ТРУДЫ

ИНСТИТУТА СИСТЕМНОГО ПРОГРАММИРОВАНИЯ РАН

PROCEEDINGS OF THE INSTITUTE FOR SYSTEM PROGRAMMING OF THE RAS

ISSN Print 2079-8156
Том 37 Выпуск 6 Часть 3

ISSN Online 2220-6426
Volume 37 Issue 6 Part 3

Институт системного
программирования
им. В.П. Иванникова РАН

Москва, 2025

ИСП РАН

ISSN 2079-8156 (Print), ISSN 2220-6426 (Online)

Труды Института системного программирования РАН
Proceedings of the Institute for System Programming of the RAS

Труды ИСП РАН – это издание с двойной анонимной системой рецензирования, публикующее научные статьи, относящиеся ко всем областям системного программирования, технологий программирования и вычислительной техники. Целью издания является формирование научно-информационной среды в этих областях путем публикации высококачественных статей в открытом доступе.

Издание предназначено для исследователей, студентов и аспирантов, а также практиков. Оно охватывает широкий спектр тем, включая, в частности, следующие:

- операционные системы;
- компиляторные технологии;
- базы данных и информационные системы;
- параллельные и распределенные системы;
- автоматизированная разработка программ;
- верификация, валидация и тестирование;
- статический и динамический анализ;
- защита и обеспечение безопасности ПО;
- компьютерные алгоритмы;
- искусственный интеллект.

Журнал издается по одному тому в год, шесть выпусков в каждом томе.

Поддерживается открытый доступ к содержанию издания, обеспечивая доступность результатов исследований для общественности и поддерживая глобальный обмен знаниями.

Труды ИСП РАН реферируются и/или индексируются в:

Proceedings of ISP RAS are a double-blind peer-reviewed journal publishing scientific articles in the areas of system programming, software engineering, and computer science. The journal's goal is to develop a respected network of knowledge in the mentioned above areas by publishing high quality articles on open access.

The journal is intended for researchers, students, and practitioners. It covers a wide variety of topics including (but not limited to):

- Operating Systems.
- Compiler Technology.
- Databases and Information Systems.
- Parallel and Distributed Systems.
- Software Engineering.
- Software Modeling and Design Tools.
- Verification, Validation, and Testing.
- Static and Dynamic Analysis.
- Software Safety and Security.
- Computer Algorithms.
- Artificial Intelligence.

The journal is published one volume per year, six issues in each volume.

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge. **Proceedings of ISP RAS** is abstracted and/or indexed in:

УДК004.45

Редколлегия

Главный редактор - Аветисян Арутюн Ишханович, академик РАН, доктор физико-математических наук, профессор, ИСП РАН (Москва, Российская Федерация)

Заместитель главного редактора – Карпов Леонид Евгеньевич, д.т.н., ИСП РАН (Москва, Российская Федерация)

Члены редколлегии

Воронков Андрей Анатольевич, доктор физико-математических наук, профессор, Университет Манчестера (Манчестер, Великобритания)
Вирбицкайте Ирина Бонавентуровна, профессор, доктор физико-математических наук, Институт систем информатики им. академика А.П. Ершова СО РАН (Новосибирск, Россия)
Коннов Игорь Владимирович, кандидат физико-математических наук, Технический университет Вены (Вена, Австрия)
Ластовецкий Алексей Леонидович, доктор физико-математических наук, профессор, Университет Дублина (Дублин, Ирландия)
Ломазова Ирина Александровна, доктор физико-математических наук, профессор, Национальный исследовательский университет «Высшая школа экономики» (Москва, Российская Федерация)
Новиков Борис Асенович, доктор физико-математических наук, профессор, Санкт-Петербургский государственный университет (Санкт-Петербург, Россия)
Петренко Александр Федорович, доктор наук, Исследовательский институт Монреаля (Монреаль, Канада)
Черных Андрей, доктор физико-математических наук, профессор, Научно-исследовательский центр CICESE (Энсенада, Баха Калифорния, Мексика)
Шустер Ассаф, доктор физико-математических наук, профессор, Технион — Израильский технологический институтTechnion (Хайфа, Израиль)

Editorial Board

Editor-in-Chief - Arutyun I. Avetisyan, Academician of RAS, Dr. Sci. (Phys.–Math.), Professor, Ivannikov Institute for System Programming of the RAS (Moscow, Russian Federation)

Deputy Editor-in-Chief – Leonid E. Karpov, Dr. Sci. (Eng.), Ivannikov Institute for System Programming of the RAS (Moscow, Russian Federation)

Editorial Members

Igor Konnov, PhD (Phys.–Math.), Vienna University of Technology (Vienna, Austria)
Alexey Lastovetsky, Dr. Sci. (Phys.–Math.), Professor, UCD School of Computer Science and Informatics (Dublin, Ireland)
Irina A. Lomazova, Dr. Sci. (Phys.–Math.), Professor, National Research University Higher School of Economics (Moscow, Russian Federation)
Boris A. Novikov, Dr. Sci. (Phys.–Math.), Professor, St. Petersburg University (St. Petersburg, Russian Federation)
Alexandre F. Petrenko, PhD, Computer Research Institute of Montreal (Montreal, Canada)
Assaf Schuster, Ph.D., Professor, Technion - Israel Institute of Technology (Haifa, Israel)
Andrei Tchernykh, Dr. Sci., Professor, CICESE Research Centre (Ensenada, Baja California, Mexico).
Irina B. Virbitskaite, Dr. Sci. (Phys.–Math.), The A.P. Ershov Institute of Informatics Systems, Siberian Branch of the RAS (Novosibirsk, Russian Federation)
Andrey Voronkov, Dr. Sci. (Phys.–Math.), Professor, University of Manchester (Manchester, United Kingdom)

Адрес: 109004, г. Москва, ул. А. Солженицына, дом 25.
Телефон: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Сайт: http://www.ispras.ru/proceedings/

Address: 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Tel: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Web: http://www.ispras.ru/en/proceedings

© Институт Системного Программирования им. В.П. Иванникова РАН, 2025

Содержание

Table of Contents

DOI: 10.15514/ISPRAS-2025-37(6)-33

CC BY

Bisimulations in Memory Finite Automata

[1] *A.N. Nepeivoda, ORCID:* 0000-0003-3949-2164 *<a_nevod@mail.ru>*
[2] *A.D. Delman, ORCID: 0009-0009-6885-8429 <adelman2112@gmail.com>*
[2] *A.S. Terentyeva, ORCID: 0009-0006-8547-3959 <mathhyyn@gmail.com>*

[1] *Ajlamazyan Program Systems Institute of the Russian Academy of Sciences, 4a, Piotr 1 st., Veskovo, Yaroslavskaya obl., 152024, Russia.*

[2] *Bauman Moscow State Technical University, 5-1, 2nd Baumanskaya st., Moscow, 105005, Russia.*

Abstract. This research aims at studying the bisimulation relation for memory finite automata, which are used as the automata model for extended regular expressions in the series of works, and encapsulate the expressiveness of the named capture groups. We propose an experimental algorithm for checking bisimulation of one-memory MFAs. For multi-memory automata, we show that, in some borderline cases, the bisimulation problem is closely related to a question of whether a parameterized word is always a solution of a given word equation of an arbitrary form.

Keywords: extended regular expressions; memory finite automata; bisimulation; word equations

For citation: Nepeivoda A.N., Delman A.D., Terentyeva A.S. Bisimulations in Memory Finite Automata. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 3, 2025, pp. 7-18. DOI: 10.15514/ISPRAS-2025-37(6)-33.

Acknowledgements. The first author was partially supported by Russian Academy of Sciences, research project No.125021302067-9.

Бисимуляции конечных автоматов с памятью

[1] *А.Н. Непейвода, ORCID:* 0000-0003-3949-2164 *<a_nevod@mail.ru>*
[2] *А.Д. Дельман, ORCID: 0009-0009-6885-8429 <adelman2112@gmail.com>*
[2] *А.С. Терентьева, ORCID: 0009-0006-8547-3959 <mathhyyn@gmail.com>*
[1] *Институт программных систем им. А.К. Айламазяна РАН,*
Россия, 152024, Ярославская обл., с. Веськово, ул. Петра Первого, д. 4а.
[2] *Московский государственный технический университет имени Н.Э. Баумана,*
Россия, 105005, Москва, 2-я Бауманская улица, д. 5. стр. 1

Аннотация. В статье рассматривается проблема бисимуляции автоматов с памятью, представляющих собой модель расширенных регулярных выражений с группами захвата в память. Построен алгоритм бисимуляции таких автоматов в случае единственной ячейки памяти. Показано, что в случае нескольких ячеек и возможности рекурсивного перезахвата в память, проблема бисимуляции включает в себя проблему проверки истинности произвольного уравнения в словах над элементами линейных контекстно-свободных языков.

Ключевые слова: расширенные регулярные выражения; конечные автоматы с памятью; бисимуляция; уравнения в словах.

Для цитирования: Непейвода А.Н., Дельман А.Д., Терентьева А.С. Бисимуляции конечных автоматов с памятью. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 7–18 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)–33.

Благодарности: Первый автор поддержан проектом НИР РАН, номер проекта 125021302067-9.

1. Introduction

Extended regular expressions have been known at least since the early 90s, when they were implemented in the text editor ed [1]. Many practical extensions of the regexes are made inside the class of regular languages, e.g., lookaheads and lookbehinds. The main exception is a back-reference support: if a capture group contains an iterated expression fragment, then back-referencing to the group may represent non-regular properties of the recognized language. For example, the expression $(a*)(b*)\backslash 1\backslash 2$ recognizes the language $\{a^n b^m a^n b^m \mid m,n \in \mathbb{N}\}$, which is a typical example of a non-context free language.

The main concern about the extended regex models is the high computational complexity of their analysis. The language inclusion problem for extended regexes even with a single memory cell is proved to be undecidable [2], the similar statement holds for language inclusion for patterns that are modelled with extended regular expressions with no restriction on memorized values and no loops and alternations in the core regex [3]. Thus, extended regex simplification tends to be a hard problem, requiring the development of approximate solutions.

It is known that some practical tools such as RE2 [4] process even academic regular expressions via non-deterministic finite automata optimizations, because they can be much faster than the exact minimization algorithm, and can preserve the structure of the regular expressions. One of such NFA optimization algorithms is merging the bisimilar state classes [5-6], which is also a well-known technique in program optimization. If some of the states in an NFA are indistinguishable from the point of view of a user of the NFA, these states can be considered as a single state, thus reducing the state space with no impact on computation traces. Equivalence of NFA, which is known to be in EXPTIME, can be tested via bisimulation as well [7]: although the bisimulation relation is finer than the language equivalence, it can be computed in polynomial time, thus giving a fast under-approximation of the equivalence test. In the case of pushdown automata, language equivalence is undecidable, while bisimulation is decidable (but non-elementary) [8-9]. Bisimulation was also applied to symbolic finite automata, i.e., finite automata with guarded transitions, in order to improve performance of extended regexes with no memory operations [10].

It seems very natural to consider bisimulation-based optimizations in the presence of capture groups, both because the bisimilarity is typically easier to compute than the language equivalence, and because the bisimulation-merging optimizations are structure-preserving, which is practical in cases when the captured data is used outside the extended regex. However, as far as we know, none of state machine formalisms supporting backreferences were considered in the papers studying the bisimulation-based optimizations and analysis. The main reason for this gap is maybe a confusion of different backreference-based formalisms for extended regular expressions, none of which is chosen as a standard nowadays, despite the fact that sometimes distinctions are minor, and some formalisms can be treated as special cases of the others [11].

It is also worth noting that the language of backref-regexes cannot be treated as a special case of a formal language with well-known bisimulation properties. It can be easily shown that $\{a^n b^n | n \in \mathbb{N}\}$, which is both context-free and Petri net language, cannot be recognized by any backref-regex. On the other hand, the language $\{\omega\omega \mid \omega \in (a|b)^*\}$ is trivially captured by the regex $\big((a|b)^*\big)\backslash 1$, while this language is known to be neither context-free nor Petri-net [12]. Thus, the backref-regexes formalism is independent from the classification of process algebras given in the paper [13].

This research aims at studying the bisimulation relation for the memory finite automata (MFA), which are used as the automata model of the extended regular expressions in the series of works [2, 14], and encapsulate the expressiveness of the named capture groups with reinitializations. We propose an experimental algorithm for checking bisimulation of one-memory MFAs, and discuss why bisimulation is hard in the case of multiple memory cells. For the latter, we show that, in some borderline cases, the bisimulation problem is closely related to a question whether a parameterized is always a solution to a given word equation of an arbitrary form.

2. Preliminaries

2.1 Bisimulation

Every state machine can be defined by its transition graph, which contains a complete description of its possible traces. If the state machine is not finite, the transition graph is infinite, taking into account the infinite set of inner states of the machine. We assume that the transition graphs are represented as labelled transition systems, in which edges are labelled by the actions possible in the state machines.

Definition 2.1.1 Given labelled transition systems τ_1, τ_2 and the action alphabet Σ, *bisimulation* is a coarsest relation ~ between states of the systems satisfying the following property.

If $q_1 \in States(\tau_1)$, $q_2 \in States(\tau_2)$, $\gamma \in \Sigma$, and $q_1 \sim q_2$, then for every transition $q_1 \xrightarrow{\gamma} {q'}_1 \in \tau_1$ there is a transition $q_2 \xrightarrow{\gamma} {q'}_2 \in \tau_2$ s.t. $\tau_1 \sim \tau_2$, and vice versa.

Systems τ_1 and τ_2 are bisimilar if and only if their starting states are in bisimulation, and, in the case of the existence of final states, any final state in τ_1 is bisimilar to a final state in τ_2, and vice versa.

State machines are bisimilar if and only if their transition graphs are bisimilar.

For most known state machine models, the bisimulation relation is strictly finer than the language equivalence relation. E.g., non-bisimilar finite automata recognizing the same language $\{a^n + 1 | \, n \in \mathbb{N}\}$ are given in Fig. 1.

A simple and natural technique for checking the bisimilarity of the transition graphs τ_1 and τ_2 can be formulated as a bisimilarity game with two players:

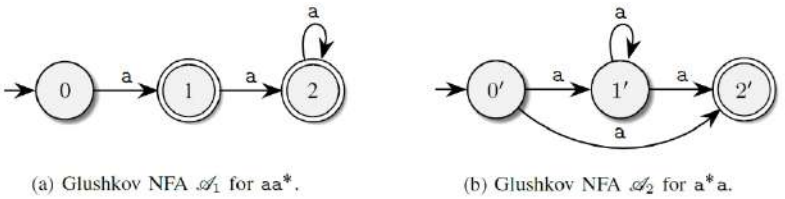
The initial player configuration is the pair $\langle q_S, {q'}_S \rangle$, where q_S, ${q'}_S$ are the starting states of τ_1 and τ_2 respectively.

Given the pair $\left\langle q_{k_i}, q_{k_j} \right\rangle$, Attacker chooses any element of the pair and a transition $q_{k_m} \xrightarrow{\gamma} q_p$ in the corresponding LTS. Defender must respond with a transition $q_{k_n} \xrightarrow{\gamma} {q'}_p$ from the remaining state in

the pair, respecting the finality of the state q_p. After this round, the player configuration includes q_p and q'_p, and the new round starts.

Attacker can play while there exists at least one transition from either q_{k_i} or q_{k_j}. If Defender cannot choose a transition at least for one player configuration reachable from the initial configuration, the LTS are not bisimilar. Otherwise, the LTS are bisimilar.

E.g., given the processes in Fig. 1, Attacker can choose the first one (namely, A_1) and play $0 \xrightarrow{a} 1$. Then, in order to respect the state finality, Defender is forced to respond with $0' \xrightarrow{a} 2'$. Since state $2'$ has no outgoing transitions, any possible second action of Attacker in the LTS makes Defender to lose.



(a) Glushkov NFA $\mathscr{A}_1$ for aa^*. (b) Glushkov NFA $\mathscr{A}_2$ for a^*a.

Fig. 1. An example of non-bisimilar equivalent NFA.

2.2 Extended Regular Expressions

The theory of the extended regular expressions followed much later than they became usual in practice. Several formalisms were proposed by different research groups, based on the details of naming capture groups and possibility of reinitialization of the groups [11]. In 2014, Markus Schmid suggested to consider only named capture groups in the extended syntax, and proposed a convenient representation in terms of state machines with restricted memory support (memory finite automata, MFA). Schmid-style regular expressions are more expressive than PCRE2-style regular expressions used in practice currently [11], but the trends of the PCRE2 development show that cyclic re-initializations are likely to appear in near future [15].

In our work, we use the latest MFA model, which includes reset memory actions [16]. Following the papers [14, 16], we also call extended regular expressions ref-words.

Definition 2.2.1 Given an input alphabet Σ and the memory set cardinality $k \in \mathbb{N}$, *a regular expression with backreferences (ref-word)* is defined recursively:

- $\gamma \in \Sigma$, ϵ, and $\&i$, where $i \leq k$, are ref-words (the latter defines reading the □-th memory cell);
- if ρ_1 and ρ_2 are ref-words, then so are $(\rho_1|\rho_2)$, $(\rho_1\rho_2)$, $(\rho_1)^*$;
- if $i \leq k$ and ρ is a ref-word including neither $\&i$ nor $[_i\tau]_i$, then $[_i\tau]_i$ is also a ref-word.

The last operation defines *capture groups*. We require memory brackets $[_i$, $]_i$ to be balanced both wrt the regular parentheses, and wrt each other. That is the only distinction from the formalism given in paper [14], which admits unbalanced capture groups.

The ref-word definition above does not specify semantics of uninitialized backreferences, e.g. in $\&1a[_1b^*]_1a\&1$. Following the terminology of the paper [11], we assume *ϵ-semantics*: all uninitialized references are meant to have the empty value. Thus, the ref-word given above recognizes the language $\{ab^nab^n \mid n \in \mathbb{N}\}$.

2.3 Memory Finite Automata

Definition 2.3.1 A *memory finite automaton* (MFA) [14] is a tuple $\langle Q, \Sigma, q_0, F, \delta\rangle$, where Q is a finite set of states, Σ is the input alphabet, $q_0 \in Q$ is a starting state, $F \subseteq Q$ are final states, and $\delta: (Q \times \Sigma \cup \{\epsilon\} \cup \{1,2,..,k\}) \to Q \times \langle o, c, r, s\rangle^k$ is a transition table. The symbols o, c, r, s are

memory instructions for opening, closing, resetting, and staying in the current memory status respectively.

An MFA *configuration* is a tuple $\langle q, \omega, (u_1, m_1), .., (u_k, m_k)\rangle$, where q is a current state, ω is an input suffix to be read, and for all $i, 1 \le i \le k$, (u_i, m_i) is an i-th memory state, consisting of a stored string u_i and a memory status $m_i \in \{O, C\}$. Given an input ω_0, the initial memory state is $\langle q_0, \omega_0, (\epsilon, C), .., (\epsilon, C)\rangle$, i.e., all the memory cells are assumed to be closed and to store empty strings.

A transition from configuration $\langle q, \upsilon\omega, (u_1, m_1), .., (u_k, m_k)\rangle$ to $\langle q', \omega, (u'_1, m'_1), .., (u'_k, m'_k)\rangle$ is possible if there is a transition rule from q to q' labelled β, and having the instructions $\langle b_1, .., b_k\rangle$ such that either:

- $\beta \in \Sigma \cup \{\epsilon\}$ and $\upsilon = \beta$ (usual transition), or
- $\beta \in \{1, .., k\}$ and $\upsilon = u_\beta$, given $m'_\beta = C$ (reading transition).

The memory states are updated as follows:

$$m'_i = b_i(m_i) = \begin{cases} C, b_i = c \vee b_i = r \vee (b_i = s \wedge m_i = C) \\ O, b_i = o \vee (b_i = s \wedge m_i = O) \end{cases}, u'_i = \begin{cases} u_i, m'_i = C \\ \epsilon, b_i = r \\ \upsilon, b_i = o \\ u_i\upsilon, b_i = s \wedge m_i = O \end{cases}$$

That is, the memory status is updated before processing the tape. If the resulting memory status is closed, the memory cell does not append the currently read input fragment υ. If the status is open, the string υ is appended to the current memory. Both reset and open instructions clear the memory, but the first sets C as a status, while the second sets O.

An example of a memory finite automaton for a nonregular language $\{a^{n+k}ba^n | n > 0\}$ is shown in Fig. 2. For convenience, the memory actions on the edges are given in the brief form: a label only lists cells that are closed, reset and opened along the edge, not mentioning the staying in the current status instruction. The MFA that are used in the examples are uniformly generated from the corresponding ref-words, using an MFA construction algorithm based on the Glushkov construction [17], and utilising the reset memory action in order to avoid ϵ-transitions.

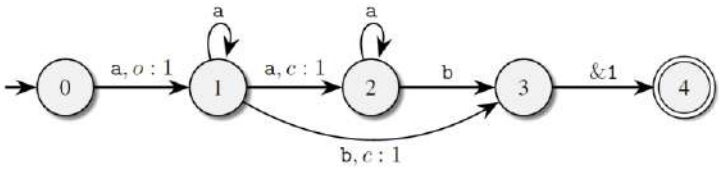


Fig. 2. An example of a memory finite automaton.

3. Bisimulation in Memory Finite Automata

Definition 3.1. Let $A = \langle Q, \Sigma, q_0, F, \delta\rangle$ be a memory finite automaton over k memory cells. Its *transition graph* $G(A)$ is defined as follows.

- $(q_0, \langle(\epsilon, C), .., (\epsilon, C)\rangle)$ is the starting node of the transition graph;
- given a configuration $N_i = (q, \langle(u_1, m_1), .., (u_k, m_k)\rangle)$, children of the node labelled with N_i are the nodes labelled with configurations reachable by all possible one-step transitions from N_i.

In most practical cases, the memory statuses in all states of A are determined by the states (i.e., given a state q, the values of $m_1, .., m_k$ can be restored independently of the previous trace). We assume that the condition holds for all MFA considered, and omit m_i values in the node configurations of transition graphs. While the notion "capture groups" is usually used in the context of ref-words, we also say that, given an MFA A the subgraphs between the open and close operations wrt the cell k

are "capture groups" for the reference k in A. Additionally, we assume that every capture group of an MFA is useful, i.e., there exists at least one path in the transition graph where the value accumulated in the group is used by a reference. In order to distinguish the actions with the same label in distinct A_1 and A_2, we sometimes mark the references in the transition graphs with the corresponding subscripts, i.e., $\&k_{A_1}$ and $\&k_{A_2}$.

Given the MFA formalism, we make the following general assumption about the input commands controlled by the user.

Assumption 3.1. Given the transition graphs $G(A_1)$ and $G(A_2)$, the transitions from nodes N_{j_1} and N_{j_2} both labelled with $\&i$ are equal if and only if the stored u_i value in the N_{j_1} configuration coincides with the stored u_i value in the configuration of N_{j_2} .

Now we are ready to give the definition of the bisimulation relation in the terms of MFA.

Definition 3.2. Given MFA A_1 and A_2, the MFA are *bisimilar* (denoted $A_1 \sim A_2$), if and only if their transition graphs $G(A_1)$ and $G(A_2)$ are bisimilar.

When considering the problem of MFA bisimulation, it is natural to assume that users have no direct access to memory operations (open, close, reset). On the other hand, a reference to a memory is an explicit control action, which is distinct from any other user action. Thus, we introduce the notion of an action automaton, which describes possible sequences of

control actions in the process graph of a MFA.

Definition 3.3. Given k-cell MFA A, its *action NFA* $\pi_M(A)$ results from A by erasing memory operations from edge labels of A. The symbols $\&i$ $(1 \leq i \leq k)$ become elements of the input action alphabet Act for $\pi_M(A)$.

If MFA A_1 and A_2 are in the bisimulation relation, then the control traces of them must coincide, thus, $\pi_M(A_1) \sim \pi_M(A_2)$. In the latter case, we say that A_1 and A_2 are action-bisimilar. This relation induces a relation on the states of the MFA themselves, however, the action-bisimilarity is not enough to provide real MFA bisimilarity: in order to imitate the action $\&k_{A_1}$ played by Attacker, not only Defender must respond with the action $\&k_{A_2}$, but also, they must guarantee that the reference reads exactly the same value. Otherwise, the two actions $\&k$ cannot be considered as equal.

For example, let us consider the MFA given in Fig. 3. Their action NFA are trivially bisimilar, being equal, but Attacker has the following winning strategy.

- Play $S' \xrightarrow{a} 1'$.
- If Defender responds with $S \xrightarrow{a} 1$, then play $1' \xrightarrow{\&1} 3'$. Defender loses, being unable to reset memory value to ϵ.
- If Defender responds with $S \xrightarrow{a} 2$, then play $1' \xrightarrow{a} 2'$. Now Defender cannot accumulate anything in the capture group.

4. Bisimulation for One-Cell Memory Finite Automata

We start with the simplest class of memory finite automata, namely, automata using a single memory cell. In order to obtain a winning strategy, Defender must be able to repeat the Attacker's decisions made inside the capture groups. The two possible sorts of decisions are:

- given a loop inside a capture group, decide whether to continue iterations or to exit;
- given a captured branching by different $\gamma_1, \gamma_2 \in \Sigma$, choose the alternation branch.

Note that, given a branching without a loop in a capture group, we are still able to capture only a finite set of possible strings, provided that the MFA has the only memory cell. Hence, the second sort of the decisions, considered separately, is somewhat "weaker" than the first: memorization of alternations without loops can be modelled by finite state models. Hence, we are interested in the

decisive actions occurring only in capture groups containing loops. If the actions are considered outside the capture groups, we mention this fact explicitly.

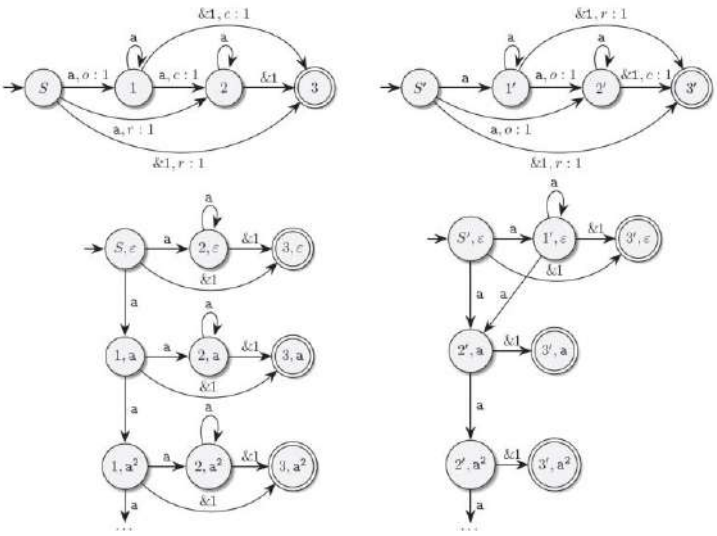


Fig. 3. Non-bisimilar MFA and their transition graphs.

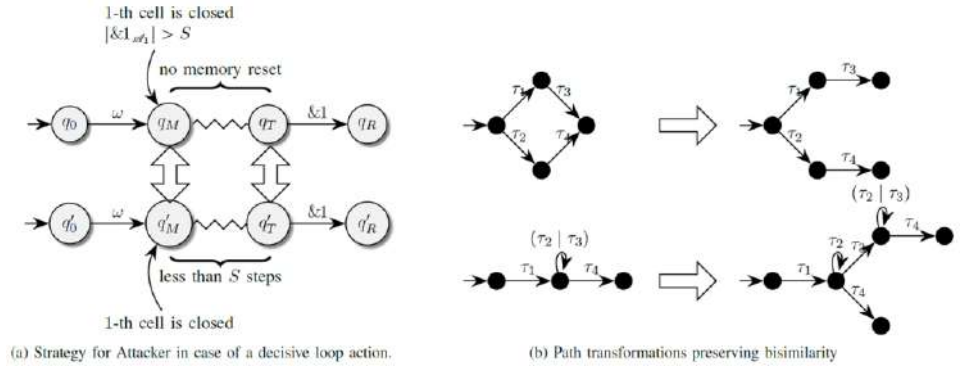
Proposition 4.1. Let $\pi_M(A_1) \sim \pi_M(A_2)$. If there exists a state q_1 in A_1 s.t. it has an outgoing decisive action (inside a capture group) and none of the states in A_2 that are action-bisimilar to q_1 has such a decisive action inside the capture group, then $\neg(A_1 \sim A_2)$.

Proof. Let us show Attacker's winning strategy in the case of the decisive action in A_1 which is not action-bisimilar to any decisive action in A_2. First, given the loop in the capture group where the decisive action occurred (or is to occur), Attacker decides to play the looping back action at least S times, where S is the sum of the number of states in A_1 and A_2, and additionally chooses such a number of iterations that the value of $\&1_{A_1}$ is not equal to $\&1_{A_2}$. Second, Attacker chooses the shortest possible path in A_1 to the action $\&1$. In order to imitate the action $\&1_{A_1}$, Defender is forced to reset the value of $\&1_{A_1}$, but the value of $\&1_{A_1}$ is longer than any possible value read along at most transitions. Due to MFA semantics, no transition captured in the memory cell 1 can refer to the memory cell, thus, no more than S letters can be captured. See Fig. 4 (a).

Proposition 4.1 demonstrates an important property of bisimilar traces within MFA semantics: if the memory cell captures any loop, then the capturing of the input fragment read along the loop must be done synchronously in both MFA. Otherwise, Attacker can apply the strategy shown in Fig.4 and force the Defender to lose, looping inside the capture long enough. Hence, in bisimilar MFA, all the traces with loops must be action-bisimilar in their "decisive" fragments, as well as synchronized in terms of write and read operations: if a trace of A_1 contains k read operations after a capturing with a loop, then the bisimilar trace of A_2 must also contain k read operations without re-capturing memory between them.

In Fig. 3, while the states 1 and $2'$ with the decisive actions are action-bisimilar, the process graphs are not. The reason is that Proposition 4.1 induces a narrowing on action-bisimilarity relation, taking into account the synchronisation that must occur in the capture groups previously visited. Namely, if the state 1 is visited by a player, then $2'$ is visited as well, and all the actions reachable after closure of the memory cell containing state 1 in its capture group must be also reachable after closure of the memory cell containing state $2'$. The given condition fails due to existence of state 2 in A_1, allowing Attacker to read additional letters a after the memory is closed, while in A_2 the only reachable state after the memory closure is the state $3'$, and Defender is unable to read any a in this state. Hence,

not only the decisive actions must be bisimilar in the bisimilar MFA, but also the sub-automata having the nodes with the decisive actions as starting states must be bisimilar.



(a) Strategy for Attacker in case of a decisive loop action.

(b) Path transformations preserving bisimilarity

Fig. 4. Attacker strategy in case of a decisive action in a loop within a capture group, and bisimulation-preserving path transformation.

Now we are to deal with the case when the memorized strings can be re-captured by Defender. According to Proposition 4.1, this case can only occur when the strings are read along the paths in that do not include any loops. The set of all possible memory values read along the paths is finite. Hence, we can annote the paths storing these values with a finite number of indexes, and then track that the indexes coincide when the memories are referenced to. One important case of such annotation is reset-annotation: that is, tracking that states on the given path store the empty value. However, in order to make the annotation, it is crucial to split the paths carrying unbounded and bounded strings in the memory cell. The following proposition is a corollary of the fact that ref-words satisfy certain Kleene algebra theorems, such as right distributivity, nesting, and fusion.

Proposition 4.2 Given any MFA A, we can transform it into a bisimilar MFA using the following transformations:

- given two paths having the same destination node, duplicate the destination node and split them apart (subfigure (b) in Fig. 4, upper diagram);
- and, given a looped path, extract a first iteration over a certain subpath into an explicit subgraph (subfigure (b) in Fig. 4, lower diagram).

Using these two transformations, we can split subpaths storing constant strings from the ones storing possibly unbounded strings, and apply the annotation procedure in order to validate constant value re-capturing.

For example, such transformations allow us to distinguish cases of bisimilar MFA based on the ref-words $a^*\big([_1 a^*]_1\big)^*\&1$ and $\big([_1 a^*]_1\big)^*\&1$ – where the latter is transformed to $\underbrace{a^*}_{\text{annotated with } \epsilon}\big([_1 a^*]_1\big)^*\&1$, and non-bisimilar MFA based on $[_1 b]_1 \underbrace{a^*}_{annotated\ with\ b}\big([_1 a^*]_1\big)^*\&1$ and $[_1 b]_1\big([_1 a^*]_1\big)^*\&1$. The bisimilarity in the second case is broken, because we cannot "unfuse" the looped capture group in $[_1 b]_1\big([_1 a^*]_1\big)^*\&1$ preserving the constant b-annotation. Hence, the path fragments preceding captured decisive actions cannot be made bisimilar by any combination of transformations given in Proposition 4.2.

It is tempting to conclude that the whole capture groups with loops are required to be action-bisimilar in order to guarantee that $A_1 \sim A_2$. Nevertheless, in some bisimilar MFA non-decisive actions can occur asynchronously even along the paths including loops. An example is given in Fig. 5. The capture groups are non-bisimilar, while the states 2 and $2'$ with action-decisive outgoing transitions are both captured. The reason for this non-trivial bisimulation is rooted in the theory of word equations. Namely, for every $\omega = a^k, a\omega = \omega a$, and that is why the values of the references $\&1_{A_1}$

and $\&1_{A_2}$ always coincide. Due to Proposition 4.1, non-bisimilarity of this sort, given a loop in the capture group, can occur only with respect to constant prefixes and suffixes surrounding the input fragment captured in the loop. That is, given the captured strings X_1, X_2, …, X_n, they can be rearranged between constant fragments in such a way that the resulting equation $v_0X_1v_1X_2 \dots X_n = X_1\omega_1X_2 \dots X_n\omega_n$, where v_i and ω_i are string constants, should hold for any $X_1, \dots, X_n$ values in the languages of the decisive fragments. We assume that the equation is minimal, that is, the equation cannot be split to lesser length-equal fragments. E.g. $aX_1bX_2 = X_1aX_2b$ can be definitely represented as a system of minimal equations $\begin{cases} aX_1 = X_1a \\ bX_2 = X_2b \end{cases}$, hence, it is not minimal.

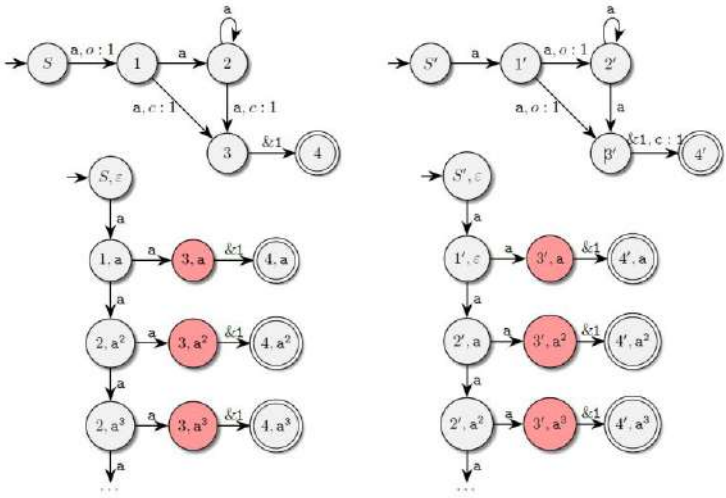


Fig. 5. Bisimilar MFA with non-bisimilar capture groups.

Equations of the form $v_0X_1v_1X_2 \dots X_n = X_1\omega_1X_2 \dots X_n\omega_n$ are well-studied [18]: any their solution X_i-component belongs to a simple regular language following the pattern $(\tau_1\tau_2)^n\tau_1$, i.e., the language of fractional powers of word $\tau_1\tau_2$. Therefore, bisimulation of MFAs with non-bisimilar capture groups including loops can occur, only if the loops themselves, in any their composition, are marked with powers of the same string.

In order to process bisimulation cases of the sort described above, we propose to use memory revision algorithm.

- All the decisive alternation fragments in capture groups are replaced with fresh string parameters.
- All the loops with no decisive alternations inside them (i.e., iterating along the constant path ξ) are replaced with the parameterized word ξ^{k_i}, where k_i is a fresh integer parameter or, in case of nested loops along the same subwords, parameter expression.
- The resulting values are unified, i.e., the parameterized and constant values are substituted instead of the path fragments. If the result of the substitution is a trivial equality, then the bisimulation can hold, otherwise, there is at least one path along action-bisimilar states that results in different values of $\&1_{A_1}$ and $\&1_{A_2}$.

Regarding nested loops, the procedure depends on their semantics. If given a loop with a minimal path reading $\xi_1\xi_2$, this loop contains an inner loop starting in ξ_1-th position, the inner loop contains a decisive alternation unless it iterates along the constant path $\xi_2\xi_1$ or a power of its primitive root ξ; in the latter case, words read along the both loops (the inner and the outer one) can be mapped to

a single parameterized word $\xi^{k_{i_1}+k_{i_2}}$, where k_{i_1} is a parameter denoting the outer iterations count, and k_{i_2} denotes the total inner iterations count.

Only if all the decisive actions are synchronized wrt the order induced by the synchronization, all the constant memory annotations are synchronized wrt each other, and all the memories are revised to be equal before referencing to them, then we can state that $A_1 \sim A_2$.

A prototype of the described algorithm, as well as the ref-word — MFA conversion and a fuzz equivalence testing module for MFA, is developed in the Chipollino formal language converter https://github.com/OnionGrief/Chipollino — the application that allows you to generate, transform and analyze various representations of formal languages (e.g., regular expressions, automata: FA, MFA, PDA). The MFA can be input by hand via a simple DSL language, or constructed from ref-words using Schmid algorithm [14], and an analogue of the Glushkov construction, merging ϵ-closures; random MFA and ref-words generators are supported in the project as well.

5. Multiple Memory Cells

In the case of multiple memory cells, the bisimulation problem meets new challenges, since the corresponding MFA paths can include references inside capture groups, as well as iterations over re-captured references. In the case of a single memory cell, at least we can assume that the capture groups used by bisimilar reference actions are action-equivalent (i.e., the languages of the corresponding action NFAs coincide). When the reference actions can occur inside the capture groups, that statement is not true. For example, values of capture groups with no decisive actions can behave the same way as the constant strings. An example of ref-words producing MFA with non-equivalent traces inside the capture groups is the pair $a[_1 a]_1 a[_2 \&1 a]_2 \&2$ and $[_1 a]_1 a[_2 a \&1]_2 a \&2$. The value stored in the cell 1 is a fixed constant, and its impact on the value stored in the cell 2 is exactly the same as in the case when this constant is explicitly stored.

If the memory cells have a cyclic dependency [19], the bisimulation problem very likely becomes undecidable, because the languages of the memory cells have even more expressible power than languages of the whole ref-words. An example is the ref-word $\left([_2 a \&1 b]_2 [_1 a \&2 b]_1\right)^*$: its memory cells 1 and 2 store the languages $\{a^{2n} b^{2n} | n \in \mathbb{N}\}$ and $\{a^{2n+1} b^{2n+1} | n \in \mathbb{N}\}$ respectively, and both can be proved to be inexpressible by any ref-word. Actually, given any linear context-free language L, a memory cell language can capture L; and, provided sequences $[_1 \Phi]_1 \Psi \&1, \Phi[_1 \Psi]_1 \&1$ depending on words in generated by these languages, the bisimulation between them holds if and only if the word equation $\Phi = \Psi$ for members of these languages is always true. This observation shows that the multi-cell bisimulation is definitely a hard problem, and maybe even undecidable.

6. Conclusion

The bisimulation problem of memory finite automata appears to be tractable at least in some practical cases. If a bisimulation is constructed on the states of A itself, then the bisimilar nodes in A can be merged with no change of the captured values, or the MFA traces. While the minimization problem for MFA is undecidable, the optimisation by bisimulation can be a decent approximation of the minimization, especially in the case when the MFA is deterministic. Efficiency estimation of this optimization is a future work of our MFA project.

Another interesting question is the decidability and complexity issue of MFA bisimulation in the general case. The existential theory of strings is known to be decidable [20-21], however, the bisimulation problem requires an algorithm not to decide a sole question whether a word equation has at least one solution, but to check if the language generated by the previous traces of MFA always satisfies this equation.

References

[1]. I. Free Software Foundation. (1993–2024) The GNU ed line editor. [Online]. Available: https://www.gnu.org/software/ed/manual/edmanual.html

[2]. D. D. Freydenberger, “Inclusion of pattern languages and related problems,” Ph.D. dissertation, Goethe University Frankfurt am Main, 2011. [Online]. Available: http://publikationen.ub.uni-frankfurt.de/frontdoor/index/index/docId/22351

[3]. T. Jiang, A. Salomaa, K. Salomaa, and S. Yu, “Inclusion is undecidable for pattern languages,” in Automata, Languages and Programming, A. Lingas, R. Karlsson, and S. Carlsson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 301–312.

[4]. Google. (2010–2023) Official public repository of RE2 library. [Online]. Available: https://github.com/google/re2

[5]. C. Bianchini, B.Riccardi, A. Policriti, and R.Romanello, “Incremental NFA minimization,” in CEUR Workshop Proceedings, 2022.

[6]. C. Fu, Y. Deng, D. N. Jansen, and L. Zhang, “On equivalence checking of nondeterministic finite automata,” in Dependable Software Engineering. Theories, Tools, and Applications, K. G. Larsen, O. Sokolsky, and J. Wang, Eds. Cham: Springer International Publishing, 2017, pp. 216–231.

[7]. F. Bonchi and D. Pous, “Checking NFA equivalence with bisimulations up to congruence,” SIGPLAN Not., vol. 48, no. 1, p. 457–468, jan 2013. [Online]. Available: https://doi.org/10.1145/2480359.2429124

[8]. C. Stirling, “Decidability of bisimulation equivalence for normed pushdown processes,” Theoretical Computer Science, vol. 195, no. 2, pp. 113–131, 1998, concurrency Theory. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397597002168

[9]. M. Benedikt, S. Goller, S. Kiefer, and A. S. Murawski, “Bisimilarity of pushdown automata is nonelementary,” in 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science, 2013, pp. 488–498.

[10]. L. D’Antoni and M. Veanes, “Forward bisimulations for nondeterministic symbolic finite automata,” in Tools and Algorithms for the Construction and Analysis of Systems, A. Legay and T. Margaria, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 518–534.

[11]. M. Berglund and B. van der Merwe, “Re-examining regular expressions with backreferences,” Theoretical Computer Science, vol. 940, pp. 66–80, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397522006570

[12]. J. L. Peterson, Petri Net Theory and the Modelling of Systems. Prentice-Hall, April 1981.

[13]. L. Aceto, A. Ingolfsdottir, and J. Srba, The algorithmics of bisimilarity, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2011, pp. 100–172. [Online]. Available: https://doi.org/10.1017/CBO9780511792588.004

[14]. M. L. Schmid, “Characterising REGEX languages by regular languages equipped with factor-referencing,” Information and Computation, vol. 249, pp. 1–17, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540116000109

[15]. P. Hazel. (1997–2021) PCRE2 electronic manual. [Online]. Available: https://www.pcre.org/current/doc/html/index.html

[16]. D. D. Freydenberger and M. L. Schmid, “Deterministic regular expressions with back-references,” Journal of Computer and System Sciences, vol. 105, pp. 1–39, 2019. [Online]. Available: https://doi.org/10.1016/j.jcss.2019.04.001

[17]. V. M. Glushkov, “The abstract theory of automata,” Uspekhi Mat. Nauk, vol. 16, pp. 3–62, 1961. [Online]. Available: http://mi.mathnet.ru/rm6668

[18]. J. D. Day, F. Manea, and D. Nowotka, “The hardness of solving simple word equations”, 42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017). Leibniz International Proceedings in Informatics (LIPIcs), Volume 83, pp. 18:1-18:14, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2017) Available: https://doi.org/10.4230/LIPIcs.MFCS.2017.18

[19]. D. Ismagilova and A. Nepeivoda, “Disambiguation of regular expressions with backreferences via term rewriting,” Modeling and Analysis of Information Systems, vol. 31 iss. 4, pp. 426-445, 2024. [Online]. https://doi.org/10.18255/1818-1015-2024-4-426-445

[20]. J. D. Day, V. Ganesh, and F. Manea, “Formal languages via theories over strings: An overview of some recent results,” Bull. EATCS, vol. 140, 2023. [Online]. Available: http://eatcs.org/beatcs/index.php/beatcs/article/view/765

[21]. G. S. Makanin, “The problem of solvability of equations in a free semigroup,” Mat. Sb. (N.S.), vol. 103(145), pp. 147–236, 1977.

Информация об авторах / Information about authors

Антонина Николаевна НЕПЕЙВОДА – научный сотрудник Института программных систем РАН. Сфера научных интересов: теория формальных языков, программная семантика, математическая логика и функциональное программирование.

Antonina Nikolaevna NEPEIVODA – researcher in the Program Systems Institute of RAS. Research interests: formal language theory, program semantics, mathematical logic, and functional programming.

Александр Дмитриевич ДЕЛЬМАН – студент Московского государственного технического университета им. Н. Э. Баумана. Сфера научных интересов: конструирование компиляторов и обработка естественного языка.

Aleksandr Dmitrievich DELMAN – student of the Bauman Moscow State Technical University. Research interests: compiler design and natural language processing.

Анна Сергеевна ТЕРЕНТЬЕВА – студент Московского государственного технического университета им. Н. Э. Баумана. Сфера научных интересов: конструирование компиляторов.

Anna Sergeevna TERENTYEVA – student of the Bauman Moscow State Technical University. Research interests: compiler design and optimization.

DOI: 10.15514/ISPRAS-2025-37(6)-34

CC BY

Типовая архитектура высокопроизводительной вычислительной системы для решения задач численного моделирования

С.Ю. Мокшин, ORCID: 0000-0002-7454-6597 <s.yu.mokshin@vniitf.ru>
А.О. Игнатьев, ORCID: 0000-0003-4902-2123 <a.o.ignatyev@vniitf.ru>
А.И. Мельников, ORCID: 0009-0008-1982-9501 <a.i.melnikov@vniitf.ru>
Д.В. Иванков, ORCID: 0000-0003-4254-0104 <d.v.ivankov@yandex.ru>

Российский Федеральный Ядерный Центр – Всероссийский научно-исследовательский институт технической физики имени академика Е.И. Забабахина,
Россия, 456770, г. Снежинск, Челябинская область, ул. Васильева, 13.

Аннотация. Работа посвящена тематике высокопроизводительных вычислительных систем (ВВС), предназначенных для решения задач численного моделирования. Приводится подробное описание архитектуры, функционального состава, а также используемого системного программного обеспечения и особенностей обработки информации в ВВС. Авторы работы опираются на собственный многолетний опыт создания ВВС для предприятий России. Данная работа может быть полезна специалистам, занимающимся разработкой и эксплуатацией современных вычислительных систем высокой производительности, предназначенных для проведения научных исследований.

Ключевые слова: архитектура вычислительной системы; высокопроизводительные вычисления; обработка данных математического моделирования; системное программное обеспечение.

Для цитирования: Мокшин С.Ю., Игнатьев А.О., Мельников А.И., Иванков Д.В. Типовая архитектура высокопроизводительной вычислительной системы для решения задач численного моделирования. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 19–44. DOI: 10.15514/ISPRAS–2025–37(6)–34

Typical HPC System Architecture for Numerical Simulation

S.Yu. Mokshin, ORCID: 0000-0002-7454-6597 <s.yu.mokshin@vniitf.ru>
A.O. Ignatyev, ORCID: 0000-0003-4902-2123 <a.o.ignatyev@vniitf.ru>
A.I. Melnikov, ORCID: 0009-0008-1982-9501 <a.i.melnikov@vniitf.ru>
D.V. Ivankov, ORCID: 0000-0003-4254-0104 <d.v.ivankov@yandex.ru>

Russian Federal Nuclear Center –
Zababakhin All-Russian Research Institute of Technical Physics,
13, Vasilieva street, Snezhinsk, Chelyabinsk region, 456770, Russia.

Abstract. The paper discusses issues related to the topic of high-performance computing (HPC) systems designed to solve numerical simulation problems. A detailed description of the architecture, functional composition, as well as the system software used and information processing features in the HPC systems is provided. The authors of the work rely on their own long-term experience in creating HPC systems for Russian scientific and industrial centers. This work may be useful to specialists involved in the development and operation of modern high-performance computing systems designed for scientific research.

Keywords: computing system; calculations; high-performance computing simulation; mathematical simulation.

For citation: Ignatyev A.O., Mokshin S.Yu., Melnikov A.I., Ivankov D.V. Typical HPC system architecture for numerical simulation. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 3, 2025. pp. 19-44 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-34

1. Введение

Понятие высокопроизводительной вычислительной системы (суперкомпьютера) появилось в конце 60-х годов и применялось в то время к уникальным по производительности вычислительным системам Сеймура Крэя [1]. Под суперкомпьютером понималась вычислительная система, создаваемая, как правило, в единственном экземпляре или в небольшом количестве, значительно превосходящая по производительности массово выпускаемые образцы и ориентированная в первую очередь на вычисления, связанные с обработкой большого количества данных за приемлемое время (это, как правило, задачи численного моделирования физических процессов).

Первоначально в суперкомпьютерах применялись специально разработанные дорогие быстродействующие процессоры с уникальными архитектурами. Суперкомпьютер тех времен состоял из одного или нескольких (до 10) центральных процессоров (как правило, с векторной архитектурой), взаимодействующих с общей оперативной памятью. Каждый суперкомпьютер работал под управлением собственной операционной системы, на нем использовались специально написанные под него компиляторы (как правило, для языков Алгол или Фортран).

Однако, появление в конце 90-х годов прошлого столетия серийных универсальных процессоров с производительностью, сравнимой с производительностью ранее выпускаемых суперкомпьютеров, а также коммуникационного оборудования, способного передавать информацию между ними с приемлемо высокой скоростью, привело к революции в суперкомпьютеростроении. С тех пор и по сегодняшний день, все суперкомпьютеры имеют кластерную архитектуру, то есть состоят из большого числа как правило однотипных серверов, объединенных высокоскоростным коммуникационным оборудованием.

Первая серьезная публикация о высокопроизводительной вычислительной системе (ВВС) такого типа на русском языке появилась в 2003 году [2]. Чуть позже вышла докторская диссертация А.О. Лациса на эту же тему, с которой авторы имели возможность ознакомиться. Многие последующие публикации, как в журналах, так и в Интернете, посвящены, как правило, либо обоснованиям необходимости применения ВВС, либо отдельным архитектурным аспектам конкретного экземпляра или семейства. Лишь в 2020 году усилиями

коллег из ФГУП «РФЯЦ-ВНИИЭФ» были разработаны ГОСТ на тематику ВВС [3, 4], в которых вводятся общие понятия о структуре ВВС и порядке проведения их испытаний. И хотя за прошедшие 20 лет общая архитектура ВВС, используемых для решения задач численного моделирования, практически устоялась, но до сих пор не существует доступного и полного ее описания.

В настоящей работе дано подробное описание архитектуры ВВС, предназначенной для решения задач численного моделирования, включающее в себя описание основных функциональных компонент ВВС, используемое системное и прикладное программное обеспечение, а также рассматриваются особенности обработки информации в ВВС такого типа. Авторы работы опираются на многолетний опыт участия сотрудников РФЯЦ-ВНИИТФ в создании ВВС и их компонент для предприятий России, а также на опыт подобных разработок за рубежом [5-10].

2. Основные компоненты ВВС

Архитектура современной ВВС, предназначенной для решения задач численного моделирования, строится исходя из основной цели создания такой ВВС – обеспечения выполнения массово-параллельных задач численного моделирования на всех доступных пользователю вычислительных ресурсах ВВС. Под архитектурой ВВС здесь понимается общее описание структуры и функций ВВС, связей между ее компонентами на уровне, достаточном для понимания принципов работы, не включающее деталей технического и физического характера. Исходя из этого, ВВС должна содержать очень большое количество вычислительных узлов, объединенных высокопроизводительной вычислительной сетью передачи сообщений, подключенных к доступным с каждого вычислительного узла файловым ресурсам и оснащенным системным и прикладным программным обеспечением, обеспечивающим заданную функциональность.

На практике вычислительным узлом ВВС является материнская плата, на которой установлены одно или несколько вычислительных устройств (процессоров), общая для них оперативная память, контроллеры ввода/вывода и управления, сетевые адаптеры и, при необходимости, дисковые накопители. Каждый узел объединяется с другими узлами ВВС посредством коммуникационного (сетевого) оборудования. В то же время, с точки зрения пользователей, ВВС является единым цельным объектом, подключенным к сети предприятия.

В зависимости от назначения, узлы могут быть разного типа и комплектации. Каждый узел работает под управлением собственного экземпляра операционной системы (ОС узла). Согласованную работу всех узлов обеспечивает системное программное обеспечение ВВС (СПО ВВС).

ВВС состоит из нескольких систем и подсистем, выполняющих различные функции:

- вычислительное поле (ВП) – основная из подсистем ВВС (с точки зрения реализации её функций), предназначена для проведения вычислений;
- система хранения (СХ) – предназначена для хранения исходных данных, промежуточных и финальных результатов вычислений;
- подсистема доступа (ПД) – обеспечивает непосредственный доступ пользователей к ВВС, подготовку начальных данных для расчетов, обработку результатов расчетов и запуск задач на ВП;
- подсистема управления и мониторинга (ПСУМ) – реализует функции общего управления работой всех узлов ВВС, включая запуск и прохождение задач, мониторинг и диагностику оборудования и системного программного обеспечения;
- сервисная подсистема (СП) – обеспечивает организацию единого пространства пользователей (подсистему учета и аутентификации пользователей), доменных

имен, информационных и некоторых других вспомогательных для ВВС сервисов;

- коммуникационная подсистема (КПС) – обеспечивает сетевое взаимодействие всех узлов ВВС между собой.

Таким образом, функциональную схему ВВС можно представить следующим образом (см. рис. 1).

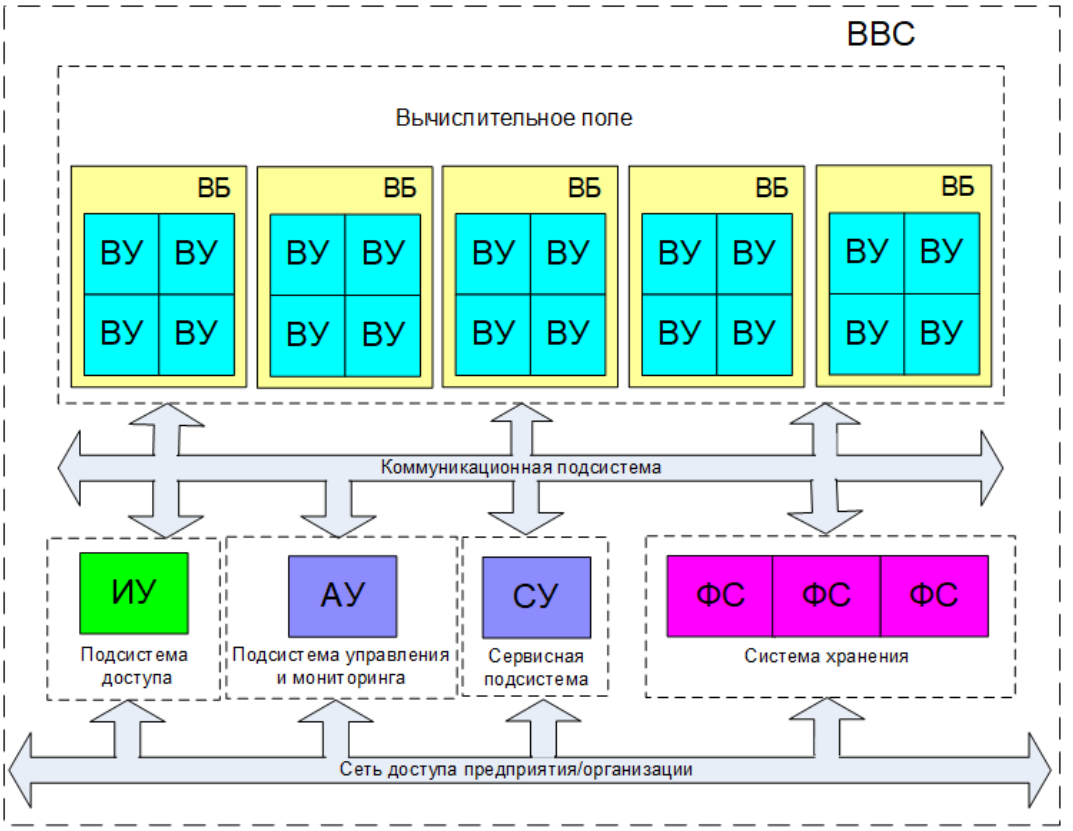


Рис. 1. Функциональная схема ВВС.
Fig.1. HPC system functional diagram.

На этом рисунке приняты следующие обозначения:

- ВУ – узлы ВП (вычислительные узлы);
- ВБ – вычислительные блоки ВП;
- ИУ – узлы подсистемы доступа (инструментальные узлы);
- СУ – узлы сервисной подсистемы (сервисные узлы);
- АУ – узлы подсистемы управления и мониторинга (административные узлы);
- ФС – узлы системы хранения (файловые серверы).

Рассмотрим каждую из этих подсистем подробнее.

2.1 Вычислительное поле

Вычислительное поле – подсистема ВВС, предназначенная для проведения расчетов, содержит в себе множество вычислительных узлов (ВУ), объединенных в вычислительные блоки (ВБ). Фактически реализует суммарную вычислительную мощность ВВС для расчета задач.

Каждый ВУ содержит один или несколько процессоров, общую для них оперативную память и устройства или каналы передачи информации (сетевые адаптеры, интегрированные порты), подключенные к коммуникационной подсистеме, также может содержать сопроцессоры или различные ускорители вычислений, которые, в некоторых случаях, могут вполне успешно использоваться для решения задач численного моделирования.

Суммарная пиковая производительность вычислительного поля Rpeak вычисляется по формуле:

$$R_{peak} = N_{CPU}*F*N_C*N_{CORE}*M$$

где:

N_{CPU} – количество процессоров в одном вычислительном узле;

F – частота процессора, ГГц;

N_C – количество конвейеров с плавающей запятой в одном ядре процессора;

N_{CORE} – количество ядер в процессоре;

M – количество вычислительных узлов в вычислительном поле.

Загрузка операционной системы на ВУ осуществляется либо с локального жесткого диска, либо по сети с системного узла. Как правило, на ВВС отдается предпочтение сетевому способу загрузки операционной системы (ОС) на ВУ, обеспечивающему высокую эффективность администрирования, гибкость управления и некоторую экономию денежных средств за счет отсутствия необходимости покупки жестких дисков для ВУ (особенно при большой размерности вычислительного поля).

При большом количестве ВУ (а на реальных ВВС в списке TOP-500 [11] это количество составляет от нескольких тысяч до десятков тысяч) вычислительные узлы целесообразно объединять в вычислительные блоки (ВБ). Такое деление на логические единицы – ВБ реализует возможность управления ими индивидуально, обеспечивая возможность проведения работ с необходимыми и конкретными ВБ в ходе эксплуатации ВВС: проведения профилактических работ, аварийного поблочного отключения ВВС при отказе систем жизнеобеспечения ВВС (энергоснабжения, охлаждения) и т.п. Кроме того, очевидно, что управлять очень большим вычислительным полем с одного системного узла в случае сетевой загрузки – невозможно. Поэтому такое разделение структуры вычислительного поля на ВБ позволяет объединять от 80-ти до 120 ВУ в один вычислительный блок, при этом загрузка ОС на ВУ данного ВБ обеспечивается с отдельного системного узла, входящего в состав этого ВБ. Таким образом, в составе вычислительного поля ВВС может быть некоторое множество ВБ и каждый ВБ содержит в себе системный узел ВБ, обеспечивающий:

- бездисковую загрузку каждого ВУ блока;
- диагностику всех вычислительных узлов блока.

Важно отметить, что такая организация структуры вычислительного поля позволяет обеспечивать и его отказоустойчивость, так как, фактически, каждый системный узел представляет собой резервную копию любого другого системного узла в составе вычислительного поля (при необходимости с минимальными отличиями в настройках системных компонентов).

Необходимо учитывать, что при сетевой загрузке, в случае использования механизма монтирования файловой системы ВУ с системного узла по протоколу NFS, количество эффективно обслуживаемых системным узлом ВУ не превышает 100-120 узлов (особенности протокола NFS, являющегося «золотым стандартом» для сетевой загрузки). Безусловно, это количество можно увеличить, используя на вычислительном поле узлы с микроядром ОС без прямого монтирования всей структуры файловой системы ВУ, но в этом случаем многие оперативные изменения в настройках ОС ВУ потребуют перезапуска ВУ. В то же время необходимость такого оперативного внесения изменений в ОС ВУ достаточно очевидна и

обусловлена, в свою очередь, необходимостью адаптации системного окружения и конфигурации системного программного обеспечения (ПО) под задачи пользователя, либо оптимизацией системных настроек при изменении состава оборудования, масштабировании ВВС. Кроме того, каждый ВУ является сетевым устройством в общей сети ВВС и администратору ВВС важно обеспечить такой IP-адрес ВУ, который будет соответствовать некоторой логической структуре ВП (блок-узел), при этом не превышающий общую размерность IP-подсети, соответствующей вычислительному блоку, поэтому превышать вышеуказанное количество ВУ в ВБ нецелесообразно.

На каждом ВУ запускается агент системы управления и клиенты сетевых файловых систем СХ, а также с помощью специального ПО обеспечивается исполнение различных процедур библиотек передачи сообщений (как правило, MPI), используемых в параллельных прикладных программах.

2.2 Система хранения

Система хранения – подсистема ВВС, обеспечивающая хранение программ, начальных данных, промежуточных и финальных результатов расчетов задач численного моделирования. СХ состоит из множества устройств хранения, коммуникационного и серверного оборудования. Входящие в состав СХ программные компоненты обеспечивают как файловый, так и объектные сервисы хранения различного назначения.

Главным элементом СХ типовой ВВС является параллельная файловая система, обладающая высокой производительностью ввода-вывода и способная оперативно обслуживать большое количество запросов, поступающих от процессов счетных задач, программ постобработки данных и других пользовательских приложений ВВС. Ресурсы хранения параллельной файловой системы представлены группой файловых серверов, оснащенных высокопроизводительными дисковыми массивами. Используемое здесь оборудование обладает большой степенью аппаратной избыточности, что обычно определяет высокую удельную стоимость хранения данных на параллельной файловой системе и, как следствие, – ограниченный объем ее ресурсов. Клиентская часть параллельной файловой системы устанавливается на всех узлах ВП и инструментальных узлах системы доступа. Такой широкий охват и высокая зависимость вычислительного техпроцесса от наличия и работоспособности оперативных файловых ресурсов делают параллельную файловую систему одной из важнейших подсистем ВВС. Примерами параллельных файловых систем, широко используемых в настоящее время в данной предметной области, являются Lustre [12], BeeGFS [13], Spectrum Scale [14].

Ввиду ограниченного объема ресурсов, а также характерного для параллельных файловых систем приоритета производительности ввода-вывода над надежностью хранения, эти файловые системы востребованы, прежде всего, на этапе расчета задач численного моделирования, когда генерируется и модифицируется основной объем расчетных данных. Поэтому параллельные файловые системы, входящие в ВВС, часто называют «рабочими», «оперативными» или даже «черновыми» (от англ. scratch - черновик). Период нахождения данных на файловой системе обычно ограничен временем выполнения расчетов соответствующей задачи численного моделирования и может составлять от суток до нескольких недель. Несмотря на столь ограниченный срок, параллельная файловая система решает практически все поставленные перед СХ задачи по обеспечению процесса вычислений ресурсами хранения данных и, тем самым, удовлетворяет большинство потребностей многих заказчиков.

Для более длительного и более надежного хранения сгенерированных задачей расчетных данных, а также для обеспечения возможности их постобработки и визуализации различными программами в структуру СХ вводят дополнительные компоненты, обладающие, помимо указанных свойств, встроенной информационной избыточностью, масштабируемостью и

способностью предоставлять прямой доступ к данным посредством стандартных сетевых протоколов не только для структурных компонентов ВВС, но и со стороны смежных с ней информационных систем. Такие ресурсы хранения часто называют «проектными», так как административно они выделяются для отдельных научных коллективов (проектов). Отдельным их подвидом можно считать домашние директории пользователей ВВС. Время жизни данных на проектных ресурсах обычно измеряется месяцами.

С технической точки зрения проектные ресурсы хранения ВВС реализуется либо с помощью специализированных файловых серверов (так называемых «NAS-файлеров»), либо на основе распределенных масштабируемых программно-определяемых «облачных» систем хранения с файловым или объектным интерфейсом. Примерами подобных систем, кроме множества коммерческих продуктов разных производителей, можно назвать Ceph [15], Openstack Swift [16], отечественная облачная система хранения данных (ОСХД) «Стриж» [17].

В зависимости от потребностей заказчика система хранения ВВС может включать в себя дополнительную подсистему, предназначенную для долговременного хранения некоторого подмножества сгенерированных задачей расчетных данных. Ввиду длительности их хранения, требуемых на протяжении всего этого времени гарантий их неизменности, а также их постоянно возрастающего объема, эта (архивная) подсистема строится на основе технологии магнито-ленточной памяти, которая обладает минимальной по индустрии удельной стоимостью гигабайта. Наличие системы такого класса позволяет эксплуатирующей ВВС организации увеличить объем данных, находящихся в СХ ВВС, и уменьшить стоимость владения ими. К примерам таких подсистем можно отнести HPSS [18], Enstore [19], и отечественную разработку – архивную систему хранения данных (АСХД) [20].

Таким образом, полную архитектуру системы хранения любой современной ВВС можно с уверенностью назвать многоуровневой/многослойной. Выбор слоя для размещения/перемещения очередного набора данных расчетной задачи определяется предъявляемыми к СХ требованиями, которые отражают текущий этап жизненного цикла этих данных. Ввиду уникальности процессов расчета задач численного моделирования, сложившихся у разных заказчиков, а также из-за различия их организационных, технических и финансовых возможностей, система хранения любой типовой ВВС является уникальной.

2.3 Подсистема доступа

Подсистема доступа – это подсистема ВВС, предназначенная для организации доступа пользователей к ресурсам ВВС, включает в себя множество инструментальных узлов (ИУ). Каждый ИУ подключен посредством коммуникационной системы к остальным узлам ВВС, а с помощью сети доступа – к сети предприятия/организации, откуда пользователи ВВС могут к ним обращаться.

На ИУ выполняются следующие виды работ:

- организация полноценного графического или консольного сеанса пользователя;
- создание и редактирование файлов программ и данных;
- компиляция и сборка программ;
- проведение расчетов на ИУ;
- запуск задач на ВП;
- обработка результатов счета.

ИУ похож на ВУ, за исключением возможно большего числа процессоров и большего объема оперативной памяти, количество которых варьируется в зависимости от количества пользователей на ИУ, а также дополнительного сетевого интерфейса для подключения к сети доступа. Кроме того, для выполнения задач визуализации результатов расчетов, на ИУ может быть установлен специализированный графический процессор (видеокарта).

При большом количестве ИУ для удобства их администрирования в состав подсистемы доступа может входить системный узел, по функциям аналогичный системному узлу вычислительного блока.

На узлах подсистемы доступа реализуются сервисы доступа, на этих узлах доступны средства разработки прикладного программного обеспечения и клиентские компоненты системы управления задачами и ресурсами, а также другие системные сервисы.

2.4 Подсистема управления и мониторинга

Подсистема управления и мониторинга обеспечивает выполнение следующих функций:

- запуск и прохождение задач на ВП (подсистема управления задачами и ресурсами);
- ведение статистики по расчетам;
- мониторинг и диагностику оборудования и системных сервисов;
- ведение статистики по состоянию оборудования;
- конфигурирование и управление (менеджмент) КПС ВВС.

В состав ПСУМ входят несколько административных узлов, оснащенных необходимым СПО. На узлах подсистемы управления и мониторинга запускаются:

- серверные компоненты подсистемы управления задачами и ресурсами;
- ПО менеджмента сетей КПС;
- серверные и клиентские компоненты подсистем мониторинга и диагностики всех узлов и функциональных подсистем ВВС.

2.5 Сервисная подсистема

В состав сервисной подсистемы входят вспомогательные, но важные для работы ВВС сервисы:

- сервис учета и аутентификации пользователей;
- сервис доменных имен;
- сервис точного времени;
- WEB-сервис;
- сервис баз данных;
- сервис ведения проектов;
- информационные сервисы: электронная почта и сервис мгновенных сообщений;
- сервис маршрутизации, реализующий функции экспорта данных из СХ на автоматизированные рабочие места (АРМ) пользователей в сети предприятия.

На узлах сервисной подсистемы запускаются серверные компоненты перечисленных системных сервисов. Все указанные сервисы могут быть виртуализованы, либо могут использовать систему контейнеризации для обеспечения удобства миграции и надежности функционирования.

2.6 Коммуникационная подсистема

Коммуникационная подсистема ВВС (КПС) – совокупность коммуникационного оборудования, линий связи и специализированного ПО, обеспечивающая передачу данных между компонентами ВВС. КПС состоит из несколько типов функциональных сетей: сети передачи сообщений, сети данных, сети управления, сети мониторинга и т.д. Может включать различные типы сетевого оборудования для выполнения разных целей (Ethernet, InfiniBand и пр.).

Сеть передачи сообщений и сеть данных, как правило, создаются на базе высокопроизводительных решений, наподобие InfiniBand. С точки зрения обеспечения высоких показателей надежности и готовности вычислительной системы КПС может содержать дублирующие сети, например, сеть данных подсистемы доступа, построенную на оборудовании Ethernet и дублирующую сеть данных на оборудовании InfiniBand.

Сети управления и мониторинга строятся на базе оборудования Ethernet и могут быть совмещены на уровне физической сети. Данные сети подключены ко всем узлам ВВС, контроллерам управления (BMC, Baseboard Management Controller) узлов, контроллерам управления коммутаторов КПС, а также контроллерам управления устройств хранения информации СХ.

3. Обработка информации в ВВС

Поскольку ВВС является сложным по своей структуре и назначению объектом, обработка информации на ней существенно отличается от обработки информации на персональных компьютерах или традиционных электронных вычислительных машинах прошлого века. Основные причины этих отличий следующие:

- ВВС является распределенной системой, поэтому разные этапы обработки информации могут выполняться на разных функциональных подсистемах ВВС.
- Вычислительное поле ВВС состоит из множества вычислительных узлов, поэтому эффективное его использование может быть получено только с использованием параллельных программ. В этом случае пользователь вынужден распараллеливать алгоритмы своей вычислительной задачи таким образом, чтобы обеспечить её запуск на общей памяти и нескольких процессорах ВУ, или на распределенной памяти и нескольких узлах ВВС (а может быть и на разных ВВС). То есть пользователь ВВС должен предусмотреть и реализовать возможные уровни распараллеливания для своей задачи. Более того, пользователь вынужден думать о том, чем именно будет заниматься тот или иной процесс вычислительной задачи на ВУ или даже на конкретном ядре процессора, сопроцессора, ускорителя.
- Архитектура ВВС оптимизируется для получения сверхвысокой производительности, но, как это всегда бывает, в ущерб универсализму. Типичные ВВС для задач численного моделирования поддерживают, как правило, только одну многоуровневую модель параллельного программирования: на верхнем уровне процессы параллельной программы обмениваются сообщениями с использованием библиотеки MPI [21], каждый процесс, в свою очередь, распараллеливается на общей памяти средствами OpenMP [22]. Кроме того, процессы могут обращаться к различным арифметическим ускорителям.
- Так как ВВС является распределенной системой, файловые ресурсы, доступные пользователю, размещаются на специальных сетевых файловых системах СХ, при этом СХ может быть многоуровневой. Такая сложная структура СХ ВВС обязывает пользователя ВВС учитывать эту особенность в процессе своей работы, заставляет планировать размещение своих файлов на различных уровнях СХ, реализовывать схемы их миграции между уровнями СХ, а также планировать файловый вывод расчетной задачи, например, выбирая схему параллельного вывода с каждого процесса, либо сбор данных и вывод с одного процесса задачи.
- ВВС является системой коллективного доступа. Как правило, ВВС создаются для нужд целой организации, научного центра, фирмы-разработчика и т.п. Количество пользователей ВВС может исчисляться сотнями и тысячами. Поэтому при создании ВВС применяются технологии работы пользователей, учитывающие организацию системы разграничения доступа. С одной стороны, пользователи могут работать над

одними и теми же данными общего доступа, с другой стороны они должны иметь доступ только к своим, совершенно конкретным данным.

В то же время, если разделить работу пользователя на основные функциональные этапы, то работа пользователя на ВВС во многом похожа на его работу на ПЭВМ.

На рис. 2 показаны основные этапы обработки информации на ВВС:

- вход в систему;
- разработка прикладного программного обеспечения;
- отладка прикладного программного обеспечения;
- подготовка начальных данных;
- запуск и контроль за выполнением вычислительных задач.

Рассмотрим эти этапы подробнее.

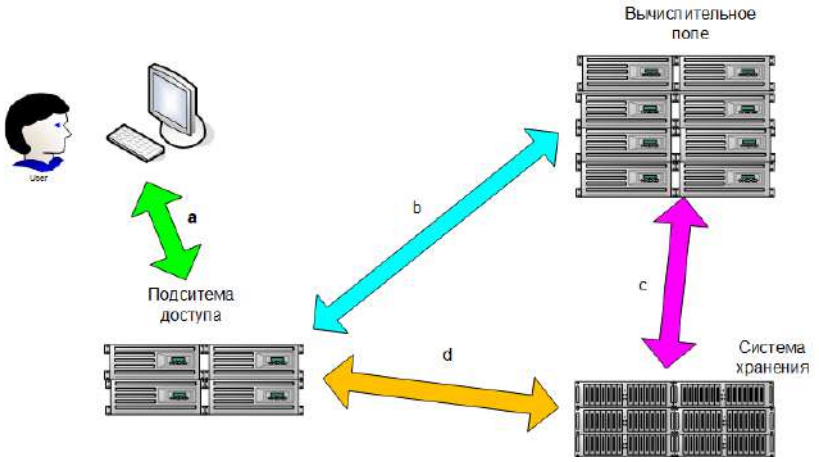


Рис. 2. Основные этапы обработки информации в ВВС.
a – вход в систему и обмен информацией между пользователем и системой;
b – запуск задач и проведение расчетов; c – обмен данными с системой хранения в процессе счета;
d – обмен данными с системой хранения в процессе разработки, отладки программ, подготовки начальных данных и обработки полученных результатов.
Fig.2. Key stages of information processing in the HPC system.
a – user login and information exchange between user and system;
b – tasks launching and performing calculations; c – data exchange with the storage system during the calculations; d – data exchange with the storage system during the development, debugging, preparation of initial data and post-processing.

3.1 Вход в систему

Пользователи получают доступ к ВВС, как правило, со своих АРМ, расположенных в сети предприятия. Существует несколько видов доступа:

- Терминальный (символьный или графический). При этом виде доступа организуется терминальный сеанс, в котором пользователь может использовать приложения из состава СПО ВВС и исполнять прикладные программы. Сеанс организуется на ИУ подсистемы доступа, как правило, с использованием протоколов SSH, XDMCP, VNC (RDP) и т.д. Перед созданием сеанса пользователь должен авторизоваться (т.е. предъявить свое учетное имя) и пройти обязательную аутентификацию (т.е. подтвердить свое имя, например, паролем или ключом доступа). За авторизацию и аутентификацию отвечает сервис учета и аутентификации пользователей (как

правило, создаваемый на основе OpenLDAP [23]). Сеанс организует один из сервисов доступа, о которых мы расскажем чуть позже в данной публикации.

- Обмен данными. Существует два вида обмена данными: передача файлов (с использованием сервисов доступа, например, на основе протоколов FTP, SFTP) или экспорт файлового пространства СХ ВВС в качестве сетевого диска на АРМ пользователя (например, на базе протоколов CIFS/SMB с использованием маршрутизаторов сервисной подсистемы).
- WEB-доступ. WEB-сервер предоставляет пользователям доступ к общедоступной информации (руководства, инструкции и другая справочная информация), а после авторизации и обязательной аутентификации – к WEB-приложениям, реализованным в ВВС (с использованием протоколов HTTP, HTTPS).
- Доступ к информационным и пользовательским сервисам. При установке соответствующих клиентских приложений на своей АРМ пользователь может подключиться к информационным и пользовательским сервисам ВВС – электронной почте, сервису мгновенных сообщений, сервису ведения проектов и т.д., для организации обмена информации (с использованием протоколов SVN, HTTPS, HTTP, SMTP, IMAP, POP3, XMPP и др.).

3.2 Разработка прикладного программного обеспечения

Разработка прикладного программного обеспечения (ППО) заключается в создании и редактировании текстов программ, компиляции и сборки программ из исходных текстов и необходимых библиотек. Разработка ППО выполняется на ИУ в терминальном режиме доступа. При разработке ППО используются текстовые редакторы и средства разработчика, компиляторы и библиотеки из состава СПО ВВС и сторонних производителей, установленные на ИУ.

Кроме ППО собственной разработки на ВВС может использоваться ППО сторонних производителей, доступное в текстах или в бинарном виде.

ППО сторонних производителей, доступное в исходных текстах, собирается аналогично собственному ПО предприятия. ППО, доступное в бинарном виде, должно быть совместимо с СПО ВВС.

3.3 Отладка прикладного программного обеспечения

Отладка прикладного программного обеспечения заключается в устранении ошибок, допущенных разработчиком ППО при создании и редактировании текстов программ, компиляции и сборки программ из исходных текстов и необходимых библиотек. Отладка ППО выполняется на ИУ в терминальном режиме доступа, либо на ВП в пакетном или интерактивном режиме запуска отлаживаемой программы через подсистему управления задачами и ресурсами. При отладке ППО используются штатные отладчики из состава СПО ВВС (например, GDB), а также коммерческое ПО сторонних производителей (например, Linaro DDT [24], Perforce Totalview [25]).

3.4 Подготовка начальных данных

Подготовка начальных данных для проведения расчетов производится на ИУ подсистемы доступа с использованием текстового редактора, либо специализированных для ППО программ (например, ПО Salome [26]). Также подготовка начальных данных может проводиться на АРМ пользователей, при условии, что формируемые данные будут располагаться на экспортируемых на АРМ пользователя ресурсах СХ ВВС.

3.5 Запуск вычислительных задач

Запуск вычислительных задач, использующих ВП для организации расчетов, производится с ИУ подсистемы доступа с использованием клиентских компонент подсистемы управления задачами и ресурсами. Подсистема управления задачами и ресурсами предоставляет пользователю информацию о состоянии запущенных и выполняемых задач, а также о доступных для его задач ресурсах ВП. Кроме того, она выполняет функции управления расчетами в части задания приоритетности той или иной задачи (группы задач) [8].

3.6 Обработка полученных результатов

Обработка полученных в ходе выполнения расчетов данных является заключительным этапом проведения расчетов. Обработка результатов расчетов может проводиться как на ИУ, так и на АРМ пользователя (при наличии возможности получения обрабатываемой информации непосредственно на АРМ). В ряде случаев, при обработке большого объема информации могут потребоваться специальные программы обработки, использующие для своей работы ресурсы вычислительного поля. В частности, при проведении графической обработки (процесса визуализации) результатов могут использоваться программы визуализации, в том числе параллельной, разработанные в российских федеральных ядерных центрах для ВВС – VIZI, ЛОГОС Scientific View, а также за рубежом – VisIt [27], ParaView [28].

3.7 Организация доступа к данным

Каждый из этапов обработки информации предполагает доступ к данным, хранимых на ресурсах ВВС. При этом, безусловно, работа на ВВС связана с некоторыми особенностями при доступе к данным, которые мы и рассмотрим далее.

Вход на ВВС предполагает создание сеанса пользователя на подсистеме доступа ВВС, при этом обрабатываются конфигурационные файлы сеанса, расположенные в домашнем каталоге пользователя на ИУ ВВС, с помощью которых формируется системное окружение пользователя на ВВС. Одной из особенностей организации доступа к данным на ВВС является то, что системное программное обеспечение ВВС конфигурируется таким образом, что данное окружение пользователя обеспечивается на всех подсистемах ВВС, задействованных в обработке его информации.

Работы по созданию прикладного ПО и подготовке начальных данных выполняются на ИУ как в домашнем каталоге пользователя, так и на файловых ресурсах оперативного уровня хранения. Учитывая очень большие объемы оперативного уровня СХ, и, как следствие, отсутствие репликации данных на этом уровне, пользователю следует хранить критичную к потере информацию либо в домашнем каталоге, либо перемещать ее на второй уровень СХ.

Данные, используемые в процессе счета задач, хранятся на оперативном уровне СХ. К ним также должно применяться следующее правило – критические важные данные должны копироваться на второй уровень СХ и резервироваться на архивном уровне СХ.

Обработка результатов счета производится на основе данных с оперативного уровня. Полученные результаты могут быть выведены на АРМ пользователя, помещены на второй уровень СХ либо на архивный уровень (для долговременного хранения).

Другой особенностью организации доступа к данным на ВВС является то, что на ВВС пользователи могут совместно работать над общими для них данными. В этом случае в рамках ВВС такие пользователи объединяются в тематические группы. На оперативном уровне хранения (а также в домашних каталогах) файловые объекты приписываются к этим тематическим группам и, соответственно, все пользователи тематической группы могут полноценно работать с таким образом организованным набором файлов. В то же время, для сторонних пользователей, не входящих в данную тематическую группу, эти данные недоступны.

4. Программное обеспечение ВВС

Системное программное обеспечение ВВС основано на дистрибутивах Linux семейства Red Hat. В основном используются дистрибутивы, доступные в исходных кодах, такие как CentOS или Rocky Linux [29,30], дополненные необходимыми компонентами из открытых источников.

Можно выделить следующие программные средства, используемые в ВВС:

- ОС узла – совокупность СПО нижнего уровня, обеспечивающая работу аппаратных составляющих узла (сервера). Для ВВС типично использование дистрибутивов на основе Linux, например, разработки РФЯЦ-ВНИИТФ – СПО Супер-ЭВМ [31], или разработки РФЯЦ-ВНИИЭФ – ЗОС Арамид [32];
- ПО, используемое для разработки прикладных программ – компиляторы с языков программирования C, C++, Fortran и др., системы разработки, прикладные, научные и графические библиотеки и пр.;
- ПО коммуникационной подсистемы – системное ПО, обеспечивающее передачу данных по каналам связи с использованием специфичных для конкретного вида сетевого оборудования протоколов (включая библиотеки, реализующие интерфейс передачи сообщений MPI);
- ПО системы хранения – системное ПО, используемое для организации доступа и хранения данных в системе хранения. Специфично для каждого уровня системы хранения;
- сервис учета и аутентификации пользователей – использует СПО, предназначенное для хранения учётных записей пользователей и реализующее механизмы, применяемые для аутентификации;
- сервис точного времени – использует СПО, обеспечивающее синхронизацию точного времени на всех подсистемах ВВС;
- сервис ведения проектов – сервис, реализующий функции структурированного хранилища программных проектов и ведения версий исходных текстов прикладного и системного программного обеспечения;
- подсистема управления задачами и ресурсами – программная подсистема, основанная на СПО, предназначенном для организации многозадачного и многопользовательского режима выполнения задач на вычислительном поле;
- средства управления и мониторинга компонент ВВС – системное ПО, используемое в процессе эксплуатации ВВС для управления узлами, диагностики работы оборудования;
- сервисы доступа к узлам ВВС – системные компоненты, установленные на узлах ВВС и обеспечивающие подключение пользователей локально или удаленно со своих АРМ, входящих в состав сопряженных автоматизированных систем, к узлам ВВС;
- WEB сервис – сервис, предоставляющий доступ к информационным ресурсам по протоколам HTTP/HTTPS, а также к компонентам подсистемы управления задачами и ресурсами;
- информационные сервисы – сервисы, предназначенные для передачи сообщений между пользователями, а также между компонентами СПО и администраторами на основе электронной почты и ПО мгновенных сообщений;
- сервис баз данных (БД) – сервис, предназначенный для хранения и обработки различной информации в базах данных;
- прикладные программы – программы, предназначенные для решения различных задач численного моделирования.

Вышеперечисленные компоненты, за исключением прикладных программ, составляют системное программное обеспечение ВВС.

Приведем краткое описание вышеперечисленных компонент СПО.

4.1 ОС узла

Операционная система узлов ВВС относится к семейству ОС Linux.

Загрузка ОС на любом узле ВВС начинается с загрузки ядра Linux. В зависимости от наличия или отсутствия загрузочного диска выделяется два типа загрузки: дисковая или бездисковая.

В случае дисковой загрузки BIOS (Basic Input Output System) узла определяет устройство, содержащее образ загрузчика, считывает данные загрузчика в память и передает ему управление. Загрузчик находит образ ядра на устройстве, загружает его в память, разархивирует и передает ему управление.

В случае сетевой загрузки первичный загрузчик находится в энергонезависимой памяти сетевого адаптера. Начальная загрузка осуществляется в соответствии со спецификацией PXE (Preboot Execution Environment). Для получения параметров загрузки (сетевой адрес сервера загрузки, имя файла вторичного загрузчика и др.) используется протокол DHCP (Dynamic Host Configuration Protocol). Для получения файлов по сети (вторичный загрузчик, ядро, начальная файловая система initramfs) используется протокол TFTP (Trivial File Transfer Protocol).

После загрузки ядро ОС производит тестирование обнаруженного оборудования, инициирует подсистемы управления памятью и процессами, виртуальную файловую подсистему, гипервизор виртуальных машин, а также запускает подсистему инициализации, которая на основании своих конфигурационных файлов запускает остальные системные процессы.

С другими программами ядро взаимодействует путем предоставления им системных вызовов, выполняющих определенные запросы от программ.

Ядро ОС на узле ВВС обеспечивает функции создания, управления и идентификации процессов, организацию виртуальной памяти, выделяемой процессам, обслуживания устройств, сетевые коммуникации, доступ к файловым объектам и прочие базовые функции, необходимые для работы вычислительной системы. Ядро представляет собой загружаемый бинарный файл, а также набор динамически подгружаемых модулей, хранящихся в сжатом виде на системных узлах ВВС и локальных дисках узлов ВВС.

4.2 ПО для разработки программ

При разработке прикладных программ на ВВС могут использоваться текстовые редакторы VI, Emacs, Gedit, mcedit из состава СПО ВВС, а также среды разработки Eclipse [33] и NetBeans [34], Qt Creator [35], компиляторы с языков высокого уровня C, C++, Fortran семейств GNU, от производителей процессоров Intel и AMD (Intel One API и AMD ROCm) [36,37], библиотеки математических функций BLAS [38], LAPACK [39], FFTW [40], и т.д., графические библиотеки GTK, Qt, OpenGL, отладчик GDB из состава СПО ВВС, и множество других открытых и проприетарных продуктов.

4.3 ПО коммуникационной подсистемы

Коммуникационное ПО включает программное обеспечение, предназначенное для реализации высокоскоростной передачи данных между абонентами в сетях InfiniBand. Данное ПО содержит средства диагностики, мониторинга, конфигурирования сетей и обеспечивает, в том числе, доступ к различным транспортным и управляющим протоколам посредством коммуникационных библиотек и модулей ядра ОС Linux, а также предоставляет возможности и инструментарий для разработки приложений с использованием коммуникационной среды InfiniBand.

Для доступа к коммуникационной подсистеме из прикладных программ используется, как правило, библиотека, реализующая стандарт передачи сообщений MPI (например, MPICH [41], MVAPICH [42], OpenMPI [43], S-MPI [44]), предоставляющая параллельным задачам возможность выполнять передачу массивов данных между процессами задачи в режиме точка-точка или в режиме выполнения коллективных операций (выполняемых одновременно на всех или группе процессов задачи).

4.4 Программное обеспечение системы хранения

Опишем программное обеспечение, которое может использоваться на разных уровнях хранения данных, более подробно.

4.4.1 NFS

NFS является сетевой файловой системой общего назначения.

NFS предоставляет клиентам прозрачный POSIX-совместимый доступ к файлам и файловой системе сервера. NFS осуществляет доступ только к тем частям файла, к которым обратился процесс, и основное достоинство NFS состоит в том, что она делает этот доступ прозрачным. Это означает, что любое приложение, которое может работать с локальным файлом, с таким же успехом может работать и с файлом NFS, без каких-либо модификаций самой программы. Однако, существующие реализации NFS имеют ограничение на количество одновременных подключений и скорость доступа со стороны параллельных задач, поскольку за обработку запросов отвечает серверная компонента на одном файловом сервере (узле хранения). Поэтому в ВВС NFS имеет ограниченное использование:

- в ВВС с числом узлов менее 100 – для домашних каталогов и на оперативном уровне;
- в ВВС с числом узлов от 100 и более – только для организации бездисковой загрузки ВУ в вычислительном блоке (для монтирования корневой файловой системы ВУ), при этом в качестве NFS-сервера выступает системный узел ВБ.

4.4.2 Файловый сервер Lustre

Lustre – это кластерная файловая система объектного типа, способная обеспечивать высокопроизводительный ввод-вывод для множества процессов как в режиме «file-per-process», так и в режиме параллельного доступа «single-shared-file».

Программное обеспечение файловой системы Lustre состоит из трех серверных компонентов, одного клиентского компонента, пользовательских и административных утилит, а также программных средств разработки. Функциональность серверных и клиентской компонентов файловой системы Lustre реализована в виде модуля ядра операционной системы Linux.

Серверные компоненты размещаются на выделенных для соответствующих их целям узлах СХ, которые оснащены локальными дисковыми ресурсами либо подключены к внешним дисковым подсистемам.

Сервер объектного хранения OSS (object storage server) – выделенный узел, выполняющий функции хранения файлового содержимого в виде уникально идентифицируемых объектов. Дисковые ресурсы, которыми оснащен каждый OSS, сгруппированы в тома файловой системы Lustre (OST – object storage targets).

Сервер хранения метаданных MDS (meta data server) – выделенный узел, выполняющий функции хранения иерархии пространства пользовательских данных ФС и метаданных. Дисковые ресурсы, которыми оснащен каждый MDS, сгруппированы в тома метаданных файловой системы Lustre (MDT – meta data targets).

Сервер хранения конфигурации ФС (management server) – выделенный или совмещенный с другой ролью узел, выполняющий функции хранения текущей конфигурации файловой системы, охватывающей все множество ее программных и аппаратных компонентов. Конфигурационные данные файловой системы Lustre размещаются на локальных для MGS блочных устройствах (MGT – management targets).

Большинство серверных узлов ВВС (вычислительные, административные, инструментальные, сервисные) являются клиентами файловой системы Lustre, которая подключается к ним путем статического монтирования с использованием высокоскоростной коммуникационной сети данных ВВС.

Размещение пользовательских файлов в файловой системе Lustre производится согласно встроенному алгоритму распределения, который «старается» равномерно заполнять тома OST. По умолчанию каждый создаваемый в файловой системе Lustre файл целиком размещается на динамически назначенном ему OST. Такой способ размещения соответствует режиму индивидуального ввода-вывода для процессов задачи («file-per-process»). Для включения режима параллельного ввода-вывода нескольких процессов задачи в один файл («single-shared-file») параметры размещения этого файла, определяемые в этом случае самим пользователем, задаются с помощью пользовательской утилиты lctl или посредством интерфейсных функций, предоставляемых программной библиотекой liblustre.so. Назначение этих параметров для отдельной директории позволяет автоматически их наследовать для всех создаваемых файловых объектов в нижележащих уровнях иерархии.

Объединение в единую систему множества томов OST позволяет не только сформировать значительный по объему ресурс хранения данных, но и суммировать пропускную способность задействованных сетевых каналов к соответствующим серверам хранения OSS. Получившаяся в результате такого объединения высокая интегральная производительность файловой системы позволяет задачам численного моделирования минимизировать длительность периодов регулярной записи своего промежуточного состояния на внешней памяти, что, в свою очередь, сокращает астрономическое время проведения расчетов и, как следствие, повышает эффективность эксплуатации ВВС.

4.4.3 ОСХД «Стриж»

«Стриж» – это объектная распределенная система хранения данных, обеспечивающая их доступность и целостность, а также способная масштабироваться как по объему, так и по пропускной способности. Основным программным способом взаимодействия с системой является объектный интерфейс доступа к неструктурированным данным и две его реализации SwiftAPI [45] и S3API [46].

ОСХД «Стриж» базируется на двух компонентах облачной платформы OpenStack [16] – сервисе аутентификации Keystone и сервисе объектного хранения Swift, а также включает в себя:

- Gate – многофункциональный пользовательский web-интерфейс управления данными, разработанный в РФЯЦ-ВНИИТФ;
- ElasticSearch [47] – документная СУБД;
- Zabbix [48] – служба сетевого мониторинга;
- HSDS [49] – сервис хранения структурированных данных с HDF5 API [50];
- и другие компоненты.

Благодаря своим характеристикам ОСХД «Стриж» способна выступать в роли площадки для организации проектных ресурсов хранения для ВВС и смежных с ней информационных систем.

4.4.4 ПО АСХД

Ключевые особенности технического решения вопроса долговременного, целостного и масштабируемого хранения «холодных» данных можно рассмотреть на примере разработанной во РФЯЦ-ВНИИТФ архивной системы хранения данных (АСХД) [20, 51].

АСХД представляет собой систему массового обслуживания, выполняющую загрузку, хранение, поиск и восстановление цифровой информации на основании пользовательских заявок. Высокая степень сохранности обрабатываемых данных обеспечивается благодаря широкому применению средств контроля целостности, встроенной репликации, использованию технологии магнитных лент, изоляции заполненных архивных носителей и самих архивных объектов. Способность хранения значительного объема информации в течение длительного времени обеспечивается как за счет применения автоматизированных ленточных библиотек различных технологических поколений, так и благодаря разработанному регламенту обмена архивными носителями между библиотеками и стеллажным хранилищем. Источниками входных информационных потоков для АСХД выступают ресурсы хранения данных с файловым и объектным интерфейсом.

В состав программного обеспечения АСХД входят: метасервер, медиасервер, транспортный агент, консольный и веб клиенты, программное обеспечение оператора хранилища архивных носителей. Технический фундамент площадки АСХД составляют группа медиасерверов и ленточных библиотек, объединенных сетью хранения SAN, а также один или несколько служебных узлов. Транспортные агенты размещаются вне контура АСХД — на серверах системы доступа и выделенных серверах, подключенных к ресурсам-источникам данных (файловые системы и ОСХД).

4.5 Сервис учета и аутентификации пользователей

Важно отметить, что в рамках ВВС организовано единое пространство пользователей и единое пространство хранения. Это означает, что на каждом узле пользователь ВВС имеет одинаковый идентификатор, обладает возможностью доступа к данным, расположенным на системе хранения ВВС.

Сервис учета и аутентификации пользователей предназначен для обеспечения идентификации и аутентификации пользователя при входе в систему, идентификации процессов пользователя, запущенных на узлах ВВС, а также для хранения информации об учетных записях пользователей ВВС.

Сервис учета и аутентификации пользователей базируется на Едином Пространстве Пользователей (ЕПП), реализованном с использованием централизованной базы данных пользователей, хранящейся в службе каталогов OpenLDAP [23] (в составе сервисной подсистемы), динамического модуля аутентификации Linux PAM (на каждом узле ВВС) и механизмов идентификации пользователей и процессов, реализованных в ядре ОС узла. Для автономных АРМ и отдельных учетных записей на узлах ВВС допускается использовать локальные учетные записи, расположенные в файле /etc/passwd.

Идентификация пользователя производится при его входе в систему (точнее, при подключении пользователя к одному из системных сервисов, размещенных в системе доступа ВВС и предоставляющих услуги по организации доступа – терминальный и графический доступ, режим передачи файлов, удаленное монтирование файловых ресурсов ВВС и пр.)

Функция идентификации и аутентификации пользователей основывается на использовании механизма PAM и наличии ЕПП на основе LDAP.

После успешной регистрации пользователя на одном из сервисов доступа, как правило, создается первичный процесс пользователя. В дальнейшем, процессы пользователя порождаются средствами ОС узла или с помощью подсистемы управления задачами и ресурсами. Основными атрибутами, идентифицирующими пользователя в процессе,

являются уникальный идентификатор пользователя (uid) и уникальный идентификатор группы (gid). Эти атрибуты используются в подсистеме разграничения доступа при организации дискреционной модели доступа.

4.6 Сервис точного времени

Сервис точного времени на основе NTP (Network Time Protocol) входит в сервисную подсистему ВВС и обеспечивает синхронизацию системного времени на всех узлах ВВС по отношению к эталонному узлу. Как правило, эталонным узлом является один из административных узлов системы управления и мониторинга, либо специально выделяемый сервер точного времени, использующий оборудование GPS или GLONASS [52,53], либо высокоточные часы с цифровым радиоприемником для получения сигнала точного времени.

4.7 Сервис ведения проектов

Сервис ведения проектов входит в сервисную подсистему и предназначен для ведения версий исходных текстов прикладного и системного программного обеспечения. Сервис ведения проектов может использовать ПО Subversion (SVN) [54] или GIT [55], доступное в исходных кодах и обладающее широким функционалом.

SVN и Git предназначены для помощи в разработке программных проектов: позволяют управлять файлами и каталогами, а также сделанными в них изменениями. Это дает возможность отслеживать изменения в исходном коде программ при разработке программных проектов, восстановить более ранние версии исходных текстов, изучить историю всех изменений.

Указанное ПО представляет собой ПО общего назначения, которое можно использовать для управления любыми типами файлов, в частности для ведения версий разрабатываемого на ВВС прикладного и системного программного обеспечения.

Основным объектом является проект – совокупность данных, описывающих структуру хранимой в проекте информации и ее версии. К каждому проекту назначается список доступных к нему пользователей.

Пользователь может экспортировать свои локальные данные в проект, импортировать данные из проекта (любой версии) в свое локальное пространство, внести изменение в проект под новой версией.

4.8 Подсистема управления задачами и ресурсами

Подсистема управления задачами и ресурсами предназначена для запуска и выполнения задач пользователей (прикладных программ) в пакетном и интерактивном режиме с использованием ресурсов вычислительного поля ВВС.

На ВВС для организации данной подсистемы в основном используется ПО Slurm (Simple Linux Utility for Resource Management) [56], являющееся высокомасштабируемой программной системой с открытым исходным кодом, предназначенной для управления ресурсами ВВС, планирования и запуска вычислительных задач. Slurm выполняет три ключевые функции:

- захватывает ресурсы (вычислительные узлы) для пользователей в необходимом количестве на определённое время;
- предоставляет средства для запуска и мониторинга задач на выделенных узлах;
- организует очередь задач, выполняет планирование запуска задач согласно настроенным правилам, предотвращает конфликты при захвате ресурсов.

Slurm состоит из следующих основных компонент:

- Центральный демон slurmctld (контроллер) – управляет работой всей подсистемы и

обрабатывает запросы пользовательских команд. Дополнительно может функционировать резервный контроллер, запущенный на другом сервере. Резервный контроллер получает управление в случае отказа основного сервера.

- Узловые демоны slurmd, работающие на каждом вычислительном узле. Все узловые демоны взаимодействуют друг с другом и другими компонентами через сеть, образуя отказоустойчивую иерархическую структуру.
- Утилиты (команды), предназначенные для взаимодействия пользователя (или администратора) с Slurm;
- Опциональный демон взаимодействия с базой данных slurmdbd, который может использоваться для сохранения учётной информации по задачам и ресурсам в базу данных. Один демон slurmdbd может обслуживать несколько ВВС.

Логические объекты, управляемые Slurm, включают следующие сущности:

- узлы (nodes) – вычислительные ресурсы;
- разделы (partitions) – узлы, логически объединённые в одно множество;
- задачи (jobs) – выделенные ресурсы для конкретного пользователя на определённое время;
- шаги задачи (job steps) – множества процессов (экземпляров программы) внутри задачи.

Разделы можно рассматривать как очереди задач, каждая из которых имеет набор ограничений, таких как: размер задачи, пользователи, для которых разрешен на данный момент запуск задач и т.д.

Узлы внутри раздела выделяются для задач до тех пор, пока достаточно ресурсов (узлов, процессоров, памяти и т.д.). После того, как под задачу выделен набор узлов, пользователь может инициировать шаги задачи (запускать параллельные программы) в любой конфигурации на выделенных ресурсах. Например, один шаг может использовать сразу все узлы, выделенные для задачи, или несколько шагов могут независимо использовать только часть узлов.

При постановке задачи пользователя в очередь на исполнение Slurm запоминает идентификатор пользователя, поставившего задачу и связанный с ним идентификатор группы. В дальнейшем, после выделения вычислительных ресурсов, Slurm выполняет запуск процессов задачи и назначает им ранее сохраненные идентификаторы пользователя и группы. Таким образом, процессы на выделенных задаче узлах имеют те же права доступа к данным, что и пользователь, запустивший задачу.

4.9 Средства управления и мониторинга компонент ВВС

Средства управления и мониторинга компонент ВВС представляют собой ПО, являющееся частью подсистемы управления задачами и ресурсами ВВС и предназначенное для выполнения следующих функций:

- анализа состояния узлов и другого оборудования ВВС;
- управления (включения, выключения, конфигурации) оборудованием ВВС;
- диагностики состояния оборудования ВВС;
- диагностики и управления системных сервисов ВВС.

Средства управления и мониторинга компонент ВВС состоят из системных компонент, взаимодействующих с контроллерами оборудования и системными журналами, и получающих от них данные диагностики. Кроме того, в состав этих средств входят

консольные утилиты, взаимодействующие с контроллерами оборудования с целью управления и конфигурации.

Для этих целей используется обычно ПО Ganglia [57], Zabbix [48], Nagios [58], а также ПО нижнего уровня для работы с интерфейсом IPMI (Intelligent Platform Management Interface), например, ipmitool [59].

4.10 Сервисы доступа

Сервисы доступа обеспечивают подключение пользователей к ВВС. Функционально данные сервисы реализуются на наборе следующих инструментов и компонентов ОС:

- login – вход на узел (АРМ) с непосредственно подключенного к узлу (АРМ) терминала. После авторизации и аутентификации на узле образуется сессия (shell), позволяющая выполнять на узле различные команды от имени пользователя.
- ssh – удаленный вход на узел с АРМ пользователя или другого узла. После авторизации и аутентификации на узле образуется сессия (shell), позволяющая выполнять на узле различные команды от имени пользователя. Для входа могут быть использованы либо встроенные средства ОС Linux, либо различные ssh-клиенты, например, PuTTY [60].
- xdm – удаленный вход на ИУ подсистемы доступа в графическом режиме с АРМ пользователя. После авторизации и аутентификации на ИУ создается графическая сессия, позволяющая пользователю запускать различные графические приложения. Для входа могут быть использованы либо встроенные средства ОС Linux, либо различные пакеты, эмулирующие X-сервер на АРМ под управлением ОС Windows – Xming [61], OpenText Exceed [62] и т.д.
- lightdm – вход на АРМ в графическом режиме. После авторизации и аутентификации на АРМ создается графическая сессия, позволяющая пользователю запускать различные графические приложения.
- vnc – удаленный вход на ИУ подсистемы доступа в графическом режиме с АРМ пользователя. Предполагает запуск на ИУ из ранее созданной терминальной сессии программы vncserver, после этого на АРМ пользователя запускается утилита vncviewer. После авторизации и аутентификации на ИУ создается графическая сессия, позволяющая пользователю запускать различные графические приложения, в том числе, ориентированные на обработку 3D-графики. Для входа могут быть использовано различное специализированное ПО, например, TurboVNC [63].
- ftp – удаленный вход с АРМ пользователя или других узлов ВВС на узлы с установленным ftp-сервером для передачи файлов. Для входа могут быть использован любой ftp-клиент.
- samba – подключение к узлам маршрутизации сервисной подсистемы для подключения ресурсов СХ в качестве сетевых дисков к АРМ пользователя. Требует авторизации и аутентификации. Для входа могут быть использованы встроенные средства ОС Linux и ОС Windows.

4.11 WEB-сервис

WEB-сервис может быть реализован на основе ПО Apache [64], входящего в состав большинства дистрибутивов ОС Linux, в том числе дистрибутива СПО Супер-ЭВМ. WEB-сервер Apache в составе сервисной подсистемы обеспечивает как авторизованный, так и неавторизованный (анонимный) доступ пользователей, администраторов и ремонтно-обслуживающего персонала к информации и системным сервисам по протоколам HTTP/HTTPS.

В роли клиента может выступать любой WEB-браузер (например, Яндекс-браузер [65]), а также любая утилита, передающая серверу запросы по протоколам HTTP/HTTPS.

При анонимном доступе пользователю предоставляется только информация открытого доступа (информация справочного характера).

При авторизованном доступе Apache запрашивает имя и пароль пользователя, затем выполняет аутентификацию с использованием PAM. Далее пользователю может быть предоставлена информация в соответствии с правилами разграничения доступа, реализованными в Apache.

На ВВС в режиме авторизованного доступа могут быть реализованы и доступны пользователям многие информационные и функциональные сервисы, например:

- WEB-интерфейс подсистемы управления задачами и ресурсами;
- WEB-интерфейс системы архивного хранения;
- WEB-интерфейс контроля ресурсов на СХ;
- портал поддержки, содержащий список вопросов и ответов по интересующим пользователей вопросам и т.д.

4.12 Информационные сервисы

Данные сервисы используются для передачи диагностических сообщений от системных сервисов администраторам и обслуживающему персоналу и реализуют функции электронной почты и сервиса мгновенных сообщений.

4.12.1 Электронная почта

Электронная почта может использовать следующие компоненты: агент передачи сообщений Postfix [66] и агент доставки сообщений Dovecot [67]. Почтовый сервер Postfix отсылает и принимает почту, а также осуществляет локальный доступ к почтовым ящикам пользователей узла. Dovecot осуществляет удалённый доступ пользователей к их почтовым ящикам по протоколам POP3 и IMAP.

Postfix – это агент передачи сообщений (MTA, Message Transport Agent), который занимается пересылкой по протоколу SMTP сообщений от пользовательского почтового агента (MUA, Mail User Agent), называемого также почтовым клиентом, к удаленному почтовому серверу. MTA также принимает сообщения от удаленных почтовых серверов и пересылает их другим MTA или доставляет в локальные почтовые ящики. Переслав или доставив сообщение, Postfix заканчивает свою работу. За доставку сообщения конечному пользователю отвечают другие серверы, например, сервер Dovecot, который передает по протоколам POP3 и IMAP сообщения различным почтовым клиентам, с помощью которых пользователь ВВС может прочитать их.

4.12.2 Сервис мгновенных сообщений

Информационный сервис Jabber обеспечивает передачу информации по протоколу XMPP. Для ВВС может быть использовано ПО Jabberd2 [68], реализующее модульный сервер XMPP, написанный на языке C.

Поддержка многопользовательских конференций в Jabberd2 реализована в виде отдельного проекта MU-Conference. Хотя XMPP не привязан к какой-то определенной сетевой архитектуре, реализация сессии осуществляется по схеме клиент-сервер, где клиент реализует подключение к серверу с помощью TCP-транспорта и сами серверы взаимодействуют друг с другом, используя протокол TCP.

Сервер ответственен за:

- установление и поддержание соединения или сессии с другими объектами, в виде

XML-потоков к или от авторизованных клиентов, серверов и прочих объектов;

- маршрутизацию корректно адресованных атомарных запросов.

Большинство клиентов подключается непосредственно к серверу посредством TCP и используют XMPP для получения полной функциональности, доступной на сервере.

4.13 Сервис баз данных

Сервис баз данных (системы управления базами данных – СУБД) является неотъемлемой частью ВВС. В ВВС данный сервис используется для хранения и обработки различной информации в базах данных. Это и статистические данные по расчетам на ВВС, и сами расчетные данные, накапливаемые при решении прикладных задач численного моделирования.

На ВВС могут применяться СУБД MariaDB [69] и PostgreSQL [70], которые являются реляционными СУБД с открытым исходным кодом.

4.14 Прикладные программы

Прикладное ПО является неотъемлемой, и даже подчас определяющей, частью ВВС. Это объясняется тем, что зачастую прикладное ПО разрабатывается отдельными специализированными научными организациями, фирмами, на протяжении многих десятков лет и зачастую никоим образом не зависит от процесса создания ВВС для нужд той или иной организации. Созданное таким образом прикладное ПО является уникальной разработкой и стоимость его иногда сравнима со стоимостью небольшой вычислительной системы. Поэтому часто потребность использования того или иного прикладного ПО определяет структуру и состав как функциональных подсистем ВВС, так и ВВС в целом. В то же время в некоторых крупных научно-исследовательских организациях, принят несколько иной подход к созданию ВВС и разработке прикладного ПО. Прикладное ПО в таких организациях разрабатывается для решения конкретных, исторически определенных узких задач самостоятельно и безусловно учитывает архитектуру и особенности ВВС, специально создаваемых для этих организаций.

Как правило, на ВВС используется ПО ведущих производителей инженерного и научного ПО: Ansys [71], Siemens [72], Dassault Systemes [73], MathWorks [74], ЛОГОС [75], OpenFOAM [76], LAMMPS [77] и многое другое. Данное ПО включает в себя множество модулей для решения задач расчета прочности конструкций, аэродинамики, газодинамики, гидродинамики, теплопроводности, молекулярной динамики, различных физико-химических процессов и т.д. Это ПО позволяет проводить массово-параллельные расчеты фактически по любой научной проблематике.

Все прикладное ПО тестируется на совместимость с системным ПО ВВС, изучаются его возможности по использованию ускорителей вычислений, различных оптимизирующих библиотек и, в соответствии с результатами, может меняться либо состав оборудования, системного программного обеспечения ВВС, либо, в свою очередь, это прикладное ПО со временем дорабатывается и оптимизируется разработчиками для эффективного использования на ВВС.

5. Заключение

Разработка архитектуры ВВС является сложной многоэтапной задачей, заключающейся в многофакторном анализе множества исходных данных: комплекса решаемых ВВС задач, наличия или отсутствия прикладного ПО для их решения, финансовых, инженерных возможностей предприятия/организации, наличия или отсутствия квалифицированного персонала по разработке ПО и эксплуатации ВВС. В настоящей работе предложено описание общей архитектуры ВВС: функциональный состав ВВС, используемое системное

программное обеспечение и особенности обработки информации в ВВС, на основе анализа информации о вычислительных системах, создаваемых и эксплуатируемых как за рубежом, так и на предприятиях России. Результаты данной работы носят, безусловно, достаточно общий и субъективный характер и отражают один из возможных подходов к построению ВВС. Тем не менее, авторы надеются, что опубликованная ими работа будет полезна специалистам, занимающимся разработкой и эксплуатацией современных вычислительных систем высокой производительности, предназначенных для проведения научных исследований. Авторы намерены продолжить публикации по данной тематике, с целью представления более развернутой информации по каждой из функциональных подсистем ВВС, основным аспектам и особенностям администрирования ВВС, особенностям разработки прикладного и системного ПО для ВВС, предназначенных для решения задач численного моделирования.

Список литературы/References

[1]. J.E. Thornton, Design of a Computer – The Control Data 6600. In the editorials series of Malcolm C Harrison, Courant Institute of Mathematical Sciences, New York University, 1970

[2]. А. О. Лацис. Как построить и использовать суперкомпьютер. - М.: Бестселлер, 2003. -240 с. 3000 экз. ISBN 5-98158-003-8.

[3]. ГОСТ Р 57700.27-2020. ВВС. Термины и определения. 2020.

[4]. ГОСТ Р 57700.26-2020 Высокопроизводительные вычислительные системы. Требования приемочных испытаний.

[5]. Глазырин А.И., Мокшин С.Ю., Доклад «Суперкомпьютер «Зубр» средней производительности» // Всероссийская Конференция «Информационные технологии в оборонно-промышленном комплексе» (17-20 мая 2016 г), г. Челябинск.

[6]. Цифровые продукты РФЯЦ-ВНИИТФ. Available at: https://vniitf.ru/rubric/tsod-i-svem, accessed 10.05.2025.

[7]. Мокшин С.Ю. Общие подходы к проектированию подсистемы доступа высокопроизводительных вычислительных систем. Труды ИСП РАН, том 32 вып. 4, 2020, стр. 41-52, DOI: 10.15514/ISPRAS-2020-32(4)-3 / Mokshin S.Yu. General ways to design the access subsystem of high performance supercomputing systems. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 4, 2020, pp. 41-52 (in Russia), DOI: 10.15514/ISPRAS-2020-32(4)-3.

[8]. Игнатьев А.О., Калинин А.А, Мокшин С.Ю. Реализация функций управления задачами и ресурсами высокопроизводительной вычислительной системы в «СПО Супер-ЭВМ». Труды ИСП РАН, том 34 вып. 2, 2022, стр. 159-178, DOI: 10.15514/ISPRAS-2022-34(2)-13 / Ignatyev A.O., Kalinin A.A., Mokshin S.Yu. Task and resources management function in HPC operation system «SPO Super-EVM». Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 2, 2022, pp. 159-178 (in Russia), DOI: 10.15514/ISPRAS-2022-34(2)-13.

[9]. Игнатьев А.О., Мокшин С.Ю., Иванков Д.В., Бекетов Е.А. Пути организации параллельного доступа к структурированным данным. Труды ИСП РАН, том 35 вып. 2, 2023, стр. 111-126, DOI: 10.15514/ISPRAS-2023-35(2)-8 / Ignatyev A.O., Mokshin S.Yu., Ivankov D.V., Beketov E.A. The parallel access to structured data organization. Trudy ISP RAN/Proc. ISP RAS, vol. 35, issue 2, 2023, pp. 111-126 (in Russia), DOI: 10.15514/ISPRAS-2023-35(2)-8.

[10]. Игнатьев А.О., Мокшин С.Ю. Типовая архитектура высокопроизводительной вычислительной системы для решения задач численного моделирования, Препринт РФЯЦ-ВНИИТФ № 265, Снежинск, 2020, 21 с. / Ignatyev A.O., Mokshin S.Yu. Base architecture of the mathematical modelling HPC system, Preprint FSUE «RFNC-VNIITF named after Academ. E.I. Zababakhin» № 265, Snezhinsk, 2020, 21 p. (in Russian).

[11]. TOP500. Available at: https://www.top500.org/, accessed 01.05.2025.

[12]. Understanding Lustre Filesystem Internals, Tech Report: ORNL/TM-2009/117, Available at: http://wiki.lustre.org/lid/index.html, accessed 10.05.2025.

[13]. BeeGFS, Available at: https://www.beegfs.io/, accessed 10.05.2025.

[14]. IBM Spectrum Scale. Available at: https://www.ibm.com/support/pages/ibm-spectrum-scale/, accessed 10.05.2025.

[15]. Ceph, Available at: https://ceph.io/, accessed 10.05.2025.

[16]. Open standard cloud computing platform. Available at: https://www.openstack.org/, accessed 10.04.2025.
[17]. ОСХД Стриж. Available at: https://vniitf.ru/data/marketing/ZOS/33.01-rukovodstvo%20programmista.pdf, accessed 10.04.2025.
[18]. High Performance Storage System. Available at: https://computing.llnl.gov/projects/hpss, accessed 10.04.2025
[19]. Enstore, Available at: https://github.com/Enstore-org/enstore/, accessed 10.05.2025.
[20]. АСХД. Программа для ЭВМ № 2018610434. Available at: https://onlinepatent.ru/software/2018610434/, accessed 10.04.2025.
[21]. MPI: The Message Passing Interface. Available at: http://parallel.ru/tech/tech_dev/mpi.html, accessed 01.05.2025.
[22]. The OpenMP API specification for parallel programming. Available at: https://www.openmp.org/, accessed 01.05.2025.
[23]. OpenLDAP. Available at: https://www.openldap.org/, accessed 01.05.2025.
[24]. Linaro DDT Debugger. Available at: https://www.linaroforge.com/linaro-ddt/, accessed 01.05.2025.
[25]. Perforce Totalview Debugger. Available at: https://totalview.io/, accessed 01.05.2025.
[26]. Salome. Available at: https://www.salome-platform.org/, accessed 01.05.2025.
[27]. VizIt. Available at: https://sd.llnl.gov/simulation/computer-codes/visit/, accessed 01.05.2025.
[28]. Paraview. Available at: https://www.paraview.org/, accessed 01.05.2025.
[29]. CentOS. Available at: https://www.centos.org/download/, accessed 01.05.2025.
[30]. Rocky Linux. Available at: https://rockylinux.org/, accessed 01.05.2025.
[31]. СПО Супер-ЭВМ. Available at: https://vniitf.ru/article/spo-super-evm, accessed 01.05.2025.
[32]. ЗОС Арамид. Available at: https://vniief.ru/researchdirections/civildevelopment/Aramid/, accessed 01.05.2025.
[33]. Eclipse. Available at: https://www.eclipse.org/, accessed 01.05.2025.
[34]. NetBeans. Available at: https://netbeans.apache.org/front/main/, accessed 01.05.2025.
[35]. Qt Creator. Available at: https://www.qt.io/product/development-tools, accessed 01.05.2025.
[36]. The Intel® oneAPI HPC Toolkit. Available at: https://www.intel.com/content/www/us/en/developer/tools/oneapi/hpc-toolkit.html, accessed 10.02.2022.
[37]. AMD ROCm. Available at: https://www.amd.com/en/products/software/rocm.html, accessed 10.04.2025.
[38]. Basic Linear Algebra Subprograms. Available at: https://netlib.sandia.gov/blas/, accessed 10.04.2025.
[39]. Linear Algebra PACKage. Available at: https://netlib.sandia.gov/lapack/, accessed 10.04.2025.
[40]. The Fastest Fourier Transform in the West (FFTW). Available at: https://fftw.org/, accessed 10.04.2025.
[41]. MPICH. Available at: https://www.mpich.org, accessed 30.05.2025.
[42]. MAVAPICH. Available at: https://mvapich.cse.ohio-state.edu/, accessed 10.04.2025.
[43]. OpenMPI. Available at: https://www.open-mpi.org/, accessed 10.04.2025.
[44]. Библиотека S-MPI. Available at: https://agora.guru.ru/abrau2013/pdf/338.pdf, accessed 10.04.2025.
[45]. Object Storage API. Swift documentation. Available at: https://docs.openstack.org/api-ref/ object-store/, accessed 10.05.2025.
[46]. Amazon S3 REST API Introduction – Amazon Simple Storage Service. Available at: https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html, accessed 10.05.2025.
[47]. ElasticSearch: The Official Distributed Search & Analytics Engine. Available at: https://elastic.co/elasticsearch/, accessed 10.05.2025.
[48]. Zabbix – open source distributed monitoring solution. Available at: https://zabbix.com/ru/, accessed 10.05.2025.
[49]. Highly Scalable Data Service (HSDS) – The HDF Group. Available at: https://hdfgroup.org/ solutions/highly-scalabale-data-service-hsds/, accessed 10.05.2025.
[50]. The HDF5 Library & File Format – The HDF Group. Available at: https://hdfgroup.org/ solutions/hdf5/, accessed 10.05.2025.
[51]. Иванков Д.В. Реализация функции долговременного хранения научных данных большого объема в вычислительном центре. Труды ИСП РАН, том 34 вып. 4, 2022, стр. 117-134. DOI: 10.15514/ISPRAS-2022-34(4)-9 / Ivankov D.V. Large-scale scientific data and long-term data storage function in a computing datacenter. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 4, 2022. pp. 117-134 (in Russia), DOI: 10.15514/ISPRAS-2022-34(4)-9.
[52]. GPS. Available at: https://www.gps.gov/, accessed 10.05.2025.

[53]. ГЛОНАСС. Available at: https://glonass-iac.ru/en/about_glonass/, accessed 10.05.2025.
[54]. SVN. Available at: https://subversion.apache.org/, accessed 10.05.2025.
[55]. Git. Available at: https://git.kernel.org/pub/scm/git/git.git, accessed 10.05.2025.
[56]. Slurm Workload Manager. Available at: https://slurm.schedmd.com/, accessed 10.05.2025.
[57]. Ganglia. Available at: https://github.com/ganglia/, accessed 10.05.2025.
[58]. Nagios. Available at: https://www.nagios.org/, accessed 10.05.2025.
[59]. IPMI tool. Available at: https://github.com/ipmitool/ipmitool, accessed 10.05.2025.
[60]. PuTTY. Available at: https://www.putty.org/, accessed 10.05.2025.
[61]. Xming X Server. Available at: https://www.straightrunning.com/XmingNotes/, accessed 10.05.2025.
[62]. OpenText Exceed. Available at: www.opentext.com/products-and-solutions/products/specialty-technologies/connectivity/exceed/, accessed 11.12.2024.
[63]. TurboVNC. Available at: https://www.turbovnc.org/, accessed 10.05.2025.
[64]. Apache Software Foundation. Available at: https://www.apache.org/, accessed 10.05.2025.
[65]. Яндекс Браузер. Available at: https://browser.yandex.ru/, accessed 10.05.2025.
[66]. Postfix. Available at: https://www.postfix.org/, accessed 11.07.2023.
[67]. Dovecot. Available at: https://www.dovecot.org/, accessed 10.05.2025.
[68]. Jabberd2. Available at: https://jabberd2.org/, accessed 10.05.2025.
[69]. MariaDB Foundation. Available at: https://mariadb.org/, accessed 10.05.2025.
[70]. PostgreSQL. Available at: https://www.postgresql.org/, accessed 10.05.2025.
[71]. ANSYS. Available at: https://www.ansys.com/, accessed 19.02.2022.
[72]. SIEMENS. Available at: https://www.siemens.com/global/en/products/software.html, accessed 10.05.2025.
[73]. Dassault Systemes. Available at: https://dassault.fr/en/subsidiary/dassault-systemes, accessed 10.05.2025.
[74]. MathWorks. Available at: https://www.mathworks.com/, accessed 10.05.2025.
[75]. ЛОГОС. Available at: https://www.logos.vniief.ru/, accessed 10.05.2025.
[76]. OpenFOAM. Available at: https://www.openfoam.com/, accessed 10.05.2025.
[77]. LAMMPS. Available at: https://www.lammps.org/, accessed 18.02.2022.

Информация об авторах / Information about authors

Сергей Юрьевич МОКШИН – начальник отдела ВНИИ технической физики имени академика Е.И. Забабахина с 2016 года. Сфера научных интересов: проектирование вычислительных систем, разработка функциональных подсистем для высокопроизводительных вычислительных систем, разработка операционных систем, методы и средства защиты информации.

Sergey Yurievich MOKSHIN – Head of the Department of Zababakhin All-Russian Scientific Research Institute of Technical Physics since 2016. Research interests: design of supercomputer systems, development of functional subsystems for high performance supercomputing systems, operating systems development, methods and means for protecting information.

Алексей Олегович ИГНАТЬЕВ – начальник лаборатории ВНИИ технической физики имени академика Е.И. Забабахина с 1998 года. Сфера научных интересов: проектирование вычислительных систем, разработка параллельных программ численного моделирования, разработка операционных систем, методы и средства защиты информации.

Alexey Olegovich IGNATYEV – Head of the Laboratory of Zababakhin All-Russian Scientific Research Institute of Technical Physics since 1998. Research interests: design of supercomputer systems, parallel numerical simulation programs development, operating systems development, methods and means of information security.

Александр Иосифович МЕЛЬНИКОВ – ведущий научный сотрудник ВНИИ технической физики имени академика Е.И. Забабахина с 2015 года. Сфера научных интересов: высокопроизводительные вычислительные системы, разработка ПО для Linux.

Alexander Iosifovich MELNIKOV – Leader scientist of Zababakhin All-Russian Scientific Research Institute of Technical Physics since 2015. Research interests: height performance computing, Linux development.

Дмитрий Владимирович ИВАНКОВ – начальник лаборатории ВНИИ технической физики имени академика Е.И. Забабахина с 2016 года. Сфера научных интересов: проектирование многоуровневых систем хранения данных, разработка высокопроизводительных систем хранения данных, исследования методов управления данными.

Dmitry Vladimirovich IVANKOV – Head of the Laboratory of Zababakhin All-Russian Scientific Research Institute of Technical Physics since 2016. Research interests: design of tiered data storage systems, development of high performance storage systems, research in data management methods.

DOI: 10.15514/ISPRAS-2025-37(6)-35

Аппаратное ускорение модуля MMU при полносистемной эмуляции aarch64 на x86-64 в эмуляторе Qemu

Д.Н. Полетаев, ORCID: 0009-0005-8872-2802 <dmitry.poletaev@ispras.ru>
П.М. Довгалюк, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>
Г.Н. Тейс, ORCID: 0009-0008-6059-2315 <george.teys@ispras.ru>
М.А. Костин, ORCID: 0009-0002-1464-8302 <maksim.kostin@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.
Новгородский государственный университет им. Ярослава Мудрого,
Россия, 173003, Великий Новгород, ул. Большая Санкт-Петербургская, д. 41.

Аннотация. В работе рассматривается оптимизация, позволяющая переложить часть вычислений, связанных с трансляцией гостевых виртуальных адресов при полносистемной эмуляции с программного MMU на MMU хостовой системы. Описан вариант применения оптимизации к эмулятору Qemu. Выполнена оценка прироста производительности от использования оптимизации, предложен вариант дальнейшего развития подхода.

Ключевые слова: виртуальные машины; полносистемная эмуляция; аппаратный MMU; ускорение эмуляции.

Для цитирования: Полетаев Д.Н., Довгалюк П.М., Тейс Г.Н., Костин М.А. Аппаратное ускорение модуля MMU при полносистемной эмуляции aarch64 на x86-64 в эмуляторе Qemu. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 45–58. DOI: 10.15514/ISPRAS–2025–37(6)–35.

Благодарности: Исследование выполнено за счет гранта Российского научного фонда № 24-11-20022, https://rscf.ru/project/24-11-20022/.

Hardware Acceleration of Qemu MMU for aarch64 on x86-64 Full System Emulation

D.N. Poletaev, ORCID: 0009-0005-8872-2802 <dmitry.poletaev@ispras.ru>
P.M. Dovgalyuk, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>
G.N. Teys, ORCID: 0009-0008-6059-2315 <george.teys@ispras.ru>
M.A. Kostin, ORCID: 0009-0002-1464-8302 <maksim.kostin@ispras.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Yaroslav-the-Wise Novgorod State University,
41, B. Sankt- Peterburgskaya st., Veliky Novgorod, 173003, Russia.

Abstract. Full system cross-ISA emulation is widely used nowdays, but is known for being slow. Major contribution to the slowdown is made by software MMU doing guest virtual addresses translation. In article we look at optimization which allows to move part of such address translation work to the hardware MMU of the host system. For this goal, extra view to the whole guest virtual address space is added to the address space of the emulator process, using mmap system call. After mapping is done there is opportunity to use fixed offset correction to guest virtual address in the translated binary code in place of dynamic search of needed offset in software TLB. Additional view of guest virtual address space maintained coherent with guest page tables. Such approach allows to use less host instructions per each guest memory instruction, which lead to notable emulation acceleration, considering the large quantity of memory instructions in the guest execution flow. Measurments show speed up as large as 271% for benchmark tests and up to 217% for the real-world program. Ideas are proposed for overcoming some limitations of described approach.

Keywords: virtual machines; full system emulation; hardware MMU; emulation speed up.

For citation: Poletaev D.N., Dovgaluk P.M, Teys G.N., Kostin M.A. Hardware acceleration of Qemu MMU for aarch64 on x86-64 full system emulation. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 3, 2025, pp. 45-58 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-35.

Acknowledgements. This work is supported by Russian Science Foundation under grant number 24-11-20022, https://rscf.ru/project/24-11-20022/.

1. Введение

Кросс-архитектурная полносистемная эмуляция активно используется в сферах разработки компьютерной аппаратуры и программного обеспечения, цифровой безопасности, моделировании систем и пр. Одним из основных неудобств, а иногда и ограничений, связанных с применением полносистемной эмуляции для этих целей, считается низкая производительность программных эмуляторов (замедление работы эмулируемого ПО в десятки раз – не редкость). Большой вклад в замедление вносит эмуляция работы виртуальной памяти, которая является неотъемлемой частью многих компьютерных архитектур, включая aarch64. По оценке из работы [1], накладные расходы на эмуляцию виртуальной памяти в полносистемных эмуляторах на основе двоичной трансляции могут достигать ~40% от всего замедления, вносимого эмуляцией. В работе рассматривается оптимизация, позволяющая использовать аппаратуру MMU хостовой (на которой работает эмулятор) системы для эмуляции работы виртуальной памяти гостевой (которая запущена в эмуляторе) системы. Для практической апробации подхода выбран эмулятор Qemu. Результаты выполненных замеров работы с применением оптимизации показывают значительное увеличение производительности эмулятора в определенных режимах работы гостевой системы.

2. Обзор предлагаемых решений

2.1 Виртуальная память

Концепция виртуальных адресов позволяет изолировать различные процессы в системе друг от друга, а также эффективнее использовать доступную физическую память. В современных ОС (и платформах, на которых они работают) виртуальная память является страничной - все адресное пространство разбивается на равные непрерывные блоки памяти, страницы. Виртуальные страницы отображаются на страницы физической памяти, причем несколько виртуальных страниц могут быть отображены на одну физическую страницу, но не наоборот. Для описания того, каким физическим адресам соответствуют виртуальные адреса, существуют системные структуры памяти, хранящиеся в памяти, называемые страничными таблицами. Таблицы настраиваются системными программами (операционной системой), согласно его стратегии работы с памятью.

При каждом обращении процессора к памяти (в том числе для выбора инструкции), виртуальный адрес преобразуется специальным аппаратным блоком – устройством управления памятью (Memory Management Unit, MMU), который просматривает таблицы трансляций, в физический, который уже и используется, непосредственно, для доступа. Для кеширования результатов трансляций в MMU присутствует кеш трансляций – буфер ассоциативной трансляции (Translation Lookaside Buffer, TLB). Буфер TLB позволяет значительно снизить нагрузку на ОЗУ от работы механизма виртуальной памяти: если трансляция уже есть в TLB, то используется она, а полный поиск в таблицах не выполняется.

Таблицы трансляций динамично меняются по мере работы программы (память выделяется, процессы переключаются), и для обеспечения синхронного состояния таблиц и записей в TLB системное программное обеспечение (ПО) удаляет неактуальные записи с помощью специальных процессорных команд. Тогда при новом обращении по измененному адресу запись в TLB не найдется, и будет выполнен полный обход таблиц, а запись с актуальной трансляцией помещена в TLB.

В ситуациях, когда в таблицах отсутствует информация, как нужно транслировать определенный виртуальный адрес в физический адрес, или параметры доступа к странице памяти нарушаются (например, запись в страницу только для чтения или доступ из пользовательского процесса к памяти привилегированного процесса), в процессоре происходит исключение, и управление передается программе-обработчику исключения.

2.2 Эмуляция виртуальной памяти в Qemu

Qemu – полносистемный кроссплатформенный эмулятор с открытым исходным кодом [2]. Он поддерживает многие популярные компьютерные архитектуры, как в качестве гостевых систем, так и в качестве хостовых.

Работа эмулятора основана на динамической двоичной трансляции кода гостевой архитектуры в код хостовой архитектуры. Но чтобы избежать поддержки большого количества трансляторов для каждого сочетания исходной и целевой архитектуры, и для эффективного выполнения некоторых оптимизаций над кодом, трансляция кода выполняется с дополнительным шагом в виде промежуточного кода. Операции, которые сложно и/или неоправданно преобразовывать с помощью кодогенерации (редкие платформозависимые инструкции, меняющие состояние гостевой системы, трансляция виртуального адреса и прочие), интерпретируются с помощью функций на языке Си, а вызовы к этим функциям встраиваются непосредственно в сгенерированный двоичный код.

2.2.1 Программное управление памятью эмулятора QEMU

Для эмуляции виртуальной памяти гостевых архитектур в Qemu присутствует модуль программного управления памятью MMU. Модуль функционально представляет собой

модель буфера TLB, с набором оптимизаций для ускорения программной трансляции адресов, интерфейс для добавления трансляций в TLB, удаления трансляций, синхронизации транслированного кодогенератором кода с гостевым (в случае изменения гостевых страниц).

Модель TLB в Qemu платформонезависимая и универсальная, позволяет абстрагироваться от формата таблиц трансляции конкретной гостевой архитектуры – непосредственно обходом таблиц трансляции в гостевой системе занимается код эмуляции конкретной архитектуры.

2.2.2 Поиск трансляции в программном TLB эмулятора Qemu

Во время выполнения транслированного кода при обращении к памяти, как и в реальных системах, поиск кешированной трансляции в буфере TLB происходит при каждом обращении к памяти. Эмуляцию обращения к памяти можно разделить на две части: быстрый путь и медленный путь. Быстрый путь выполняется целиком в сгенерированном коде. В случае, если трансляция для адреса отсутствует в программном TLB или существуют какие-то особенности доступа к этой памяти, выполняется длинный путь, с передачей управления хелперу, который загрузит отсутствующую трансляцию в буфер TLB или обработает сложную ситуацию, и вернет управление обратно в транслированный код (или не вернет, в случае гостевого исключения). Для примера, рассмотрим транслированную в код хостовой системы x86-64 инструкцию загрузки переменной из памяти ldr архитектуры aarch64, показанную на рис 1.

В инструкциях 0-3 осуществляется поиск соответствующей адресу обращения записи в программном TLB. В младших битах адреса в записи хранятся флаги доступа, вроде TLB_DISCARD_WRITE (если страница доступна только на чтение), TLB_WATCHPOINT (если на странице установлена точка останова), TLB_MMIO (если на странице есть блок регистров устройств) и т. п. В инструкциях 4-5 адрес обращения подготавливается к сравнению с адресом, хранящимся в записи TLB. В 6-7 выполняется сравнение двух значений, и, если они отличаются, выполнение идет по медленному пути. Таким образом, в быстром пути выполняется доступ только к страницам, где ни один из флагов не установлен, то есть к нормальной памяти, без сторонних эффектов.

2.2.3 MMU контексты в Qemu

Для каждого виртуального процессора в Qemu существует сразу несколько виртуальных буферов TLB (MMU контекстов), трансляции в которых существуют одновременно друг с другом, и используются в разных режимах работы гостевого процессора. Разные контексты нужны для того, чтобы снизить количество очисток программного TLB из-за переключения режимов работы процессора. Как правило, гость использует несколько контекстов для своей работы. MMU контексты – это абстракция модели программного буфера TLB Qemu. Хотя и, в некоторой степени, похожи, контексты не эквивалентны режимам трансляции в aarch64. Также, контексты Qemu совсем не связаны с гостевым идентификатором процесса aarch64 (в разных MMU контекстах могут храниться трансляции как для разных процессов, так и для одного).

2.3 Описание оптимизации

Понимая, как работает программный MMU в Qemu, можно отметить основной его недостаток: поиск нужной трансляции в виртуальном TLB во время выполнения транслированного кода, работающего с памятью, вносит много накладных расходов (около 10 инструкций хостовой системы на одно обращение к памяти гостевой системы). А поскольку в RISC архитектурах, в среднем, больше трети от числа всех выполненных инструкций работают с памятью [3], замедление становится ощутимым. Предлагаемая оптимизация предполагает отказ от программного поиска нужной трансляции в виртуальном TLB, перекладывая работу по трансляции на аппаратный MMU хостовой системы. В

транслированном коде же остается, непосредственно, доступ к памяти с простой корректировкой адреса сложением, и, возможно, проверки над адресом, которые нельзя или нецелесообразно выполнять в таком сочетании гость/хост аппаратно. Достигается это путем размещения дополнительного вида на виртуальное адресное пространство гостевой системы, транслированного кода и инфраструктурного кода для его выполнения в одном адресном пространстве хостовой системы, как правило, в процессе эмулятора (рис. 2).

Рис. 1. Транслированный код для инструкции ldr.
Fig. 1. Translated code for ldr instruction.

По рис. 2 видно, что для использования подхода гостевое адресное пространство должно быть меньше, чем адресное пространство, доступное эмулятору. Размещение всего адресного пространства непрерывным куском требуется, чтобы избежать дополнительного программного уровня трансляции гостевого адреса при выделении его частями, что неприемлемо снизит потенциальный эффект ускорения. Размещение адресного пространства гостя в том же хостовом адресном пространстве, где работает транслированный код, необходимо из-за того, что переключение контекстов - относительно долгая операция, и положительный эффект от аппаратной трансляции адресов ее не перекроет.

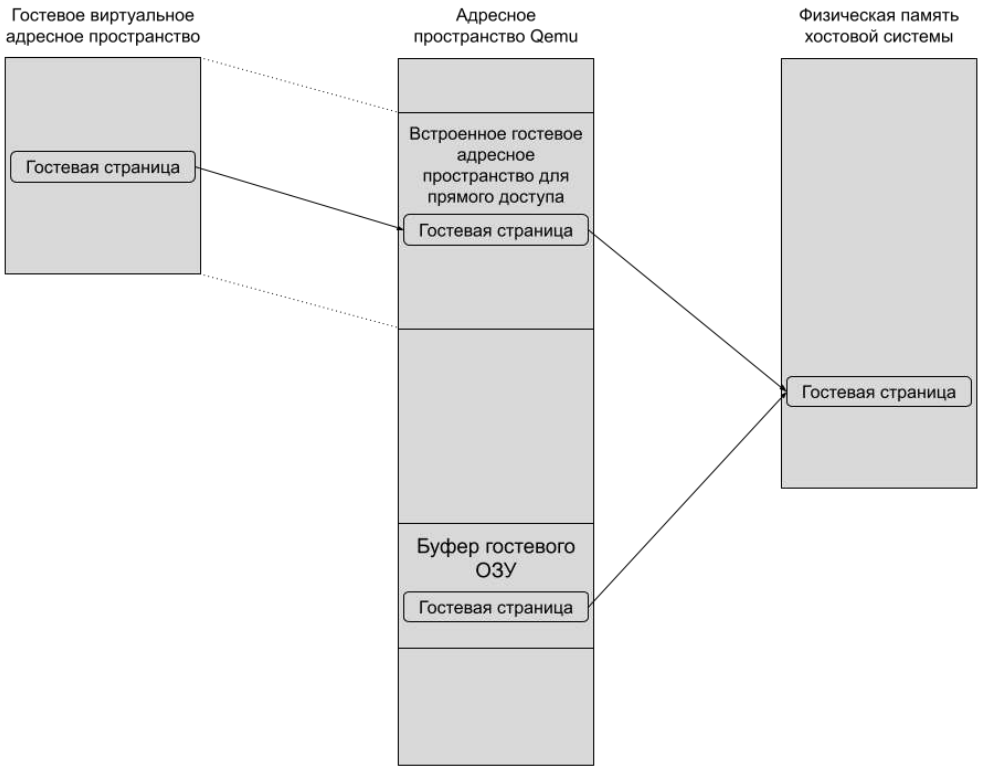


Рис. 2. Дополнительный вид на гостевое вирнуальное адресное пространство.
Fig. 2. Additional view of the guest virtual address space.

Общий алгоритм у всех подобных решений примерно следующий. Вначале, доступ ко всем страницам в диапазоне, выделенном для гостевых адресов, запрещен. Гостевая программ, обращаясь к какой-то странице, вызывает страничное исключение в хостовой системе, после чего эмулятор отображает страницу, вызвавшую исключение на страницу физической памяти хостовой системы, хранящую соответствующие данные физической памяти гостевой системы. Таким образом, следующий доступ из транслированного кода получит прямой доступ к необходимым данным, без необходимости искать трансляцию для адреса в программном TLB. Когда гостевая трансляция становится некорректной (из-за изменений в страничных таблицах), ее отображение на память хостовой системы удаляется. А значит, следующий доступ гостевого кода к этой странице снова вызовет исключение в хостовой системе и добавление уже актуального, корректного отображения.

Манипуляции с отображением страниц в системе – привилегированная операция и эмулятор, как пользовательский процесс, не имеет прямого доступа к таблицам трансляции. В работе [4] для модификации страничных таблиц хостовой системы использовали отдельный модуль ядра, который использовал внутренний интерфейс (Application Programming Interface, API) ядра для этой цели, и управлялся эмулятором. Основными недостатками подхода с модулем ядра является снижение отказоустойчивости системы и необходимость поддержки кода модуля, так как интерфейс ядра может меняться от версии к версии. В дальнейших работах, вместо модуля ядра, люди стали использовать, для той же цели, системный вызов `mmap` [3, 5]. В некоторых работах [6-7] транслированный код с отображенным адресным пространством гостевой системы запускают под аппаратной виртуализацией со вторым аппаратным уровнем трансляции адресов из гостевых физических в хостовые физические.

Такой подход обеспечивает больший контроль над выполнением кода и отображением адресов, но сильно увеличивает сложность разработки в виртуализированном окружении и подразумевает борьбу с замедлением, вызванным переключением процессора между режимами работы, поскольку большая часть функционала эмулятора остается работать без аппаратной виртуализации (из-за отсутствия инфраструктуры для этого под виртуализацией). Так же, для подобного исполнения в процессоре хостовой системы необходима поддержка аппаратной виртуализации со вторым уровнем трансляции [8].

В описанных исследованиях оптимизация используется для эмуляции 32-битных гостевых систем на x86-64 хостовой системе, у которой реальная разрядность виртуального адреса 48 бит. В работе [7] применили вариацию подхода для эмуляции системы архитектуры x86-64 на системе с архитектурой aarch64. Хотя в обеих архитектурах разрядность виртуального адреса одинаковая (48 бит), но в архитектуре aarch64 одновременно используются 2 таблицы страниц, то есть, реально, виртуальное адресное пространство вдвое больше, чем в x86-64, что оставляет достаточно места, чтобы структуры эмулятора не пересекались с адресами гостевой системы.

3. Особенности реализации

В нашем подходе мы решили использовать технику на основе страничных исключений и системного вызова `mmap`. К интересующей нас комбинации архитектур (aarch64 на x86-64) в общем виде оптимизация неприменима, так как процессу эмулятора доступна только младшая половина виртуальных адресов хостовой системы, то есть 47 бит. Часть адресов использует сам эмулятор и библиотеки, при компактном расположении которых, для виртуального адреса гостевой системы остается 46 бит. В гостевой системе с архитектурой aarch64 напротив корректных адресов в 2 раза больше, чем 48 бит (за счет использования двух страничных таблиц).

Однако, часто гостевые ОС (например, на базе Linux) делят адресное пространство aarch64 на старшую половину адресов для привилегированного режима и младшую для пользовательского и не используют их одновременно. К тому же, архитектура aarch64 позволяет отключать часть уровней таблиц трансляции, что эффективно уменьшает разрядность адреса. Например, отключение одного уровня таблиц оставляет разрядность в 39 бит, что позволяет адресовать 512 ГиБ памяти, достаточных для большинства задач. К тому же, это стандартная конфигурация ядра Linux при четырехкилобайтных страницах.

Поэтому принято допущение, что гостевое ПО одновременно использует только одну половину виртуального адресного пространства (старшую или младшую) разрядностью 39 бит, что позволяет выделить немного меньше 2^8 диапазонов под гостевые виртуальные адресные пространства в адресном пространстве эмулятора, оставляя запас для дополнительных оптимизаций.

Архитектура решения допускает, что гостевое ПО может использовать любой адрес в своей работе, не только из ожидаемого диапазона, но в такой ситуации обращение будет обработано через медленный путь, хотя, потенциально, могло быть осуществлено быстрее через быстрый путь при использовании программного буфера TLB. При большом количестве таких адресов, скорость эмуляции может упасть даже больше, чем при отключенной оптимизации, так что стоит избегать подобных ситуаций.

3.1 Аппаратный буфер TLB

Для отображения гостевых виртуальных страниц на хостовые в Qemu, в дополнение к программному TLB добавлен аппаратный TLB. Термин “аппаратный TLB” используется для контраста с программным TLB, его не следует путать с реальным буфером TLB хостовой

системы, который тоже участвует в работе аппаратного буфера TLB эмулятора Qemu. Аппаратный TLB работает параллельно программному TLB и хранит трансляции для страниц с нормальной памятью без сторонних эффектов, к которым возможен непосредственный доступ из транслированного кода. Отключить программный TLB при этом нельзя, так как, помимо транслированного кода, трансляция адресов также требуется кодогенератору, периферийным устройствам, отладчику и пр., и отсутствие кеша трансляций в этих случаях значительно замедлит эмуляцию.

Аппаратный TLB, как и программный, обеспечивает добавление записей в буфер TLB, удаление сразу всех записей или постранично. Для прямого отображения гостевых виртуальных страниц на буфер с данными гостевой памяти используется системный вызов `mmap` с соответствующими флагами доступа (PROT_READ, PROT_WRITE). Для удаления трансляции из аппаратного TLB используется `mmap` с флагом PROT_NONE (то есть обращение к странице памяти вызовет исключение).

Интересной особенностью такого распределенного между таблицами трансляции и реальным TLB системы виртуального TLB является его, условно бесконечный размер (отсутствует вытеснение старых трансляций из-за добавления новых трансляций с таким же ключом). Это позволяет, иногда, избегать повторных избыточных трансляций, в отличие от программного TLB.

3.2 Модификация кодогенерации

Быстрый путь, при программном TLB выполняет несколько операций:

1) проверка, что трансляция существует, и доступ разрешен;
2) проверяет, что доступ выровненный;
3) корректирует виртуальный гостевой адрес для получения виртуального хостового (используя информацию в программном TLB);
4) осуществляет доступ к памяти.

Для аппаратного TLB шаги другие:

1) проверить, что адрес не выходит за границы поддерживаемого диапазона;
2) проверить, что доступ выровнен;
3) прибавить адрес к базе диапазона в адресном пространстве эмулятора;
4) выполнить доступ к памяти.

Первый пункт необходим, потому что гостевое ПО может использовать любое 64-битное значение в качестве адреса, и обратиться к запрещенной для него памяти. Проверку на выравнивание доступа можно совместить, в случае младших адресов, с проверкой адреса. Для старших гостевых адресов – это отдельные инструкции. Адрес базы диапазона прибавляется в x86-64 одновременно с доступом к памяти, однако, адрес базы необходимо перед этим загрузить в регистр. Во время доступа к памяти аппаратно проверяется, что трансляция для страницы существует, и доступ к ней разрешен. Во всех пунктах, если условие не удовлетворяется, происходит переход к медленному пути (через ветвление или исключение).

На рис. 3 показан транслированный код для инструкции `ldr` со включенным аппаратным TLB.

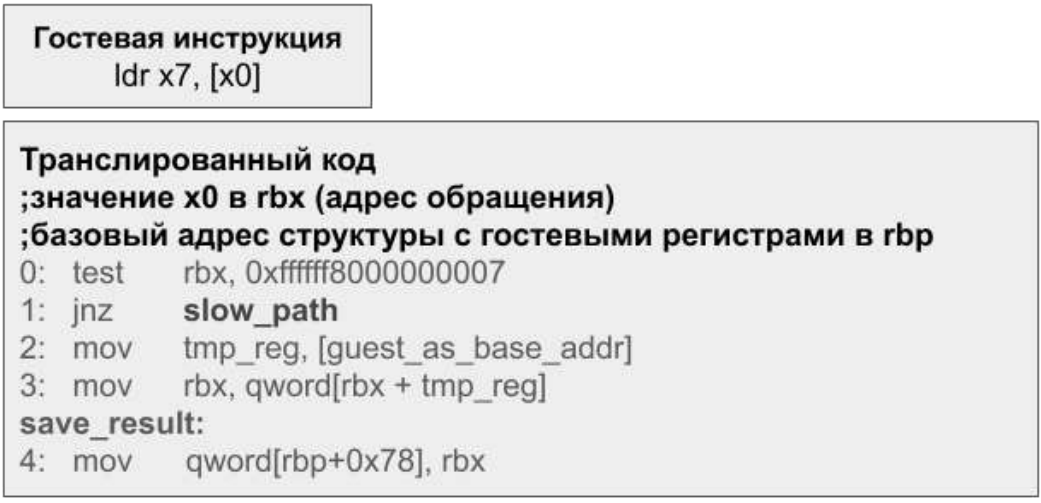


Рис. 3. Транслированная инструкция `ldr` *для использования с аппаратным TLB.*
Fig. 3. Translated `ldr` *instruction for hardware TLB usage.*

3.3 Отключение проверок в транслированном коде

Хотя наличие в быстром пути проверок на выравнивание адреса и попадание адреса в ожидаемый диапазон обеспечивает корректность эмуляции и стабильность работы эмулятора, замечено, что на правильно работающих программах эти проверки никогда не срабатывают. Поэтому, если потребность в ускорении перекрывает потенциальную, но маловероятную, некорректность и нестабильность работы эмулятора, проверки можно исключить из быстрого пути, сократив его. Результат удаления проверок из быстрого пути для инструкции `ldr` показан на рис. 4.

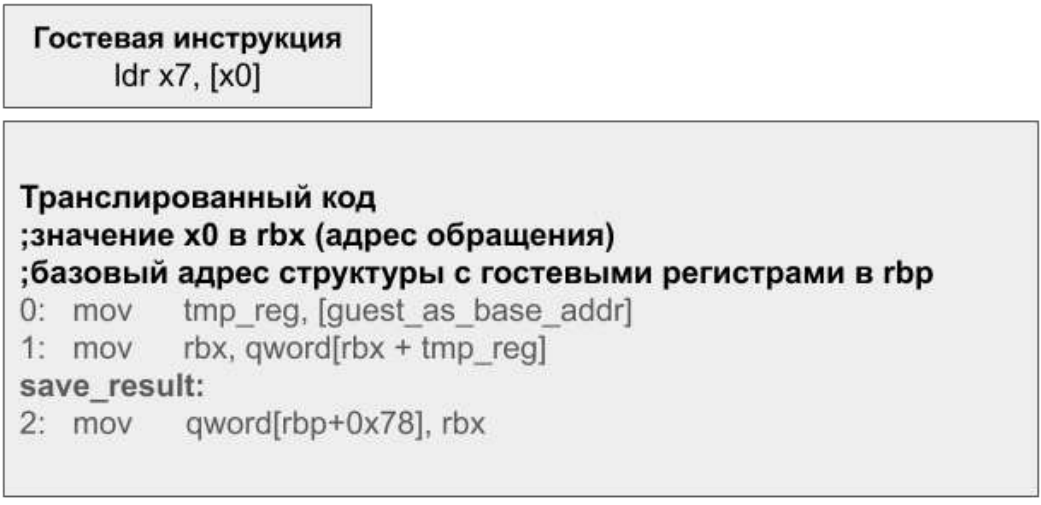


Рис. 4. Код инструкции `ldr` *для аппаратного TLB без проверок.*
Fig. 4. Hardware TLB `ldr` *code without checks.*

3.4 Использование индекса MMU для избирательного применения оптимизации

Поскольку медленный путь, при использовании подхода становится еще медленнее, чем с программным MMU (за счет обработки через прерывание, вместо простого вызова процедуры), сценарии с большим количеством медленных путей плохо подходят для алгоритма. Часто, работа через медленный путь вызвана обращением гостевых программ к устройствам периферийным устройствам. В [3] было замечено, что программы не имеют прямого доступа к устройствам, а делают это посредством привилегированного ПО, которое работает в режиме ядра. Значит, можно не включать оптимизацию для кода ядра гостевой системы, а продолжать использовать модуль программного MMU эмулятора, а для пользовательского гостевого кода использовать аппаратный буфер TLB. К тому же, код ядра обычно располагается в старшей части адресного пространства, а, как упомянуто ранее, транслированный код для доступа к старшим адресам длиннее, чем к младшим, из-за более длинной проверки и корректировки гостевого виртуального адреса, поэтому потенциальное ускорение выше при обращении к младшим адресам, а для старших адресов сходит на нет.

Поскольку решить, использовать оптимизацию или нет надо на этапе трансляции кода, в качестве критерия используется MMU индекс, который с достаточной точностью позволяет отличать ядерный и пользовательский режим работы.

Другим примером исключения MMU индекса из работы алгоритма могут служить индексы, которые не используются из сгенерированного кода, а только хелперами и инфраструктурой эмулятора. Нет смысла поддерживать аппаратный TLB для такого индекса, поскольку он никогда не будет использоваться.

3.5 Отслеживание гостевых контекстов

Внесение изменений в аппаратный TLB - относительно дорогая операция, в сравнении с модификацией программного TLB. Поэтому, если выполнять очистку аппаратного TLB каждый раз, когда гостевая система переключает процесс, как это делает программный TLB в Qemu, накладные расходы на очистку и повторную трансляцию адресов будут неоправданными. Для сглаживания отрицательного эффекта от переключения процессов в аппаратный TLB добавлено отслеживание идентификатора процесса, по типу того, как это происходит в реальном TLB современных машин. При смене таблиц трансляций во время переключения контекстов, трансляции предыдущего процесса не отбрасываются, а сохраняются для будущего использования в пуле контекстов до момента, когда процесс станет активен снова.

Для отслеживания текущего процесса используется, как и в реальном TLB, архитектурно определенный в системе команд aarch64 идентификатор процесса `asid`. Кешированные трансляции процесса удаляются при очистках кеша, не связанных с переключением контекста. В случае, отсутствия свободных контекстов в пуле контекстов, самый старый контекст заменяется новым.

Пул контекстов формируется из доступных диапазонов гостевых виртуальных адресных пространств. Для 39-битного гостя и эмулятора на хостовой системе x86-64, максимальный размер пула для каждого виртуального процесса, при многопоточной работе кодогенератора: $\texttt{floor}(2^8/N_{cpus})$, где `floor` – функция округления вниз, а N_{cpus} – количество виртуальных процессоров. В случае с однопоточной кодогенерацией, максимальный размер пула немногим меньше 2^8. Однако при работе реальных гостевых систем он не используется целиком, потому что полная очистка буфера (`tlb flush`) случается намного чаще, чем переключение между 2^8 уникальными гостевыми контекстами, а в этом случае все гостевые адресные диапазоны возвращаются обратно в пул. По результатам наблюдений, оптимальный размер пула на поток кодогенератора находится в диапазоне от 8 до 16.

4. Производительность

Для оценки производительности предложенного решения в качестве гостевой системы была выбрана виртуальная машина raspberry 3b [9] (одноплатный компьютер с 4 ядерным SoC с архитектурой aarch64, 1 Гб ОЗУ и набором периферии). В качестве гостевой ОС использовалась Raspberry Pi OS Lite [10] (основана на ОС Debian 11 Bullseye). Параметры хостовой системы: Intel® Core™ i7-8700 CPU @ 3.20GHz, 32Гб ОЗУ, 500Гб SSD.

Для тестовых примеров использовались несколько реальных сценариев работы: загрузка ОС, скачивание файла из сети, установка программы, архивация/деархивация файла, а также эталонный тест Nbench. Для большей достоверности, результаты измерений подвергались статистической обработке. Nbench измеряет производительность в итерациях в секунду. В остальных же тестах измерялось время работы, но для унифицированного представления полученные значения инвертировались. Результаты измерений представлены в табл. 1.

Табл. 1. Сравнение производительности базовой версии Qemu и с включенной оптимизацией.
Table 1. Performance comparison of vanilla Qemu and Qemu with abovementioned optimization.

Название теста	Базовая Qemu 9.1.50	Qemu 9.1.50 с аппаратным TLB	Производительность (больше 100 – ускорение, меньше – замедление), %
Загрузка ОС	22.02	18.69	84.87
Скачивание файла ~50 МБ	3.80	3.81	100.26
Установка программы	11.01	10.83	98.36
Многопоточное сжатие файла ~50 МБ	15.67	34.12	217.74
Извлечение файла ~50 МБ	28.24	30.76	108.92
Make (программы nbench)	17.36	16.31	93.95
Nbench общее	1.91	3.11	162.82
NUMERIC SORT (Nbench)	185.79	504.01	271.27
STRING SORT (Nbench)	65.18	107.13	164.36
FP EMULATION (Nbench)	128.22	284.59	221.95
FOURIER (Nbench)	6009.23	6608.07	109.96
ASSIGNMENT (Nbench)	9.03	11.51	127.46
IDEA (Nbench)	3122.83	3788.79	121.32
HUFFMAN (Nbench)	1082.52	1355.60	125.22
NEURAL NET (Nbench)	4.68	5.36	114.52
LU DECOMPOSITION (Nbench)	190.67	220.29	115.53
INTEGER INDEX (Nbench)	27.94	44.59	159.59
FLOATING-POINT INDEX (Nbench)	7.89	9.04	114.57
MEMORY INDEX (Nbench)	6.48	9.15	141.20
BITFIELD (Nbench)	204681940	253949410	124.07

По измерениям видно, что производительность со включенной оптимизацией зависит от типа нагрузки на гостевую систему. Большое ускорение получено при работе бенчмарков (ускорение до 2.71 раз, 1.62 раза в среднем), и на мультипоточном сжатии файла (2.17 раз). В остальных реальных примерах производительность изменилась незначительно, кроме теста с

загрузкой ОС, где производительность снизилась на 15%. Замедление при загрузке можно объяснить тем, что эффект ускорения от использования аппаратного TLB не перевесил добавившиеся накладные расходы на его работу.

5. Заключение

В статье рассмотрена оптимизация трансляции виртуальных адресов при кроссплатформенной полносистемной эмуляции, позволяющая перенести часть нагрузки с программного MMU эмулятора на аппаратуру трансляции адресов хостовой системы. Выполнена практическая реализация описанного подхода на эмуляторе Qemu. Замеры производительности итогового решения показывают большое ускорение при использовании оптимизации, в том числе и для реальных программ, а не только бенчмарков.

Предложенное исполнение аппаратного TLB для ускорения трансляции гостевых адресов может применяться для эмуляции aarch64 на x86-64, что раньше в работах не встречалось.

Хотя описанный подход нельзя назвать универсальным, потому что он предполагает определенные настройки гостевой операционной системы, часто такая настройка возможна, либо уже используется в выбранной системе.

Для снятия ограничения на 3 уровня таблиц трансляций в гостевой системе возможно использование расширение процессоров x86-64, увеличивающего разрядность виртуального адрес хостовой системы до 57 бит [11]. Это обеспечит достаточный запас виртуальных адресов для применения описанного выше подхода для систем aarch64 с полным 48-битным адресом. В перспективе мы собираемся проверить этот подход на практике.

Список литературы / References

[1]. Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. 2014. Efficient memory virtualization for Cross-ISA system mode emulation. In Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '14). Association for Computing Machinery, New York, NY, USA, 117–128.

[2]. “Qemu – a generic and open source machine emulator and virtualizer,” https://www.qemu.org/, accessed: 22.05.2025.

[3]. Antoine Faravelon. Acceleration of memory accesses in dynamic binary translation. Operating Systems [cs.OS]. Université Grenoble Alpes, 2018. English.

[4]. Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. Efficient memory virtualization for cross-isa system mode emulation. In ACM SIGPLAN Notices, volume 49, pages 117–128. ACM, 2014.

[5]. Zhe Wang, Jianjun Li, Chenggang Wu, Dongyan Yang, Zhenjiang Wang, WeiChung Hsu, Bin Li, and Yong Guan. Hspt: Practical implementation and efficient management of embedded shadow page tables for cross-isa system virtual machines. In ACM SIGPLAN Notices, volume 50, pages 53–64. ACM, 2015.

[6]. Tom Spink, Harry Wagstaff, and Björn Franke. Hardware-accelerated crossarchitecture full-system virtualization. ACM Transactions on Architecture and Code Optimization (TACO), 13(4):36, 2016.

[7]. S. Rodzevich, K. Batuzov, D. Koltunov, A. Cheremnov and I. Shlyapin, "Efficient MMU Emulation in Case of Cross-ISA Dynamic Binary Translation," 2024 Ivannikov Ispras Open Conference (ISPRAS), Moscow, Russian Federation, 2024, pp. 1-6, doi: 10.1109/ISPRAS64596.2024.10899135.

[8]. “AMD-V Nested Paging White Paper,” https://www.cse.iitd.ac.in/~sbansal/csl862-virt/readings/NPT-WP-1%201-final-TM.pdf, accessed: 22.05.2025.

[9]. “Raspberry Pi 3 Model B,” https://www.raspberrypi.com/products/raspberry-pi-3-model-b/, accessed: 22.05.2025.

[10]. “Raspberry Pi OS (Legacy) Lite,” https://downloads.raspberrypi.com/raspios_oldstable_lite_arm64/images/raspios_oldstable_lite_arm64-2025-05-07/2025-05-06-raspios-bullseye-arm64-lite.img.xz, accessed: 22.05.2025.

[11]. “Introduction to 5-Level Paging in 3rd Gen Intel Xeon Scalable Processors with Linux,” https://lenovopress.lenovo.com/lp1468-introduction-to-5-level-paging, accessed: 22.05.2025.

Информация об авторах / Information about authors

Дмитрий Николаевич ПОЛЕТАЕВ – разработчик программного обеспечения, сотрудник Института системного программирования РАН. Сфера научных интересов: обратная разработка, анализ бинарного кода, виртуальные машины.

Dmitry Nikolaevich POLETAEV – software developer at Ivannikov Institute for System Programming of the Russian Academy of Sciences. Research interests: reverse engineering, binary code analysis, virtual machines.

Павел Михайлович ДОВГАЛЮК – инженер, кандидат технических наук. Сфера научных интересов: интроспекция и инструментирование виртуальных машин, динамический анализ кода, отладчики, эмуляторы.

Pavel Mikhailovich DOVGALYUK – Cand. Sci. (Tech.), engineer. Research interests: virtual machines introspection and instrumentation, dynamic analysis of code, debuggers, emulators.

Георгий Николаевич ТЕЙС – инженер по тестированию Института системного программирования РАН с 2022 года. Сфера научных интересов: автоматизация процессов в среде системного программирования.

Georgiy Nikolaevich TEYS – QA engineer of the Institute for System Programming of the RAS since 2022. His research interests include processes automation in system programming.

Максим Алексеевич КОСТИН – инженер отдела компиляторных технологий ИСП РАН. Научные интересы включают эмуляторы, динамическую двоичную трансляцию, оптимизации.

Maksim Alekseevich KOSTIN – engineer at Compiler Technology department of ISP RAS. His research interests include emulators, dynamic binary translation, optimizations.

DOI: 10.15514/ISPRAS-2025-37(6)-36

CC BY

Интроспекция виртуальной машины на основе мониторинга системных вызовов и структур данных ядра

В.М. Степанов, ORCID: 0000-0003-2141-3333 <vladislav.stepanov@ispras.ru>
П.М. Довгалюк, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>
Н.И. Фурсова, ORCID: 0000-0001-6817-4670 <natalia.fursova@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.
Новгородский государственный университет им. Ярослава Мудрого,
Россия, 173003, Великий Новгород, ул. Большая Санкт-Петербургская, д. 41.

Аннотация. Семантический разрыв представляет собой одну из ключевых проблем в разработке решений полносистемного динамического анализа кода. Она заключается в том, что на уровне гипервизора инструмент имеет доступ только к низкоуровневым бинарным данным выполняемого кода, в то время как для анализа требуется высокоуровневая информация о состоянии объектов гостевой операционной системы. Данную проблему решают подходы интроспекции виртуальной машины. К сожалению, реализации существующих подходов сталкиваются с проблемами производительности и недостатка функционала, требуют от пользователя внедрять в образ виртуальной машины специальные агенты или иметь в наличии отладочные символы к ядру, а также оказываются заточенными под специфичные системы и архитектуры процессоров. В статье представлен ряд решений, помогающих снизить накладные расходы и добиться большей универсальности для инструмента анализа. Особенность разработанного подхода интроспекции заключается в том, что необходимую для анализа информацию он собирает в процессе запуска системы на эмуляторе, не требуя при этом каких-либо дополнительных действий со стороны пользователя.

Ключевые слова: виртуальные машины; динамический анализ; эмулятор QEMU; интроспекция.

Для цитирования: Степанов В.М., Довгалюк П.М., Фурсова Н.И. Интроспекция виртуальной машины на основе мониторинга системных вызовов и структур данных ядра. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 59–72. DOI: 10.15514/ISPRAS–2025–37(6)–36.

Благодарности: Исследование выполнено за счет гранта Российского научного фонда № 24-11-20022.

Virtual Machine Introspection Based on System Calls and Kernel Data Structures

V.M. Stepanov, ORCID: 0000-0003-2141-3333 <vladislav.stepanov@ispras.ru>
P.M. Dovgalyuk, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>
N.I. Fursova, ORCID: 0000-0001-6817-4670 <natalia.fursova@ispras.ru>

Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn str., Moscow, 109004, Russia.
Yaroslav-the-Wise Novgorod State University,
41, B. Sankt- Peterburgskaya st., Veliky Novgorod, 173003, Russia.

Abstract. The semantic gap is one of the key problems in developing solutions for full-system dynamic analysis. At the hypervisor level, tools have access only to low-level binary data, while analysis requires high-level information about the state of guest operating system objects. Virtual machine introspection approaches solve this problem. Unfortunately, implementations of existing approaches face performance issues and lack of functionality. They require the user to embed special agents into the virtual machine image or have debugging symbols for the OS kernel. They also turn out to work only for specific systems and processor architectures. The article presents a number of solutions that reduce overhead and increase the versatility of the analysis tool. The peculiarity of the developed introspection approach is that it does not require any additional actions from the user, collecting the information necessary for analysis during the OS boot on the emulator.

Keywords: virtual machine; dynamic analysis, QEMU; virtual machine introspection.
For citation: Stepanov V.M., Dovgalyuk P.M., Fursova N.I. Virtual Machine Introspection Based on System Calls and Kernel Data Structures. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, part 3, 2025, pp. 59-72 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-36.

Acknowledgements. The work was partially supported by the Russian Science Foundation (grant No. 24-11- 20022).

1. Введение

Технология интроспекции виртуальной машины имеет широкое применение в инструментах динамического анализа. Она реализует преобразование низкоуровневых бинарных данных, доступных на уровне гипервизора, в высокоуровневую информацию об объектах ОС. В инструменте Natch [1] с ее помощью собирается информация о поверхности атаки, т.е. об исполняемых файлах и функциях, которые отвечают за обработку данных, полученных из недоверенных источников. Это достигается за счет комбинирования интроспекции с отслеживанием помеченных данных.

Основная проблема существующих методов интроспекции виртуальных машин заключается в их узкой специализации и сложности применения. Архитектура операционных систем может сильно различаться в зависимости от семейства, версии ядра, флагов компиляции и других факторов. Поэтому построение универсального подхода, способного осуществлять мониторинг на любых гостевых ОС, является практически недостижимой целью. Большинство работ в данной сфере ограничиваются только реализацией интроспекции для ОС семейства Linux. Существующие инструменты также часто требуют наличия отладочных символов к анализируемому ядру [2-4], либо внедрения в это ядро специальных модулей, собирающих о нем информацию [4-6].

Мы предлагаем способ сбора необходимых для интроспекции данных о системе на основе простого запуска ОС на виртуальной машине. Это упрощает процесс подготовки образа виртуальной машины к анализу, ведь пользователю конечного инструмента потребуется только подождать, пока не закончится автоматическая настройка. По завершению такой настройки генерируется конфигурационный файл, называемый профилем интроспекции.

Существует ряд работ, в которых задача построения профиля интроспекции для системы решается на основе анализа снимков памяти с применением специально подготовленных разработчиком эвристик [7, 8]. В других исследованиях процесс построения эвристик автоматизируется на основе статического анализа бинарного кода [9], либо применения машинного обучения [10]. Есть также работы демонстрирующие эффективность применения для данной задачи динамического анализа, например подходы Katana [11] и ORIGEN [12].

В данной статье описывается подход интроспекции виртуальной машины для гостевых ОС Linux, Windows и FreeBSD. Он опирается на отслеживание изменений в структурах данных ядра на протяжении сеанса работы эмулятора. Для достижения универсальности подхода вводятся абстрактные типы структур данных ядра, которые обобщают реальные типы структур разных семейств ОС. Таким образом, становится возможной единая реализация алгоритмов интроспекции без привязки к конкретным архитектурам. С использованием этих абстрактных типов также реализуется набор тестов, проверяющих корректность извлекаемой из структур данных ядра информации. Эти тесты применяются для подбора смещений полей и определения размеров структур, которые по ходу настройки заносятся в генерируемый профиль интроспекции.

2. Обзор существующих решений

Инструменты TEMU [4], DECAF [5] и PANDA [6] реализуют задачи интроспекции виртуальной машины на базе эмулятора QEMU [13]. Во всех трех работах применяются внедряемые в гостевую ОС модули ядра. В TEMU такой модуль используется во время исполнения для извлечения высокоуровневых данных о состоянии Windows. В DECAF и PANDA внедряемые агенты применяются только для создания конфигурационного файла, применяемого затем в анализе без осуществления вмешательства в гостевую ОС.

Инструменты DRAKVUF [2] и RTKDSM [3] работают на основе эмулятора Xen, из-за чего ограничены в применении только архитектурой процессоров x86. Для своей работы они требуют наличия отладочных символов ядра, которые используются для извлечения параметров объектов ОС из структур данных в гостевой памяти. В DRAKVUF отладочные символы применяется также для перехвата системных функций, на основе которых инструмент отслеживает происходящие изменения в системе. В свою очередь RTKDSM выполняет мониторинг объектов ОС, отслеживая модификации страниц памяти со структурами данных ядра.

Табл. 1. Сравнение подходов интроспекции.
Table 1 Comparison of introspection approaches.

	TEMU	DECAF	PANDA	RTKDSM	Drakvuf
Внедряет гостевые драйвера	для исполнения (в Windows)	для настройки	для настройки	нет	нет
Требует наличия отладочных символов	да (для Linux)	нет	нет	да	да
Эмулятор	QEMU	QEMU	QEMU	Xen	Xen
Гостевые ОС	Windows, Linux	Windows, Linux	Windows, Linux, FreeBSD	Windows, Linux	Windows, Linux
Архитектуры процессоров	x86	x86, Arm	x86, Arm, Mips	x86	x86
Отслеживает системные вызовы	нет	нет	да	нет	да

При работе с вышеперечисленными инструментами необходимо иметь в наличии отладочные символы ядра или внедрять в ядро гостевой ОС специальные агенты. В первом случае может возникнуть проблема, когда разработчик дистрибутива не предоставляет такие символы к своей системе, или возможно даже не хранит их для старых версий. Что касается внедряемых агентов, то с некоторыми образами виртуальной машины произвести подобные модификации оказывается затруднительно или невозможно.

Ряд инструментов умеет выполнять задачу построения профиля интроспекции только на базе анализа бинарного кода. Например, подходы Katana[11] и RAMAnalyzer [13] для этого перехватывают вызовы определенных функций ядра, выполняющих доступ к искомым параметрам. При этом для распознавания функций ядра они извлекают отладочные символы Linux из механизма kallsyms. Проблема таких решений в том, что kallsyms может быть отключен в конфигурации ядра, а функции могут быть сильно изменены при переходе на новые версии. Для решения последней проблемы Katana применяет анализ исходного кода Linux, позволяющий автоматически определять функции ядра, на основе которых в ходе динамического анализа определяются смещения полей структур данных.

Подход ORIGEN [12] применяет сопоставление бинарного кода разных версий ядра ОС для переноса отладочной информации с одной системы на другую. Он использует динамический анализ для нахождения операций доступа к искомым полям структур данных в эталонном ядре, затем находит эквивалентные инструкции в целевом ядре, по которым восстанавливает смещения.

3. Natch

Далее обсудим полностью автоматизированный подход восстановления архитектуры ОС, работающий исключительно на основе анализа бинарного кода, а значит не требующий внедрения гостевых агентов и наличия отладочных символов ядра. Это решение легло в принцип работы инструмента определения поверхности атаки Natch.

3.1 Общая структура подхода

Реализация подхода представляет собой набор плагинов к эмулятору QEMU, которые осуществляют инструментирование выполняемого кода. В силу того, что такой способ сбора данных о работе гостевой системы замедляет работу эмулятора, плагины для мониторинга объектов ОС предпочтительно применять совместно с детерминированным воспроизведением. Это нужно для того, чтобы замедление от средств анализа не оказывало влияние на поведение приложений в гостевой системе.

Исходя из этого, сценарий работы с инструментом включает в себя следующие шаги:

1) Подготовка образа и запуск гостевой ОС для построения профиля интроспекции.
2) Запись сценария работы с объектами оценки средствами эмулятора QEMU.
3) Воспроизведение записанного сценария со сбором данных о состоянии объектов гостевой ОС.

На рис. 1 можно увидеть, что за чтение структур данных во время мониторинга и за создание профиля интроспекции отвечают два отдельных плагина. Они опираются на общую библиотеку, описывающую для каждого семейства ОС алгоритмы извлечения параметров структур данных. Плагины мониторинга процессов и файлов в своей работе взаимодействуют с плагинами трассировки системных вызовов и чтения структур данных ядра. Другие плагины опираются на мониторинг объектов для выполнения некоторых специфичных задач, таких как отслеживание исполняемых модулей, пометка файлов и построение графов распространения помеченных данных. По завершению мониторинга объектов гостевой ОС формируются лог-файлы и графы, которые затем могут быть приведены к более удобному для аналитика формату с помощью средств визуализации.

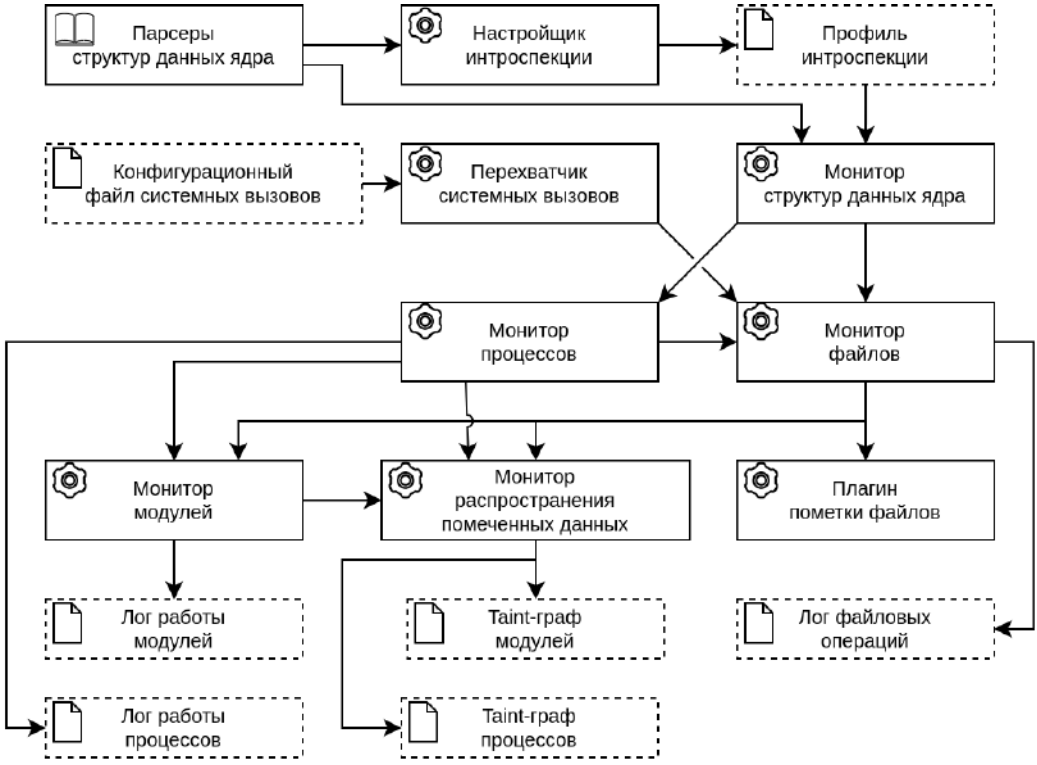


Рис. 1. Схема взаимодействий плагинов интроспекции.
Fig. 1. Introspection plugin interaction diagram.

3.2 Мониторинг объектов ОС

К основным объектам ОС, которые представляют интерес для отслеживания, относятся процессы, потоки, файлы, каталоги, системные вызовы. Задача интроспекции ВМ предполагает преобразование низкоуровневых данных, таких как потоки инструкций и снимки памяти в высокоуровневую информацию об этих объектах.

Свойства объектов описываются в специальных структурах данных ядра, называемых дескрипторами. Для нахождения этих дескрипторов в памяти используется профиль интроспекции анализируемой ОС. Также существует способ отслеживания информации об объектах без использования структур ядра. Он заключается в перехвате системных функций с обработкой их входных и выходных данных. Системные вызовы, как правило, менее изменчивы по сравнению со структурами данных, поэтому одна реализация их трассировки может работать на большом диапазоне версий ядра ОС [13].

Объекты ОС взаимосвязаны и иногда дублируют одни и те же данные. Например, имена пользовательских процессов могут быть извлечены из дескриптора процесса, из дескриптора исполняемого файла, а также из входных данных системной функции execve. Благодаря этому свойству можно проверять корректность параметров на специальном наборе тестов, в ходе которых сопоставляются полученные из разных мест данные. Такие наборы тестов позволяют удостовериться в соответствии профиля интроспекции анализируемой гостевой ОС.

Между семействами ОС Linux, Windows и FreeBSD существует множество различий с точки зрения способа хранения метаданных объектов в памяти. Для упрощения работы с ними мы определяем абстрактные типы структур данных, которые обобщают некоторые структуры

ядра разных ОС. Для извлечения параметров объектов задается общий интерфейс, каждая функция которого реализуется отдельно под каждую архитектуру. В свою очередь плагины эмулятора, выполняющие мониторинг за объектами, описываются на более высоком уровне с использованием абстрактных типов, поэтому оказываются абстрагированы от особенностей реализации гостевых систем.

Как показано на рис. 2, мониторинг процессов может отслеживать их имена, идентификаторы, состояние и иерархию. Дескриптор процесса представляет собой структуру *task_struct* в ядре Linux, *EPROCESS* в Windows, *Process* в FreeBSD. Определить местоположение дескриптора текущего выполняемого потока или процесса в каждом случае возможно по значению регистра GS для платформы x86, либо *TPIDR_EL1* для ARM. Через параметры текущего процесса удается выполнить переход к структурам ядра, описывающим все остальные объекты системы.

Рис. 2. Диаграмма структур данных ядра.
Fig. 2. Kernel data structures diagram.

Разбор таблицы объектов позволяет определить метаданные открытых процессом файлов. Кроме обычных файлов в той же таблице могут быть найдены сокеты и каналы, используемые для осуществления взаимодействий между процессами. Операции с файлами, такие как чтение и запись, отслеживаются по вызовам системных функций. Для средств взаимодействия также определяются параметры объектов, с которыми происходит соединение и обмен данными.

Дескрипторы областей виртуальной памяти организуются в виде деревьев поиска. По ним могут быть определены диапазоны адресов всей выделенной процессу памяти. В том числе можно найти исполняемый код процесса, стек, динамические библиотеки, разделяемую память и многое другое. Разбор этих структур ядра применяется в мониторинге исполняемых модулей, определении командной строки запуска процесса и отслеживании способов передачи при распространении помеченных данных (рис. 3).

3.3 Теневые структуры данных

Чтение данных из гостевой памяти является достаточно ресурсоёмкой операцией, вследствие чего ее частое применение может приводить к сильному замедлению работы виртуальной машины. Для ускорения анализа, а также корректной обработки изменений, все извлекаемые метаданные объектов следует кэшировать. На старте анализа выполняется сбор информации обо всех запущенных процессах и их ресурсах. Далее на протяжении сеанса работы виртуальной машины отслеживаются изменения в состояниях объектов ОС, в соответствии с чем обновляются данные кэша.

Изменения, как правило, сопровождаются системными вызовами. При этом в соответствии с типом системного вызова и его аргументами легко определить, в каких конкретно

объектах ОС что-то поменялось, и обновить только их. Файловый монитор, например, может обходиться совсем без чтения структур данных ядра, извлекая имена файлов из аргументов вызова open, а параметры сокетов из вызовов socket, bind и connect. Тем не менее часть изменений в объектах может происходить в отсутствие системных вызовов, что приводит к необходимости применения альтернативного метода отслеживания их состояний.

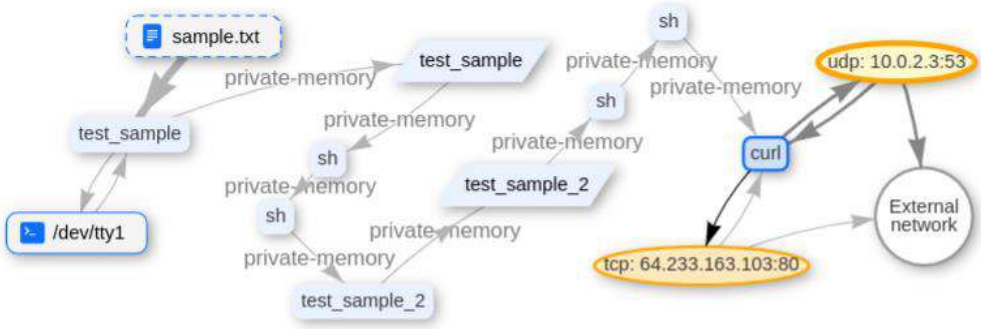


Рис. 3. Граф распространения помеченных данных по процессам.
Fig. 3. Taint graph for processes.

Другой метод реагирования на изменения опирается на перехват операций записи в структуры ядра. Так как запись каких-либо данных в память является крайне часто встречающейся операцией, инструментирование всех операций записи очевидно приводит к большим временным издержкам. Значительно облегчает задачу наличие в эмуляторе механизма точек наблюдения. Этот механизм позволяет инструментировать запись в определенные диапазоны памяти. При этом замедлению поддаются только те операции записи, которые направлены на физические страницы памяти с такими точками наблюдения.

Точки наблюдения удобно применять для отслеживания состояний процессов. Когда происходит завершение процесса, в его поле состояния записывается флаг смерти. Таким же образом может отслеживаться переход процесса в состояние зомби. Аналогичная идея: определять необходимость удаления файлового дескриптора из кэша по перехвату записи нулевого значения в его счетчик указателей.

Еще один способ применения точек наблюдения – это отслеживание записи в примитивы синхронизации структур данных ядра. Поля структур данных ядра часто оказываются защищены от возможного «состояния гонки» с помощью спин-блокировок, мьютексов или семафоров. Соответственно момент разблокировки таких примитивов может послужить для инструмента анализа сигналом к повторному чтению метаданных объекта ОС из гостевой памяти и обновлению теневых структур данных ядра.

3.4 Построение профиля интроспекции

Задача нахождения смещений полей структур данных в нашей работе решается на основе набора тестов, позволяющих удостовериться в корректности извлекаемых из гостевой памяти данных. Тесты организуются таким образом, чтобы проверки, требующие наименьшего количества известных смещений полей, выполнялись в первую очередь. Для каждого теста при этом выполняется перебор всех возможных значений смещений проверяемых им полей. Большинство извлекаемых из ядра параметров последовательно проверяются сразу на нескольких тестах, в результате чего отсеиваются наборы смещений, прошедшие первые проверки по ошибке.

Что представляет собой тест? Это небольшой алгоритм, выполняемый в некотором остановленном состоянии виртуальной машины. Как правило, в ходе выполнения теста

происходит попытка извлечения из гостевой памяти параметров ядра. Корректность этих параметров проверяется по области допустимых значений, либо по совпадению значений с некоторыми известными из другого источника данными.

Тесты можно поделить на две категории: обязательные к прохождению и эвристические. Первые всегда должны выдавать положительный результат при корректно подобранных смещениях полей. За счет этого они позволяют отсеивать все варианты значений, с которыми результат проверок отрицательный. С эвристическими проверками предполагается, что даже при корректно подобранных смещениях полей результат может оказаться отрицательным. Поэтому такие проверки нужно выполнять несколько раз на разных состояниях виртуальной машины, пока не удастся найти набор смещений, с которым они будут пройдены.

Тесты связываются между собой в соответствии с их зависимостями от полей структур данных и передаваемых параметров. Из этих связей формируется дерево. Корневым узлом, например, становится проверка, в ходе которой извлекается адрес текущего дескриптора процесса. Тесты, проверяющие отдельные параметры процесса, такие как имя, родитель, уникальный идентификатор, оказываются связаны с корнем. За счет этого они используют уже извлеченный адрес дескриптора процесса из предыдущей выполненной проверки, а не читают его из гостевой памяти вновь. По итогу, при удачном выполнении всех связанных тестов удается подтвердить, что проверяемая структура данных содержит все те поля, которые должны присутствовать в дескрипторе текущего процесса, а значит наиболее вероятно она им и является.

Некоторые проверки требуют своего выполнения на определенных состояниях виртуальной машины. Например, тест идентификатора процессов выполняется в момент завершения системного вызова `getpid` (В случае Linux и FreeBSD). В ходе его работы извлекаемый из структуры ядра идентификатор `pid` сравнивается с возвращаемым значением системной функции. Другой пример, смещения параметров файловых структур данных подбираются при перехвате вызова `open`. Здесь проверяется, что извлекаемое из структур ядра имя открываемого файла совпадает с тем, которое передается в системную функцию в качестве аргумента. Смещения параметров областей виртуальной памяти в свою очередь подбираются при перехвате вызова `mmap`. Для этой задачи реализуется тест, в ходе которого находится структура данных ядра, описывающая создаваемое отображение файла в память.

Основная идея подхода генерации профиля интроспекции заключается в последовательном нахождении смещений всех параметров структур ядра на базе ранее найденных. Так подбор смещений для файловых структур данных выполняется только после нахождения указателя на дескриптор текущего процесса. А тесты для нахождения дескрипторов областей виртуальной памяти и дескрипторов сокетов работают на базе ранее найденных файловых структур. Эти особенности выражаются в виде связей между тестами.

Зависимости, наборы перебираемых смещений и передаваемые между тестами параметры описываются в декларативной форме представления. За счет этого, при разработке алгоритмов генерации профиля интроспекции удается абстрагироваться от точной последовательности шагов по подбору смещений и описывать по большей части только сами проверки, включающие в себя извлечение из гостевой памяти искомых параметров. Также используемый формат организации тестов открывает возможность проверять с их помощью соответствие гостевой ОС уже готовому профилю интроспекции, либо начинать подбор смещений с середины, то есть на базе какого-то незаконченного профиля. Этапы переборов смещений в этих случаях просто пропускаются, т.к. их значения уже известны.

На рис. 4 в виде схемы представлен пример декларативного описания проверок. Узлы, обозначенные в виде скругленного прямоугольника, представляют собой точки останова, в которых выполняются следующие по цепочке проверки. В данном случае это системные вызовы `getpid`, `open` и `mmap`. Обычные прямоугольники описывают извлечение параметров из структур ядра и выполнение тестов с ними. С помощью стрелок указывается

последовательность выполнения, а также направление передачи всех вычисленных переменных, которые могут поступать на вход функциям проверки. Стрелки с пунктирной линией обозначают передачу найденных наборов смещений с одного узла на другой. Ромбы представляют собой операцию конъюнкции между разными ветвями тестов.

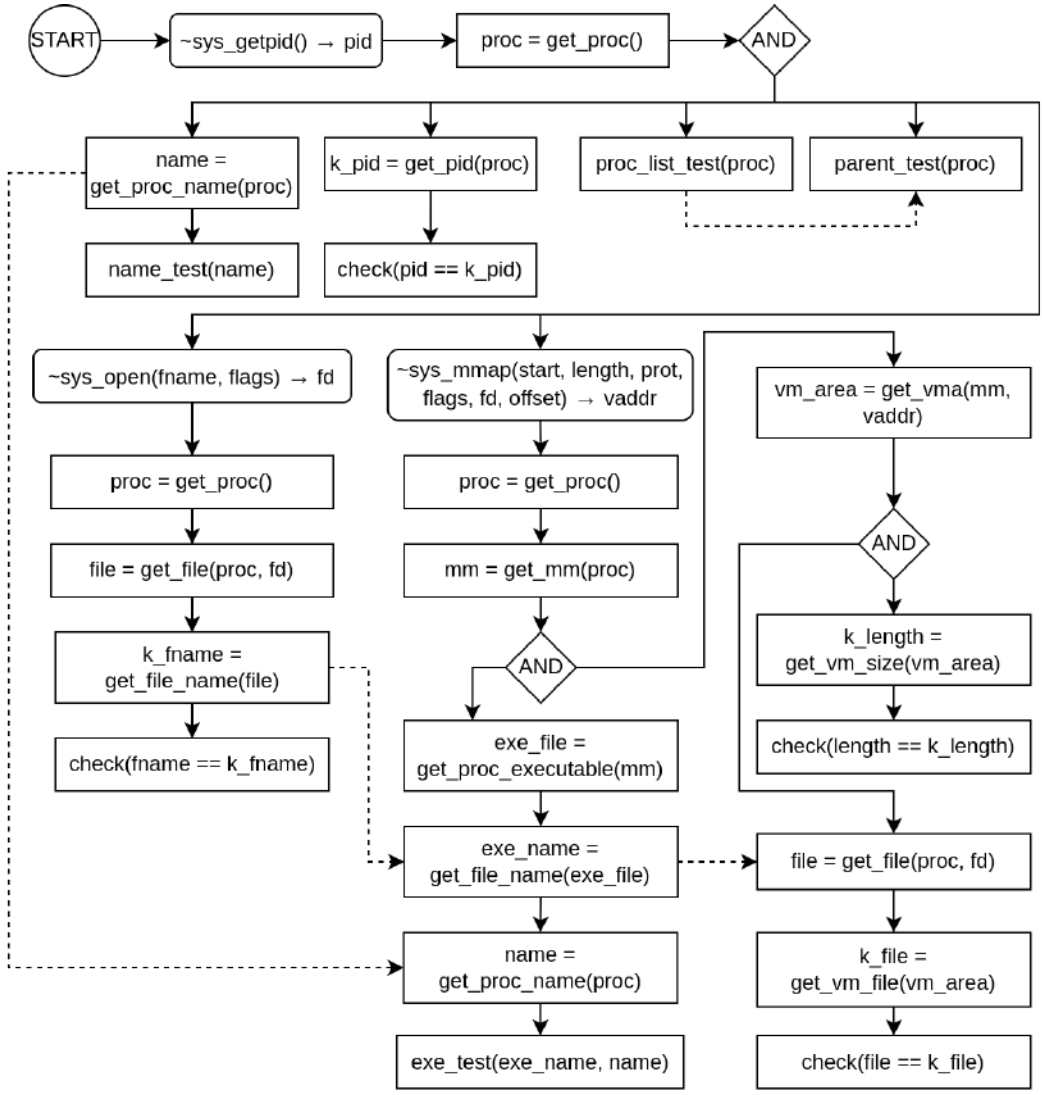


Рис. 4. Схема проверок для генерации профиля интроспекции.
Fig. 4. Scheme of checks for generating an introspection profile.

Тесты разделяются на разные ветви в зависимости от того, какие наборы смещений они проверяют и какие переменные они принимают на вход. Если тесты не зависят от проверяемых смещений друг друга, то их следует располагать на разных ветвях. За счет этого общее количество переборов смещений может значительно сокращаться.

Связанные между собой тесты и переборы проверяемых ими значений выполняются в порядке обхода дерева в глубину. При этом дочерние ветви узлов с переборами обрабатываются повторно на каждое выдаваемое значение. При прохождении тестов, расположенных на листовых узлах дерева, определяются конечные наборы смещений

параметров. В случае корректно составленных тестов такой набор смещений должен быть только один. В обратном же случае необходимы дополнительные тесты, которые уточнят местоположение искомых параметров. Для этой задачи был реализован механизм, позволяющий перед любым тестом в графе выполнить перебор наборов смещений с какого-либо другого узла. Таким образом, на более поздних этапах генерации профиля удается дополнительно протестировать ранее найденные параметры и отсечь некорректные варианты.

Приведем пример. В начале настройки наш алгоритм может найти несколько возможных вариантов смещения для параметра имени процесса. На этом этапе он проверяется исключительно по области допустимых значений и служит подтверждением того, что структура данных, в которой он находится, действительно является дескриптором процесса. Позже, когда становятся известны смещения в файловых структурах, выполняется подбор параметра исполняемого файла процесса. В ходе проверки все возможные варианты имени процесса сопоставляются с вариантами полного пути к исполняемому файлу. Как итог, при их совпадении получается подтвердить местоположение обоих искомых параметров.

Для повышения надежности подбора смещений также используется сбор статистики. Если проверки иногда дают ложно положительный результат, но работают корректно в большинстве случаев, то достаточно выполнить их несколько раз и выбрать тот набор смещений, который будет встречаться наиболее часто.

Говоря о подборе смещений полей, следует отметить, что в большинстве случаев их значения достаточно перебирать в рамках диапазона максимального размера структуры данных с определенным шагом, обычно равным размеру параметра. Когда вариантов немного, их бывает удобнее перебирать по некоторому списку возможных значений. Также есть параметры, смещения которых следует вычислять по формуле вместо каких-либо переборов. В качестве примера последних можно привести поле состояния процесса. Для его нахождения выполняется перехват системного вызова exit. Далее отслеживаются операции записи в область памяти дескриптора процесса. Выполняемый тест проверяет, что записываемое такой операцией значение представляет собой флаг смерти процесса, и в случае успеха вычисляет из адреса записи смещение искомого поля.

Для ускорения подбора смещений и построения профиля интроспекции в целом реализуется ряд оптимизаций. Они затрагивают механизмы обхода графа и чтения памяти, но не сами реализации проверок. Опишем некоторые из них.

Во время переборов смещений тесты могут обращаться множество раз к одним и тем же адресам гостевой памяти на одном и том же состоянии виртуальной машины. В связи с этим в рамках каждой остановки эмулятора выполняется кэширование ранее прочитанных из памяти значений, за счет чего параметры читаются из памяти значительно быстрее.

При поиске в структуре данных указателя на другую структуру может возникать ситуация, когда попадается несколько указателей с одним и тем же значением. Это может приводить к тому, что ряд последующих тестов, рассчитанных на поиск и проверку полей той другой структуры, будет повторно выполняться на одних и тех же данных, что потенциально может приводить к большому количеству лишних вычислений. В связи с этим алгоритм перебора отслеживает такие ситуации и выполняет последующие тесты только один раз. Для таких случаев также реализуются дополнительные проверки, которые определяют какой из дублирующихся указателей является истинным искомым параметром.

В тестах для извлечения параметров из гостевой памяти применяется тот же интерфейс с абстрактными дескрипторами объектов, что используются и для задач мониторинга. Это приводит к тому, что вызываемые функции извлечения параметров могут читать сразу несколько разных полей структур данных за раз. В ряде случаев при чтении какого-то определенного поля структуры бывает очевидно, что его смещение некорректно. Например, указатель на структуру может хранить неправильный адрес памяти. Тогда оказывается, что в

переборе значений последующих смещений полей нет смысла, ведь при всех их возможных значениях будет возникать одна и та же ошибка. В качестве решения проблемы для попыток извлечения параметров была реализована обработка ошибок, в ходе которой определяется какое конкретно поле структуры данных некорректно. Смещение этого поля при этом переключается на следующее его возможное значение, пропуская переборы других смещений. Во множестве случаев такая оптимизация сильно сокращает общее количество перебираемых комбинаций значений.

4. Тестирование

Проверка работоспособности реализации предлагаемого подхода интроспекции виртуальной машины проводилась с различными гостевыми ОС. В качестве таких систем использовались дистрибутивы Linux (Ubuntu, Debian, Fedora, Centos и др.) с версиями ядра от 2.4 до 6.8, Windows с версиями ядра NT от 6.1 до 10.0 и FreeBSD с версией 13.1. Тестирование во всех случаях выполнялось с эмуляцией архитектуры x86_64. Интроспекция Linux также проверена для платформы AArch64.

Результаты построения профиля интроспекции сравнивались с наборами смещений, извлекаемыми из отладочных символов. Эти наборы смещений сильно варьируются в зависимости от версии и опций компиляции ядра, и тестирование показало полное соответствие найденных значений эталонным в каждом случае. Корректность кэширования данных тестировалась на различных состояниях виртуальной машины путем автоматического сопоставления метаданных объектов в кэше с данными, читаемыми из гостевой памяти. Выполняемые в ходе тестов сценарии включали в себя запуск на гостевой ОС программ, работающих с файлами и сетью. Лог файлы, получаемые путем мониторинга объектов, проверялись вручную с использованием инструмента визуализации этих данных.

В ходе измерения производительности было установлено, что совместная работа плагинов мониторинга в среднем замедляет воспроизведение записанного на эмуляторе QEMU сценария примерно в 2,5 раза. Применение точек наблюдения за примитивами синхронизации и переменными состояния оказывает замедление на систему в пределах 5% по сравнению с подходом, где обновление кэша выполняется только по системным вызовам. При этом точки наблюдения позволили повысить точность мониторинга, т.к. с их помощью успешно отслеживаются изменения в объектах ОС, которые происходят в обход системных вызовов.

Время генерации профиля интроспекции зависит от скорости загрузки гостевой ОС на эмуляторе. В большинстве случаев все параметры определяются до момента начала авторизации пользователя. В случае некоторых особенно легковесных дистрибутивов Linux системных событий во время загрузки может не хватать на восстановление всех искомых смещений. С такими системами требуются дополнительные действия со стороны пользователя, такие как ввод логина и пароля, запуск каких-либо программ или перезагрузка ОС. Как правило, весь процесс генерации профиля для любой гостевой ОС занимает всего несколько минут.

5. Заключение

По итогам данной работы был разработан и проверен ряд новых решений для задачи интроспекции виртуальной машины. Представлен подход построения профиля интроспекции, реализованный на базе полносистемного динамического анализа. Он работает с гостевыми ОС семейств Linux, Windows и FreeBSD на архитектурах x86 и ARM. Для его функционирования не требуются отладочные символы ядра и какие-либо подготовительные работы с образом виртуальной машины со стороны пользователей.

Разработанные алгоритмы автоматической настройки также упрощают расширение функционала мониторинга за объектами гостевой системы. Реализация новых возможностей

осуществляется через добавление функции чтения нового параметра из гостевой памяти и набора тестов для его проверки.

Кэширование метаданных объектов ОС сыграло важную роль в реализации задач мониторинга, а точки наблюдения позволили более корректно и быстро отслеживать изменения в структурах данных ядра, после чего своевременно обновлять кэш.

Список литературы / References

[1]. Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. 2014. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14). Association for Computing Machinery, New York, NY, USA, 386–395. DOI: 10.1145/2664243.2664252.

[2]. Jennia Hizver and Tzi-cker Chiueh. 2014. Real-time deep virtual machine introspection and its applications. In Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '14). Association for Computing Machinery, New York, NY, USA, 3–14. DOI: 10.1145/2576195.2576196.

[3]. TEMU: The BitBlaze dynamic analysis component. Available at: http://bitblaze.cs.berkeley.edu/temu.html, accessed 07.10.2025.

[4]. Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014). Association for Computing Machinery, New York, NY, USA, 248–258. DOI: 10.1145/2610384.2610407.

[5]. B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, R. Whelan. Repeatable Reverse Engineering with PANDA. 5th Program Protection and Reverse Engineering Workshop, Los Angeles, California, December 2015.

[6]. Richard Golden, Andrew Case, and Lodovico Marziale. 2010. Dynamic Recreation of Kernel Data Structures for Live Forensics. Digital Investigation 7(2010), pp. 32–40.

[7]. Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing system-wide information flow for malware detection and analysis. In Proceedings of the 14th ACM Conference on Computer & Communications Security. pp. 116–127.

[8]. Shuhui Zhang, Xiangxu Meng, and Lianhai Wang. 2016. An adaptive approach for Linux memory analysis based on kernel code reconstruction. EURASIP Journal on Information Security 2016, 1 (2016), p. 14.

[9]. Fellicious, Christofer & Reiser, Hans & Granitzer, Michael. (2025). Bridging the Semantic Gap in Virtual Machine Introspection and Forensic Memory Analysis. Available at: 10.48550/arXiv.2503.05482, accessed 07.10.2025.

[10]. Fabian Franzen, Tobias Holl, Manuel Andreas, Julian Kirsch, and Jens Grossklags. 2022. Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots. In 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus. ACM, New York, NY, USA, 18 p.

[11]. Qian Feng, Aravind Prakash, Minghua Wang, Curtis Carmony, and Heng Yin. 2016. Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. pp. 11–22.

[12]. Qemu. A generic and open source machine emulator and virtualizer. Available at: https://www.qemu.org/, accessed 07.10.2025.

[13]. Shuhui Zhang, Xiangxu Meng, and Lianhai Wang. 2016. An adaptive approach for Linux memory analysis based on kernel code reconstruction. EURASIP Journal on Information Security 2016, 1 (2016), 14.

Информация об авторах / Information about authors

Владислав Михайлович СТЕПАНОВ – разработчик программного обеспечения. Сфера научных интересов: отладка, интроспекция и инструментирование виртуальных машин, динамический анализ бинарного кода, эмуляторы.

Vladislav Mikhailovich STEPANOV is a software developer. Research interests: debugging, introspection and instrumentation of virtual machines, dynamic analysis of binary code, emulators.

Павел Михайлович ДОВГАЛЮК – кандидат технических наук, инженер. Сфера научных интересов: интроспекция и инструментирование виртуальных машин, динамический анализ кода, отладчики, эмуляторы.

Pavel Mikhailovich DOVGALYUK – Cand. Sci. (Tech.), engineer. Research interests: virtual machines introspection and instrumentation, dynamic analysis of code, debuggers, emulators.

Наталья Игоревна ФУРСОВА – кандидат технических наук, инженер. Сфера научных интересов: интроспекция и инструментирование виртуальных машин, динамический анализ кода, эмуляторы.

Natalia Igorevna FURSOVA – Cand. Sci. (Tech.), engineer. Research interests: virtual machines introspection and instrumentation, dynamic analysis of code, emulators.

DOI: 10.15514/ISPRAS-2025-37(6)-37

CC BY

Перенос обучения в сетевых системах обнаружения вторжений: обзор методов и подходов

[1] *А.Ю. Покидько, ORCID: 0009-0008-8981-8429 <a.pokidko@ispras.ru>*
[1,2] *И.А. Степанов, ORCID: 0009-0003-1964-5001 <ivan_mipt@ispras.ru>*
[1,2,3,4] *А.И. Гетьман, ORCID: 0000-0002-6562-9008 <ever@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*

[2] *Московский физико-технический институт, 141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9.*

[3] *Национальный исследовательский университет «Высшая школа экономики», 101978, Россия, г. Москва, ул. Мясницкая, д. 20.*

[4] *Московский государственный университет имени М.В. Ломоносова, Россия, 119991, Москва, Ленинские горы, д. 1.*

Аннотация. Статья представляет обзор современных методов переноса обучения (transfer learning) в сетевых системах обнаружения вторжений (СОВ), ориентируясь на проблему устойчивости моделей в условиях дрейфа сетевых данных, изменчивости трафика и появления новых типов атак. Рассматриваются основные парадигмы переноса – параметрический, признаковый и основанный на отношениях – и их адаптация к задаче обнаружения аномалий и классификации сетевых вторжений. Особое внимание уделено различиям между методами на основе анализа статистических свойств сетевых потоков и методами на основе анализа пакетов. На основе анализа существующих работ демонстрируется, что использование переноса обучения позволяет существенно повысить устойчивость сетевых СОВ к изменениям инфраструктуры и распределений данных, однако сталкивается с проблемами негативного переноса, недостатка репрезентативных источников домена и усложнения архитектур. В завершение формулируются ключевые направления дальнейших исследований, включая адаптивные модели с учётом дрейфа, перенос в условиях ограниченных данных и интеграцию с потоковыми методами машинного обучения.

Ключевые слова: сетевая система обнаружения вторжений (СОВ); перенос обучения.

Для цитирования: Покидько А.Ю., Степанов И.А., Гетьман А.И. Перенос обучения в сетевых системах обнаружения вторжений: обзор методов и подходов. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 73–90. DOI: 10.15514/ISPRAS–2025–37(6)–37.

Благодарности. Результаты получены с использованием услуг Центра коллективного пользования Института системного программирования им. В.П. Иванникова РАН – ЦКП ИСП РАН.

Transfer Learning in Network Intrusion Detection Systems: a Review of Methods and Approaches

[1] *A.Y. Pokidko, ORCID: 0009-0008-8981-8429 <a.pokidko@ispras.ru>*
[1,2] *I.A. Stepanov, ORCID: 0009-0003-1964-5001 <ivan_mipt@ispras.ru>*
[1,2,3,4] *A.I. Getman, ORCID: 0000-0002-6562-9008 <ever@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *Moscow Institute of Physics and Technology (National Research University), 9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.*

[3] *National Research University «Higher School of Economics», 20, Myasnitskaya ulitsa, Moscow, 101000, Russia.*

[4] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia.*

Abstract. This article provides an overview of modern transfer learning methods in network intrusion detection systems (IDS), focusing on the problem of model stability in conditions of network data drift, traffic variability, and the emergence of new types of attacks. The main transfer paradigms – parametric, feature-based, and relationship-based – and their adaptation to the task of anomaly detection and network intrusion classification are considered. Particular attention is paid to the differences between methods based on the analysis of statistical properties of network flows and methods based on packet analysis. Based on an analysis of existing work, it is demonstrated that the use of transfer learning can significantly improve the robustness of network IDSs to changes in infrastructure and data distributions, but faces problems of negative transfer, lack of representative domain sources, and architectural complexity. Finally, key directions for further research are formulated, including adaptive models that account for drift, transfer under limited data conditions, and integration with streaming machine learning methods.

Keywords: network intrusion detection system (NIDS); transfer learning.

For citation: Pokidko A.Y., Stepanov I.A., Getman A.I. Transfer learning in network intrusion detection systems: a review of methods and approaches. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 3, 2025, pp. 73-90 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-37.

Acknowledgements. The results were obtained using the services of the Ivannikov Institute for System Programming (ISP RAS) Data Center.

1. Введение

С ростом объемов и сложности сетевого трафика в современных информационных системах возрастает необходимость в эффективных методах его анализа. Такие задачи, как обнаружение аномалий, классификация трафика, выявление вредоносной активности сетевых атак, требуют высокой точности и адаптивности используемых сетевых систем обнаружения вторжений (СОВ, IDS).

Существующие сетевые СОВ делятся на 3 типа в зависимости от метода обнаружения атак [1]:

1. Обнаружение на основе сигнатур: метод, отслеживающий пакеты в сети и сравнивающий их с предварительно настроенными и заранее определенными шаблонами атак, известными как сигнатуры. Такие СОВ отличаются высокой точностью обнаружения известных атак, но требуют постоянного ручного обновления базы сигнатур.

2. Анализ протокола с сохранением состояния (Stateful protocol analysis, SPA): IDS может знать и отслеживать состояния протокола (например, связывая запросы с ответами). Как правило, модели сетевых протоколов в SPA изначально основаны на

стандартах протоколов международных организаций по стандартизации, например, IETF. Такие СОВ способны обнаруживать новые и неизвестные атаки (если они нарушают спецификацию), но имеют высокую ресурсоёмкость и могут не распознать атаку, которая формально корректна по спецификации, но содержит вредоносную полезную нагрузку (например, SQL-инъекция внутри корректного HTTP-запроса).

3. Статистическое обнаружение аномалий: сетевая СОВ, основанная на аномалиях, будет отслеживать сетевой трафик и сравнивать его характеристики с установленным базовым уровнем. Такие системы способны обнаруживать неизвестные или новые типы атак, но часто дают много ложных срабатываний.

В этой статье будут рассмотрены статистические сетевые СОВ на основе нейронных сетей, так как они позволяют извлекать как простые статистические взаимосвязи, так и сложные нелинейные зависимости из данных. Однако обычно такие подходы зависят от большого количества размеченных данных и плохо переносятся на новые сетевые условия, где данные либо ограничены, либо имеют иную структуру.

Перенос обучения (Transfer learning или TL) предлагает эффективное решение этой проблемы, позволяя использовать знания, полученные в одной задаче или домене, для ускорения обучения и повышения точности в другой. Применение методов переноса обучения в анализе сетевого трафика открывает новые перспективы для создания универсальных и устойчивых систем сетевой безопасности, способных адаптироваться к изменяющимся условиям и различным типам сетей.

В данной статье рассматриваются ключевые подходы переноса обучения в контексте анализа сетевого трафика. Анализируются существующие исследования, обсуждаются преимущества и ограничения методов, а также направления для дальнейших исследований и применения на практике. Таким образом цель работы – провести систематический обзор методов переноса обучения, применимых к системам обнаружения вторжений и заложить основу для будущих исследований в этой области. В отличие от существующих обзоров, посвящённых либо общим методам переноса обучения в кибербезопасности, либо применению глубокого обучения в NIDS, данная работа систематизирует методы переноса обычения именно для сетевых систем обнаружения вторжений, предлагая сквозную классификацию по типу переносимых знаний и характеру сетевых данных (потоки или пакеты), а также оценивая применимость каждого подхода в практических сценариях

Остальная часть работы структурирована следующим образом. В разделе 2 содержатся основные определения и классификация методов переноса обучения. В разделе 3 приводится анализ существующих работ, относящихся к различным типам переноса обучения. В заключении приводятся выводы и возможные направления для будущих исследований.

2. Обзор

В этом разделе будут даны определения переноса обучения и сопутствующих терминов, а также представлена классификация методов переноса обучения.

2.1 Основные определения

В переносе обучения изученные знания передаются из исходной области (исходного домена) в целевую для улучшения процесса обучения целевой задачи. Таким образом, прежде всего будут даны определения «домена» и «задачи» [2].

Определение 1 (Домен) Доменом D является пара $D = \{\mathcal{X}, P(X)\}$, где $\mathcal{X}$ – признаковое пространство, $P(X)$ – маргинальное (или частное распределение), а X – множество объектов, т.е. $X = \{x \mid x_i \in \mathcal{X}, i = 1, \dots, n\}$.

Определение 2 (Задача) В домене D задача T состоит из двух частей: 1) пространства меток Y и 2) функции принятия решений $f(\cdot)$. Функция принятия решений обучается на основе пар векторов признаков и пространства меток, т.е. $\{x_i, y_i\}$, где $x_i \in X$ и $y_i \in Y$. Другими словами, задача определяется как $T = \{Y, f(\cdot)\}$.

В общем случае, функция принятия решений представляет собой предсказание соответствующей метки $f(x_i)$ для экземпляра x_i. В этом случае предсказательная функция может быть определена как $f(x_i) = \{P(y_k|x_i)|\ y_k \in Y, k = 1, \dots, |Y|\}$.

Определение 3 (Перенос обучения) Даны исходный домен D_S и задача T_S, а также целевой домен D_T и задача T_T. Перенос обучения направлен на улучшение качества обучения целевой предсказательной функции $f(\cdot)$ в области D_T за счет использования знаний, полученных в D_S и T_S, при условии, что $D_S \neq D_T$ или $T_S \neq T_T$.

2.2 Классификация

Перенос обучения представляет собой набор методов, позволяющих использовать знания, извлечённые из одной задачи или домена, для улучшения обучения в другой задаче. В отличие от традиционного машинного обучения, где обучающие и тестовые данные предполагаются независимыми и идентично распределёнными, TL допускает смещение распределений и различия между источником и целевой задачей [3]. Для достижения переноса обучения используется множество техник в зависимости от доменов и задач исходного и целевого доменов, их различий и способа переноса обучения. Таким образом, существует несколько вариантов классификации методов переноса [2].

2.2.1 Классификация по типу переносимых знаний

Первый и наиболее фундаментальный способ классификации методов переноса обучения – по типу переносимых знаний.

Перенос признаков (feature-based TL) предполагает, что перенос осуществляется через построение нового признакового пространства, в котором различия между исходным и целевым доменами минимизируются. Так, например, в задачах распознавания сигналов можно отобразить данные из разных частотных диапазонов в общее пространство признаков, что упрощает классификацию. Его можно разделить на две подкатегории: симметричный подход и асимметричный. В случае симметричного происходит преобразование признаков обеих областей в общее скрытое пространство. В ассиметричном напротив, преобразование только исходных признаков так, чтобы они соответствовали целевым.

В случае **переноса параметров** (parameter-based TL) перенос знаний осуществляется на уровне параметров или гиперпараметров моделей. Например, использование моделей, предобученных на ImageNet, для задач компьютерного зрения в смежных областях. В задачах анализа сетевого трафика это может означать перенос параметров модели, обученной на сетевом трафике одной организации, для анализа сетевого трафика другой.

Для **переноса отношений** (relation-based TL) перенос знаний осуществляется через использование связей и отношений между объектами. В отличие от предыдущих подходов, здесь внимание уделяется не только самим данным, но и структуре их взаимосвязей. Например, в социальных сетях это может быть перенос знаний о связях между пользователями; в беспроводных сетях – использование корреляций между узлами при решении задач маршрутизации или управления ресурсами.

Перенос экземпляров (instance-based TL) – знания переносятся на уровне исходных примеров (instances). Вместо прямого использования всех данных из исходного домена, методы этого типа направлены на перераспределение весов исходных образцов данных, чтобы сократить различия между маргинальными распределениями источника и цели.

2.2.2 Классификация в зависимости от соотношения доменов и задач

Вторым из наиболее распространённых критериев классификации переноса обучения является характер взаимосвязи между исходной и целевой задачами, а также наличие или отсутствие разметки в целевой области. На этой основе выделяют три ключевых типа переноса обучения: индуктивный, трансдуктивный и без учителя [3].

Определение 4 (Индуктивный перенос обучения, Inductive Transfer Learning). Пусть даны исходный домен D_S с соответствующей исходной задачей T_S и целевой домен D_T с соответствующей целевой задачей T_T. Индуктивный перенос обучения направлен на улучшение обучения целевой предсказательной функции $f_T(\cdot)$ целевого домена D_T на основе знаний, полученных в D_S и T_S, в которых исходная и целевая задача различаются, то есть $T_S \neq T_T$.

Индуктивный перенос обучения делится на два вида: самообучение (Self-taught learning) и многозадачное обучение (Multitask learning). Самообучение используется, когда в исходной области нет размеченных данных. Используются неразмеченные данные источника, чтобы построить более высокоуровневое представление признаков (снижается размерность пространства), а затем классификация проводится с использованием размеченных данных целевой области. Многозадачное обучение – когда и в исходной, и в целевой области есть размеченные данные. Обе задачи обучаются одновременно, что позволяет улучшать результаты друг друга.

В индуктивном переносе обучения знания могут быть перенесены через перенос признаков, перенос параметров, перенос отношений или перенос экземпляров.

Перенос признаков создаёт новое пространство X_{new}признаков для перевода объектов исходного и целевого домена в X_{new} . Перенос параметров предполагает, что отдельные модели для связных задач имеют общие параметры или распределение гиперпараметров. Подход на основе переноса отношений не предполагает, что данные, из каждого домена, являются независимыми. Наконец, перенос экземпляров позволяет переносить часть данных из исходного домена в целевой домен.

Определение 5 (Трансдуктивный перенос обучения, Transductive Transfer Learning) Пусть даны исходный домен D_S с соответствующей исходной задачей T_S и целевой домен D_T с соответствующей целевой задачей T_T. Трансдуктивный перенос обучения направлен на улучшение обучения целевой предсказательной функции $f_T(\cdot)$ целевого домена D_T на основе знаний, полученных в D_S и T_S, где исходный домен и целевой домен различны, т. е. $D_S \neq D_T$, а исходная и целевая задача одинаковы, т. е. $T_S = T_T$.

В данном сценарии в исходном домене имеется большой объем размеченных данных, в то время как в целевом домене метки отсутствуют вовсе. Ключевое различие доменов при идентичности задач обычно обусловлено либо несовпадением пространств признаков (например, разные языки в текстовых задачах), либо различием в маргинальных распределениях вероятностей входных данных, когда $P(X_S) \neq P(X_T)$. Данный подход широко известен как адаптация домена (Domain Adaptation). В трансдуктивном переносе основное внимание уделяется минимизации расхождения между доменами. Это достигается либо через взвешивание экземпляров исходного домена (Instance-based), чтобы они больше соответствовали распределению целевого, либо через поиск инвариантного представления признаков (Feature-based), которое одинаково эффективно описывает данные обоих доменов, несмотря на их исходные различия

Определение 6 (Перенос обучения без учителя, Unsupervised Transfer Learning) Пусть даны исходный домен D_S с соответствующей исходной задачей T_S и целевой домен D_T с соответствующей целевой задачей T_T. Перенос обучения без учителя направлен на улучшение обучения целевой предсказательной функции $f_T(\cdot)$ целевого домена D_T на основе знаний, полученных в D_S и T_S, в которых исходная и целевая задача различаются, т. е. $T_S \neq T_T$, а размеченные данные Y_S и Y_T не поддаются наблюдению.

В переносе обучения без учителя знания могут быть перенесены только через перенос признаков. В этом случае переносимыми знаниями являются структурные и представительные характеристики данных. Модель, выявляющая скрытые закономерности или латентные структуры в одном наборе данных, может быть использована для анализа другого набора данных без явных меток. Такой тип переноса полезен для кластеризации, снижения размерности или предварительного обучения представлений в целевом домене.

Сравнение методов переноса с точки зрения переносимых знаний представлено в табл. 1.

Табл. 1. Сравнение методов переноса знаний.
Table 1. Comparison of knowledge transfer methods.

	Перенос признаков	Перенос параметров	Перенос отношений	Перенос экземпляров
Индуктивный перенос обучения	+	+	+	+
Трансдуктивный перенос обучения	+	-	-	+
Перенос обучения без учителя	+	-	-	-

Таким образом наиболее гибким является индуктивным перенос обучения, поддерживающий перенос любых типов знаний. Менее гибким является трансдуктивный перенос, который не поддерживает перенос параметров и перенос отношений.

2.2.3 Стратегии переноса в глубоком обучении

Перенос в глубоком обучении или глубокий перенос обучения (англ. Deep Transfer Learning, DTL) – рассматривает перенос знаний для глубоких нейронных сетей, обученных на одном домене или задаче, в другой домен/задачу. В отличие от классического TL, здесь используются предобученные глубокие модели.

Разделяют 4 основных типа [4]:

- Готовые предварительно обученные модели (Off-the-shelf pretrained models);
- Извлечение признаков с помощью предобученной модели (Feature extraction);
- Дообучение (Fine-tuning);
- Гибридные методы.

Стратегия готовых предварительно обученных моделей представляет из себя прямое использование предварительно обученной модели для целевой задачи, без модификаций. Этот метод эффективен, если источник и целевая задача совпадают или очень близки, т.е. $D_S \cong D_T$ и $T_S \cong T_T$.

При использовании предобученной модели для извлечения признаков обученная модель используется для получения более абстрактных и обобщенных представлений признаков. Этот метод эффективен при ограниченных данных в целевом домене и снижает вычислительные затраты при обучении.

Глубокое обучение можно рассматривать как иерархическое обучение. В частности, отдельный слой DNN обучается различным признакам, от общих, т. е. низкого уровня, до более конкретных, т. е. высокого уровня, по мере углубления в DNN. Было доказано, что

признаки, обученные DL, более переносимы, что означает, что их легче повторно использовать в похожих доменах [5].

Дообучение представляет собой стратегию передачи знаний, при которой предварительно обученная нейронная сеть дообучается под конкретную целевую задачу. Этот подход заключается в том, что веса предварительно обученной модели используются как начальная точка для обучения новой модели, что значительно сокращает время и ресурсы, необходимые для достижения высокой точности. У данной стратегии есть 2 основных подхода: инициализация весов и выборочное дообучение. При инициализации весов веса предварительно обученной модели используются как начальные значения для целевой модели, а затем происходит обучение модели на целевых данных. В выборочном дообучении предполагается дообучение только части модели, оставляя другие слои замороженными. Это особенно эффективно при ограниченных данных.

3. Применимость к задаче обнаружения вторжений

Одной из ключевых проблем построения сетевых систем обнаружения вторжений, основанных на методах машинного обучения, является ограниченная переносимость обученных моделей. Наборы данных для обучения классификаторов сетевого трафика часто создаются в строго определённой среде (например, в конкретной корпоративной сети, в лабораторных условиях или с использованием специализированных бенчмарков, таких как CICIDS или UNSW-NB15 [6]). Однако в реальных сценариях эксплуатации трафик подвержен постоянным изменениям: появляются новые протоколы, меняется поведение приложений, возникают ранее неизвестные атаки. Также возникает проблема при использовании пред обученных моделей так как они обучались в иной среде. В этой ситуации возникает проблема смещения распределений, которая резко снижает эффективность детекторов, обученных в одной среде и перенесённых в другую [7].

Методы переноса обучения позволяют существенно смягчить эту проблему, обеспечивая адаптацию моделей к новым условиям без необходимости полного переобучения «с нуля». Их применимость в контексте СОВ можно рассмотреть через призму используемых представлений входных данных, так как именно они во многом определяют эффективность и тип используемого переноса знаний.

3.1 Статистические признаки потоков

В традиционных системах сетевого мониторинга часто используются агрегированные статистические признаки потоков (flow-based features), такие как количество пакетов, средний размер сегмента, длительность соединения, дисперсия межпакетных интервалов и др. Данные признаки относительно компактны, легко интерпретируемы и позволяют применять широкий спектр классических алгоритмов машинного обучения. Однако такие признаки подвержены высокой зависимости от среды: одни и те же типы трафика могут иметь существенно разные статистические характеристики в разных сетях.

Здесь перенос обучения на основе переноса экземпляров (например, методы повторного взвешивания образцов или фильтрации нерелевантных потоков) позволяет адаптировать распределения признаков источника к целевой сети. Кроме того, перенос обучения на основе переноса признаков (например, доменная адаптация через автоэнкодеры) помогает проектировать более устойчивое представление потоков, снижая чувствительность к конкретным условиям трафика.

Примером переноса обучения на основе переноса экземпляров является работа [8], предложившая перенос обучения на основе перераспределения весов образцов между доменами. Авторы рассматривают проблему, типичную для NIDS на основе анализа статистических признаков потоков: статистические параметры потоков в целевой сети могут

существенно отличаться от тех, что использовались при обучении модели, т.е. домены различны. Для решения этой проблемы предлагается **перенос обучения на основе экземпляров**, в котором каждому образцу исходного домена назначается параметр-вес, определяющий его релевантность целевой задаче. В основе метода лежит идея о том, что часть данных источника близка к данным целевой сети, а часть – вводит шум и снижает точность. Авторы не рассматривают случаи появления новых типов трафика, т.е. $T_S \neq T_T$. Следовательно, рассматривается **трансдуктивный перенос обучения**.

Для вычисления оптимальных весов авторы применяют критерий максимального среднего расхождения (MMD), который минимизирует расстояние между распределениями признаков источника и цели. Далее классификатор обучается с учётом сгенерированных весов: релевантные примеры усиливают влияние при обучении, а нерелевантные – подавляются. Такой подход позволяет адаптировать модель без изменения её архитектуры, используя только перераспределение важности отдельных потоков. Ключевым преимуществом является высокая совместимость с классическими методами анализа данных со статистическими признаками сетевых потоков (градиентный бустинг, SVM, нейросети малой глубины). Итоги экспериментов показали повышение устойчивости к сетевым сдвигам, особенно при переносе между наборами данных с разной плотностью трафика. Однако точность сильно зависит от качества оценки расстояния между доменами: при высокой неоднородности потоков в целевой сети подбор весов становится менее стабильным.

В работе [9] был предложен гибридный подход, сочетающий генеративно-состязательную сеть (GAN), многоядерное максимальное среднее расхождение (MK-MMD) и оптимизацию признаков для выявления аномалий в сетевом трафике. На первом этапе выполняется оптимизация признаков с помощью ансамбля агентов на основе обучения с подкреплением (Collaborative Learning Automata) которые взаимодействуют с классификатором как со стохастической средой и итеративно корректируют вероятности выбора отдельных признаков в зависимости от полученной точности классификации. Такая совместная адаптация нескольких автоматов позволяет исключить коррелированные и избыточные признаки, сократить размерность пространства и улучшить информативность входных данных. Полученное оптимальное подмножество (около 9 признаков) используется на втором этапе, где применяется генеративно-состязательная сеть (GAN). В её структуре генератор создаёт синтетические примеры сетевого трафика на основе случайного шума и условных меток, а дискриминатор учится одновременно различать реальные и сгенерированные образцы и классифицировать их по типу атаки. Таким образом, GAN выполняет роль **переноса обучения без учителя** (semi-supervised transfer-learning), расширяя выборку и повышая устойчивость модели при ограниченном количестве размеченных данных. Для дальнейшего повышения обобщающей способности в архитектуру внедрён модуль многоядерного варианта меры максимального среднеквадратичного расхождения (multiple kernel variant of maximum mean discrepancy, MK-MMD) [10], минимизирующий различия между распределениями исходной (source) и целевой (target) областей в пространстве признаков; тем самым реализуя доменную адаптацию – перенос знаний о нормальном и атакующем трафике между различными сетевыми средами. Обучение всех компонентов проводится совместно: формирование оптимизированного вектора признаков, генерация дополнительных образцов, выравнивание их распределений с помощью MK-MMD и затем классификатор на выходе (16-мерный Softmax) выдаёт окончательные вероятности классов. Итоговая система продемонстрировала высокую точность (около 91,7 % для бинарной и 91,5 % для многоклассовой классификации), низкий уровень ложных срабатываний и устойчивость при переносе между доменами, объединяя в едином конвейере принципы обучения с подкреплением, генеративного моделирования и адаптивного переноса признаков.

Таким образом сильной стороной предложенного авторами метода является эффективная компенсация недостатка данных об аномалиях и общее улучшение устойчивости

классификатора к сдвигам во входных данных. Однако генерация синтетических признаков потоков может не полностью отражать реальные зависимости между признаками при эксплуатации данного метода в реальных условиях.

К аналогичному направлению относится работа [11], однако здесь авторами предложен метод обнаружения неизвестных сетевых атак на основе **трансдуктивного переноса признаков**, направленный на уменьшение зависимости от конкретной среды и набора данных. Авторы используют набор NSL-KDD, содержащий 41 признак, и формируют два домена: исходный, включающий известные атаки, и целевой, содержащий другие типы атак, которые моделируют «неизвестные» угрозы. Основная идея работы заключается в создании общего латентного пространства, в котором данные из исходного и целевого доменов становятся сопоставимыми. Первым вариантом метода является HeTL [12], который может находить общее скрытое подпространство двух разных атак и обучаться оптимизированному представлению, которое было инвариантным к изменениям поведения атак. Он находит две матрицы отображения, которые минимизируют искажение исходных данных и одновременно уменьшают расстояние между представлениями двух доменов. Однако HeTL чувствителен к выбору гиперпараметра, определяющего близость доменов, и требует ручной настройки. Чтобы устранить эту проблему, авторы создают улучшенный метод CeHTL, который автоматически определяет соответствие между доменами с помощью кластеризации. Этот подход позволяет избежать ручной настройки параметров, делает метод более устойчивым и позволяет работать даже при разных наборах признаков в доменах.

Экспериментальная часть показывает, что обычные классификаторы (SVM, KNN, деревья решений) плохо справляются с обнаружением новых атак, в то время как HeTL и особенно CeHTL существенно повышают accuracy, F1 и AUC.

В свою очередь, исследование [13] развивает идею **переноса отношений**. Статья предлагает модель, предназначенную для того, чтобы устранить ключевую проблему современных систем обнаружения вторжений на основе машинного обучения: их статичность и неспособность своевременно адаптироваться к новым атакам. Авторы используют два типа данных – признаки потоков и структурированные CTI-отчёты (Cyber Threat Intelligence, Аналитика киберугроз) VirusTotal. Первый набор данных включает почти три миллиона сетевых потоков с большим набором статистических признаков, полученных из корпоративного трафика, а второй состоит более чем из двух тысяч отчётов, содержащих до сотни атрибутов, включая временные метрики, репутацию IP и результаты анализа различными анализаторами. На основе этих данных создаётся гибридная модель обнаружения вторжений, сочетающая метод опорных векторов (SVM) и метод ближайших соседей (K-means): первая отвечает за классификацию известных угроз, вторая выявляет аномалии и формирует класс выбросов (outlier), который запускает обращение к CTI. При обнаружении неопределённого трафика из него извлекается IoC, производится CTI поиск (CTI-lookup), после чего CTI Transfer Model на основе метода ближайших соседей анализирует отчёт, определяет характер IoC и сопоставляет его с исходными сетевыми наблюдениями, автоматически формируя новое обучающее наблюдение. Этот процесс обеспечивает непрерывное дообучение IDS. Использование CTI приводит к росту F1-метрики на 9,29% по сравнению с моделью без CTI, а применение ML внутри CTI-модуля даёт выигрыш 30,92% по сравнению с простыми эвристическими правилами. Особенно значимыми оказываются данные из класса выбросов, поскольку именно по ним модель испытывает наибольшую неопределённость и получает максимальное улучшение после интеграции CTI. В результате DICI демонстрирует способность распознавать сложные динамические атаки, которые ускользают от традиционных IDS, оставаясь при этом вычислительно лёгкой и пригодной для онлайн-обновления.

Такой подход обеспечивает постоянную адаптацию к текущей обстановке в сети, снижая задержку между появлением новой угрозы и её детектированием. В работе реализованы

механизмы контроля качества обновлений и отката к предыдущим версиям модели, что делает систему эксплуатационно надёжной. Вместе с тем, зависимость от полноты и достоверности CTI-источников создаёт риск некорректных обновлений, а интеграция такого подхода в корпоративную инфраструктуру требует значительных ресурсов.

В статье [14] рассматривается построение IDS на основе **параметрического переноса обучения**, при котором из исходного домена в целевой переносится не структура данных и не примеры, а параметры глубокой модели – архитектура и обученные веса CNN-LSTM. Данные в обоих доменах представлены одинаково – как статистические признаки сетевых потоков, однако распределения отличаются: исходный домен использует обучающую и валидационную выборки, а целевой – полностью невидимые тестовые данные.

Такой перенос относится к **индуктивному переносу обучения**: задача классификации одинакова в обоих доменах, но данные не идентичны, поэтому модель использует заранее выученные параметрические представления признаков для работы в новой среде. В исходном домене CNN-LSTM обучается на полном наборе данных и формирует внутренние представления трафика; в целевом домене она применяет эти же веса без дообучения, что позволяет сохранить точность и существенно ускорить обработку при минимальных ресурсах.

Результаты показывают, что параметрический перенос обеспечивает точность выше 98% и повышает скорость инференса в целевом домене, демонстрируя, что заранее обученная глубинная модель способна эффективно работать в условиях ограниченных ресурсов и умеренного сдвига распределений.

В работе [15] авторы используют форму переноса обучения, которая сочетает перенос параметров и перенос представлений. С точки зрения соотношения доменов и задач их подход относится к индуктивному переносу, поскольку исходная и целевая задачи остаются одинаковыми – это многоклассовая классификация сетевого трафика, однако домены различаются, потому что Bot-IoT и TON-IoT формируют разные распределения признаков и содержат отличающиеся варианты атак. Модель, обученная на Bot-IoT, служит источником уже сформированных признаковых представлений, и эти представления переносятся на целевой домен, где присутствуют новые типы поведения тех же классов и иная статистика трафика. Замороженная свёрточная основа фактически переносит знания о структуре сетевого трафика, зафиксированные в весах, тогда как новый классификатор обучается на TON-IoT, адаптируя общее представление к характеристикам целевого домена.

Преимущество такого подхода заключается в том, что он позволяет использовать накопленные знания об общих паттернах сетевого поведения и при этом адаптировать модель к изменениям без переобучения всей сети. Он уменьшает требования к объёму данных и вычислительным ресурсам и компенсирует нехватку размеченных примеров в целевом датасете. Однако эта стратегия уязвима к сильному расхождению доменов: если распределения в Bot-IoT и TON-IoT различаются слишком существенно, перенесённая основа может кодировать устаревшие или нерелевантные признаки, ограничивая качество обновлённой модели. Кроме того, за счёт заморозки основной части сети остаётся риск того, что модель будет хуже адаптироваться к принципиально новым видам аномалий, если они требуют изменения низкоуровневых признаков, а не только обновления классификатора.

3.2 Последовательность пакетов

Современные подходы также предлагают представление сетевых данных в виде «сырых» пакетов, рассматривая первые n байт полезной нагрузки и заголовков в качестве входных данных для нейронных сетей. Такой подход ближе к компьютерному зрению или обработке текста, где модели могут самостоятельно выявлять значимые паттерны без ручной инженерии признаков.

В статье [16] анализируются атаки на автомобильную шину CAN (Controller Area Network, сеть контроллеров), где новые варианты вторжений появляются часто, а данные для обучения минимальны. Авторы используют реальные массивы CAN-трафика и преобразуют каждый кадр в компактное числовое представление из одиннадцати признаков, включая интерпакетный интервал и нормализованные байты полезной нагрузки. Из таких последовательностей формируются временные окна, которые затем переводятся в двумерное пространственно-временное представление, подходящее для сверточной модели с долгой краткосрочной памятью. Такая модель обучается на нормальном трафике и известной DoS-атаке, после чего используется один из методов **индуктивного переноса экземпляров**, а именно одномоментный перенос обучения (one-shot transfer learning), позволяющий дообучить систему на единственном примере нового типа атаки. Такой тип переноса позволяет переносить уже извлечённые закономерности на новую задачу, в которой имеется крайне мало размеченных данных. После обучения на известных вторжениях и дообучения на одной выборке новой атаки модель достигла F1=88,47% при обнаружении новых атак, что демонстрирует потенциал переноса экземпляров в задачах сетевой безопасности. Его достоинства проявляются в резком увеличении точности обнаружения новых атак, снижении количества ложных срабатываний и отсутствии необходимости собирать крупные датасеты для каждого нового класса. Основным недостатком является высокая зависимость результата от качества исходного представления данных и предварительного обучения, а также риск того, что единственный пример новой атаки может оказаться нерепрезентативным, что приведёт к деградации обобщающей способности модели.

Дополнительно возможен перенос из смежных доменов – например, использование архитектур, обученных на задаче классификации протоколов или идентификации приложений, для задачи обнаружения аномалий. В таких сценариях извлечение признаков оказывается особенно полезным: первые слои сети фиксируются как универсальные извлекатели признаков, а финальные слои перенастраиваются на задачу обнаружения атак.

В более поздних работах идея переноса репрезентаций для задач сетевой безопасности получила развитие в контексте интернета вещей (IoT), где ограниченность вычислительных ресурсов и разнообразие сетевых топологий требуют особой гибкости моделей.

В работе [17] авторы строят IDS-фреймворк на основе **индуктивного переноса признаков**, где перенос осуществляется между двумя наборами сетевого трафика, имеющими одинаковое пространство признаков (15 общих параметров), но различное распределение и различное происхождение. В этом подходе предварительно обученная на BoT-IoT[18] модель CNN служит источником сверточных фильтров, а затем эти слои переносятся в целевой домен UNSW-NB15[19], где обучается только классификатор.

В статье подчеркивается, что использование переноса обучения позволяет извлечь устойчивые закономерности сетевого поведения из масштабного BoT-IoT и перенести их на менее репрезентативный UNSW-NB15, где часть атак относится к атакам нулевого для (zero-day). В результате предварительно обученная сверточная база служит универсальным извлекателем признаков, а дообучаемые полносвязные слои адаптируются к целевому распределению. Такой подход обеспечивает высокую точность при обнаружении неизвестных атак, поскольку модель опирается на общие низкоуровневые сетевые паттерны, а не только на специфические сигнатуры.

Достоинством метода является то, что он позволяет компенсировать нехватку данных в целевом домене и значительно улучшает показатели обнаружения новых атак благодаря переносу универсальных признаков, извлечённых из источника с большим количеством экземпляров. Дополнительным преимуществом становится уменьшение переобучения: замороженные сверточные слои создают устойчивое основание для классификатора, что снижает чувствительность к шуму и дисбалансу данных. Недостатком является то, что перенос возможен только при совпадении признакового пространства, поэтому авторы были

вынуждены сократить данные до набора из 15 общих признаков, что автоматически означает потерю потенциально важной информации. Ограничением остаётся и то, что оба датасета являются синтетическими; распределения сетевого трафика в реальных IoT-средах значительно сложнее, и перенос, выполненный в однородном признаковом пространстве, может оказаться менее эффективным при переходе от симулированного домена к реальному. Кроме того, рассматривается бинарная классификация, и такой тип TL не решает задачу точного различения множества конкретных семейств атак, что ограничивает практическое применение на уровне оперативного реагирования.

Таким образом, среди исследуемых работ пакетный анализ представлен двумя направлениями: анализ последовательностей низкоуровневых кадров в CAN-сетях и анализ IoT-пакетов на уровне байтовых структур. Оба подхода демонстрируют преимущества в контекстах, где статистические потоки либо невозможно сформировать, либо они теряют важную информацию о вредоносных изменениях в полезной нагрузке. Однако пакетные методы требуют больших вычислительных ресурсов, чувствительны к изменению формата пакетов и часто менее устойчивы при переносе между доменами по сравнению с потоково-ориентированными системами, что делает задачу переноса обучения особенно важной.

Другим примером индуктивного обучения является работа [20]. В этой работе авторы решают смежную с обнаружением вторжений задачу, а именно классификацию веб-приложений. На этапе предобучения решаются вспомогательные задачи самоконтролируемого обучения (self-supervised tasks) – распознавание зашифрованных пакетов по значению энтропии и определение непрерывности потока, тогда как целевая задача – это многоклассовая классификация приложений. Таким образом, переносимые знания представляют собой неспецифические высокоуровневые представления, отражающие статистические и структурные закономерности зашифрованного трафика: распределение байтовых значений, временные паттерны, слабые межпакетные зависимости. Эти признаки извлекаются свёрточной нейронной сетью (CNN) на уровне отдельных пакетов и обобщаются кодировщиком на основе архитектуры BERT на уровне последовательностей пакетов.

Достоинством такого подхода является высокая адаптивность: модель не требует размеченных данных на этапе предобучения, а дообучение (fine-tuning) на целевом наборе данных возможно даже при небольшом объёме размеченных примеров (20–40%), что особенно важно в практических сценариях, где аннотация сетевого трафика трудоёмка и дорогостояща. Кроме того, отсутствие зависимости от заголовков пакетов – в том числе от пятёрки ключевых полей (5-tuple: исходный IP, исходный порт, целевой IP, целевой порт, транспортный протокол) – делает перенос между доменами более устойчивым: например, становится возможной адаптация между наборами данных, собранными в различных сетевых средах.

Недостатком является высокая вычислительная сложность самой архитектуры CBD: большое число параметров, особенно при использовании 12-уровневого кодировщика BERT, делает обучение чувствительным к объёму размеченных данных в целевом домене; без предобучения модель склонна к переобучению или несходимости даже на относительно простых задачах. Также следует отметить, что предложенный подход к переносу обучения остаётся замкнутым по отношению к неизвестным классам поскольку как предобучение, так и дообучение ориентированы на фиксированный набор классов, и полученные обобщённые признаки могут оказаться недостаточными для обнаружения принципиально новых, ранее не встречавшихся типов трафика.

3.3 Преимущества и ограничения переноса обучения для NIDS

Классификация приведенных выше примеров с указанием типа анализируемых данных (полужирным выделены методы на основе анализа статистических признаков) приведена в табл. 2.

Табл. 2. Классификация примеров.
Table 2. Classification of examples.

	Перенос признаков	Перенос параметров	Перенос отношений	Перенос экземпляров
Индуктивный перенос обучения	[17], [20], **[15]**	**[14]**	**[13]**	[16]
Трансдуктивный перенос обучения	**[11]**	–	–	**[8]**
Перенос обучения без учителя	**[9]**	–	–	–

Методы, основанные на анализе отдельных сетевых пакетов, обладают явным преимуществом в полноте информации: они позволяют извлекать признаки непосредственно из структуры заголовков и полезной нагрузки, что обеспечивает выявление сложных атак, скрытых на уровне байтовых последовательностей. Благодаря этому подходы на основе анализа пакетов особенно эффективны против угроз ориентированных на полезную нагрузку, таких как внедрение SQL-кода, инъекции кода и атаки на протоколы прикладного уровня. Дополнительным преимуществом является высокая чувствительность к малозаметным закономерностям в поведении атакующих, которые могут быть потеряны при агрегации в сетевые потоки. В контексте переноса обучения наличие богатого низкоуровневого представления создаёт условия для переноса универсальных признаков, особенно связанных с протокольной структурой и вредоносными шаблонами полезной нагрузки.

Однако высокая детализация ведёт к существенным минусам. Анализ пакетов требует значительно большего объёма вычислительных ресурсов, как для хранения, так и для обработки данных. Модели на таких входах склонны сильнее деградировать при смене домена, поскольку структура пакетов может существенно различаться между сетями, прошивками, устройствами и версиями протоколов. Это снижает устойчивость переноса обучения на основе анализа пакетов и требует дополнительного дообучения. Кроме того, многие современные сети используют шифрование, что делает полезную нагрузку недоступной для анализа и резко снижает эффективность методов, основанных на содержимом пакета.

Методы, основанные на признаках потоков, напротив, обладают высокой масштабируемостью и существенно меньшими требованиями к ресурсам. Они дают компактное обобщение поведения соединения, что делает такие модели более стабильными и переносимыми между различными сетями. Они устойчивы к шифрованию, поскольку используются только метаданные (размеры пакетов, время межпакетных интервалов, количество направленных пакетов), что позволяет эффективно работать в условиях современного TLS-ориентированного трафика. На практике именно методы на основе анализа признаков потоков чаще демонстрируют хорошую адаптивность к новым сетям и реже требуют переработки признаков.

Однако агрегированная природа потоков неизбежно приводит к потере информации. Потери касаются прежде всего полезной нагрузки и последовательности пакетов внутри соединения, что делает такие модели уязвимыми к атакующим, использующим нестандартные атаки или атакующие последовательности, проявляющиеся только на уровне отдельных пакетов. Таким

образом, такие методы часто уступают в обнаружении сложных и ранее неизвестных атак, особенно тех, что не изменяют метаданные соединения.

Отдельно следует учитывать риск негативного переноса [21]. При использовании моделей, обученных на пакетных данных, негативный перенос проявляется особенно часто: низкоуровневые признаки, извлечённые из заголовков и полезной нагрузки, могут быть тесно связаны с конкретными конфигурациями сети, типами устройств, версиями прошивок или даже характерными шаблонами трафика отдельных инфраструктур. При переносе на новую сеть такие признаки перестают быть релевантными и могут вводить модель в заблуждение, снижая точность и увеличивая число ложных срабатываний. В потоковых методах негативный перенос выражен слабее, однако и здесь обобщённые статистические характеристики соединений могут систематически отличаться между доменами: например, варьируются профили задержек, средние размеры пакетов или характер распределения длительностей соединений. В результате модель начинает интерпретировать безвредный трафик как аномальный или игнорировать признаки реальных атак. Таким образом, негативный перенос остаётся фундаментальной проблемой для обоих подходов, особенно при использовании данных из сильно разнородных сетевых сред.

3.4 Обобщение методов

Обобщив примеры и анализ выше, рассмотрим основные виды переноса, применяемые в NIDS, их ключевые идеи, преимущества и ограничения. Такое сравнение позволяет оценить, какие методы наиболее практичны в условиях ограниченных данных, меняющихся распределений трафика и появления неизвестных атак.

Индуктивный перенос признаков использует предобученную CNN с замороженными слоями, обученную на BoT-IoT и адаптированную на UNSW-NB15. Такой подход улучшает качество при нехватке данных и снижает переобучение, но требует совпадения признаков и в основном подходит для бинарных задач.

Индуктивный перенос параметров основан на передаче знаний от модели, обученной на исходной задаче, к новой задаче через инициализацию или частичную донастройку весов. Веса свёрточной части CNN, предобученной на Bot-IoT, фиксируются и используются как обобщённые признаковые экстракторы, а верхние полносвязные слои дообучаются на небольшом целевом датасете (TON-IoT), что позволяет эффективно адаптировать модель к новым атакам без полного переобучения и с минимальными вычислительными затратами.

Индуктивный перенос отношений сочетает SVM, K-means и KNN, используя корпоративный трафик и отчёты VirusTotal. Он улучшает F1 и качество обнаружения новых атак, но сильно зависит от качества CTI-данных и требует значительных ресурсов.

Индуктивный перенос экземпляров применяет Conv-LSTM с обучением по одному примеру атаки (one-shot) на реальном CAN-трафике. Он обеспечивает высокую точность и низкое число ложных срабатываний, но чувствителен к качеству единственного обучающего примера.

Трансдуктивный перенос признаков (HeTL и CeHTL) работает на NSL-KDD, выравнивая скрытые пространства исходного и целевого доменов. Он улучшает обнаружение новых атак; CeHTL не требует ручного тюнинга, тогда как HeTL чувствителен к параметрам.

Трансдуктивный перенос экземпляров использует перевзвешивание данных через MMD в сочетании с классическими моделями. Он помогает адаптироваться к дрейфу распределений, но зависит от точности измерения расстояния между доменами.

Перенос без учителя объединяет GAN, RL-оптимизацию признаков, MMD и Softmax-классификацию, применяя это к многоклассовому трафику. Он обеспечивает высокую точность и снижает смещение распределений, хотя синтетические данные не всегда полностью отражают реальные зависимости.

Рассмотренные подходы к переносу обучения демонстрируют, что использование знаний из других доменов позволяет существенно повысить качество и устойчивость систем обнаружения вторжений. Каждый тип переноса решает свою задачу: одни помогают справиться с нехваткой данных, другие – адаптироваться к новым атакам или изменению распределений трафика. Несмотря на имеющиеся ограничения, методы переноса уменьшают потребность в полном переобучении моделей и делают NIDS более гибкими и эффективными в условиях быстро меняющейся сетевой среды.

4. Заключение

Перенос обучения в сетевых системах обнаружения вторжений представляет собой ключевое направление, позволяющее преодолеть фундаментальные ограничения традиционных NIDS, связанные с изменчивостью трафика, дрейфом распределений и появлением новых типов атак. Анализ представленных работ показывает, что различные парадигмы переноса – от переноса признаков и моделей до переноса отношений и экземпляров – по-разному раскрывают потенциал адаптации систем обнаружения вторжений к новым сетевым средам, обеспечивая устойчивость и повышенную обобщающую способность при работе в условиях ограниченной разметки и неоднородности доменов.

Методы, основанные на статистических признаках потоков, демонстрируют высокую переносимость и устойчивость к шифрованию, что делает их особенно привлекательными для практического применения. При этом существенным ограничением остаётся неизбежная потеря информации о структуре отдельных пакетов, что снижает способность обнаруживать сложные аномалии. Напротив, подходы, использующие последовательности пакетов и их низкоуровневое представление, обладают большей детализированностью и позволяют выявлять тонкие закономерности поведения атакующих, однако значительно чувствительнее к смене домена и требуют больших вычислительных ресурсов.

Показательно, что в современных исследованиях всё чаще встречаются гибридные и многоуровневые стратегии переноса, сочетающие методы доменной адаптации, генеративные модели, оптимизацию признаков и непрерывное обновление на основе CTI-данных. Такие системы демонстрируют способность к эволюции в реальном времени, что критически важно для противодействия динамическим и малоизвестным угрозам.

Тем не менее, перенос обучения для NIDS сталкивается с рядом ограничений: риском негативного переноса, зависимостью от степени близости доменов, высокой сложностью архитектур, а также отсутствием универсальных наборов данных, адекватно отражающих разнообразие реальных сетевых условий. Решение этих проблем требует развития методов, способных учитывать дрейф, работать с частично размеченными или полностью неразмеченными данными, а также интегрироваться с потоковыми алгоритмами машинного обучения.

Таким образом, перенос обучения формирует основу для нового поколения интеллектуальных NIDS, способных адаптироваться к быстро меняющимся условиям и противостоять угрозам, ранее недоступным для классических методов, однако дальнейший прогресс в этой области требует как теоретических исследований, так и систематических практических оценок в реалистичных сетевых сценариях.

Список литературы / References

[1]. Liao H. J. et al. Intrusion detection system: A comprehensive review //Journal of network and computer applications. – 2013. – Т. 36. – №. 1. – С. 16-24.

[2]. Zhuang F. et al. A comprehensive survey on transfer learning //Proceedings of the IEEE. – 2020. – Т. 109. – №. 1. – С. 43-76.

[3]. Pan S. J., Yang Q. A survey on transfer learning //IEEE Transactions on knowledge and data engineering. – 2009. – Т. 22. – №. 10. – С. 1345-1359.

[4]. Nguyen C. T. et al. Transfer learning for wireless networks: A comprehensive survey //Proceedings of the IEEE. – 2022. – Т. 110. – №. 8. – С. 1073-1115.
[5]. Yosinski J. et al. How transferable are features in deep neural networks? //arXiv preprint arXiv:1411.1792. – 2014.
[6]. Ring M. et al. A survey of network-based intrusion detection data sets //Computers & security. – 2019. – Т. 86. – С. 147-167.
[7]. Wang M. et al. On the robustness of ML-based network intrusion detection systems: An adversarial and distribution shift perspective //Computers. – 2023. – Т. 12. – №. 10. – С. 209.
[8]. Wu P., Guo H., Buckland R. A transfer learning approach for network intrusion detection //arXiv preprint arXiv:1909.02352. – 2019.
[9]. Ma W. et al. Abnormal traffic detection based on generative adversarial network and feature optimization selection //International Journal of Computational Intelligence Systems. – 2021. – Т. 14. – №. 1. – С. 1170-1188.
[10]. Gretton A. et al. A kernel two-sample test //The journal of machine learning research. – 2012. – Т. 13. – №. 1. – С. 723-773.
[11]. Zhao J. et al. Transfer learning for detecting unknown network attacks //EURASIP Journal on Information Security. – 2019. – Т. 2019. – №. 1. – С. 1-13.
[12]. Zhao J., Shetty S., Pan J. W. Feature-based transfer learning for network security //MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM). – IEEE, 2017. – С. 17-22.
[13]. Lin Y. D. et al. Evolving ML-based Intrusion Detection: Cyber Threat Intelligence for Dynamic Model Updates //IEEE Transactions on Machine Learning in Communications and Networking. – 2025.
[14]. Dhillon H., Haque A. Towards network traffic monitoring using deep transfer learning //2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). – IEEE, 2020. – С. 1089-1096.
[15]. Idrissi I., Azizi M., Moussaoui O. Accelerating the update of a DL-based IDS for IoT using deep transfer learning //Indones. J. Electr. Eng. Comput. Sci. – 2021. – Т. 23. – №. 2. – С. 1059-1067.
[16]. Tariq S., Lee S., Woo S. S. CANTransfer: Transfer learning based intrusion detection on a controller area network using convolutional LSTM network //Proceedings of the 35th annual ACM symposium on applied computing. – 2020. – С. 1048-1055.
[17]. Rodríguez E. et al. Transfer-learning-based intrusion detection framework in IoT networks //Sensors. – 2022. – Т. 22. – №. 15. – С. 5621.
[18]. BoT IoT Dataset, Available at: https://research.unsw.edu.au/projects/bot-iot-dataset, accessed 20.11.2025.
[19]. Moustafa N., Slay J. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set) //2015 military communications and information systems conference (MilCIS). – IEEE, 2015. – С. 1-6.
[20]. Hu X. et al. CBD: A deep-learning-based scheme for encrypted traffic classification with a general pre-training method //Sensors. – 2021. – Т. 21. – №. 24. – С. 8231.
[21]. Wang Z. et al. Characterizing and avoiding negative transfer //Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. – 2019. – С. 11293-11302.

Информация об авторах / Information about authors

Антон Юрьевич ПОКИДЬКО – стажер-исследователь отдела компиляторных технологий ИСП РАН. Научные интересы: дрейф в машинном обучении и нейронных сетях, перенос обучения, анализ сетевого трафика.

Anton Yurevich POKIDKO – research intern at Compiler Technology department of ISP RAS. Research interests: drift in machine learning and neural networks, transfer learning, network traffic analysis.

Иван Александрович СТЕПАНОВ – аспирант, стажёр-исследователь ИСП РАН, ассистент кафедры информатики и вычислительной математики МФТИ. Сфера научных интересов:анализ сетевого трафика с помощью машинного обучения.

Ivan Alexandrovich STEPANOV – postgraduate student of the ISP RAS, intern researcher at ISP RAS, an assistant at the Department of Computer Science and Computational Mathematics at MIPT. Research interests: network traffic analysis using machine learning.

Александр Игоревич ГЕТЬМАН – кандидат физико-математических наук, старший научный сотрудник ИСП РАН, ассистент ВМК МГУ, доцент ВШЭ и МФТИ. Сфера научных интересов: анализ бинарного кода, восстановление форматов данных, анализ и классификация сетевого трафика.

Aleksandr Igorevich GETMAN – Cand. Sci. (Phys.-Math.), senior researcher at ISP RAS, assistant at CMC MSU, associate professor at HSE and MIPT. Research interests: binary code analysis, data format recovery, network traffic analysis and classification.

DOI: 10.15514/ISPRAS-2025-37(6)-38

CC BY

Адаптация алгоритма ThreadSanitizer для обнаружения гонок по данным в ядре ОСРВ

Е.С. Ельчинов, ORCID: 0000-0003-4555-1204 <elchinov@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Дизайн и реализация корректных алгоритмов многопоточной синхронизации являются неотъемлемой частью разработки современных операционных систем реального времени. Тестирование корректности алгоритма в модели памяти языка – одна из важнейших задач на этом пути. В статье описывается интеграция широко используемого алгоритма обнаружения гонок данных ThreadSanitizer из программной инфраструктуры LLVM в систему сборки и тестирования ядра операционной системы реального времени и его преимущества и недостатки в сравнении с другими подходами обнаружения ошибок многопоточной синхронизации. Среди прочего рассматривается определение семантики управления прерываниями и работы с физическими ядрами в контексте синхронизации в модели «выполняется прежде» (happens-before). В заключение приводятся результаты интеграции инструмента ThreadSanitizer в ядро операционной системы реального времени CLOS в сравнении с существующими подходами обнаружения ошибок в ядре данной операционной системы.

Ключевые слова: многопоточная синхронизация; динамический анализ; операционные системы; алгоритм thread sanitizer; гонки по данным.

Для цитирования: Ельчинов Е.С. Адаптация алгоритма ThreadSanitizer для обнаружения гонок по данным в ядре ОСРВ. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 91–108. DOI: 10.15514/ISPRAS–2025–37(6)–38.

Adaptation of the ThreadSanitizer Algorithm for Data Race Detection in a RTOS Kernel

E.S. Elchinov, ORCID: 0000-0003-4555-1204 <elchinov@ispras.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Correct design and implementation of concurrent algorithms is a crucial part of modern real-time operating system development. One of the main steps along this way is a verification of such algorithms within the programming language memory model. The article describes an integration of the ThreadSanitizer – broadly used LLVM tool for data race detection – into the RTOS kernel environment and discusses its advantages and disadvantages over other tools for data race detection. Among other things, the semantics of context switches and interrupt management within the happens-before synchronization model is considered. In conclusion the results of a ThreadSanitizer tool integration are provided compared to current approaches of concurrency bugs detection in RTOS kernel.

Keywords: mutlithreaded synchronization; dynamic analysis; operating systems; thread sanitizer algorithm; data races.

For citation: Elchinov E.S. Adaptation of the ThreadSanitizer algorithm for data race detection in a RTOS kernel. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, psrt 3, 2025, pp. 91-108 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-38.

1. Введение

Алгоритмы многопоточной синхронизации находят широкое применение в разработке современных операционных систем, в особенности, поддерживающих симметричную многопроцессорность и вытеснение на уровне задач ядра. В то же время, нетривиальные алгоритмы многопоточной синхронизации существенно усложняют процесс тестирования кода ядра и могут содержать трудно отлаживаемые без дополнительных инструментов ошибки.

В пользовательском окружении для отладки и верификации систем процессов, коммуницирующих через разделяемую память, существует множество подходов, использующих различные методы статического и динамического анализа кода и его исполнений. Некоторые операционные системы общего назначения, например, Linux, поддерживают собственные инструменты и для анализа кода ядра (такие как LKMM [1] или KCSAN [2]).

В данной работе рассматривается расширение возможностей динамического анализа применительно к поиску ошибок многопоточной синхронизации в ядре операционных систем реального времени. В сравнении с операционной системой (ОС) общего назначения, ОС реального времени (ОСРВ) характерно имеют статическую настройку виртуальной памяти, меньший объём кода ядра и строгие ограничения на время исполнения задач ядра и затрачиваемую память. Рассматриваемая в статье ОС имеет все описанные характерные для систем реального времени черты. С учётом особенностей пространства ядра ОС и, в частности, требований операционных систем реального времени, существующие подходы к динамическому анализу кода требуют адаптации и переработки.

В данной работе описывается адаптация алгоритма детектора гонок данных ThreadSanitizer [3] и необходимые оптимизации его библиотеки времени исполнения для работы в пространстве ядра упомянутой ОСРВ, а также результаты его интеграции в систему тестирования ядра. Для определения требований к алгоритму вначале определяется семантика управления прерываниями в терминах отношения *«выполняется прежде»*

(*happens-before*). Также описывается влияние свойств систем реального времени на успешность применения различных подходов к поиску ошибок синхронизации.

2. Гонки по данным

Гонки по данным – один из наиболее распространённых типов ошибок в алгоритмах многопоточной синхронизации. По стандарту языков C и C++, на которых написана значительная часть системного программного обеспечения, они приводят к неопределённому поведению [4]. Для определения корректности программы в конкурентном исполнении многие современные языки программирования внедрили в свои стандарты так называемые модели памяти – наборы правил для каждой операции чтения, определяющих все возможные модифицирующие операции, результаты которых она может вернуть. Вторая задача модели памяти – определить все корректные исполнения программы с точки зрения многопоточной синхронизации.

2.1 Математические порядки над множеством операций над памятью

В модели памяти языка C для каждого исполнения программы определяется несколько частичных порядков над выполняемыми операциями над памятью.

Для дальнейшего изложения понадобятся следующие отношения и определения:

- *happens-before* – отношение «выполняется прежде» – для двух операций **A** и **B** верно **A** *happens-before* **B**, если из результата исполнения операции **B** следует факт завершения операции **A**.
- *sequenced-before* (также, *program order*) – отношение «предшествует в одном потоке» – для двух операций **A** и **B** верно **A** *sequenced-before* **B**, если операции **A** и **B** исполняются в одном потоке, причём операция **A** предшествует операции **B**.
- *synchronizes-with* – отношение «причинности» операций синхронизации – для двух операций **A** и **B** над одной переменной синхронизации верно **A** *synchronizes-with* **B**, если **A** – операция чтения некоторого состояния, **B** – операция записи (изменения) состояния, и операция **A** читает результат записи операции **B**.
- *atomic*-операции – специальный тип операций над памятью, имеющих, согласно стандарту, определённое поведение при конкурентном исполнении конфликтующих операций.
- *release*-операции – операции записи (изменения) состояния некоторого объекта синхронизации.
- *acquire*-операции – операции чтения состояния некоторого объекта синхронизации.

Согласно стандарту, если **B** – *release*-операция и **A** – *acquire*-операция, читающая значение, записанное операцией **B**, то выполняется отношение **A** *synchronizes-with* **B**.

2.2 Определение гонки данных

Стандарты языка C, начиная с версии C11 определяют гонку данных как два одновременных конфликтующих неатомарных обращения к памяти. Согласно стандарту, две попытки доступа конфликтуют, если соответствующие им ячейки памяти имеют непустое пересечение, и хотя бы одна из попыток модифицирующая [4]. Две операции могут считаться одновременными в случае, когда они не упорядочены порядком *happens-before*, согласно его определению в модели памяти – иными словами, если в исполнении в промежутке между этими операциями нет наблюдаемой синхронизации соответствующих им потоков.

3. Динамические методы обнаружения гонок данных

Для поиска гонок данных в коде применяется два основных класса методов – алгоритмы статического и динамического анализа кода.

Алгоритмы статического анализа кода ищут ошибки, опираясь на исходный код, без непосредственного исполнения алгоритма. Среди статических инструментов для поиска гонок данных в коде ядра ОС можно выделить LKMM [1] из ядра ОС Linux.

В отличие от статических методов, при использовании динамического анализа алгоритм обнаружения ошибок внедряется в анализируемый код на этапе сборки и производит поиск ошибок на основе анализа текущего исполнения на некоторых тестовых сценариях.

Среди основных преимуществ динамического анализа:

- Динамический анализ позволяет практически исключить получение ложноположительных результатов, ограничиваясь небольшим (в сравнении со статическим анализом) количеством аннотаций к анализируемому коду.
- Многие динамические алгоритмы требуют затрат по памяти и времени исполнения, пропорциональных требованиям анализируемой системы, что позволяет анализировать сложные исполнения.

Из минусов динамического подхода, можно отметить:

- Так как динамические методы рассматривают лишь конкретное исполнение, качество анализа напрямую зависит от качества тестового покрытия анализируемого кода.
- Динамический анализ не способен обеспечить строгие гарантии корректности кода, так как в тестовых сценариях могут реализовываться не все возможные исполнения.

Для динамического поиска гонок в коде ядра ОС Linux поддерживается инструмент KCSAN (kernel consurrency sanitizer) [2]. Для пользовательских приложений инфраструктура сборки LLVM предоставляет инструмент динамического анализа Thread Sanitizer [5].

На момент написания статьи существует несколько подходов к динамическому анализу программ на предмет гонок данных, среди которых можно выделить алгоритмы, основанные на использовании точек останова по данным (watchpoint-based) и алгоритмы, использующие для поиска гонок упомянутое ранее отношение *happens-before* над операциями с памятью, а также метод, основанный на построении множества активных критических секций (lockset). По определению, гонка данных – это одновременный конфликтующий небезопасный доступ. Упомянутые подходы различаются алгоритмом обнаружения одновременных операций над одной ячейкой памяти.

Основанные на точках останова (watchpoint-based) алгоритмы находят одновременные доступы путём приостановки программы в местах доступа к памяти на псевдослучайный промежуток времени и обработки всех доступов, произошедших к данной ячейке памяти в обозначенный промежуток, как одновременных с первым доступом. Для доступов к интересующему адресу такие алгоритмы используют механизм точек останова по данным (watchpoint).

Детекторы гонок по данным, основанные на построении порядка операций (happens-before-детекторы), производят поиск одновременных доступов по определению из стандарта, то есть поддерживают в памяти часть истории доступов к памяти и сжатое представление порядка *happens-before* и проверяют, упорядочен ли каждый следующий доступ с конфликтующими с ним сохранёнными доступами.

Алгоритм построения множества активных критических секций (lockset) для каждого доступа к памяти определяет все активные на этот момент критические секции и проверяет для каждой переменной, что все конфликтующие доступы к ней разделяют между собой хотя бы одну общую критическую секцию.

Соответственно, преимуществами подхода, основанного на точках останова являются:

- Минимальные затраты памяти – необходимо лишь хранить информацию о текущих точках останова по данным;
- Отсутствие необходимости в аннотациях к кодовой базе, так как доступы, происходящие в момент ожидания на точках останова, всегда не упорядочены с обрабатываемым доступом в смысле порядка *happens-before* в модели памяти.

В свою очередь, из плюсов подхода, основанного на построении во время исполнения представления частичного порядка *happens-before*, можно отметить:

- Способность обнаруживать гонки по их определению в модели памяти языка – то есть независимо от целевой платформы и её модели памяти.
- Обработка каждой операции над памятью детерминированным образом – то есть для верификации конкретного сценария исполнения программы его достаточно проанализировать один раз.

Таким образом, детекторы, основанные на построении порядка *happens-before*, в сравнении с подходом, основанным на точках останова, позволяют независимо от текущей целевой платформы находить более широкий спектр ошибок, в том числе гонок данных, не приводящих к некорректным значениям в анализируемом исполнении, но требуют более детальных аннотаций и дополнительных ресурсов по времени и памяти.

По сравнению с алгоритмами, основанными на точках останова по данным либо построении порядка *happens-before*, метод, основанный на построении множества активных критических секций (lockset) обеспечивает меньшие затраты по памяти, чем метод, основаный на *happens-before* и, в отличие от детекторов, основанных на точках останова, позволяет детектировать потенциальные гонки, не реализовавшиеся в анализируемом исполнении. Однако, этот метод не позволяет корректно анализировать алгоритмы неблокирующей синхронизации, а также, по сути подхода, способен генерировать ложные сообщения об ошибках.

К подходам, основанным на точках останова, относятся такие алгоритмы как, например, RaceHound [6] и Kernel Concurrency Sanitizer [2] в ядре Linux. Алгоритм Thread Sanitizer [3], в свою очередь, является детектором, основанным на построении порядка *happens-before*. Среди детекторов, использующих построение множества активных критических секций, можно выделить Eraser [7].

4. Алгоритм Thread Sanitizer

Алгоритм поиска гонок Thread Sanitizer относится к классу динамических алгоритмов поиска гонок, основанных на анализе порядка операций *happens-before* и, соответственно, должен поддерживать в своей памяти некоторое сжатое представление упомянутого порядка над всеми хранимыми операциями с памятью. С этой целью каждой операции с памятью ставится в соответствие её эпоха – целое неотрицательное число, а каждому потоку и переменным синхронизации соответствуют векторные часы [8].

4.1 Модель синхронизации алгоритма ThreadSanitizer

В модели памяти языка C операции внутри одного потока упорядочены полным порядком *sequenced-before* (см. п. 2.1), известным также как *program order*. Отношение *happens-before* в модели памяти определяется как транзитивное замыкание *sequenced-before*, *synchronizes-with* и ещё нескольких порядков, где каждая дуга *synchronizes-with* формируется *release*-операцией и соответствующей ей (наблюдающей её эффект) последующей *acquire*-операцией над некоторой переменной синхронизации.

В модели синхронизации алгоритма Thread Sanitizer порядок *happens-before* выражается в виде транзитивного замыкания порядков *sequenced-before* и *synchronizes-with*, что является упрощением модели памяти языка C.

4.2 Эпохи и векторные часы

Поскольку в модели синхронизации внутри каждого потока отношение *sequenced-before* определяет полный порядок на операциях, алгоритм ThreadSanitizer сопоставляет каждой операции её эпоху – некоторое целое неотрицательное число, являющееся мерой прогресса потока к моменту данной операции, аналогично логическим часам Лэмпорта [9]. В дескрипторе каждой обрабатываемой операции над памятью, Thread Sanitizer, помимо прочих характеристик, сохраняет эпоху этой операции и идентификатор её потока.

В качестве сжатого представления порядка *happens-before* в ThreadSanitizer используются векторные часы [8], являющиеся широко используемым обобщением логических часов Лэмпорта. Если **T** – некоторый поток исполнения, то векторные часы потока **T** – список, хранящий для каждого активного потока в системе эпоху его последней операции, упорядоченной до текущей операции в потоке **T** в смысле порядка *happens-before*.

4.3 Синхронизация

Для корректной работы алгоритма необходим способ пересчёта сжатого представления графа отношения *happens-before* после операций синхронизации (то есть *release* и *acquire* операций над переменными синхронизации). С этой целью каждой переменной синхронизации – примитивам синхронизации и атомарным (*atomic*) переменным – также ставятся в соответствие векторные часы, хранящие для каждого потока последнюю его эпоху, предшествующую или равную (в смысле отношения *happens-before*) некоторой *release*-операции над этой переменной.

Для каждой *release*-операции потока **T** над переменной синхронизации **S** её векторные часы $\mathbf{V_S}$ (рис. 1) обновляются поэлементным максимумом (далее – операция $\max^*$) с векторными часами $\mathbf{V_T}$ потока **T**. Для *acquire*-операций, наоборот, векторные часы $\mathbf{V_T}$ обновляются поэлементным максимумом с $\mathbf{V_S}$. Таким образом, для дуги отношения *synchronizes-with* между операциями *release*(**S**) (далее – сокр. **rel**) в **T1** и *acquire*(**S**) (далее – сокр. **acq**) в **T2** верно $\mathbf{V'_{T2}} = \max^*(\mathbf{V_{T2}}, \mathbf{V'_S}) > \max^*(\mathbf{V_{T2}}, \max^*(\mathbf{V_{T1}}, \mathbf{V_S})) > \mathbf{V_{T1}}$ (где $\mathbf{V'_{T2}}$ и $\mathbf{V'_S}$ – обновлённые значения $\mathbf{V_{T2}}$ и $\mathbf{V_S}$ соответственно) (рис. 1).

Такое поведение согласуется с определениями векторных часов и отношений *synchronizes-with* и *happens-before* в модели памяти.

4.4 Теневая память и поиск гонок

Для хранения информации об обработанных доступах к памяти Thread Sanitizer использует отдельный регион памяти – так называемую теневую память – такой, что каждой ячейке машинного слова в основной памяти соответствует ячейка теневой памяти.

Каждая ячейка теневой памяти хранит необходимую информацию про несколько (в реализации в LLVM – 4) последних доступов к соответствующему машинному слову в основной памяти. Для каждой операции (листинг 1) в теневой памяти сохраняется её идентификатор потока, текущая эпоха этого потока, затронутые байты и тип доступа (рис. 2).

Таким образом, в момент обработки текущей операции **X** над памятью теневая память определяет потенциально конфликтующие предшествующие операции $\mathbf{Y_i}$, а векторные часы позволяют проверить, что между каждой конфликтующей операцией $\mathbf{Y_i}$ и **X** присутствует отношение *happens-before* (то есть, **X** *happens-before* $\mathbf{Y_i}$). В противном случае инструмент генерирует сообщение об ошибке.

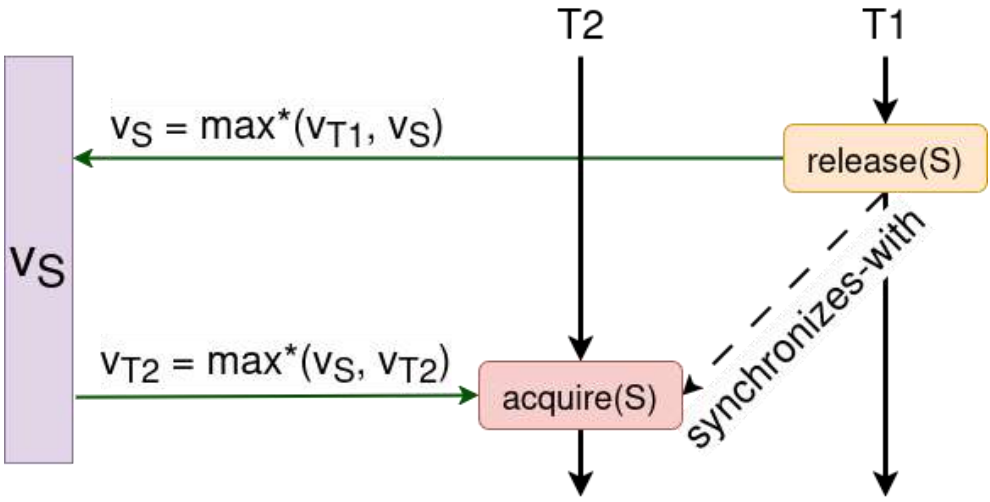


Рис. 1. Векторные часы.
Fig. 1. Vector clock.

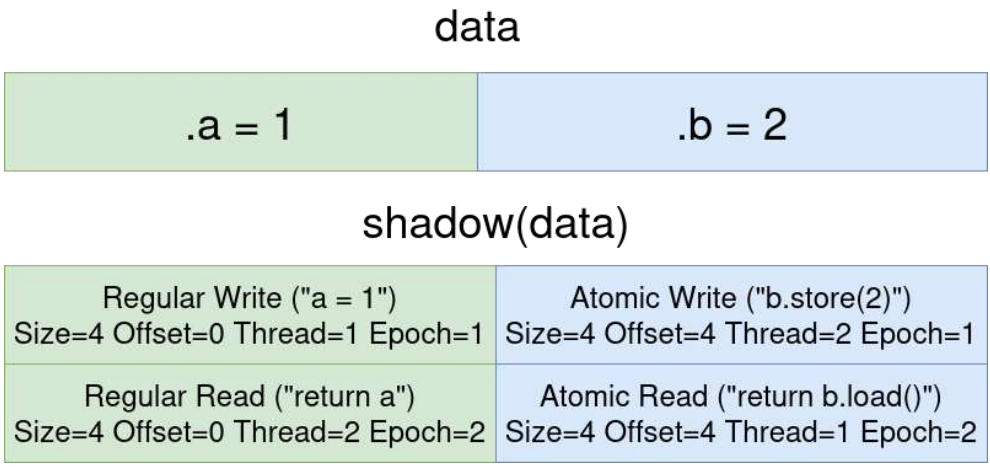


Рис. 2. Теневая память.
Fig. 2. Shadow memory.

```
struct {
    int a;
    std::atomic<int> b;
} data;
int thread1 () {
    data.a = 1;
    return data.b.load();
}
int thread2 () {
    data.b.store(2);
    return data.a;
}
```

Листинг 1. Пример заполнения теневой памяти.
Listing 1. Shadow memory update example.

5. Оптимизации, используемые в LLVM Thread Sanitizer

С целью уменьшения расходов по памяти и поддержки потенциально неограниченного числа потоков в текущей версии алгоритма LLVM Thread Sanitizer используется несколько механизмов, в числе которых обновление теневой памяти, трассировка операций и слоты синхронизации.

5.1 Трассировка операций и теневой стек

В трассах операций для каждого потока сохраняется порядок операций над памятью относительно вызовов процедур, что, среди прочего, позволяет восстановить стек вызовов любой операции из теневой памяти. Теневой стек поддерживает текущий стек вызовов в виде массива адресов в коде, предоставляя возможность быстро генерировать сообщения об ошибках (рис. 3).

Thread Sanitizer внедряет в код вызовы библиотеки времени исполнения в момент входа в каждую функцию и возврата из неё, что позволяет поддерживать теневой стек и учитывать вызовы и возвраты из функций в трассах потоков.

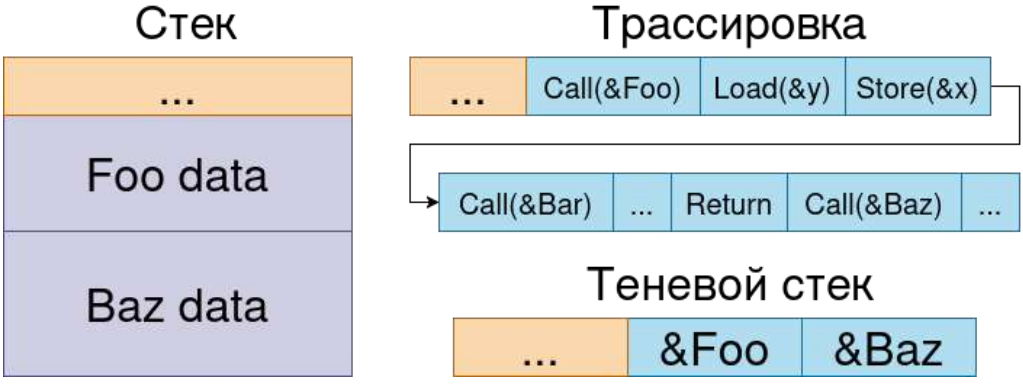


Рис. 3. Трассировка и теневой стек.
Fig. 3. Tracing and shadow stack.

5.2 Слоты синхронизации

Для эффективной обработки большого количества потоков в последней версии библиотеки времени исполнения Thread Sanitizer была сделана оптимизация, позволяющая ограничить размер векторных часов и количество бит идентификатора потока в теневой памяти.

Вместо потенциально неограниченного по величине целочисленного идентификатора потока реализация ThreadSanitizer поддерживает фиксированное количество так называемых слотов – логических единиц исполнения, имеющих локальный счётчик эпохи (рис. 4). Библиотека времени исполнения ThreadSanitizer ставит каждому исполняемому на данный момент потоку в соответствие некоторый слот. Каждый слот, в свою очередь, поддерживает историю потоков, когда-либо исполнявшихся в привязке к этому слоту. Эпоха слота считается эпохой привязанного к нему потока, а векторные часы потоков и переменных синхронизации хранят последнюю наблюдаемую эпоху каждого слота. В теневой памяти вместо идентификатора потока хранится индекс слота, соответствующего потоку, исполнившему операцию.

В такой модели наличие дуги отношения *synchronizes-with* между некоторыми потоками **T1** и **T2** влечёт за собой наличие той же дуги между соответствующими слотами потоков. В случае если количество потоков превосходит количество слотов, такой подход вызывает паразитную синхронизацию между потоками, разделяющими один слот в течение времени исполнения, что может привести к ложноотрицательным вердиктам в некоторых

исполнениях. Однако, если количество активных потоков не превосходит количества слотов, данный подход не приводит к потере информации в сжатом представлении порядка *happens-before* текущего исполнения.

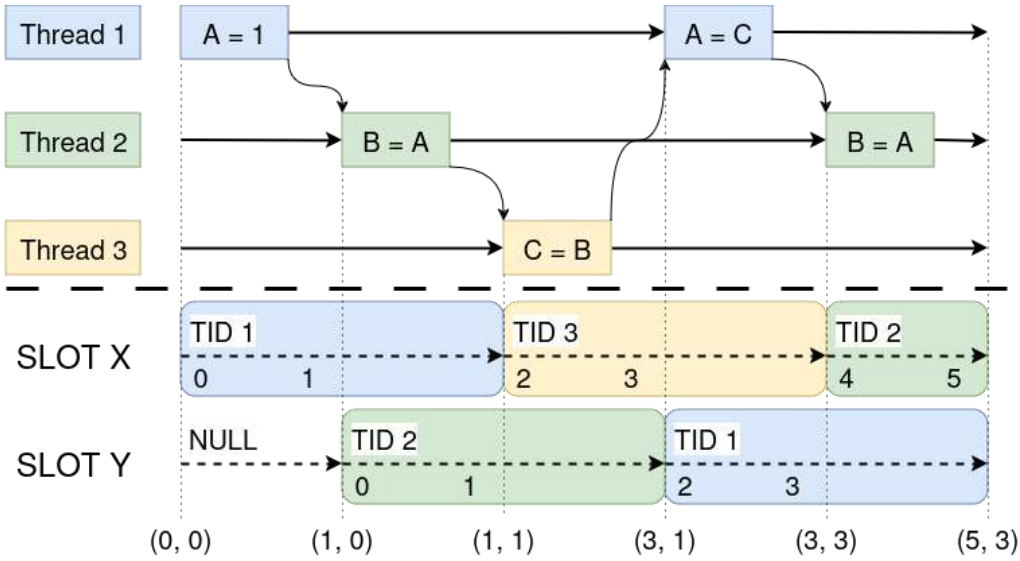


Рис. 4. Слоты синхронизации.
Fig. 4. Synchronization slots.

6. Интеграция Thread Sanitizer в ядро ОСРВ

6.1 Требования при интеграции в ядро ОСРВ

Особенности пространства ядра операционной системы накладывают ряд требований на реализацию алгоритма Thread Sanitizer:

- Большое количество потоков в сравнении с пространством пользователя – требуется умение обрабатывать синхронизацию сотен потоков, не увеличивая затраты памяти и времени для поддержания механизма векторных часов.
- Использование нестандартных механизмов синхронизации (барьеров памяти и управления прерываниями) – требуется определить их семантику в отношении порядка *happens-before* и интегрировать поддержку описанных механизмов в библиотеку времени исполнения.
- Переход в пространство пользователя и обработка асинхронных прерываний – требуется поддержать механизм поиска гонок между обработчиком прерывания и кодом прерванного потока.

Помимо описанных требований, свойства ОС реального времени накладывают на реализацию дополнительные ограничения:

- Ограниченный и фиксированный объём физической и виртуальной памяти – объём теневой памяти в отношении к основной памяти ядра не должен превосходить 1:1.
- Ограничения по затратам по времени в худшем случае – алгоритмы, опирающиеся на гарантии реального времени, должны корректно работать в комбинации с инструментом динамического анализа.

Для уменьшения требований алгоритма по дополнительной памяти за основу реализации библиотеки времени исполнения была взята описанная выше версия библиотеки времени

исполнения LLVM Thread Sanitizer, поддерживающая трассировку операций и слоты синхронизации, которая позволяет поддерживать произвольно большое количество логических потоков при фиксированном размере векторных часов и корректно обрабатывать переполнение счётчиков эпох потоков.

6.2 Оптимизация расходов по памяти

В текущей реализации LLVM Thread Sanitizer теневая память состоит из двух регионов: в первом, в соотношении 2:1 к основной памяти, хранится информация о последних доступах к памяти, во втором – метаинформация о переменных синхронизации, в соотношении 1:1 к основной памяти. В пространстве ядра ОСРВ нет механизмов динамического управления виртуальной памятью, а объём физической памяти на целевых платформах зачастую не позволяет иметь теневую память в соотношении большем чем 1:1 к основной.

Трёхкратная экономия потребляемой теневой памяти достигается за счёт уменьшения количества слотов потоков и максимального значения эпохи потока, а также замены теневого региона для метаинформации хеш-таблицей, хранящей для каждого адреса переменной синхронизации её векторные часы.

6.3 Обработка барьеров памяти и ассемблерного кода

Барьеры памяти, почти не используемые в пользовательском коде, находят широкое применение в коде ядра и должны учитываться библиотекой времени исполнения как точки синхронизации. Принцип обработки барьеров описан авторами KTSAN [10] и был адаптирован для текущей версии библиотеки времени исполнения.

С целью корректной обработки ассемблерного и прочего кода, не подвергающегося автоматическому аннотированию во время сборки (т.н. инструментации), Thread Sanitizer поддерживает явные вызовы интерфейса библиотеки времени исполнения – аннотации.

6.4 Оптимизация обработки переполнения счётчика эпохи

Для корректности свойств векторных часов и алгоритма обнаружения гонок данных требуется монотонность эпох операций в каждом слоте. Поскольку, с целью оптимизации затрат по памяти, размер счётчика эпохи в текущей реализации, в сравнении с LLVM ThreadSanitizer, был уменьшен до 6 бит, на любом реальном исполнении будет возникать переполнение счётчика эпохи. После переполнения счётчика эпохи нарушается свойство монотонности, и вся сохранённая информация, содержащая номера старых эпох и данные векторных часов, становится несогласованной с текущим состоянием структур Thread Sanitizer. Таким образом, для корректной обработки переполнения эпохи требуется выполнить сброс всего текущего состояния.

Обработка переполнения счётчика эпохи происходит в два этапа: сначала производится очистка трасс потоков и таблицы переменных синхронизации, затем операция обновления тени.

Меньший диапазон возможных эпох потоков, помимо уменьшения размера ячейки теневой памяти, даёт разумное ограничение на размер хеш-таблицы для метаинформации. Однако такое решение приводит к частым операциям обновления внутреннего состояния санитайзера, в том числе обнуления (сброса) теневой памяти с целью обработки переполнения счётчика эпохи потока. В LLVM Thread Sanitizer для Linux обнуление теневой памяти реализовано через системный вызов mmap. ОСРВ не поддерживает такое решение в силу статической конфигурации виртуальной памяти.

В связи с этим, для сброса теневой памяти состояние страниц тени (уже обновлена страница, или требует обнуления) эмулируется программным образом с использованием буфера флагов (рис. 5). При необходимости доступа к странице, требующей сброса данных, поток сначала

обнуляет её данные, затем сбрасывает флаг состояния страницы. Так как сбросы страниц теневой памяти конфликтуют с чтением информации о доступах из тени, возможные ложные гонки фильтруются с помощью подсистемы трассировки доступов к памяти.

Описанная реализация позволяет не останавливать прогресс системы во время операции обновления тени, что важно для динамического анализа кода, опирающегося на гарантии реального времени.

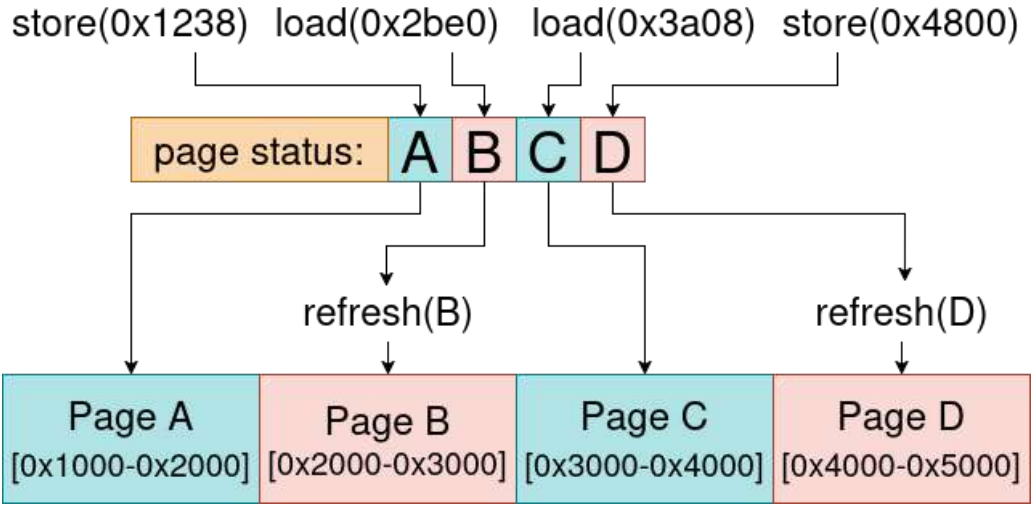


Рис. 5. Постраничное обновление тени.
Fig. 5. Per-page shadow reset.

7. Поддержка прерываний в реализации ThreadSanitizer

Одним из ключевых отличий пространства ядра от пользовательского пространства является наличие прерываний и управления контекстом исполнения.

7.1 Обработка процедур обработчиков прерываний

Поскольку внутри ядра ОС основная часть взаимодействия с пользователем и аппаратурой построена на механизме прерываний, код обработки прерываний может как вносить вклад в синхронизацию потоков, так и содержать гонки данных, которые необходимо своевременно обнаруживать. Таким образом, ThreadSanitizer обязан внедрять соответствующие проверки в код, исполняемый в процессе обработки прерываний.

Так как текущая реализация алгоритма ThreadSanitizer содержит блокировки и критические секции, вызовы библиотеки времени исполнения должны исполняться с запретом асинхронных прерываний. Запрет прерываний на время вызовов функций библиотеки времени исполнения Thread Sanitizer гарантирует, что на одном физическом ядре обработчики асинхронных прерывания обрабатываются в изоляции относительно кода проверок из библиотеки времени исполнения Thread Sanitizer для родительского потока.

Также, процедуры переключения контекстов нарушают предположение подсистемы трассировки вызовов Thread Sanitizer, что вызов функции и возврата из неё происходят в одном потоке и требуют отключения внедрения проверок в их коде.

7.2 Динамический анализ обработчиков асинхронных прерываний

Поскольку асинхронные прерывания исполняются конкурентно с кодом родительского потока, между ними существует возможность возникновения гонок по данным. С этой целью

обработчик прерывания следует рассматривать как отдельный логический поток, имеющий другой идентификатор доступов, хранящихся в теневой памяти, то есть привязанный к отдельному слоту потока.

Так как в каждый момент времени каждый поток исполняет либо собственный код, либо код обработчика прерывания, логический поток обработчика прерывания может разделять с родительским потоком теневой стек и буфер трассировки (рис. 6). Таким образом, поддержка асинхронных прерываний почти не требует дополнительной памяти. Также описанный подход упрощает диагностику при трассировке стека доступа, вызвавшего гонку, так как будет учтён стек вызовов не только обработчика прерывания, но и прерванного потока.

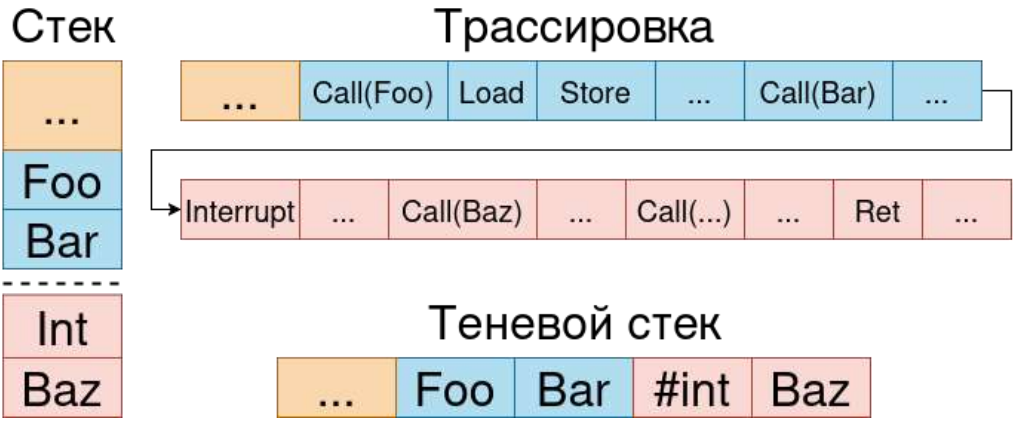


Рис. 6. Трассировка и теневой стек обработчика прерывания.
Fig. 6. Tracing and shadow stack with interrupt handlers.

В описанной реализации локальное хранилище данных структуры потока в реализации ThreadSanitizer в ядре ОС привязано к идентификатору соответствующего контекста исполнения, а текущий слот логического потока зависит от наличия прерываний.

Алгоритм, реализующий поддержку обработчиков прерываний в качестве отдельных логических потоков, должен определить семантику операций переключения контекста в ядре ОС относительно синхронизации в модели памяти. В реализации ядра ОС присутствуют три основных вида переключений контекста: переключение на исполнение другого потока ядра, переключение в пространство пользователя и, в момент возникновения прерывания, переключение на точку входа в его обработчик и возврат из прерывания. Прерывания, которые необходимо обрабатывать, могут быть как прерываниями из пространства пользователя, так и асинхронными прерываниями из пространства ядра.

7.3 Модель переключения контекстов в ядре ОС CLOS

Обработка прерываний и переключений контекстов исполнения в ядре ОС требует описания семантики работы каждой затрагиваемой операции, её предусловия и эффекта в некоторой модели состояния системы. В упрощённой модели далее рассматривается следующее состояние, локальное для каждого ядра центрального процессора:

- **context**: *integer* – ID текущего потока ядра ОС
- **ienable**: *bool* – флаг разрешения асинхронных прерываний
- **is_user**: *bool* – флаг исполнения пользовательского кода
- **async_int**: *integer* – уровень вложенности текущего обработчика асинхронного прерывания ядра (0 для случая исполнения кода потока), локальный для текущего context

Для простоты рассуждений и реализации модель не разделяет различные номера прерываний. В данной модели верны следующие инварианты:

- is_user ⇒ ienable
- async_int > 0 ⇒ ¬is_user

Определим в описанной модели обрабатываемые операции управления прерываниями (см. табл. 1). Нетрудно заметить, что вышеописанные инварианты согласуются с описанными операциями.

Табл. 1. Операции в модели переключения контекстов ОС CLOS.
Table 1. Operations in the CLOS OS model of context switching.

Операция	Описание	Предусловие	Эффект
async_enable	Разрешение асинхронных прерываний	¬is_user ∧ ¬ienable	ienable = 1
async_disable	Запрет асинхронных прерываний	¬is_user ∧ ienable	ienable = 0
switch_context(T)	Переключение на поток T	¬is_user ∧ ¬ienable	context = T ∧ async_int = async_int(T)
switch_to_user	Переключение в код пользователя	¬is_user ∧ ¬ienable	is_user = 1 ∧ ienable = 1 ∧ async_int = 0
kernel_sync_int	Вход в синхронное прерывание ядра	¬is_user	∅
kernel_sync_ret	Возврат из синхронного прерывания ядра	¬is_user	∅
kernel_async_int	Вход в асинхронное прерывание ядра	¬is_user ∧ ienable	async_int = async_int+1
kernel_async_ret	Возврат из асинхронного прерывания ядра	¬is_user ∧ async_int > 0	async_int = async_int−1
user_int	Вход в (любое) прерывание пользователя	is_user	is_user = 0 ∧ ienable
user_ret	Возврат из (любого) прерывания пользователя	¬is_user ∧ ¬ienable ∧ async_int = 0	is_user = 1 ∧ ienable = 1

7.4 Семантика асинхронных прерываний в коде ядра

Операция разрешения прерываний *async_enable* имеет семантику *release*-операции (см. определения ранее), поскольку, помимо управления прерываниями, служит аппаратным барьером для операций записи (т. н. *release*-барьером). Аналогично, операция запрета прерываний *async_disable* имеет эффект аппаратного барьера для операций чтения (т. н. *acquire*-барьера). Таким образом, точка входа в прерывание *kernel_async_int* и операция возврата из него *kernel_async_ret* имеют, соответственно, семантику *acquire* и *release* операций.

Операция *kernel_async_int* требует **¬is_user∧ienable**, а значит, наблюдает эффект **ienable = 1** при **¬is_user** некоторой операции включения прерываний *async_enable* на данном ядре.

Операция *async_disable* имеет эффект **ienable = 0**, а значит, наблюдает эффект всех обработанных прерываний на данном ядре, в силу предусловия **ienable** операции *kernel_async_int*, её эффекта **async_int = async_int + 1** и предусловия **async_int > 0** операции *kernel_async_ret*.

Операции *kernel_sync_int* и *kernel_sync_ret* не меняют состояния модели и не обрабатываются в реализации.

Таким образом, на одном ядре CPU верно, что

- *async_enable* synchronizes-with *kernel_async_int*
- *kernel_async_ret* synchëronizes-with *async_disable*

С целью поддержания этого отношения, в структуре-дескрипторе физического ядра выделяются отдельные векторные часы, **«core» vclock**, по аналогии с обработкой регулярной переменной синхронизации (рис. 7).

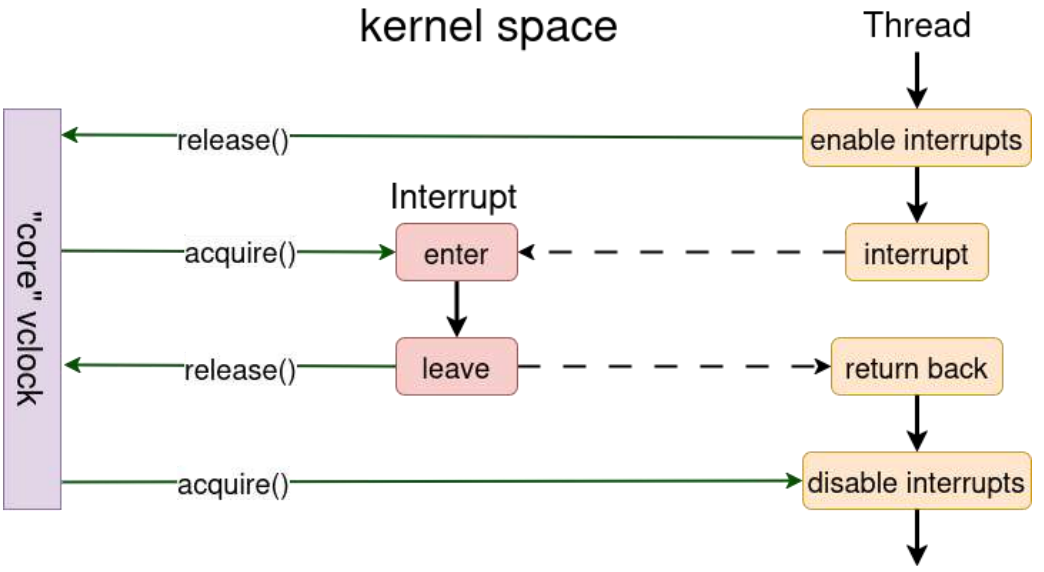


Рис. 7. Асинхронные прерывания.
Fig. 7. Asynchronous interrupts.

7.5 Семантика переключений контекста исполнения

При синхронном переключении с контекста потока **Thread 1** (далее **T1**) на контекст другого потока ядра **Thread 2** (далее **T2**) – *switch_context(***T2***)* – все последующие операции потока **T2** наблюдают эффект всех операций текущего потока **T1**, исполненных к данному моменту.

Так как для *switch_context* выполняется предусловие **¬is_user∧¬ienable**, верно, что *async_disable* sequenced-before *switch_context*, и для синхронизации потоков **T1** и **T2** возможно использовать те же векторные часы **«core» vclock**, что и в предыдущем пункте (рис .8).

7.6 Семантика прерываний в коде пользователя

Переключение в пространство пользователя останавливает прогресс кода потока ядра,д однако ядро продолжает обрабатывать прерывания пользовательского кода, как асинхронные

(например, прерывание таймера), так и синхронные (например, системные вызовы). По аналогии с обработкой прерываний в ядре, операции *switch_to_user* и *user_ret* имеют release-семантику, а операция *user_ret* имеет acquire-семантику.

Так как *user_int* имеет предусловие **is_user**, а операции *switch_to_user* и *user_ret* имеют эффект **is_user = 1**, верно, что *user_int* наблюдает эффект *switch_to_user* и *user_ret*.

Также *switch_to_user* имеет предусловие **¬is_user∧¬ienable**, то есть, наблюдают эффект **ienable = 0** операции *async_disable*. В свою очередь, *async_disable* наблюдает эффект всех обработанных прерываний на данном ядре (см. раздел 7.4).

Таким образом, на одном ядре CPU выполняется

- *switch_to_user* synchronizes-with *user_int*
- *user_ret* synchronizes-with *user_int*

Для обработки этого отношения, аналогичным с **«core» vclock** образом, в структуре-дескрипторе физического ядра выделяются ещё одни векторные часы, **«user» vclock** (рис. 9). Свойство *async_disable* synchronizes-with *switch_to_user* обеспечивается синхронизацией **«user» vclock** с **«core» vclock**.

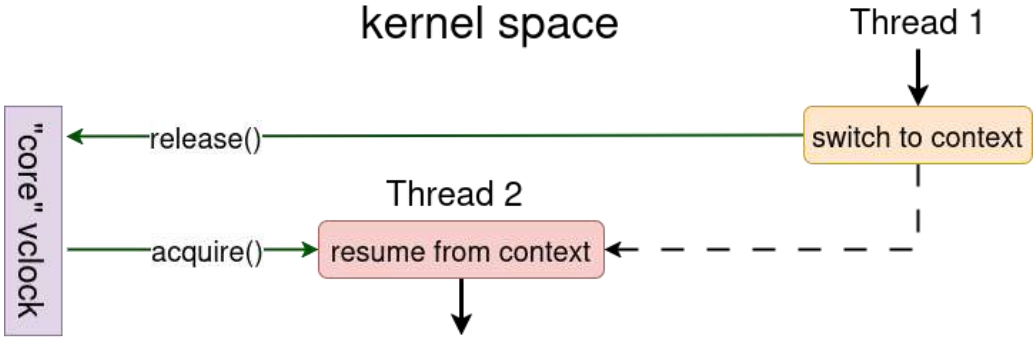


Рис. 8. Переключение контекста.
Fig. 8. Context switch.

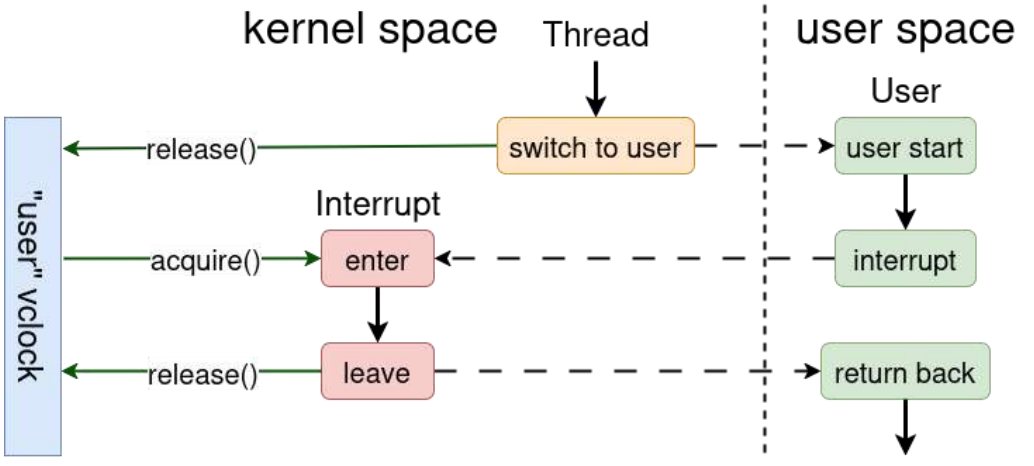


Рис. 9. Пользовательские прерывания.
Fig. 9. Userspace interrupts.

8. Результаты

8.1 Производительность

В связи с приоритетом оптимизации расходов по памяти, адаптированный к коду ядра ОСРВ алгоритм ThreadSanitizer заметно уступает по производительности версии ThreadSanitizer из набора инструментов программной инфраструктуры LLVM. На рис. 10 показано относительное увеличение времени чтения и записи сообщения размером 1024 байта для различных портов (примитивов взаимодействия между процессами в ОСРВ CLOS). Замедление интерфейсов большинства примитивов синхронизации также лежит в пределах от 100 до 150 раз по отношению к их неинструментированным версиям.

Несмотря на приоритет экономии ресурсов памяти над оптимизацией производительности, текущая реализация алгоритма позволяет успешно тестировать код ядра ОСРВ в различных сценариях работы, обнаруживать ошибки многопоточной синхронизации и предоставлять детальную диагностику их места возникновения.

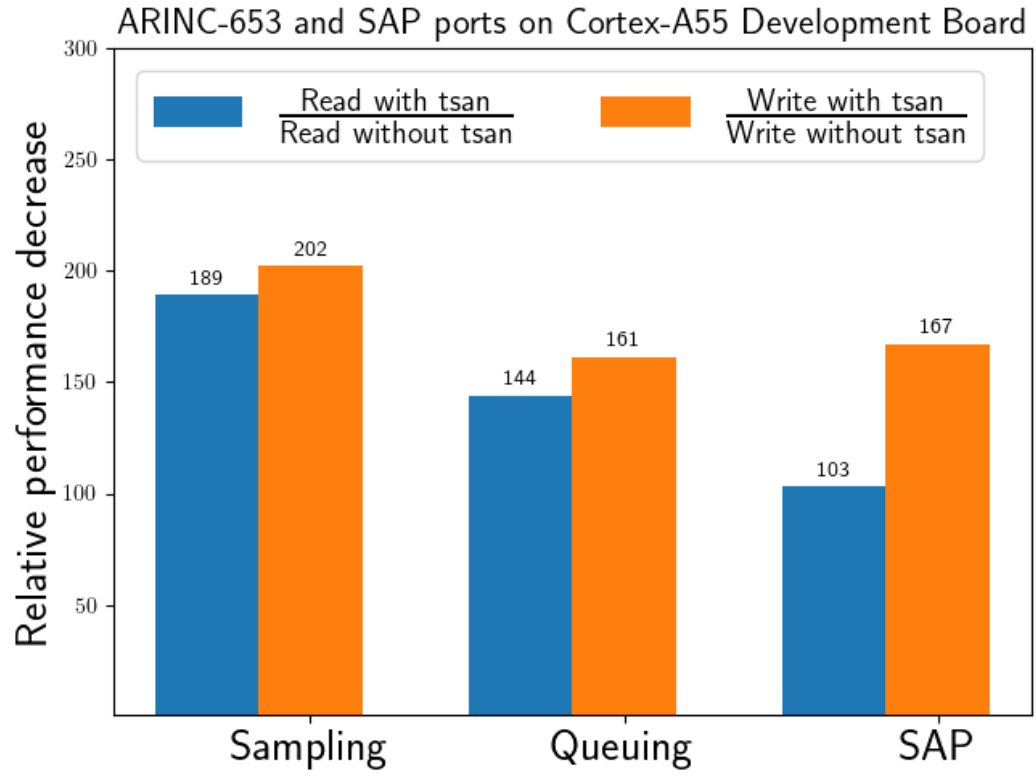


Рис. 10. Замедление процедур интерфейсов ARINC-653 и SAP портов.
Fig. 10. Performance decrease for ARINC-653 and SAP port interfaces.

8.2 Обнаруженные ошибки многопоточной синхронизации

Поскольку до интеграции Thread Sanitizer код ядра ОСРВ CLOS тестировался с применением детектора гонок по данным RaceHunter, основанного на методе точек останова по данным, а также в силу сравнительно небольшого объёма кода, алгоритм Thread Sanitizer способен найти сравнительно немного новых ошибок в коде ядра. Также, частые операции обработки переполнения эпохи теоретически способны привести к необнаружению гонок в некоторых исполнениях.

Несмотря на это, в результате интеграции в систему сборки и тестирования ядра ОС реального времени CLOS алгоритма Thread Sanitizer была подтверждена одна и обнаружены ещё две гонки по данным – как между потоками ядра, исполняющимися на различных процессорных ядрах, так и между кодом потока и обработчиком прерывания на одном ядре (листинг 2).

```
[TSAN] race condition on addr 0x822034e0 :
[tid=9 pattern = ".......X" kind="Regular Write"] ,
[tid=8 pattern = ".......X" kind="Regular Read "]
[TSAN] tid 9 backtrace
[00] 0x0000000009003c8ac
[01] 0x0000000009004c604
[02] 0x0000000090047bdc
[03] 0x0000000009002eac8
...
[09] 0x0000000009001e5c0
[10] 0x0000000000000004
[TSAN] tid 8 backtrace
[00] 0x00000000900329e8
[01] 0x00000000900522b4
[02] 0x0000000090052988
[03] 0x0000000009002eac8
...
[09] 0x0000000009001e5c0
[10] 0x0000000000000000
```

Листинг 2. Сообщение об обнаруженной гонке по данным.
Listing 2. Detailed log message about a data race detected.

Также, для тестирования инструмента в систему тестирования ядра ОСРВ добавлен сценарий, содержащий гонки по данным, и позволяющий проверять корректность работы Thread Sanitizer в коде ядра.

9 Заключение

В данной работе показано, что алгоритм поиска гонок по данным, основанный на последней версии LLVM ThreadSanitizer, может быть адаптирован к интеграции с системой сборки и тестирования ядра ОС реального времени. Возникающие при этом задачи соответствия жестким требованиями на предсказуемость времени исполнения и затраты требуемой памяти имеют решения, позволяющие обнаруживать гонки по данным, оказывая ограниченное влияние на эти свойства исполнения.

В сравнении с ОС Linux, где в качестве инструмента динамического анализа для поиска гонок по данным в ядре был выбран подход, основанный на точках останова по данным (KCSAN), в ОС специального назначения, в частности, в ОС реального времени, проблемы при интеграции детекторов, основанных на построении отношения *happens-before* преодолимы, что позволяет применять оба семейства детекторов гонок по данным, и анализировать больший спектр исполнений.

Список литературы / References

[1]. J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern, Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel, SIGPLAN Not., vol. 53, pp. 405–418, Mar. 2018.

[2]. M. Elver, Concurrency bugs should fear the big bad data-race detector. https://lwn.net/Articles/816850/ , 2020. [Online; accessed 21-May-2025].

[3]. K. Serebryany and T. Iskhodzhanov, Threadsanitizer – data race detection in practice, pp. 62–71, 12 2009.

[4]. ISO Central Secretary, Information technology – Programming Languages – C, Standard ISO/IEC 9899:2024, International Organization for Standardization, Geneva, CH, 2024.
[5]. D. Vyukov, Llvm thread sanitizer. https://github.com/google/sanitizers/wiki/threadsanitizercppmanual, 2020. [Online; accessed 22-May-2025].
[6]. N. Komarov, On the implementation of data-breakpoints based race detection for linux kernel modules, 2013.
[7]. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, Eraser: a dynamic data race detector for multithreaded programs, ACM Trans. Comput. Syst., vol. 15, p. 391–411, Nov. 1997.
[8]. F. Mattern, Virtual time and global states of distributed systems, 01 2004.
[9]. L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM, vol. 21, p. 558–565, July 1978.
[10]. A. Konovalov, Kernel thread sanitizer. https://github.com/google/kernel-sanitizers/blob/master/KTSAN.md , 2015. [Online; accessed 24-April-2025].

Информация об авторах / Information about authors

Егор Сергеевич ЕЛЬЧИНОВ – старший лаборант отдела технологий программирования Института системного программирования. Сфера научных интересов: методы динамического анализа ПО, алгоритмы многопоточной синхронизации, операционные системы.

Egor Sergeevich ELCHINOV – Senior Lab Assistant of the Department of Programming Technologies of the Institute for System Programming of the RAS. Research interests: methods for software dynamic analysis, concurrency algorithms, operating systems.

DOI: 10.15514/ISPRAS-2025-37(6)-39

CC BY

Devirtualization-Based Python Static Analysis

[1] *A.L. Galustov, ORCID: 0009-0001-9591-5873 <artemiy.galustov@ispras.ru>*
[1,2] *K.I. Vihlyantsev, ORCID: 0009-0007-4292-0891 <vikhliantsev.ki@phystech.edu>*
[1] *A.E. Borodin, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1,3] *A.A. Belevantsev, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*

[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn str., Moscow, 109004, Russia.*

[2] *Moscow Institute of Physics and Technology, 9, Institutskiy Pereulok, Dolgoprudny, Moscow Oblast, 141701, Russia.*

[3] *Lomonosov Moscow State University, Leninskie Gory, Moscow, 119991, Russia.*

Abstract. In this paper we present an approach to static analysis of Python programs based on a low-level intermediate representation and devirtualization to provide interprocedural and intermodule analysis. This approach can be used to analyze Python programs without type annotations and find complex defects inaccessible to traditional AST-based analysis tools. Using CPython bytecode as a base, the representation suitable to static analysis is constructed and call resolution is performed via an interprocedural devirtualization algorithm. We implemented the proposed approach in a static analyzer for finding errors in C, C++, Java, and Go programs and achieved good results on open-source projects with minimal modifications to existing detectors. The detectors that are relevant to Python had a true positive rate from 60% up to 96%. This demonstrates that our approach allows to apply techniques used for analysis of statically typed languages to Python.

Key words: static analysis; Python; devirtualization.

For citation: Galustov A.L., Vihlyantsev K.I., Borodin A.E., Belevantsev A.A. Devirtualization-based Python static analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 3, 2025, pp. 109-120. DOI: 10.15514/ISPRAS-2025-37(6)-39.

Статический анализ языка Python с использованием девиртуализации

[1] *Галустов А.Л., ORCID: 0009-0001-9591-5873 <artemiy.galustov@ispras.ru>*
[1,2] *Вихлянцев К.И., ORCID: 0009-0007-4292-0891 <vikhliantsev.ki@phystech.edu>*
[1] *Бородин А.Е, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1,3] *Белеванцев А.А., ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*

[2] *Московский Физико-Технический Институт, 141701, Россия, г. Долгопрудный, ул. Первомайская, д. 5.*

[3] *Факультет ВМК МГУ им. М.В. Ломоносова, 119991, Россия, г. Москва, ул. Колмогорова, д. 1, стр. 52.*

Аннотация. В статье предлагается подход к статическому анализу программ на языке Python на основе низкоуровневого внутреннего представления и девиртуализации, который позволяет выполнять межпроцедурный и межмодульный анализ. Подход применим к программам, не содержащим ручных аннотаций типов, и может быть использован для поиска сложных ошибок, которые не ищутся популярными инструментами на основе анализа АСД. Представление для анализа строится по байткоду CPython, затем в результате работы межпроцедурного алгоритма девиртуализации разрешаются вызовы. Предлагаемый подход к девиртуализации уже реализован для языков C, C++, Java, Go и показал хорошие результаты без необходимости изменения существующих детекторов. После адаптации алгоритма для языка Python доля истинных срабатываний детекторов для Python составила от 60% до 96%. Таким образом, изначально предложенный для статически типизированных языков алгоритм оказался применимым к языку Python.

Ключевые слова: статический анализ; язык программирования Python; девиртуализация.

Для цитирования: Галустов А.Л., Вихлянцев К.И., Бородин А.Е., Белеванцев А.А. Статический анализ языка Python с использованием девиртуализации. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 109–120 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)–39.

1. Introduction

The Python Programming language is consistently one of if not the most popular of the last decade. Popularity indices like TIOBE [1] place it in first by popularity among programmers in the last few years and it remained firmly in top 10 for almost the past two decades. Despite this and the fact that some studies have shown that Python programs may be more prone to software defects then software in other languages [2] there are relatively few static analysis tools for Python. Those that exist can be divided into two broad categories: utilizing AST-based analysis (tools like Pylint [3] and Bandit [4]) and utilizing type annotations (MyPy [5], Pyre [6]). AST based analysis tools may be more popular and easier to implement but they fail to detect more complex interprocedural and intermodular defects. Tools that utilize type annotations are better suited to find complex defects but are limited by the fact that vast majority of Python programs don't utilize type annotations as indicated by [7] which discovered that less than 4% of open-source projects on GitHub use type annotations in their projects.

In this paper we present our approach to static analysis of Python programs based on low-level intermediate representation using devirtualization algorithm to provide precise analysis. Our approach allows to leverage existing detectors implemented in Svace [8-9] – a static analysis tool aimed at finding complex defects in source code of programs on a variety of languages. Currently Svace supports C, C++, Java, Kotlin, Scala, and Go. All of these are statically typed compiled languages.

2. Building IR

To work with a wide variety of different languages Svace relies on a low-level intermediate representation Svace IR. This representation is represented as a Control Flow Graph (CFG) where instructions are in Static Single Assignment form (SSA) and branch conditions are denoted as assume instructions in target blocks. List of instructions relevant to Python analysis is presented in Table 1.

Python source files must be compiled into a suitable format that can then be transformed into Svace IR. *CPython bytecode* is well-suited for this purpose due to its similarity in structure and semantics to Svace IR. The main problem with this representation is lack of stability. For example, Python 3.11 introduced a new way of handling exceptions incompatible with the old one. Because of this we restrict Python version to *3.12* and focus on specifics of this version in this paper.

To obtain the bytecode, we utilize the standard Python library module **dis** [10], which provides a straightforward way to read and manipulate the bytecode. Moreover, by incorporating specific additional instructions into Svace IR, we can achieve full compliance with Python's semantics, further solidifying its utility for our analysis tasks.

Svace IR is constructed in two phases. First source directories are traversed recursively and for each Python source file the bytecode is saved to a file without any significant modification. Additionally directory structure that contains files found is also saved to facilitate import resolution (see section 3.5). Second when analysis starts bytecode is read from these files and transformed to Svace IR. This is where most of transformations outlined below take place.

Table 1. Main Svace IR instructions.

`a = alloca()`	memory allocation
`a = b`	SSA-assignment
`a = *b`	pointer dereference
`*a = b`	pointer assignment
`a = b.field`	read object attribute
`a.field = b`	write attribute
`a = makeclosure func, (`c_0`, `c_1`, …)`	closure/lambda creation
`a = ptr (`a_0`, `a_1`, …)`	virtual function call
`assume condition`	condition true on the execution path

2.1 SSA Form

Svace operates on an intermediate representation in Static Single Assignment (SSA) form, whereas Python bytecode does not conform to this format. Consequently, a translation process is necessary to reconcile this discrepancy. To achieve this, we employ a two-stage approach. Initially, we perform analysis of the string table within the bytecode, focusing on identifying variables that appear on the left-hand side of assignment operations with multiplicity greater than one. These variables are modeled via references to memory allocated via **alloca** instructions at the start of function. Single-instance variables are directly represented via variables in Svace IR. The outcome of this pass is a symbol table that serves as the foundation for constructing the intermediate representation. Subsequently, we execute a bytecode-to-Svace IR conversion process, where each instruction is individually modeled to accommodate its SSA-form requirements, effectively transforming the original bytecode into Svace IR. An example of source code, its Python bytecode and resulting Svace IR can be seen in fig. 1.

Additional consideration is global and non-local (captured) variable handling. At the level of Python bytecode all variables accessed using **STORE_NAME** and **LOAD_NAME** in module functions are

considered global and in normal functions special **LOAD_GLOBAL** and **STORE_GLOBAL** are used. Unlike Python bytecode Svace IR does not have special instruction for global variables handling and instead has special type of global symbols which are equal when used in different functions. Therefore, when converting to SSA form a special context is maintained per Python source file which allows tracking of all global variables across different functions. Non-local variables behave differently. Unlike globals, they are not just available to all inner functions. Instead, each **MAKE_FUNCTION** accepts a tuple argument with all captured variables. This information is saved in the **makeclosure** Svace IR instruction and is later used by devirtualization and main analysis.

Python source	Python bytecode	Svace IR

`def foo(n):`	`RESUME`	`n = alloca ()`
		`*n = arg_n`
		`s = alloca()`
		`return_value = alloca()`
		`*return_value = None`

`if n < 0:`	`LOAD_FAST (n)`
	`LOAD_CONST (0)`
	`COMPARE_OP (>)`
	`POP_JUMP_IF_FALSE (to 16)`
		`n_1 = *n`

`s = n`	`LOAD_FAST (n)`
	`STORE_FAST(s)`
		`assume n_1 > 0`
		`n_2 = *n`
		`*s = n_2`

`s += 2`	`LOAD_FAST_CHECK (s)`
	`LOAD_CONST (2)`
	`BINARY_OP (+=)`
	`STORE_FAST (s)`
		`s_1 = *s`
		`s_2 = s_1 + 2`
		
		`*s = s_2`

`return s`	`LOAD_FAST (s)` `RETURN_VALUE`	`s_3 = *s` `*return_value = s_3`

Fig. 1. Python source code translation example.

2.2 Exception handling

Exception handling in Python is done using an auxiliary structure called *exception table*. This structure stores the start and end offsets of the instruction blocks that can throw an exception and corresponding offset of block that handles the exception. During evaluation if an exception occurs and exception tables has a matching range, the code execution goes to the handler. However, this approach has several problems when converting to Svace IR: this creates implicit control flow not represented by any instructions and this potentially means that each instruction inside block defined in exception table can throw an exception. Solution to first problem is creating an explicit split in control flow graph (CFG) after each instruction and adding **assume** instructions in each execution path that denote whether this is normal or exception path (fig. 2) [11].

This approach is very convenient for static analysis but exacerbates the second problem, adding a split in CFG after each instruction in try block leads to extremely complex graph which will slow analysis down significantly. At the same time while many instructions in Python bytecode may throw exception this is not modelled the same way in Svace IR. Instead, we opted to assume that only call instructions may throw an exception (fig. 3).

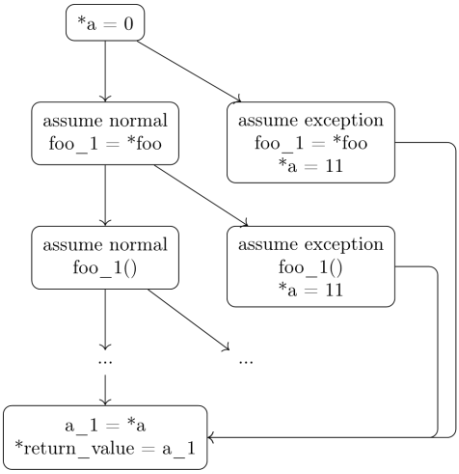


Fig. 2. CFG for try-except.

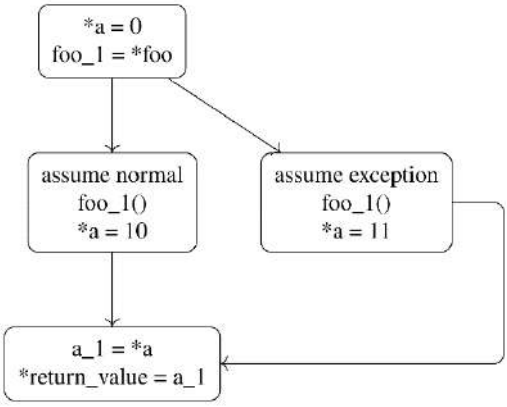


Fig. 3. Simplified CFG for try-except.

2.3 Classes and object instantiation

Classes in Python are special functions which when called produce an instance of the class [12, page 3.2.8.8]. Additionally behavior of instance creation can be modified with special methods like **__init__** and **__new__**. As specified in Python reference when class object is called first instance of class is created, possibly via the **__new__** method and then **__init__** method is invoked in instance. However, the **__call__** method of class object is not contained anywhere in CPython bytecode. Consider the following example:

```
class Example:
    a = 1
    def __init__ (self, b, c):
        self.b = a
        self.c = b
    def foo (self):
        print (self.a)
```

Bytecode generated for this example will contain a **<class Example>** block that initializes methods and fields, and functions **__init__** and **foo**. However, the factory function that corresponds to the

Example itself is not present (note that it is not the same as **<class Example>**, it does not invoke **__new__** or **__init__**). Actually, that function is generated using a special **__build_class__** built-in that is written in C[1] that accepts **<class Example>** function as input and produces class object. The **__call__** method of class is also a built-in function written in C (known as **type_call**) that is common for all classes (unless custom metaclass is used) [12, page 3.3.3.1].

This behavior is quite complex and will not work well with algorithms insensitive to caller context such as devirtualization algorithm used in Svace (see Section 3). Instead, our analysis models this by adding the missing **type_call** function for each class to Svace IR during build process and reproduce required behavior in it:

```
def Example (a, b, c):
        class = < class Example > ()
        instance = class.__new__ ()
        instance.__init__ (b, c)
        return instance
```

Adding such a function for each class (not just a common one as part of metaclass implementation) improves analysis accuracy by removing the need for flow and context sensitivity to analyze class creations.

3. Analyzing calls and imports

As previously mentioned, our solution does not construct any call graph while building IR. Other tools also use algorithms separate from construction of internal representation (be that some low level SSA form or AST) to perform two tasks: import resolution and qualified name resolution. This algorithms by themselves are quite difficult to implement due to intricacies of Python scoping rules and dynamic nature of language. However, they are only effective at working with calls to "basic" function not giving any insight into method calls or lambda invocations. Using typing annotation is more useful but as mentioned earlier small amount of Python programs use them.

We argue that more suitable approach is to use a virtual call resolution algorithm, i.e., devirtualizing for all functions. Svace already has a robust and extensible algorithm for multilingual devirtualization [13]. However, Python differs significantly from any languages that Svace already supported in two key ways: dynamic typing and no static linking. This warrants some additions to devirtualization algorithm.

Devirtualization algorithm in Svace is performed in two parts [13]:

1. Analysis on individual functions building *summaries*. Summaries are language agnostic representation of type aliases in functions and dependencies or updates to values external to the function. For example, assigning or reading global variables, using function arguments or calling other functions etc. When a summary is constructed each variable in a function is assigned to multiple alias classes of two possible types: direct and transitive. A direct alias represents that a variable contains a certain function, an instance of certain class etc., while a transitive alias represents that variable receives data from some external function.
2. Iterative algorithm that manipulates a dependency graph of function summaries to resolve global interprocedural dataflow. Summaries are processed to see which global values are updated by this summary and then algorithm proceeds to update function summaries affected and so on until algorithm eventually converges.

To add support for Python in this algorithm Python specific constructions need to be converted to this language-agnostic representation.

[1] This built-in function is so special that it even has a dedicated **LOAD_BUILD_CLASS** opcode.

3.1 Functions and lambdas

Python is often described as a functional programming language [14]. This is often attributed to first-class function support like lambdas. However, at Python bytecode level all functions, not only lambdas but also ones defined with **def** keywords are treated as values. This is in stark contrast to languages like C/C++, Go and others where distinction between regular calls and calls to function pointers, virtual methods etc. In fig. 4 CPython bytecode and Svace IR for simple function call is presented.

Python source	Python bytecode	Svace IR

`def foo:`
`  ...`
	`LOAD_CONST (<foo>)`
	`MAKE_FUNCTION`
	`STORE_FAST (foo)`
		`mkcls_res = makeclosure foo`
		`*foo = mkcls_res`

`foo()`	`LOAD_CONST (<foo>)`
	`MAKE_FUNCTION`
	`STORE_FAST (foo)`
		`foo_1 = *foo`
		
		`foo_1()`

Fig. 4. Simple function call IR.

MAKE_FUNCTION instruction produces an object corresponding to the function which is stored in variable **foo**. Any subsequent calls to this function will retrieve value from variable to stack and **CALL** uses this object to invoke function. This is the only way to create and invoke functions in Python[2]. This means that devirtualization in Python has one source of direct aliases **makeclosure** Svace IR instruction and one type of call virtual call of functional object, represented as pointer call Svace IR instruction. When **makeclosure** instruction is encountered result is assigned an alias of respective function. While this seemingly simplifies IR over languages like C/C++, Go and JVM-based languages where multiple types of calls can occur [13] the fact that even the simplest cases like the one above require dataflow analysis to resolve increases complexity of devirtualization for Python.

3.2 Captured variables

As mentioned above Python has a special way of passing data to a function in place where it is created via captured variables. This data is associated with created function object in **makeclosure** instruction. During execution the data stored in object is passed to function only at call site. However, in context- and flow-insensitive devirtualization algorithm this data can be passed directly to the **makeclosure** instruction. Just like an external dependency is established for arguments of **pcall** an external dependency is established for each captured variable passed to the created function. This removes the need to track captured variables along with the created function when modelling dataflow, but provides correct results due to context-insensitivity of analysis.

3.3 Attributes

Modelling composite objects in dataflow analysis can be done in many different ways. The most precise approach is to model both individual objects and their individual components. This is done during the main analysis in Svace but this is quite a heavy approach that does not scale well to global dataflow analysis algorithms. To simplify one can either treat all components of object as one (this is useful for modelling arrays, treating all elements of array as one) or treat all instances as the same object (this is useful for modelling classes, structures etc.). Devirtualization algorithm treats all fields of the same type as the same location. This works well in languages like C/C++, Java and Go where

[2] As mentioned earlier Python bytecode utilizes two instructions to denote calls: **CALL** and **CALL_KW**. In Svace both are translated to pointer call instruction.

type of object field of which is accessed is known due to static typing. In Python type is not known and needs to be inferred. Consider the following example:

```
class Data:
    pass
def assign (obj):
    obj.a = "string"
def read (obj):
    a = obj.a
    print (a.islower ())
obj = Data ()
assign (obj)
read (obj)
```

Here the algorithm first infers that **obj** argument of both **assign** and **read** functions is an alias to class **Data**. Then during analysis of **assign** function an update to all readers of field **a** of class **Data** is issued notifying them that field now aliases class **str** and finally **read** function is analyzed again and can resolve the call to **islower** method as type of **obj.a** is resolved. This also seamlessly handles that Python attributes (unlike fields in statically typed languages) can be assigned or read without any prior declaration. Note that if calls to **assign** and **read** receive objects of different types:

```
assign (Data())
read (OtherData())
```

then call to **a. islower ()** will not be resolved due to types of **obj** being different, as expected.

3.4 Classes

As mentioned earlier in Section 2.3 classes are initialized by a complex sequence of function invocations and more specifically methods of classes are initialized in a special **<class>** function by assigning function to respective attributes of class. The fact that this function is unique per class declared in program makes it a good identifier for class type alias used in devirtualization. Thus concrete aliases to class type reference respective **<class>** function and return values of special constructor functions generated for each class are assigned this alias. Any variable assigned in **<class>** function (using **STORE_NAME** instruction) are automatically assigned to respective attribute as illustrated in fig. 5.

Python source	Python bytecode	Svace IR
`class Example`	`RESUME`	
`a = 1`	`LOAD_CONST (1)` `STORE_NAME (a)`	`clazz.a = 1`

`def __init__():`
`  ...`
	`LOAD_CONST (<__init__>)`
	`MAKE_FUNCTION`
	`STORE_NAME (__init__)`
		`mkcls_res_1 =`
		`makeclosure __init__`
		`clazz.__init__ =`
		`mkcls_res_1`

`def foo(self)`	`LOAD_CONST (<foo>)`
	`MAKE_FUNCTION`
	`STORE_NAME (foo)`
		`mkcls_res_2 =`
		`makeclosure foo`
		`clazz.foo =`
		`mkcls_res_2`

Fig. 5. Class declaration translation example.

Consequently, method call **c. foo(args)** in Svace IR is represented as two instructions: **c_foo = c. foo** and **c_foo(args)**[3]. This means that previously described approach to modelling attributes models Python method semantics perfectly without the need to additionally model method tables like in C++, Java and Go where methods are a distinct entity [13]. This approach is also able to model some Python specific tricks like overwriting methods after object creation (a.k.a "monkey patching"):

```
class Example:
    def foo (self):
        pass

o = Example ()
o.foo = lambda: print ("Hello, world!")
```

In this example there are two assignments to **foo** attribute of **Example** class. One is a method that does nothing and one is a lambda that prints "Hello, world!". This is a valid Python code and is sometimes used in real Python project as an alternative to overriding methods. Our devirtualization algorithm can handle this situation correctly because it is no different semantically than method assignment in class creation process.

3.5 Module imports

The final piece to full Python analysis is intermodularity. In the most basic case the **import** statement receives a single argument, which is a path to the module, and binds it to the current function as a variable [12, page 5]. Imported modules in Python are also regular objects with attributes. Each module gets its own type based on **<module>** function for direct aliases encoding type information of corresponding object (just as each class does based on its **<class>** function). Each assignment to variable in module function is also modeled as an assignment to attribute of module with the same name. Then when encountering import instructions result must get an alias to corresponding module. As mentioned earlier information about directory structure is saved during the build phase. This information is stored as a filesystem tree and when an import instruction is processed, this tree is traversed according to rules outlined in Python documentation [12, page 5.5] and respective module is selected. Then the result of import call is assigned alias to appropriate module. More complex forms of imports such as **import a from b as c** then can be represented as a simple import and number of attribute reads for "import from" statements and assignments for "as" directives.

Similarly to class modelling this makes devirtualization able to deal with "monkey patching" and also other unusual applications of modules when they are treated as regular objects. For example, modules that are passed as function arguments:

```
def foo (obj):
    return obj.urlopen ("https://example.com")

def bar():
    import urllib.request
    return foo (urllib.request)
```

Here **urllib.request** argument is modeled as an alias to **<module request>** object and in **foo** function attribute **urlopen** is resolved to correct function.

[3] This corresponds to CPython opcodes **LOAD_ATTR** and **CALL**. Interestingly before Python 3.12 special **LOAD_METHOD** and **CALL_METHOD** opcodes that combined the two existed.

4. Results

To test the approach, we have analyzed a set of open source projects of various sizes ranging from tens of thousands to one and a half million lines of code. Projects, their versions and source code sizes are outlined in Table 2.

Table 2. Analyzed projects.

Project	Version	Size (KLOC)
home-assistant/core	2023.9.1	1537
pytorch	2.5.1	1304
tensorflow	2.18.0	926
plotly.py	5.14.1	747
cpython	3.11.5	709
django	4.2.5	372
jax	0.4.14	182
matplotlib	3.8.0	180
fastapi	0.101.1	83
ipython	8.12.2	54

To perform analysis a server with 16 CPU cores and 64 Gb of RAM was used. Warnings for a set of detectors were reviewed and results are presented in Table 3. Detectors chosen are the ones that do not depend on particular details of analyzed language and thus could be used for Python without any major modifications. Additionally, Table 4 contains devirtualization algorithm performance statistics for various projects.

Large number of emitted warnings and high rate of true positives demonstrates that approach to analysis of Python presented in this paper can effectively transform Python programs in a form suitable for complex static analysis using the same tools that are applied to statically typed languages like C, C++, Java etc. that are handled well by Svace.

Table 3. Results summary.

Warning	Total warnings	TP	TP%
DEREF_AFTER_NULL	47	23	82%
DEREF_OF_NULL	94	38	64%
DIVISION_BY_ZERO	235	133	78%
NULL_AFTER_DEREF	115	47	64%
REDUNDANT_COMPARISON	358	147	75%
TAINTED_PTR	110	91	96%
UNUSED_FUNC_RES	456	224	93%
UNUSED_VALUE	448	190	81%

5. Conclusion

This paper describes an approach to static analysis of Python programs. Using specific techniques during intermediate representation construction and applying a devirtualization algorithm allows to create representation similar to those generated for statically typed languages and is a critical

component for interprocedural and intermodular analysis. This representation can be efficiently used by existing static analysis tools such as Svace to detect many types of errors in programs without any additional modifications to existing detectors.

Table 4. Devirtualization statistics.

Project	Total time (sec)	Devirt time (sec)	Devirt time %	Virtual call number	Resolved	Resolved %
home-assistant/core	504	13.0	2.5%	805260	53330	6.6%
pytorch	644	15.9	2.4%	705661	46484	6.6%
tensorflow	370	9.1	2.5%	601024	68880	11.4%
plotly	404	6.9	1.7%	124635	14808	11.8%
cpython	626	11.5	1.8%	449269	24903	5.5%
django	282	7.1	2.5%	220678	15955	7.2%
jax	215	6.1	2.8%	120423	7717	6.4%
matplotlib	157	3.5	2.2%	82679	12503	15.1%
fastapi	68	1.3	1.9%	19359	1634	8.4%
ipython	90	1.4	1.5%	21897	1484	6.7%
scrapy	34	0.8	2.3%	27120	1232	4.5%

References

[1]. Tiobe index. Available at: https://www.tiobe.com/tiobe-index/ (accessed 23.02.2025).

[2]. B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pp. 155-165, 2014.

[3]. Pylint documentation. Available at: https://pylint.readthedocs.io/en/latest/ (accessed 03.04.2025).

[4]. Bandit documentation. Available at: https://bandit.readthedocs.io/en/latest/ (accessed 03.04.2025).

[5]. Mypy - Optional Static Typing for Python. Available at: https://mypy-lang.org/ (accessed 03.04.2025).

[6]. Quickstart | Pyre – pyre-check.org. Available at: https://pyre-check.org/docs/pysa-quickstart/ (accessed 03.04.2025).

[7]. I. Rak-Amnouykit, D. McCrevan, A. Milanova, M. Hirzel, and J. Dolby. Python 3 types in the wild: a tale of two type systems. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, pp. 57-70, 2020.

[8]. Ivannikov V., Belevantsev A., Borodin A., Ignatiev V., Zhurikhin D., and Avetisyan A. Static analyzer svace for finding defects in a source program code. *Programming and Computer Software*, 40(5). pp. 265-275, 2014.

[9]. A. Borodin and I. Dudina. Intraprocedural Analysis Based on Symbolic Execution for Bug Detection. *Programming and Computer Software*, 47(8), pp. 858-865, 2021.

[10]. Dis – disassembler for python bytecode. Available at: https://docs.python.org/3.12/library/dis.html (accessed 03.04.2025).

[11]. Афанасьев В.О., Дворцова В.В., and Бородин А.Е. Статический анализатор для языков с обработкой исключений. Труды Института системного программирования РАН, 34(6):7-28, 2022. / Afanasyev V.O., Dvortsova V.V., Borodin A.E. Static analysis for languages with exception handling. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 6, 2022. pp. 7-28 (in Russian). DOI: 10.15514/ISPRAS-2022-34(6)-1.

[12]. The python language reference. Available at: https://docs.python.org/3.12/reference/ (accessed 03.04.2025).

[13]. A. Galustov, A. Borodin, and A. Belevantsev. Devirtualization for static analysis with low level intermediate representation. In *2022 Ivannikov Ispras Open Conference (ISPRAS)*, pp. 18-23. IEEE, 2022.

[14]. G. Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41 of number 1, pp. 1–36. Santa Clara, CA, 2007.

Информация об авторах / Information about authors

Артемий Львович ГАЛУСТОВ – магистр, стажёр-исследователь ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Artemiy Lvovich GALUSTOV – masters graduate, researcher at ISP RAS. Research interests: static analysis for finding errors in source code.

Константин Игоревич ВИХЛЯНЦЕВ – бакалавр Московского физико-технического института (факультет радиотехники и кибернетики). Научные интересы включают статический анализ и профилирование динамических языков программирования.

Konstantin Igorevich VIHLYANTSEV – master’s student at the Moscow Institute of Physics and Technology (Faculty of Radio Engineering and Cybernetics). His research interests include static analysis and profiling of dynamic programming languages.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Alexey Evgenevich BORODIN – Cand. Sci. (Phys.-Math.), researcher. Research interests: static analysis for finding errors in source code.

Андрей Андреевич БЕЛЕВАНЦЕВ – доктор физико-математических наук, член-корреспондент РАН, ведущий научный сотрудник ИСП РАН, профессор кафедры системного программирования ВМК МГУ. Сфера научных интересов: статический анализ программ, оптимизация программ, параллельное программирование.

Andrey Andreevich BELEVANTSEV – Dr. Sci. (Phys.-Math.), Prof., corresponding Member RAS, leading researcher at ISP RAS, Professor at Moscow State University. Research interests: static analysis, program optimization, parallel programming.

DOI: 10.15514/ISPRAS-2025-37(6)-40

CC BY

PereFlex: A Tool for Automated Evaluation of Error Recovery in Parsers

[1] *O.I. Bachishche, ORCID: 0009-0007-5247-4362 <bachisheo@yandex.ru>*
[2] *Y.S. Vorobiev, ORCID: 0009-0007-5271-5066 <yaroslav.vorobev-2015@mail.ru>*
[1] *G.R. Raykin, ORCID: 0009-0004-7783-7818 <gregra@mail.ru>*
[1] *D.V. Vasina, ORCID: 0009-0001-0728-956X <dashavasina625@gmail.com>*
[1] *D.S. Shushakov, ORCID: 0009-0006-9546-0174 <shushakov4@ya.ru>*
[3] *S.V. Grigoriev, ORCID: 0000-0002-7966-0698 <s.v.grigoriev@spbu.ru>*

[1] *ITMO University, Kronverksky Pr. 49, bldg. A, St. Petersburg, 197101, Russia.*

[2] *HSE University, 11 Pokrovsky Bulvar, Moscow, 109028, Russia.*

[3] *St. Petersburg State University,*
7-9, Universitetskaya Embankment, St Petersburg, 199034, Russia.

Abstract. Error recovery is a critical component of parsing technology, particularly in applications such as IDEs and compilers, where a single syntax error should not prevent further analysis of the input. This paper presents PereFlex – a tool for extensive experimental evaluation of error recovery in JVM-based parsers. Our evaluation is based on real-world parsers for Java and users' erroneous programs. The results demonstrate that while some strategies are fast, they often fail to provide meaningful recovery, whereas advanced methods offer better recovery quality at the cost of increased computational overhead.

Keywords: error recovery; parsing; IDE; evaluation.

For citation: Bachishche O.I., Vorobiev Y.S., Raykin G.R., Vasina D.V., Shushakov D.S., Grigoriev S.V. PereFlex: A tool for automated evaluation of error recovery in parsers. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 3, 2025, pp. 121-132. DOI: 10.15514/ISPRAS-2025-37(6)-40.

PereFlex: инструмент для автоматической оценки восстановления после ошибок в синтаксических анализаторах

[1] *О.И. Бачище, ORCID: 0009-0007-5247-4362 <bachisheo@yandex.ru>*
[2] *Я.С. Воробьев, ORCID: 0009-0007-5271-5066 <yaroslav.vorobev-2015@mail.ru>*
[1] *Г.Р. Райкин, ORCID: 0009-0004-7783-7818 <gregra@mail.ru>*
[1] *Д.В. Васина, ORCID: 0009-0001-0728-956X <dashavasina625@gmail.com>*
[1] *Д.С. Шушаков, ORCID: 0009-0006-9546-0174 <shushakov4@ya.ru>*
[3] *С.В. Григорьев, ORCID: 0000-0002-7966-0698 <s.v.grigoriev@spbu.ru>*

[1] *Университет ИТМО,*
Россия, 197101, г. Санкт-Петербург, Кронверкский проспект, д.49, литер А.
[2] *НИУ ВШЭ, Россия, 109028, г. Москва, Покровский бульвар, д. 11.*
[3] *СПбГУ, Россия, 199034, г. Санкт-Петербург, Университетская наб., д. 7-9.*

Аннотация. Восстановление после ошибок – один из ключевых компонентов технологии синтаксического анализа, особенно в таких приложениях, как IDE и компиляторы, где синтаксические ошибки не должны блокировать анализ входных данных. В данной статье представлен PereFlex – инструмент для экспериментальной оценки восстановления после ошибок в анализаторах, работающих на платформе JVM. Оценка основана на реальных парсерах для Java и ошибочных пользовательских программах. Полученные данные указывают на обратную зависимость между скоростью работы и качеством восстановления: продвинутые методы, обеспечивающие качественное восстановление, сопряжены с высокой вычислительной сложностью.

Ключевые слова: восстановление после ошибок; синтаксический анализ; IDE.

Для цитирования: Бачище О.И., Воробьев Я.С., Райкин Г.Р., Васина Д.В., Шушаков Д.С., Григорьев С.В. PereFlex: инструмент для автоматической оценки восстановления после ошибок в синтаксических анализаторах. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 121–132 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)–40.

1. Introduction

An integrated development environment (IDE) is a tool designed to assist developers in writing code efficiently. The features and inspections provided by an IDE are largely based on the abstract syntax tree (AST), which serves as a structural representation of the source code generated by a parser or syntax analyzer. During the coding process, programmers do not always maintain syntactically correct code.

To work effectively under such circumstances, IDE parsers must be capable of recovering from errors and generating an AST that is, if not entirely accurate, at least a reasonable approximation of the correct structure. Efficient error recovery ensures that minor syntax errors do not interrupt the development process [1], enabling features such as autocompletion, real-time syntax highlighting, and quick fixes to function smoothly.

Evaluation of error recovery is important from both scientific and practical standpoints. Parser developers need reliable benchmarks and metrics to compare new recovery algorithms against existing solutions. Likewise, practitioners who build tools such as IDEs, linters, or static analyzers must be able to choose a parser that meets their needs. This requires access to measurable, reproducible information about how well a parser handles erroneous input. Without a clear evaluation framework, it is difficult to make informed decisions or track progress in error recovery techniques.

Despite its practical importance, evaluating the quality of error recovery remains a challenging and largely open problem. First, there is no standardized metric for measuring recovery quality. Different approaches employ a variety of techniques, including analysis of error type distributions [2], exact

match metrics [3], and manual assessment [4-5]. Second, not all metrics are universally applicable. Some are suited to specific parsing algorithms [6], while others depend on the structure of the recovered AST [5, 7]. Different parsers often produce different ASTs for the same incorrect input. This variation arises from differences in their grammars, error recovery strategies, and design decisions. As a result, directly comparing parsers becomes a challenging task.

In addition to metric design, dataset construction presents another fundamental challenge. Many prior studies rely on synthetic datasets, fully generated or partially modified by mutators [5, 7-8]. These datasets make it possible to automatically label error types and run controlled experiments. However, they can create error patterns that do not reflect real-world code, which limits how well the results apply in practice. To address this, recent studies [1, 6, 8] have started using datasets of real code written by developers [9]. These datasets provide more realistic testing conditions and lead to more reliable evaluations.

This work addresses these challenges by introducing a unified way to evaluate parser error recovery. The proposed method does not depend on a specific parsing technique or AST representation and can be applied to a wide range of real-world codebases.

To ensure an efficient and systematic evaluation, we first review existing approaches to assessing error recovery quality. Building upon this foundation, we present a novel tool that automatically evaluates the error recovery quality of parsers targeting the JVM platform. Notably, the tool can operate without access to a parser's internal structure, including its parsing algorithm or specific error representations.

Additionally, we analyze the error recovery capabilities of widely used JVM-based parsers integrated into IDEs such as VS Code and Eclipse. To facilitate benchmarking, we have annotated a real-user dataset [9] of Java source files containing syntactic errors.

This research contributes to the field of parsing and syntax analysis in the following ways.

- **Evaluation Metrics**: We review existing approaches to error recovery evaluation and summarize their advantages and limitations.
- **Tool Implementation**: We present **PereFlex** [10] (**Flex**ible **P**arser **E**rror **R**ecovery **E**valuation), a tool that computes a subset of these metrics for any given JVM-based parser without requiring internal access to its implementation.
- **Analysis of Real-World Parsers**: We apply PereFlex to evaluate both generated and standalone parsers used in real-world development environments, including Visual Studio Code [11] and Eclipse [12], providing insights into their recovery quality and limitations.

2. Evaluation Methodology

Many techniques exist for evaluating error recovery [13]. We limit our scope to parser recovery in general, excluding lexer-level errors.

2.1 Quality Metrics

The central question in quality assessment is how closely the recovered input matches the original user input or a correct reference version. Several evaluation strategies are widely used.

1) The simplest approach involves **manual evaluation** [4-5]. However, this method is time-consuming and prone to subjective bias. Additionally, developers may be influenced by error-messages output from the tools used during verification [6]. Despite its limitations, manual inspection remains a valuable method, particularly in cases where no baseline file with corrected errors is available. To enhance precision, some studies classify recovery quality using coarse categories such as "excellent", "good", and "poor" [4]. However, this classification lacks granularity and does not allow detailed analysis.

2) **AST comparison** assesses structural similarity between recovered and expected syntax trees [5, 7]. This method is accurate but sensitive to AST format differences and parser-specific representations. This method also requires a dataset with reference recovered code to compare trees in the same format.
3) **Edit distance** [14] quantifies the number of modifications needed to transform one string into another. In error recovery algorithms, particularly in syntax analysis, edit distance is employed to identify the optimal recovery path with minimal data loss [6]. While useful, it may not fully reflect structural differences in complex codebases, requiring careful application.
4) **Exact match** [15] is 1 if the recovered code exactly matches the marked target code, and 0 otherwise. Because it is based on a labeled dataset, false positives may occur when performing alternative but syntactically correct fixes.
5) **Cascade errors**, where one error triggers additional parsing failures, are measured by counting total error locations [6]. Yet, fewer errors do not always imply better recovery, as some parsers may miss detecting them entirely.
6) **First-error stopping** [2, 16] limits recovery assessment to the initial error, ignoring parser behavior for subsequent code, which is vital in IDE scenarios.
7) **Error type distribution** [2] reflects the frequency and kinds of errors encountered. Though often mixing compile-time and parsing errors for Java code, it reveals patterns that highlight parser weaknesses.

In our approach, we selected **edit distance** and **error distribution** as our primary evaluation metrics. Edit distance provides a general, language-independent measure of how well the recovered input matches the original one, while error distribution highlights systematic flaws in recovery strategies, providing insight into the parser's robustness and practical utility.

2.2 Performance and Memory Usage

The most straightforward way to evaluate recovery performance and memory consumption is to measure the difference between parsing benchmarks for a correct file and the same file containing errors [5, 7]. This approach eliminates common overhead costs but is only applicable to generated datasets where an ideal solution is known. When working with real-world datasets, performance trends must be compared across different parsers rather than against a predefined baseline.

In some cases, performance measurement can be further refined. For instance, if recovery is a distinct module within a parsing algorithm, its execution time can be measured separately [6]. However, this is only applicable to parsers with dedicated recovery modules.

2.3 JVM-specific Aspects of Evaluation

In this study, we analyze the performance of the JVM (Java Virtual Machine). Our methodology is based on "Pro.NET Benchmarking" [17], which provides an approach to experiment design, statistical analysis, and bias mitigation that can be applied to any virtual machine.

- **Use release builds with optimizations**: this ensures that benchmarks reflect real-world tool performance.
- **Prevent compiler optimizations from eliminating measurements**: partial results should be stored in variables to ensure correctness.
- **Repeat benchmarks multiple times**: each performance measurement should be executed at least 30 times, with 5–10 warm-up iterations to stabilize results.
- **Trigger garbage collection (GC) between measurements**: since GC can introduce variability, forcing a collection cycle before measurement ensures consistency.

Since this study focuses on evaluating JVM-based parsers, memory measurement must account for JVM-specific constraints. The most critical metric is peak memory usage during parsing. JVM-based programs do not have a fixed point where peak memory usage is guaranteed, as GC can reclaim memory at any time.

One possible solution is to use specialized profiling libraries such as JMH [18]. However, these tools are designed for microbenchmarking and are not well suited for experiments involving large datasets or extensive computations. An alternative approach is to constrain the JVM's maximum memory allocation and use binary search to determine the minimum required memory for processing a given file.

In summary, a tool for reliable parser evaluation on the JVM requires awareness of its runtime behaviors, especially GC and JIT optimizations.

3. Error Recovery Analysis Tool for JVM Platform

PereFlex provides an efficient and structured way to benchmark error recovery capabilities across different parsers on the JVM. It operates as a console application, enabling users to evaluate multiple parsers for different programming languages with precise control over execution parameters.

3.1 System Architecture

The high-level architecture of the benchmarking tool consists of the following key components shown in Fig. 1.

- **RecoveryAnalyzer** An API for evaluating any JVM parser in a benchmark system and its implementation for different parsers.
- **Benchmark Runners**: Kotlin module that performs multiple measurement algorithms with a given implementation of *IRecoveryAnalyzer* and dataset. This module can be extended by users.
- **Data Representations**: Python library processes the collected benchmark data. This module generates graphs for a specific parser or compares graphs for all analyzers and calculates some statistics to obtain test results.

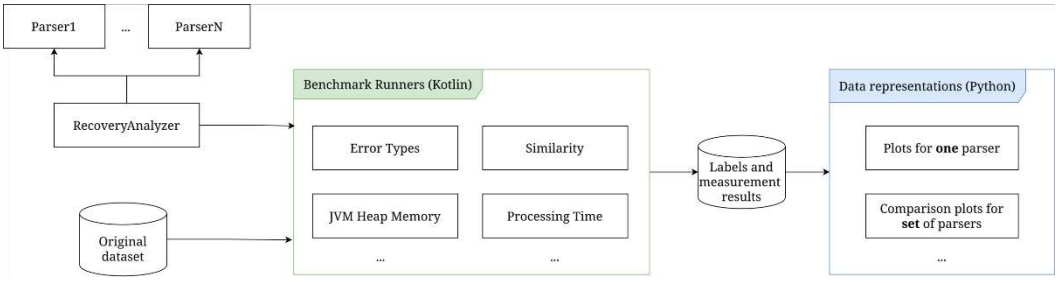


Fig. 1. High-level architecture of the benchmarking tool.

3.2 Key Features

The benchmarking tool currently implements the following evaluation metrics for error recovery in parsers.

- **Error Types**: Categorization and analysis of errors encountered during parsing.
- **Similarity**: Similarity score is calculated between recovered output and expected result. In current implementation the original token sequence is used as the expected result. The system can be extended to use a reference-fixed version, enabling broader dataset evaluation and experiment design flexibility.
- **JVM Heap Memory Usage** Tracks peak JVM-heap memory consumption during parsing.

- **Processing Time**: Execution time is measured multiple times for each benchmarked input. Each measurement is recorded, allowing users to specify the number of warm-up runs and actual executions even after evaluation.

The system is designed to be extensible, allowing users to define and integrate their own evaluation metrics. By following the existing benchmarking framework, users can implement custom analyzers to gather additional insights, such as parser-specific error handling strategies, CPU usage, or other performance indicators. This flexibility makes the tool adaptable for different research needs and evolving parsing technologies.

3.3 Conclusion

The evaluation tool described in this paper is a practical solution for researchers and developers working on JVM-based parsing. The tool is also scalable, allowing users to add their own JVM-based parsers and define custom statistical measurements, making it adaptable for different research needs and parser implementations.

4. Experiment Design

This section describes our approach to measuring the quality of error recovery in parsers.

4.1 Dataset

To evaluate the quality of error recovery, we use the real-world dataset from Blackbox, a repository of events from the BlueJ Java IDE [19]. This dataset has previously been used for experiments in the field of syntactic analysis [1, 6, 8], enabling direct comparisons with prior research.

4.2 Research Questions

We design experiments to address the following research questions.

RQ1: How does parsing implementation affect error detection quality?

RQ2: Does similarity score reflect the quality of error recovery?

4.3 Evaluation Metrics

In this study, we adopt **edit distance** and **error type distribution** as the principal evaluation metrics.

The **edit distance** (*ED*) is zero for identical code; however, in our approach, we invert this metric so that the similarity score equals zero if the token sequences are completely different:

$$1 - \left(\frac{ED(original, recovered)}{max(|original|, |recovered|)}\right)$$

The *original* token sequence is obtained from the lexer phase, while the *recovered* token sequence is collected from parsing results, including error tokens if error nodes appear.

To evaluate the quality of error recovery by **error type distribution**, the dataset must be annotated with error types that should be detected. Since the target language is Java, we use the *javac* [20] compiler as a reference. *javac* is part of the Java Development Kit (JDK) so it serves as the de facto standard for compiling Java source files. It provides a well-defined set of diagnostics [21] with human-readable error descriptions [22].

A key challenge is that *javac* does not strictly separate parsing and compilation errors. For instance, a parser error like *illegal start of expression* and a compilation error such as *classes: {0} and {1} have the same binary name* are defined in the same file and referenced by property names. To ensure accurate evaluation of parsing errors, we manually exclude errors from later compilation stages. As *javac* is our reference, the metric for error recovery is the number of errors detected by *javac* that were missed by other parsers.

4.4 Implementation Details

All parser metric evaluations are abstracted using the *IRecoveryAnalyzer* interface, which provides the following methods.

- *getLexerTokens* – returns the set of original tokens obtained after lexing.
- *getParserTokens* – computes the set of tokens after error recovery.
- *measureParseTime* – performs parsing once and returns the execution time.

To support a custom JVM-based parser, it is necessary to implement these methods. During this research, we developed implementations for commonly used parsers and parser generators.

- **Tree-sitter** [23] – A widely used parser generator, notably in VSCode. Originally written in C, it includes optional JVM bundling [24]. Due to its non-heap memory usage, JVM-based memory profiling is not representative, but parsing speed and quality evaluations remain applicable. Since it lacks a built-in lexer API, we implemented an external lexer (based on JFlex for Java) for similarity distance calculation.
- Two versions of **ANTLR** [25] were generated for Java:
  - **ANTLR-Java8-spec** – Based on the official Java 8 specification [26].
  - **ANTLR-Java** – Based on an optimized Java grammar [27].
- **JDT** (Java Development Tools) – Used in Eclipse, JDT differs from conventional parsers by storing keyword tokens (e.g., *if, else, catch, public*) as AST node fields instead of leaves. Extracting token sequences for comparison requires a custom AST traversal mechanism. Fortunately, JDT provides the *NaiveASTFlattener* visitor, which traverses the AST, applies error recovery strategies, and reconstructs the correct code.

In approbation purposes, we analyzed the parsing speed for each parser on the BlackBox dataset. The evaluation results are shown in Fig. 2. This demonstrates that our tool enables researchers to compare different parsers in a benchmarking ecosystem with minimal overheads.

5. Evaluation

In this section, we present the evaluation results and address our research questions.

RQ1: How does parsing implementation affect error detection quality?

To assess the impact of parsing implementation on error detection quality, we compare different parsers in terms of missed error messages relative to the reference parser from the javac compiler.

Fig. 3 presents the distribution of missed errors across multiple parsing implementations.

The results reveal differences in error detection effectiveness. The most notable finding is the variation in the number of missed errors, particularly for common syntax issues such as *not a statement* and most common Java error [2] *`;` expected*. Tree-sitter, for instance, exhibits a high rate of missed *variable declaration not allowed here* and *class, interface, enum, or record expected* errors, indicating its limitations in handling incorrect declarations. ANTLR-Java and ANTLR-Java8-spec, while similar in parsing algorithm, show discrepancies in handling punctuation-related errors such as missing colons and parentheses. JDT, by contrast, appears to have more balanced performance but still misses certain structural errors.

These differences suggest that parsing strategies play a crucial role in error detection quality. Parsers optimized for flexibility, such as Tree-sitter or Antlr-java, may allow faster parsing but at the cost of missing critical syntactic violations. Conversely, stricter parsers such as ANTLR-Java8-spec can

enforce the Java grammar more rigorously, but as we can see in Fig. 2, they may struggle with parsing performance.

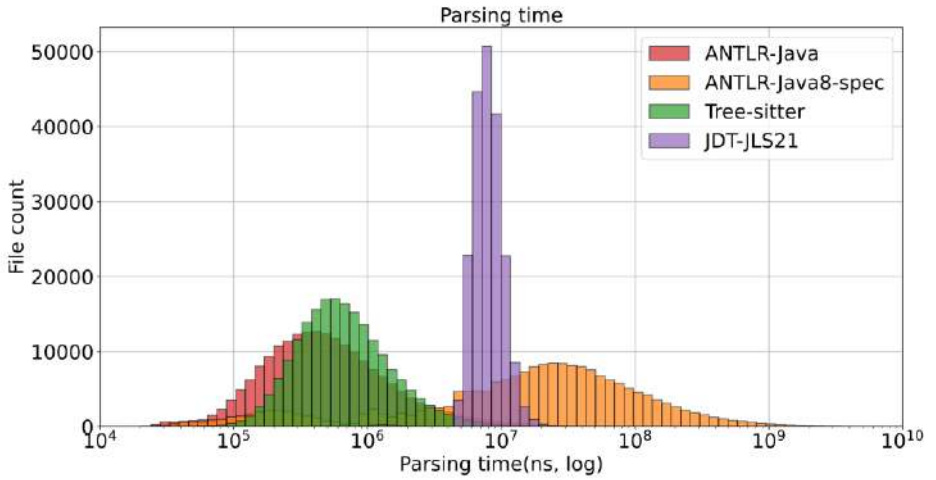


Fig. 2. Parsing time measurement on the BlackBox dataset.

In summary, the choice of parsing implementation directly affects error detection quality. The trade-off between strictness and recovery flexibility determines whether a parser will correctly diagnose syntax issues or silently fail to recognize them.

RQ2: Does similarity score reflect the quality of error recovery?

The similarity score, based on edit distance, measures how closely the recovered code matches the original input. We measured this metric using a BlackBox dataset to evaluate error recovery, as shown in Fig. 4. To better understand the quality of the similarity score, we compared its results with other metrics, such as performance and error distribution.

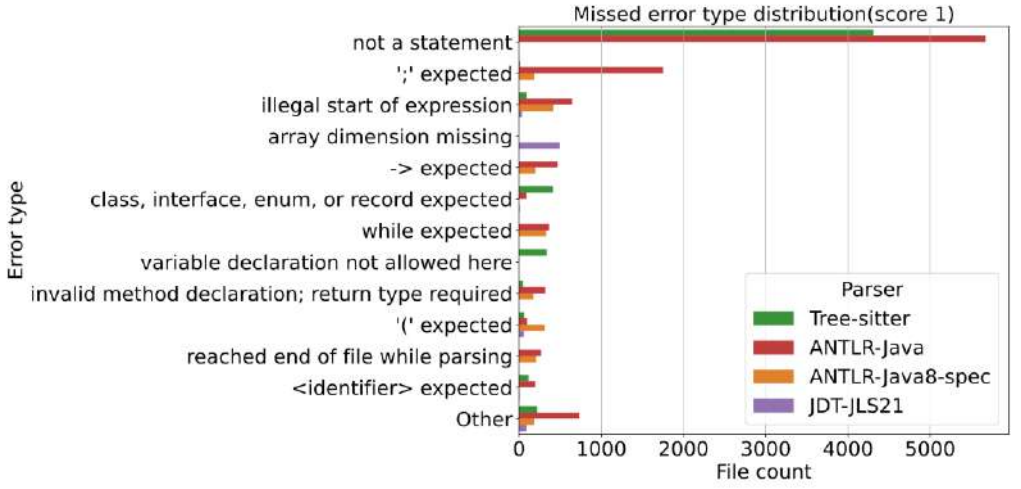


Fig. 3. Missing error distribution in comparison with javac.

Our analysis showed that although the similarity score provides a useful quantification, it does not always reflect the true quality of error recovery.

- **ANTLR-Java** and **Tree-sitter** show high similarity scores, but they tend to miss many syntax errors, as seen in Fig. 3. Their outputs look close to the original, yet they often fail to identify and correct issues.
- **JDT-JLS21** has lower similarity scores because it carefully prunes incorrect subtrees during recovery. This leads to more edits but ensures that invalid constructs are not preserved.
- **ANTLR-Java8-spec** closely follows the Java language specification and detects many errors, resulting in more differences from the original and thus lower similarity scores.

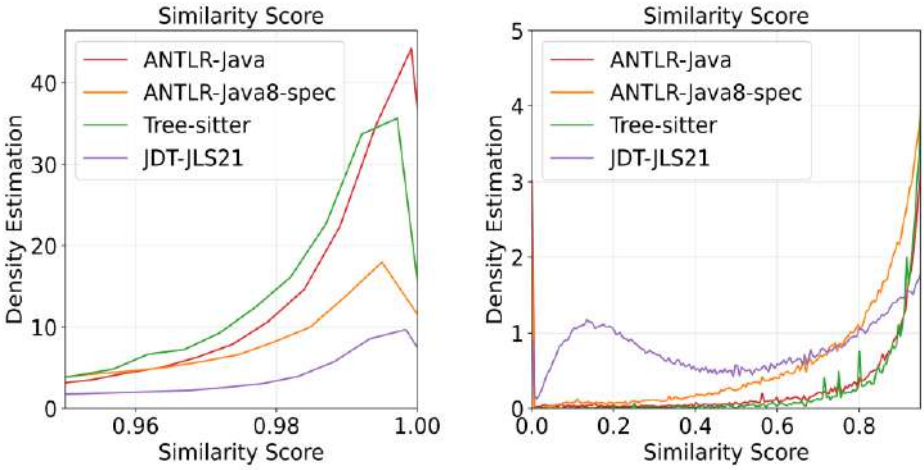


Fig. 4. Similarity score for Blackbox dataset.

In conclusion, while the similarity score alone is not sufficient to assess recovery quality, it still offers valuable insights. It is simple to compute, enables easy comparison between parsers, and it indicates how the recovery process is performed in general. However, since high similarity may also result from ignoring errors rather than fixing them, it must be used alongside correctness and structural validation metrics to accurately evaluate recovery behavior.

6. Conclusion

In this work, we surveyed and analyzed existing approaches to measuring error recovery quality in parsers. To support a subset of this analysis, we developed an open-source tool that enables researchers to evaluate both performance and recovery quality benchmarks for any JVM-based parser. The tool features an extensible architecture, allowing additional benchmarks and data analysis modules to be integrated using a consistent API.

Using this framework, we evaluated a real-world parser on actual erroneous user input. Our results demonstrate that parser recovery quality cannot be fully understood using a single metric alone. Instead, a comprehensive analysis across multiple metrics is necessary to draw meaningful conclusions.

This multifaceted evaluation approach allows not only for a deeper understanding of newly developed parsers but also facilitates practical comparisons between existing ones. Such comparisons can guide the selection of parsers best suited for real-world applications, depending on whether performance or error recovery is a higher priority for the end user.

7. Future Work

Impact on Performance: Understanding the impact of error recovery on performance requires proper benchmarking. Establishing a baseline for each example allows us to isolate whether performance variations stem from error recovery or differences in the parsing algorithm.

Cross-Language Evaluation: Extending our framework to other programming languages, particularly those using Parsing Expression Grammars (PEGs) [28-29], would provide broader insights into recovery strategies and language-specific challenges.

AI-Powered Recovery: AI-assisted tools demonstrate promising results in error recovery by analyzing the AST and suggesting fixes [8, 30-31]. Future work could explore integrating AI-driven recovery into JVM-based parsers and comparing its quality and effectiveness with traditional methods.

Evaluate More Real-World IDE Parsers: Some parsers used in real-world projects are not standalone tools but are instead tightly integrated components of larger systems. A key engineering challenge in benchmarking such parsers lies not in supporting them within the benchmarking framework, but in isolating and executing them independently of the full application. Notable examples include the parser from the IntelliJ IDEA IDE and the one embedded in the *javac* Java compiler.

References

[1]. I. Karvelas, J. Dillane, and B. A. Becker, “Programmers’ views on IDE compilation mechanisms,” in Proceedings of the ACM Conference on Global Computing Education Vol 1, ser. CompEd 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 98–104, Available: https://doi.org/10.1145/3576882.3617915.

[2]. D. Pritchard, “Frequency distribution of error messages,” in Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools, ser. PLATEAU 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–8., Available: https://doi.org/10.1145/2846680.2846681.

[3]. X. Zhou, S. Cao, X. Sun, and D. Lo, “Large language model for vulnerability detection and repair: Literature review and the road ahead,” ACM Trans. Softw. Eng. Methodol., Dec. 2024, just Accepted, Available: https://doi.org/10.1145/3708522.

[4]. T. J. Pennello and F. DeRemer, “A forward move algorithm for lr error recovery,” in Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ser. POPL ’78. New York, NY, USA: Association for Computing Machinery, 1978, p. 241–254., Available: https://doi.org/10.1145/512760.512786.

[5]. M. de Jonge, L. C. L. Kats, E. Visser, and E. S¨oderberg, “Natural and flexible error recovery for generated modular language environments,” ACM Trans. Program. Lang. Syst., vol. 34, no. 4, Dec. 2012., Available: https://doi.org/10.1145/2400676.2400678.

[6]. L. Diekmann and L. Tratt, “Don’t Panic! Better, Fewer, Syntax Errors for LR Parsers,” in 34th European Conference on Object-Oriented Programming (ECOOP 2020), ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Hirschfeld and T. Pape, Eds., vol. 166. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum f¨ur Informatik, 2020, pp. 6:1–6:32., Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2020.6.

[7]. M. de Jonge and E. Visser, “Automated evaluation of syntax error recovery,” in Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ser. ASE ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 322–325., Available: https://doi.org/10.1145/2351676.2351736.

[8]. E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, “Syntax and sensibility: Using language models to detect and correct syntax errors,” in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 311–322.

[9]. N. C. C. Brown, M. K¨olling, D. McCall, and I. Utting, “Blackbox: a large scale repository of novice programmers’ activity,” in Proceedings of the 45th ACM Technical Symposium on Computer Science Education, ser. SIGCSE ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 223–228., Available: https://doi.org/10.1145/2538862.2538924.

[10]. PereFlex source code, Available at: https://github.com/dsult/parser-compare, accessed 12.05.2025.
[11]. Visual Studio Code code editor, Available at: https://code.visualstudio.com, accessed 12.05.2025.
[12]. Eclipse IDE for Java, Available at: https://eclipseide.org, accessed 12.05.2025.
[13]. P. Degano and C. Priami, “Comparison of syntactic error handling in lr parsers,” Softw. Pract. Exper., vol. 25, no. 6, p. 657–679, Jun. 1995., Available: https://doi.org/10.1002/spe.4380250606.
[14]. P. Medvedev, “Theoretical analysis of edit distance algorithms,” Commun. ACM, vol. 66, no. 12, p. 64–71, Nov. 2023., Available: https://doi.org/10.1145/3582490.
[15]. B. Berabi, A. Gronskiy, V. Raychev, G. Sivanrupan, V. Chibotaru, and M. Vechev, “Deepcode ai fix: Fixing security vulnerabilities with large language models,” 2024., Available: https://arxiv.org/abs/2402.13291.
[16]. B. A. Becker, C. Murray, T. Tao, C. Song, R. McCartney, and K. Sanders, “Fix the first, ignore the rest: Dealing with multiple compiler error messages,” in Proceedings of the 49th ACM Technical Symposium on Computer Science Education, ser. SIGCSE ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 634–639., Available: https://doi.org/10.1145/3159450.3159453.
[17]. A. Akinshin, Pro.NET Benchmarking: The Art of Performance Measurement. Apress Berkeley, CA, 2019. 687 p
[18]. JMH source code, Available at: https://github.com/openjdk/jmh, accessed 12.05.2025.
[19]. I. Utting, N. Brown, M. K¨olling, D. McCall, and P. Stevens, “Web-scale data gathering with bluej,” in Proceedings of the Ninth Annual International Conference on International Computing Education Research, ser. ICER ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1–4., Available: https://doi.org/10.1145/2361276.2361278.
[20]. Javac compiler, Available at: https://docs.oracle.com/javase/8/docs/technotes/guides/javac, accessed 12.05.2025.
[21]. Javac diagnostics, Available at: https://openjdk.org/groups/compiler/doc/hhgtjavac/diagnostics.html, accessed 12.05.2025.
[22]. javac errors, Available at: https://github.com/openjdk/jdk/blob/master/src/jdk.compiler/share/classes/com/sun/tools/javac/resources/compiler.properties, accessed 12.05.2025.
[23]. Tree-sitter parser generator, Available at: https://tree-sitter.github.io/tree-sitter, accessed 12.05.2025.
[24]. Java bundle for Tree-sitter, Available at: https://github.com/tree-sitter/tree-sitter-java, accessed 12.05.2025.
[25]. T. Parr and K. Fisher, “Ll(*): the foundation of the antlr parser generator,” in Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 425–436., Available: https://doi.org/10.1145/1993498.1993548.
[26]. ANTLR grammar based on Java 8 specification, Available at: https://github.com/antlr/grammars-v4/tree/master/java/java, accessed 12.05.2025.
[27]. Optimized ANTLR grammar for Java, Available at: https://github.com/antlr/grammars-v4/tree/master/java/java8, accessed 12.05.2025.
[28]. S. Queiroz de Medeiros, G. de Azevedo Alvez Junior, and F. Mascarenhas, “Automatic syntax error reporting and recovery in parsing expression grammars,” Sci. Comput. Program., vol. 187, no. C, Feb. 2020., Available: https://doi.org/10.1016/j.scico.2019.102373.
[29]. S. Q. de Medeiros and F. Mascarenhas, “Towards automatic error recovery in parsing expression grammars,” in Proceedings of the XXII Brazilian Symposium on Programming Languages, ser. SBLP ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 3–10., Available: https://doi.org/10.1145/3264637.3264638.
[30]. G. Sakkas, M. Endres, P. J. Guo, W. Weimer, and R. Jhala, “Seq2parse: neurosymbolic parse error repair,” Proc. ACM Program. Lang., vol. 6, no. OOPSLA2, Oct. 2022., Available: https://doi.org/10.1145/3563330.
[31]. R. Gupta, A. Kanade, and S. Shevade, “Deep reinforcement learning for syntactic error repair in student programs,” in Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, ser. AAAI’19/IAAI’19/EAAI’19. AAAI Press, 2019., Available: https://doi.org/10.1609/aaai.v33i01.3301930.

Информация об авторах / Information about authors

Ольга Игоревна БАЧИЩЕ – аспирант института прикладных компьютерных технологий Университета ИТМО. Сфера научных интересов: статический анализ программ.

Olga Igorevna BACHISHCHE – postgraduate student at the Institute of Applied Computer Technologies, ITMO University. Research interests: static program analysis.

Ярослав Станиславович ВОРОБЬЕВ – магистр НИУ ВШЭ. Сфера научных интересов: анализ данных.

Yaroslav Stanislavovich VOROBIEV – master at HSE University. Research interests: data analysis.

Григорий Романович РАЙКИН – аспирант института компьютерных наук ИТМО. Сфера научных интересов: статический и динамический анализ программ, фаззинг, формальная спецификация программ.

Grigoriy Romanovich RAYKIN – postgraduate student at the ITMO Institute of Computer Science. Research interests: static and dynamic software analysis, fuzzing, formal software specification.

Дарья Владимировна ВАСИНА – ведущий инженер Санкт-Петербургской лаборатории средств разработки облачного ПО компании Huawei. Окончила факультет компьютерных технологий и управления Университета ИТМО по направлению «Информатика и вычислительная техника». Специализируется на создании инструментов для разработки программного обеспечения с интеграцией технологий искусственного интеллекта.

Darya Vladimirovna VASINA is a Lead Engineer at the St. Petersburg Cloud Software Development Tools Laboratory of Huawei. She graduated from the Faculty of Computer Technologies and Control at ITMO University, majoring in Computer Science and Engineering. She specializes in creating software development tools with the integration of artificial intelligence technologies.

Даниил Сергеевич ШУШАКОВ – магистр Университета ИТМО. Специализируется в разработке интегрированных сред разработки и статического анализа кода.

Daniil Sergeyevich SHUSHAKOV – master at ITMO University. Specializes in the development of integrated development environments and static code analysis.

Семён Вячеславович ГРИГОРЬЕВ – кандидат физико-математических наук, доцент кафедры системного программирования СПбГУ. Сфера научных интересов включает в себя статический анализ программ, алгоритмы и инструменты синтаксического анализа, высокопроизводительный анализ графов.

Semyon Vyacheslavovich GRIGORIEV – Cand. Sci. (Phys.-Math.), an associate professor in the Department of Software Engineering at St. Petersburg State University. His research interests include static program analysis, parsing algorithms and tools, and high-performance graph analysis.

DOI: 10.15514/ISPRAS-2025-37(6)-41

CC BY

Bounding Thread Switches in Dynamic Analysis of Multithreaded Programs

[1] *V.P. Rudenchik, ORCID: 0009-0000-6719-2594 <rudenchik@ispras.ru>*
[1] *P.S. Andrianov, ORCID: 0000-0002-6855-7919 <andrianov@ispras.ru>*
[1,2] *V.S. Mutilin, ORCID: 0000-0003-3097-8512 <mutilin@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *Moscow Institute of Physics and Technology, 9, Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.*

Abstract. For detecting race conditions in multithreaded programs, dynamic analysis methods can be used. Dynamic methods are based on observing program behavior on real program executions. Since analyzing all possible execution paths is generally infeasible (due to the combinatorial explosion of possible thread interleavings), dynamic methods can overlook certain bugs that manifest only within the specific conditions or thread interleavings. This limitation applies, for instance, to the approach implemented in the previous version of RaceHunter tool, which demonstrates the ability to effectively detect race conditions, but may still miss certain cases. To address the combinatorial explosion problem, context bounding analysis can be used. Context bounding is a dynamic analysis technique that limits the number of thread switches in each explored execution path, enabling more scalable exploration. This method is able to detect bugs missed by other techniques with the bound of only two preemptive thread switches.

In this work, we present an implementation of context bounding within the RaceHunter tool, which provides a unified framework for describing various dynamic analysis techniques. The evaluation shows that the proposed approach is able to detect race conditions that other methods missed, though at the cost of significantly increased analysis time. As expected, this increase in analysis time is caused by repeated executions. Still, the implementation is an important foundation for future integration with other race detection techniques, specifically with the approach already implemented in the RaceHunter tool.

Keywords: verification; dynamic analysis; context bounding.

For citation: Rudenchik V.P., Andrianov P.S., Mutilin V.S. Bounding Thread Switches in Dynamic Analysis of Multithreaded Programs. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 6, part 3, 2025. pp. 133-148 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-41.

Ограничение количества переключений потоков при динамическом анализе многопоточных программ

[1] *В.П. Руденчик, ORCID: 0009-0000-6719-2594 <rudenchik@ispras.ru>*
[1] *П.С. Андрианов, ORCID: 0000-0002-6855-7919 <andrianov@ispras.ru>*
[1,2] *В.С. Мутилин, ORCID: 0000-0003-3097-8512 <mutilin@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский физико-технический институт, 141701, Московская область, г. Долгопрудный, Институтский переулок, д. 9.*

Аннотация. Для обнаружения состояний гонки в многопоточных программах могут использоваться методы динамического анализа. Динамические методы основаны на наблюдении за поведением программы при её реальном выполнении. Так как анализ всех возможных путей выполнения в общем случае неосуществим (из-за комбинаторного взрыва числа возможных чередований потоков), динамические методы могут упускать определённые ошибки, проявляющиеся только при специфических условиях или порядке выполнения потоков. Это ограничение относится, например, к подходу, реализованному в предыдущей версии инструмента RaceHunter, который демонстрирует способность эффективно выявлять состояния гонки, но всё же может пропускать отдельные случаи. Для решения проблемы комбинаторного взрыва может использоваться анализ с ограничением числа переключений потоков (англ. context bounding). Этот метод подразумевает исследование только тех путей выполнения, в которых ограничено число переключений потоков, что позволяет сделать анализ более масштабируемым. Анализ с ограничением числа переключений потоков способен выявлять ошибки, пропускаемые другими методами, при ограничении всего в два принудительных переключения потоков.

В данной работе мы представляем реализацию анализа с ограничением количества переключений потоков в инструменте RaceHunter, который обеспечивает единую платформу для описания различных методов динамического анализа. Оценка показала, что предложенный подход способен выявлять состояния гонки, которые были пропущены другими методами, хотя и ценой значительного увеличения времени анализа. Как и ожидалось, это увеличение времени связано с повторными запусками программы. Тем не менее, реализация представляет собой важную основу для будущей интеграции с другими техниками обнаружения состояний гонки, в частности с подходом, уже реализованным в инструменте RaceHunter.

Ключевые слова: верификация; динамический анализ; ограничение контекста.

Для цитирования: Руденчик В.П., Андрианов П.С., Мутилин В.С. Ограничение количества переключений потоков при динамическом анализе многопоточных программ. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 133–148 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)–41.

1. Introduction

The verification of multithreaded (parallel) programs is more complex than that of single-threaded programs. This is primarily due to race conditions – a type of error unique to multithreaded programs and notoriously difficult to detect. Race conditions occur when multiple threads access shared memory simultaneously and can lead to nondeterministic program behavior. The problem of race detection is NP-hard [1]. As a result, numerous techniques for automatic race detection have been proposed and continue to be actively developed.

Approaches for detecting race conditions are generally classified as static or dynamic. Static methods analyze source code without executing the target program. They can detect potential race conditions across all possible executions but often produce false positives due to over-approximation. Dynamic methods, on the other hand, detect race conditions by monitoring program executions on specific inputs. This paper focuses on dynamic methods.

The main benefit of dynamic analysis is its high precision: it generally reports issues that actually occur during execution. And its main limitation is incompleteness: it can only detect bugs that

manifest under the specific inputs and execution paths explored during analysis. As a result, it may miss rare or nondeterministic bugs unless executions are carefully guided or repeated under varying conditions.

Analyzing all possible execution paths of a multithreaded program is generally infeasible due to the combinatorial explosion of possible thread interleavings. Context bounding is a dynamic analysis technique that addresses this problem by limiting the number of thread switches in each explored execution path. It systematically explores all execution paths that satisfy a specified bound. The method is scalable and has been shown to detect many real-world bugs even with relatively small bounds [2]. However, its high analysis time, caused by repeated executions, and its implementation as a stand-alone tool [3], limit its applicability.

The RaceHunter tool [4] provides a unified framework for describing and implementing various dynamic analysis methods. Each method fits a common interface and is implemented as an independent unit. This allows it to be used independently, and at the same time greatly simplifies its combination with other methods.

The main contribution of the paper is an implementation of context bounding analysis in the existing dynamic analysis tool, RaceHunter.

The paper is organized as follows. Section 2 provides a brief overview of existing dynamic program analysis methods and the tools that implement them. Section 3 presents the RaceHunter framework. Section 4 describes the implemented approach in terms of the RaceHunter framework. Finally, the evaluation of the implemented approach is presented in Section 5.

2. Preliminaries

We focus on multithreaded programs - that is, programs that contain more than one execution thread. The central problem is detecting race conditions in such programs. To simplify the description, we will consider data races, which are a narrower class of errors. However, the presented approach is able to detect general race conditions. A *data race* is formally defined as follows: two accesses from different threads to the same memory location are called conflicting if at least one of the accesses is a write. Two conflicting accesses are said to form a data race if the order of their execution is undefined.

Dynamic analysis techniques are used to detect race conditions. Dynamic analysis is a type of program analysis that is based on observing program executions on concrete inputs. A *program execution* refers to a specific run of the program, during which a particular sequence of operations is performed, determined by the program's logic, input data, and runtime environment. The sequence of operations in a program execution is referred to as an *execution path*. It represents one possible interleaving of operations across all threads of a multithreaded program. Note that we assume sequential consistency and do not consider weak memory models.

Below is a brief overview of several dynamic analysis methods, along with examples of tools that implement them.

2.1 Lockset and happens-before algorithms

Most dynamic data race detection tools are based on one of the following algorithms: happens-before [5], lockset [6], or a combination of both [7].

The happens-before algorithm is based on introducing a partial order over the set of memory accesses. In particular, two conflicting accesses are considered to form a race condition if they are not comparable under this order.

The lockset algorithm is based on tracking the set of acquired locks. In the lockset algorithm, two conflicting accesses are considered to form a race condition if their sets of acquired locks do not intersect.

To detect data races, a combination of these algorithms can be used [7]. The combination of the lockset and happens-before algorithms was implemented in ThreadSanitizer [8], a tool for dynamic analysis. However, the combination showed limited practical benefit due to false positives. That is why later versions of ThreadSanitizer do not rely on the lockset algorithm to find data races.

In ThreadSanitizer, memory accesses are grouped into segments - sequences of events within a single thread that contain only memory access events (i.e., no synchronization events). Each memory access event belongs to exactly one segment. A partial order is established on these segments using happens-before relations. The state of ThreadSanitizer is a set of unordered segments for each memory location. On every memory access (which are tracked using instrumentation), the state is updated and the algorithm performs a check for conflicting accesses (accesses from different threads with at least one of them being a write). Finally, if no happens-before relationship exists between two accesses and they are conflicting, a data race is reported.

2.2 Breakpoint-watchpoint approach

Another approach used for race detection is the breakpoint-watchpoint approach. It is based on placing breakpoints and watchpoints on a certain set of memory accesses. When a breakpoint is triggered for the memory that is being accessed, a watchpoint is set for the same memory, and a short waiting period follows for a thread that triggered the breakpoint. If the watchpoint is triggered in this period, which means that some other thread is accessing this memory, a race condition is detected.

This approach is implemented in the tools KCSAN [9], DataCollider [10], and RaceHunter [4]. In KCSAN and DataCollider, the set of memory accesses where breakpoints are placed is chosen randomly. In RaceHunter, each pair of conflicting memory accesses is analyzed.

Let's take a closer look at how the breakpoint-watchpoint approach in the RaceHunter tool works. During the first execution of the program (monitoring phase), various events are tracked for each thread, particularly accesses to shared memory. The resulting trace of events is then analyzed to identify pairs of conflicting memory accesses. For each such pair, the program is executed again (race provocation phase), with breakpoints set at the two conflicting accesses. When one of the breakpoints is triggered, the corresponding thread pauses for a limited time. If the second breakpoint is subsequently triggered, the race condition between the two conflicting accesses is considered detected.

The enhanced capability of RaceHunter to identify data races is achieved at the cost of increased verification time (primarily due to repeated executions) and memory consumption during the monitoring phase.

2.3 Context Bounding

All of the approaches described above can miss race conditions due to exploring only a fraction of possible execution paths of a multithreaded program.

A complete analysis of all possible thread interleavings is practically impossible in most cases because of the number of such thread interleavings. Consider a program with n threads where each thread executes k atomic operations. In this case, the number of possible interleavings of these operations exceeds $(n!)^k$. This means that the total number of interleavings grows at least exponentially with respect to both n and k. Analyzing such a vast number of execution paths is feasible only for the simplest examples and does not scale to real-world programs.

The number of analyzed execution paths can be significantly reduced by using iterative context bounding [2], in which only paths with a limited number of thread switches are analyzed.

Consider a context bound c, which is the maximum number of preemptive thread switches. In a program with n threads, where each thread executes k operations, the number of possible

interleavings with the bound c grows as $(n^2k)^c n!$ [2]. It is a polynomial growth with respect to k, making context bounding a scalable technique for larger programs.

Context-bounding is implemented in the CHESS tool [3]. CHESS systematically explores all possible thread interleavings using iterative context bounding. That means that it starts by analyzing execution paths without preemptions ($c = 0$), and after analyzing all paths with c preemptions, it proceeds to analyze those with $c + 1$ preemptions. For each execution path, CHESS uses Goldilocks [11] algorithm to check for data races. In theory, such an approach could take as long as an exhaustive search, but in practice, CHESS is capable of detecting race conditions missed by other algorithms even with a bound $c < 3$.

The benefits of this approach include its scalability to large programs and the ability to detect all race conditions that can be achieved with a given context bound c.

However, it also has drawbacks: the verification time is increased due to the large number of repeated executions, and the standalone implementation of the method makes integration with other approaches challenging.

3. Framework Overview

As previously stated, the goal of this work was to implement context bounding analysis within the existing dynamic analysis tool, RaceHunter. To describe the proposed context-bounding-based approach, we must first introduce the RaceHunter framework into which it is integrated. RaceHunter relies on instrumentation to control program execution, as detailed in Subsection 3.1. The required structure of test programs is described in Subsection 3.2. The RaceHunter algorithm is presented in Subsection 3.3, and the *analysis*, which is an abstraction for representing a dynamic analysis method, is defined in Subsection 3.4.

3.1 Instrumentation

In order to intercept program operations and control program execution, instrumentation is used. At the compilation stage, calls to the **OnEvent** function (see Subsection 3.3) from the RaceHunter library are automatically inserted immediately before each memory access event using ThreadSanitizer's default instrumentation. That allows RaceHunter to acquire control over program execution just before a memory access is performed, enabling the analysis to react based on information about the upcoming event and without executing it yet.

The **OnEvent** function is called concurrently for events from different threads, without the use of synchronization primitives. This concurrency allows RaceHunter to control the execution of the target program: based on the intercepted event, the analysis decides whether the event should be executed immediately or later. In the former case, **OnEvent** returns and the thread proceeds; in the latter, the thread is temporarily suspended within RaceHunter while other threads continue. This mechanism enables the analysis to enforce a specific execution path.

3.2 Test structure

The target test program must have a specific structure. To enable repeated execution, a test function should be wrapped in a loop. An example of such a program is shown in Listing 1. The `main` function contains a loop with calls to the `Reset`, **OnStart**, and **OnRestart** interface functions. The function-under-test `test_func` performs some concurrency operations. Note that the loop body is not limited to a single test function and may include an arbitrary set of operations.

The `test_func` function is called from the loop's body, allowing it to be executed repeatedly. Each iteration of the loop corresponds to a single execution of the test function. The condition for executing the loop's body is the function **OnRestart** (line 27 in Listing 1) of RaceHunter algorithm. The `Reset` function (line 24) resets the values of the shared variables to the default ones. The **OnStart** function (line 25) allows the algorithm to track and handle the start of each iteration.

```
1  int n = 0;
2
3  void Reset() { n = 0; }
4
5  void func1() {
6    tmp1 = n + 1;
7    n = tmp1;
8  }
9
10 void func2() {
11   tmp2 = n + 2;
12   n = tmp2;
13 }
14
15 void test_func() {
16   thread th1(func1);
17   thread th2(func2);
18   th1.join();
19   th2.join();
20 }
21
22 int main() {
23   do {
24     Reset();
25     OnStart();
26     test_func();
27   } while (OnRestart());
28 }
```

Listing 1: A test program.

3.3 RaceHunter algorithm

Algorithm in Listing 2 is the key component of the RaceHunter tool. Its input is an abstract *analysis*, which is used for program analysis, e.g. breakpoint-watchpoint approach or context bounding approach. Its attributes are a current *state*, a current *target* and the sets of targets - *waitlist* and *reached*. Three functions - **OnEvent**, **OnStart**, and **OnRestart** - are called from the instrumented target program.

For each iteration of the algorithm - equivalently, for each execution of the target program - a *target* is specified. Informally, a *target* represents the objective of the current iteration of the analysis. The set *waitlist* contains targets that are yet to be explored, while the set *reached* contains all targets for the analyzed program, both explored and unexplored. By definition, *waitlist* $\subseteq$ *reached*.

At the start of the analysis, *reached* contains only the initial target provided by the *analysis*, *waitlist* is empty, and *target* is the initial target. Both *reached* and *waitlist* are populated with targets by the `GetNewTargets` operator of the analysis, which is called from the **OnEvent** function.

The **OnRestart** function, which is called at the end of each iteration, updates the *target* of the algorithm for the next iteration. It selects a new target from the *waitlist* and assigns it to *target*. If the *waitlist* is empty, the analysis stops.

The *state* in Listing 2 represents the current state of the analyzed program. The **OnStart** function, which is called at the start of each iteration, resets the *state* to its initial value.

The *state* is updated via the `Transfer` operator of the analysis and is used to create new targets through the `GetNewTargets` operator.

The **OnEvent** function of the algorithm is called for each memory access event, as described in Subsection 3.1. It takes as input an event *e* and a thread identifier *tid*.

The **OnEvent** function does the following. First, it calls the `Transfer` operator of the *analysis*. This operator updates the *state* and returns a Boolean value *res*, which is then used to determine whether the current thread should continue execution or be paused. Because of the concurrent nature of **OnEvent**, when a thread is paused within **OnEvent**, other threads proceed and potentially modify the global *state*. As a result, when the paused thread resumes after a timeout and calls `Transfer` again, the state is likely to have changed, which may affect the outcome of the subsequent `Transfer` call. Target program progression and no looping are ensured by each specific implementation of the `Transfer` operator.

After that, the **OnEvent** function calls the `GetNewTargets` operator of the analysis. This operator returns new targets, which are then added to the sets *reached* and *waitlist*. Finally, the **OnEvent** function returns, allowing the intercepted event to be executed.

```
 1 waitlist := ∅
 2 reached := {analysis.getInitialTarget()}
 3 target := analysis.getInitialTarget()
 4 state := analysis.getInitialState()
 5 function OnStart
 6   state := analysis.getInitialState()      # Reset state
 7 end function
 8 function OnRestart
 9   if !waitlist.Empty() then
10     target := waitlist.Pop()
11     return true              # Restart with a new target
12   end if
13  return false                # No more targets ⇒ stop analysis
14 end function
15 function OnEvent
16   res := analysis.Transfer(state, e, tid, target)
17   while !res do              # While Transfer not successful
18     WaitWithTimeout()        # Pause thread
19     res := analysis.Transfer(state, e, tid, target)
20   end while
21   newTargets := analysis.GetNewTargets(state, e, tid, target)
22   for t : newTargets do
23     if !reached.contains(t) then
24       waitlist.add(t)
25       reached.add(t)
26     end if
27   end for
28 end function
```

Listing 2: RaceHunter algorithm.

3.4 Dynamic Analysis

As previously stated, in the RaceHunter framework, each approach is implemented as an instance of an abstract *analysis*. Each *analysis* must define a *state* and a *target* and implement each of the following interface functions: `Transfer`, `GetNewTargets`, `GetInitialTarget`, and `GetInitialState`. These functions are used by the algorithm, and although their specific implementations depend on the concrete analysis, they can be generally described as follows:

- `Transfer`: updates the *state* of the analysis for a given event *e* in thread *tid* and the *target*, and determines whether the thread *tid* should be suspended.
- `GetNewTargets`: generates new targets based on the current *state*, the given event, and the *target*.

- `GetInitialTarget`: returns the initial target for the first iteration.
- `GetInitialState`: is used to reset the *state* at the beginning of each iteration.

The `Transfer` operator not only updates the *state* for a given target program event, but also determines whether the thread should pause or continue execution, as described in Section 3.3. It returns a Boolean value indicating whether the current thread should be suspended or allowed to proceed. Note that the *state* is passed to `Transfer` by reference, allowing the operator to modify it directly.

In the breakpoint-watchpoint approach implemented within the RaceHunter framework, the *target* is defined as a pair of conflicting accesses to shared memory. The *state* tracks which memory accesses have been executed so far. When one of the accesses from the *target* is encountered by the `Transfer` operator, the operator temporarily suspends the corresponding thread. During this suspension, `Transfer`, when called for other threads, checks whether the other access in the pair is about to be executed. If it is, a race condition is reported. The `GetNewTargets` operator for a given shared memory access identifies new pairs of conflicting accesses, which then become new targets for the next iterations.

4. Implementation

The proposed approach performs a systematic analysis of all possible program executions such that the number of thread switches in the execution path does not exceed a specified bound. The description of the implementation of this approach in the framework described in the previous section is presented in Subsection 4.1. Subsections 4.2-4.4 introduce the auxiliary class Scheduler used in the implementation. The limitations of the approach are discussed in Subsection 4.5, and some implementation features are provided in Subsection 4.6.

4.1 Context Bounding analysis

We now describe context bounding analysis. To do so, we need to define the four functions introduced in Section 3.4 along with the *state* and the *target* of the analysis. We begin by defining the *target* and the *state*.

The goal is to analyze all possible execution paths in which the number of thread switches does not exceed a given context bound. Accordingly, in our analysis, the *target* of each iteration represents a unique execution path. Similarly, the *state* of the analysis represents the current partial execution path.

Each execution path is described by a sequence of events. Since the events themselves are not important in our implementation, the sequence of events is abstracted to a sequence of corresponding thread identifiers. We refer to such sequences of thread identifiers as prefixes. Both the *target* and the *state* in our analysis are represented by such prefixes.

The `GetInitialState` operator returns a *state* with an empty prefix.

The `GetInitialTarget` operator returns a *target* with an empty prefix, which, in the case of targets, corresponds to an arbitrary execution path. Listing 3 provides a description of `Transfer` and `GetNewTargets` operators.

The `Transfer` operator updates the *state* (line 3 in Listing 3) by appending a thread identifier to the prefix. It does so as follow. For a given event, the `Transfer` operator decides if the event should be executed. The decision process, which includes selecting the thread necessary to recreate the execution path specified by the *target*, is described in detail in Sections 4.2, 4.3, and 4.4. When execution is permitted, the `Transfer` operator adds the corresponding thread identifier to the end of the *state*'s prefix and returns `True`; otherwise, the prefix remains unchanged and `False` is returned. As a result, by the end of an iteration, the *state*, which was empty at the start of the iteration, accumulates the full path of this execution.

```
 1 function Transfer(&state, e, tid, target)
 2   if scheduler.ResumeThread(state, tid, target) then
 3     state.Update()
 4     return true
 5   else
 6     return false
 7   end if
 8 end function
 9
10 function GetNewTargets(state, e, tid, target)
11   if TargetRecreated(state, target) then
12     for id : {AllIds \ state.LastThreadId} do
13       newTarget = copy(state)
14       newTarget.swapLastElementTo(id)
15       if newTarget.abide(bound) then
16         targets.add(newTarget)
17       end if
18     end for
19     return targets
20   else
21     return ∅
22   end if
23 end function
```

Listing 3: Context Bounding analysis functions

The `GetNewTargets` operator generates new *targets* only if the *target* from the current iteration is already recreated (see the `TargetRecreated` function at line 11). It does so as follows: it replaces the last thread identifier (line 14) in the updated state's prefix (just added by the `Transfer` operator) with each of the other available thread identifiers. Note that `GetNewTargets` does not modify the *state*. All resulting prefixes that satisfy the context bound (line 15) are returned as new targets (line 19).

We state that in our implementation, each *target* generated by the `GetNewTargets` operator during analysis uniquely corresponds to a specific execution path. The *target* prefix defines the beginning of the path but does not explicitly define the rest of it. However, the way the rest of the path is constructed during execution ensures its uniqueness. Moreover, presuming that all execution paths are finite, we state that each execution path that satisfies the given context bound is represented by some *target* generated by the `GetNewTargets` operator and is explored during the analysis.

To better illustrate this, let us consider the tree of possible execution paths shown in Fig. 1 for the program in Listing 1. For simplicity, only operations from the concurrent functions `func1` (thread 1) and `func2` (thread 2) are shown. Each edge represents a program operation, and each vertex represents a prefix. Vertex *A* corresponds to the beginning of an execution, while vertices *N* through *S* correspond to the end of an execution. An execution path is a sequence of edges from the root *A* to one of the leaves *N*–*S*. There is a one-to-one correspondence between all execution paths and the leaves *N*–*S*, so we refer to each execution path by its corresponding leaf. There are a total of six execution paths in this example: *N*, *O*, *P*, *Q*, *R*, and *S*.

First iteration. The analysis starts with an empty *target* prefix (*target* = {}) and an empty *state* prefix (*state* = {}), both of which correspond to the vertex *A* in the example. Assuming there is no context bound, our goal is to explore each of the six execution paths exactly once.

The `Transfer` operator selects the next thread to execute and appends its thread identifier to the *state*. The `GetNewTargets` operator generates new targets by appending all other possible thread

identifiers to the original *state*. In the example, the `Transfer` operator selects one of the two threads, updating the *state* to correspond to either vertex *B* or *C*. Suppose that `Transfer` selects the first thread, so the *state* now corresponds to vertex *B* (*state* = {1}). The `GetNewTargets` operator then replaces the last thread identifier in the *state* with the other available one, generating a new *target* corresponding to vertex *C* (new *target* = {2}).

After that, the `Transfer` operator selects a thread to execute again. Suppose it selects the first thread once more, updating the *state* to correspond to vertex *D* (*state* = {1, 1}). The `GetNewTargets` operator then generates a new *target* corresponding to vertex *E* (new *target* = {1, 2}). Then, the execution continues in the only remaining possible way and finishes, updating the *state* to correspond to vertex *N* (*state* = {1, 1, 2, 2}).

The result of the first iteration is as follows: the execution path *N* is explored, and two new targets are generated: {1, 2} and {2}, corresponding to vertices *E* and *C*, respectively.

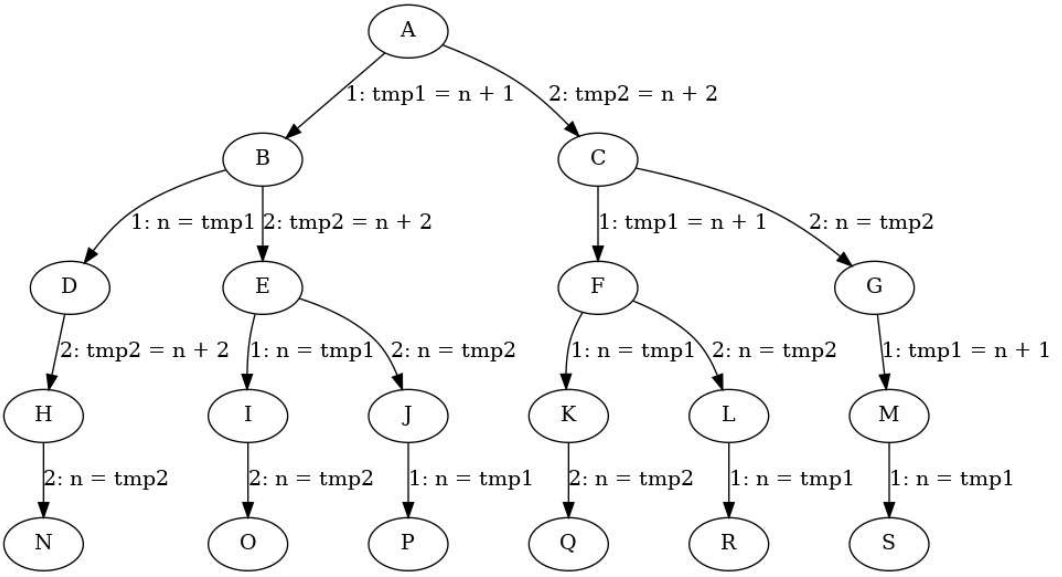


Fig. 1. Execution paths tree.

Second and subsequent iterations. Next, the analysis explores one of the new targets. Suppose that it explores *target* = {1, 2}, which corresponds to vertex *E*. In this iteration, one of the two possible execution paths, *O* or *P*, will be explored, and the other one will be represented as a new target that corresponds to vertex *I* or *J* and will be explored in a future iteration. Similarly, during the iteration with the target corresponding to vertex *C*, one of the execution paths *Q*, *R*, or *S* from this subtree (the subtree with the root *C*) will be explored, and the remaining paths will be represented as new targets and explored later.

Summing up, each iteration starts with the *target* prefix recreation, which corresponds to going down the tree to an unexplored subtree represented by the *target*. In the first iteration, the entire tree is unexplored - this is precisely what an empty target represents. For such an unexplored subtree, the `Transfer` operator incrementally selects a path from the root of the subtree to a leaf, while the `GetNewTargets` operator keeps track of all other possible paths in this subtree by representing each unexplored sub-subtree as a new *target*. This process ensures that each execution path is explored exactly once.

4.2 Scheduler

In this section, we describe the scheduler - an auxiliary component in our implementation. Its internal attributes can be viewed as part of the *state*, and its methods as helper functions for the context bounding *analysis*.

The scheduler's main purpose is to choose which thread will proceed next at a given point during an iteration. This is done through a call to the scheduler's boolean function `ResumeThread` from the `Transfer` operator, as shown in Listing 3.

The `ResumeThread` function ensures *target* prefix recreation. It does so by identifying a thread required to recreate the *target* from the current *state* and allowing only this thread to proceed.

The *target* prefix is recreated in our implementation as follows. At the start of each iteration, the scheduler checks if the *target* prefix is empty. If it is not (as is the case in all iterations except the first), the prefix has to be recreated first. For this, the scheduler creates an iterator over the *target* prefix. This iterator tracks the part of the prefix that has already been recreated and points to the identifier of the next thread required for recreation. When the `ResumeThread` function is called for a thread *tid*, it first checks whether the iterator still points to an identifier to see if the prefix has been already recreated. If it is not the case, the `ResumeThread` function then compares *tid* to the identifier to which the iterator is pointing to. If they are equal, then the iterator moves to the next identifier and the scheduler allows the thread to proceed. Otherwise, the thread is suspended, and the scheduler waits for another thread.

Once the *target* prefix has been fully recreated, the `ResumeThread` function chooses the next thread according to the scheduler's strategy, which is described in the next section.

4.3 Scheduler strategies

After the *target* prefix has been recreated, the `Transfer` operator guides the program execution to the end, thereby exploring a unique execution path. Theoretically, the specific choice of which path is executed has no effect on the correctness of the analysis - it only determines the order in which the execution paths are explored. Regardless of this order, each execution path will be explored during the analysis exactly once. However, in practical applications, the choice of execution path can be important. To demonstrate this, we implemented three scheduler strategies for selecting the next thread to execute: the *random strategy*, the *minimum switches strategy*, and the *fair scheduling strategy*.

The *random strategy* does not suspend any thread and allows every event to be executed straight away, thus performing essentially a random execution path. This strategy is natural and straightforward, but it has an important flaw. The number of thread switches in a random execution path is typically rather big. Therefore, the executed path will likely not satisfy the context bound and the analysis will waste time exploring it. That applies to essentially all of the iterations, where the *target* prefix represents a partial and not full execution path. Although the *target* prefix satisfies the context bound, the rest of the path will be a random sequence of events, and the total number of thread switches in the full execution path is likely to exceed the bound.

Essentially, with this strategy, the analysis will initially explore irrelevant execution paths that do not satisfy the context bound. However, on each iteration, it will generate *targets* with increasingly longer prefixes that do satisfy the bound. Eventually, a large enough part of the path will be defined by the *target* prefix rather than by the strategy, making it more likely for the full path to satisfy the bound. Ultimately, all possible execution paths that satisfy the context bound will be explored, but the analysis is likely to first spend a significant amount of time exploring irrelevant paths. Therefore, this strategy is not well-suited for our purposes.

The *minimum switches strategy* chooses the same thread that was executed most recently, attempting to continue execution without performing a thread switch. The scheduler tries to do so as long as it is possible. When forced to switch to another thread, it chooses any other available thread.

The idea behind this strategy is that by avoiding thread switches, the execution is likely to follow a path that satisfies the context bound. In contrast to the *random strategy*, this strategy significantly reduces the number of iterations and, consequently, the overall analysis time. A disadvantage of this strategy is that the scheduler can start unrolling a loop in the target program that can only be exited by an action from another thread, causing the analysis to be stuck in an infinite loop. This problem can be avoided by the next strategy.

The *fair scheduling strategy* chooses the same thread that was executed most recently, but only if the number of consecutive executions of events in this thread does not exceed a certain limit. That helps avoid unrolling infinite loops by enforcing a switch to another thread after some time, while still executing a path with relatively few thread switches that is likely to satisfy the context bound.

4.4 Ensuring analysis progression

During target program execution, situations may occur where the thread that should be selected by the scheduler is not available for execution - for example, if a thread is waiting on a lock in the target program. Waiting indefinitely for this thread and not allowing other threads to progress would lead to a deadlock. However, the scheduler accounts for such situations and includes a mechanism to avoid them.

As described in Section 3.3, threads that were not selected by the scheduler are suspended with a certain timeout. After the timeout expires, these threads call the scheduler (through `Transfer`) again to check if the scheduler will allow them to proceed now. In our implementation, the scheduler counts the number of such calls. If this number exceeds a certain threshold and the originally selected thread has not become available during that time, the scheduler assumes that the selected thread is blocked and chooses a different thread to execute. Note that the threshold is proportional to the number of threads in the target program. In this way, the scheduler ensures that the target program progresses.

4.5 Limitations of the approach

As mentioned above, each execution path is represented as a sequence of events, which in our implementation is abstracted as a sequence of corresponding thread identifiers. The main drawback of this approach is that it does not account for potential non-determinism in the behavior of the target program.

Our implementation is based on the assumption that thread scheduling is the only source of non-determinism, and therefore that the number of events in each thread remains the same across iterations. In practice, however, this is not always the case. When such extraneous non-determinism occurs during analysis, the scheduler may execute a path that is different from the one intended.

Adequate handling of non-determinism could be achieved by tracking and interleaving specific events rather than relying on the consistency of the number of events (i.e., performing a thread switch after a predetermined event rather than after a fixed number of events). This is one of the directions for future improvement.

Note that one of the differences from the implementation of context bounding in the CHESS tool is the definition of the bound: in CHESS, the bound limits the number of preemptive thread switches, whereas in the version of our analysis used in this paper, the bound applies to the total number of switches.

Another important consideration is the choice of events to be treated as interleaving points. In the current implementation, the sequence of thread identifiers abstracts the sequence of all memory access events in the target program. That means that every memory access is considered a possible interleaving point, which is excessive. For instance, authors of [2] show that it is sufficient to insert scheduling points before synchronization operations to not miss any errors in the program. Reducing the set of interleaving points - particularly through partial-order reduction techniques - is one of the directions for future optimization.

4.6 Implementation Features

One common source of non-determinism in the program behavior is lazy initialization that causes a data structure to be initialized during the first execution and not beforehand. Because of that, the number of events can vary between the first execution and subsequent ones, potentially affecting scheduling in our implementation. To address this problem, we added preparatory iterations before the analysis begins. During these iterations, the target program is executed in full, ensuring that all initializations occur on these iterations, before the actual analysis. A similar issue and solution are present in the CHESS tool [3].

Unlike the context bounding analysis implemented in the CHESS tool (see Section 2.3), the proposed method is not iterative; that is, it does not increment the bound after analyzing all execution paths for a given bound. This limitation is mostly technical, stemming from an implementation decision. The proposed method can be easily modified to perform iterative context bounding if needed.

5. Evaluation

We evaluated the approach on two benchmark sets. The first one contains small manually prepared tests. The second one is based on existing tests for an industrial virtual machine. We used a machine with an Intel® Core™ i5-8250U CPU (8 cores, 1.60 GHz) and 8 GiB of memory running Ubuntu 22.04.5 LTS (64-bit).

5.1 Small tests

The implemented approach explores all possible execution paths but does not itself detect or report race conditions. To detect race conditions, we combine it with ThreadSanitizer's data race detection. This combination can be done easily in the RaceHunter framework.

Our context bounding approach, in combination with ThreadSanitizer (**CB + TSan**), was evaluated against three other methods:

- plain ThreadSanitizer (**TSan**);
- the breakpoint-watchpoint approach implemented in the RaceHunter tool (**WP**);
- a combination of WP and TSan (**WP + TSan**).

The reason we evaluated our approach against the **WP + TSan** combination is to ensure that the **CB + TSan** combination does not find a race that **TSan** alone misses simply due to a few additional executions. The **WP + TSan** combination provides a few additional executions that can improve **TSan**'s ability to find races. We want to distinguish this factor from the benefit provided by the **CB** analysis.

By default, the context bounding analysis was used with a bound of two thread switches. The default scheduling strategy (see Section 4.3) was the *minimum switches strategy*. If this strategy resulted in unrolling an infinite loop, the *fair scheduling strategy* was used instead, with a limit of 1000 consecutive events in one thread.

The benchmark set is composed of two parts. 15 of the tests are from the RaceHunter repository.

The other 32 tests were adapted from the *sctbench* repository [12] to satisfy the required test structure (see Section 3.2). Some of the 32 adapted tests contain assertions that check if the existing race condition alters the expected program behavior. A violation of such an assertion means that a race condition does exist in the test and that it did affect the program's outcome. One of the tests contains a deadlock. This test was excluded from the benchmark set as irrelevant. Therefore, the overall number of tests is 46.

The results for 46 of the tests are presented in Table 1. For each method, the total analysis time was calculated as the average over three runs. The total number of reported races was determined as the maximum across three runs. For each method, the number of reported races in any single run differed

from the maximum by no more than three, meaning that only a few of those reports appear non-deterministically. Note that some of the reported races are false positives.

Table 1. Results for small benchmarks.

Method	Total time	Races reported
TSan	9.889s	28
WP	24.051s	30
WP+TSan	26.24s	36
CB+TSan	22m 10.907s	42

The **CB** + **TSan** combination can detect race conditions in two ways: either TSan can report a race, or **CB** can explore an execution path in which an assertion that indicates a race condition is violated. Table 2 groups all 46 tests by how the **CB** + **TSan** combination compares to the others: the first column shows the result category, and the second column shows how many tests fall into that category.

Table 2. Summary of results.

Result of **CB+TSan**	Number of tests (out of 46)
New races reported by **TSan**	3
New race-indicating assertion violation	7
Same as plain **TSan**	31
Same as **WP+TSan**	3
Non-unique race-indicating assertion violation	1
Analysis error	1

In 3 of the tests, **TSan** in combination with **CB** was able to report true data races that were missed by all of the other approaches. In 7 other tests, the combination of **CB** and **TSan** resulted in an assertion violation that indicates a race condition. This assertion was not triggered by any of the other approaches, and this is the main benefit of the **CB** approach.

For 31 out of 46 tests, the **CB** + **TSan** combination reported the same data races as plain **TSan**, meaning that context bounding did not provide any improvement for those tests.

For 3 of the tests, the **CB** + **TSan** combination reported the same data races as **WP** + **TSan**, which were not detected by plain **TSan**. This suggests that TSan did not necessarily need context bounding to report these races, but rather benefited from the additional execution paths explored when combined with the **WP** approach. Note that for one of these three tests, the reported races were false positives.

In one of the tests, an assertion violation indicating a race condition was triggered by both the **CB** + **TSan** combination and the **WP** approach. The analysis of another test was terminated due to a violation of an internal assertion of the context bounding analysis.

For the excluded test with a deadlock, **TSan** successfully detected and reported the deadlock even without any of the added methods. The context bounding analysis, however, was able to explore an

execution path that led to the deadlock and, therefore, did not complete. This raises an interesting question regarding the applicability of context bounding analysis for deadlock detection.

In summary, for 10 out of 46 tests, the use of context bounding analysis resulted in the detection of race conditions that were missed by other approaches. In 7 of these tests, the context bounding analysis was able to find and explore an execution path in which the race condition directly causes incorrect behavior, resulting in an assertion violation. The availability of such a path to an assertion violation clearly points to a real bug in the program and significantly simplifies the debugging process.

5.2 ARK VM tests

The benchmark set is based on existing TypeScript tests for the ARK VM [13]. The ARK VM was modified to support the RaceHunter tool.

The benchmark set contains 22 tests, but the majority of them require significant resources. Thus, we selected only 4 of them that required less time for 1000 iterations. The main goal is to evaluate time consumption.

The tool configuration uses plain context bounding analysis without ThreadSanitizer. Thus, no data races are reported, and the tool just explores different executions. Note that the breakpoint-watchpoint approach was successfully applied to the VM and found races, but this is beyond the scope of this paper. The context bounding analysis is limited by 2 thread switches and the scheduling strategy is the *fair scheduling strategy* with a limit of 10 000 consecutive events in one thread.

The results are presented in Table 3. The table contains an approximate number of memory accesses (rounded to the nearest 1000) for one iteration and the time required to explore the first 1000 iterations for a total of 4 tests. The time required to explore one iteration exceeds one second even for the smallest of the tests.

Table 3. Results on ARK VM tests.

Test name	Approximate number of memory accesses	Time for 1000 iterations
async_call_3	49000	25m28s
Await	45000	27m46s
concurrent_start_gc	1300000	2h22m41s
launch_instruction_3	24000	20m31s

The estimate of the total number of iterations is tens of thousands. So, the estimate of the total time is multiple hours for simple tests and multiple days for more complicated tests. It is barely acceptable, as the analysis would still take too long to complete. However, the approach fundamentally can be applied to complicated software and the further problem is scalability. For instance, the problems arise with overflowing some variables and using too much memory, since the framework was not initially designed for these many iterations. This shows that the problem of reducing the state space is relevant and should be explored in future work.

6. Conclusion

We presented a context bounding approach which is integrated into the RaceHunter tool. The most promising further direction is a combination of the context bounding analysis with the watchpoint-based approach.

The context bounding approach shows fundamental applicability to industrial software. However, it definitely requires further optimizations to reduce the number of considered executions. One

possible idea is to develop a coverage-based approach such as fuzzing. The target is considered only if new coverage is discovered.

A problem with non-determinism can be mitigated by more control over the target program. One possible idea is to implement a “step-by-step” version of the context bounding analysis. This approach produces only sequentialized executions. However, the main research question in this case is its scalability.

References

[1]. R. H. B. Netzer and B. P. Miller, “What are race conditions? Some issues and formalizations”, LOPLAS, vol. 1, pp. 74–88, 1992.
[2]. M. Musuvathi and S. Qadeer, “Iterative context bounding for systematic testing of multithreaded programs”, vol. 42, p. 446–455, jun 2007.
[3]. M. Musuvathi, S. Qadeer, and T. Ball, “Chess: A systematic testing tool for concurrent software”, Tech. Rep. MSR-TR-2007-149, November 2007.
[4]. E. A. Gerlits, “Racehunter dynamic data race detector”, Programming and Computer Software, vol. 50, pp. 467–481, Dec 2024.
[5]. L. Lamport, “Time, clocks, and the ordering of events in a distributed system”, Commun. ACM, vol. 21, p. 558–565, jul 1978.
[6]. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A dynamic data race detector for multi-threaded programs”, SIGOPS Oper. Syst. Rev., vol. 31, pp. 27–37, Oct. 1997.
[7]. R. O’Callahan and J.-D. Choi, “Hybrid dynamic data race detection”, inProceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’03, (New York, NY, USA), p. 167–178, Association for Computing Machinery, 2003.
[8]. K. Serebryany and T. Iskhodzhanov, “Threadsanitizer – data race detection in practice”, in Proceedings of the Workshop on Binary Instrumentation and Applications, (NYC, NY, U.S.A.), pp. 62–71, 2009.
[9]. Kernel Concurrency Sanitizer (KCSAN) google.github.io”, https://google.github.io/kernel-sanitizers/KCSAN.html. Accessed 05-05-2025.
[10]. J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, “Effective data-race detection for the kernel”, pp. 151–162, jan 2010.
[11]. T. Elmas, S. Qadeer, and S. Tasiran, “Goldilocks: Efficiently computing the happens-before relation using locksets”, pp. 193–208, 01 2006.
[12]. https://github.com/mc-imperial/sctbench.git. Accessed 05-05-2025.
[13]. https://gitee.com/openharmony/arkcompiler_runtime_core.git. Accessed 05-05-2025.

Информация об авторах / Information about authors

Вероника Павловна РУДЕНЧИК – стажер-исследователь ИСП РАН. Научные интересы: статический и динамический анализ программ.

Veronika Pavlovna RUDENCHIK – researcher at ISP RAS. Research interests: static and dynamic program analysis.

Павел Сергеевич АНДРИАНОВ – научный сотрудник ИСП РАН, кандидат физико-математических наук. Научные интересы: статическая верификация, параллельные программы.

Pavel Sergeevich ANDRIANOV – researcher in ISP RAS, Ph.D. Research interests: software model checking, parallel programs.

Вадим Сергеевич МУТИЛИН – кандидат физико-математических наук, ведущий научный сотрудник Института системного программирования им. В.П. Иванникова РАН и доцент Московского физико-технического института. Сфера научных интересов: статический и динамический анализ программ.

Vadim Sergeevich MUTILIN – Cand. Sci. (Phys.-Math.), leading researcher at Ivannikov Institute for System Programming of the RAS and associate professor at Moscow Institute of Physics and Technology. Main research interests: static and dynamic program analysis.

DOI: 10.15514/ISPRAS-2025-37(6)-42

CC BY

Clarifying Knowledge about Early Contacts of Native Speakers of the Proto-Finno-Volgaic Language Using Neural Networks

[1,2] *Ju.V. Normanskaja, ORCID: 0000-0002-2769-9187 <julianor@mail.ru>*
[1] *O.V.Goncharova, ORCID: 0000-0002-8665-6240 <oxanavgoncharova@gmail.com>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*
[2] *Institute of Linguistic RAS, 1, B. Kislovskiy lane. Moscow 125009, Russia.*

Abstract. The article explores the potential of artificial intelligence for discovering new etymologies. It consists of two parts: the first describes the structure of the neural network, while the second provides examples of new types of etymologies, including Erzya additions to existing well-known etymologies, separate Finnic-Erzya parallels, and new hypotheses regarding borrowings from Baltic and Germanic languages. The purpose is to demonstrate the kinds of new etymologies that can be proposed within a relatively short time frame for languages with an established etymological tradition through the use of a neural network. The study utilizes a Finnish-Russian dictionary containing 17,212 lexemes and an Erzya-Russian dictionary comprising 8,512 lexemes, both hosted on the LingvoDoc platform. A neural network capable of proposing new etymologies for dictionaries on the lingvodoc.ispras.ru platform has been developed. Using this tool, Finnish and Erzya dictionaries were processed, resulting in the identification of over 100 new etymologies. Among these, 16 etymologies are discussed in the article, pertaining both to native Finno-Ugric vocabulary and borrowings.

Keywords: neural network; Finnish language; Erzya language.

For citation: Normanskaja Ju.V., Goncharova O.V. Clarifying knowledge about early contacts of native speakers of the Proto-Finno-Volgaic language using neural networks. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 3, 2025, pp. 149-162. DOI: 10.15514/ISPRAS-2025-37(6)-42.

Acknowledgements. This work was supported by a grant RSCF № 25-78-20002. The results were obtained using the services of the Ivannikov Institute for System Programming (ISP RAS) Data Center.

Уточнение информации о ранних контактах носителей прафинно-волжского языка с помощью нейросети

[1,2] *Ю.В. Норманская, ORCID: 0000-0002-2769-9187 <julianor@mail.ru>*
[1] *О.В. Гончарова, ORCID: 0000-0002-8665-6240 <oxanavgoncharova@gmail.com>*
[1] *Институт системного программирования им. В.П. Иванникова РАН,*
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.
[2] *Институт языкознания РАН,*
Россия, 125009, Москва, Б. Кисловский пер., д. 1.

Аннотация. Статья посвящена возможностям технологии искусственного интеллекта для поиска новых этимологий. Она состоит из двух частей, первая посвящена описанию того, как устроена нейросеть, во второй части приведены примеры типов новых этимологий: эрзянские дополнения для существующих общеизвестных этимологий, сепаратные финско-эрзянские параллели, новые гипотезы о заимствовании из балтийских и германских языков. Целью работы - показать, какие новые этимологии возможно предложить в достаточно сжатые сроки для языков с разработанной этимологической традицией благодаря использованию нейросети. Материалы исследования: финско-русский словарь, который содержит 17 212 лексем, и эрзянско-русский словарь, объемом 8 512 лексем, размещенные на платформе ЛингвоДок. В результате работы создана нейросеть, которая может предлагать новые этимологии для словарей, размещенных на платформе lingvodoc.ispras.ru, с ее помощью обработаны словари финского и эрзянского языков, выявлено более 100 новых этимологий, из них в статье разобрано 16 этимологий, которые относятся как к исконной финно-угорской лексике, так и к заимствованиям.

Ключевые слова: нейросеть; финский язык; эрзянский язык.

Для цитирования: Норманская Ю.В., Гончарова О.В. Уточнение информации о ранних контактах носителей прафинно-волжского языка с помощью нейросети. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 149–162 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)–42.

Благодарности. Работа выполнена при поддержке Российского научного фонда, проект № 25-78-20002**.** Результаты получены с использованием услуг Центра коллективного пользования Института системного программирования им. В.П. Иванникова РАН – ЦКП ИСП РАН.

1. Introduction

In the present day, artificial intelligence (AI) technologies are actively addressing significant social challenges and have developed into a substantial industry. Linguistics has amassed a substantial amount of material that is suitable for the application of AI technologies. However, to the best of our knowledge, neural networks have not yet been utilized in comparative-historical linguistics in Russia.

Globally, the first studies on the automatic computational detection of cognates across languages appeared roughly twenty-five years ago. Initially, it was attempted to identify cognate words in the basic vocabulary of different languages using clustering methods (see [1]). These methods rely on comparing words with the same meaning from the basic vocabulary. Phonemes are grouped into clusters, which can vary greatly in size – for instance, clusters of vowels in medial position or consonants in final position may be quite large, whereas clusters of plosives or affricates may be relatively small.

In 2012, German researchers J.M. List and R. Forkel developed the well-known LexStat algorithm [2], which is available online as part of their collection of related tools, some of which were built upon this algorithm. This program, which combines a clustering algorithm with optimally calibrated weights, determines which words from two lists of basic vocabulary are cognates and which are not.

In a 2017 paper, G. Jäger developed a machine vector model and state-of-the-art (SOTA) phonetic alignment algorithms that make it possible to identify cognates in multilingual wordlists [3]. Based on basic vocabulary lists, it was shown that the algorithm performs rather poorly on small datasets, but its accuracy improves as the number of lexicons compared increases. Interestingly, the evaluation was conducted using data from three language families – Indo-European, Altaic, and Austronesian. The highest error rate was observed for the Indo-European languages, which have the longest tradition of study. This demonstrates that even clearly related words, for example in the Indic and Germanic branches, can differ phonetically to a considerable degree. The authors of this highly cited paper on clustering methods concluded that such methodologies cannot replace comparative-historical linguistic analysis.

As far as we know, one of the earliest attempts to use neural networks for cognate detection was undertaken in 2007 by Russian programmers working at the time in the United Kingdom and Bulgaria (see [4]). Their method was tested on English, French, German, and Spanish text corpora. The aim was to train a neural network to distinguish cognate words from those that merely exhibit "false similarity." The accuracy of cognate detection ranged from 81% for the German-English pair to 89.6% for the English-French pair. Overall, considering that the method was applied not only to basic vocabulary but also to broader text corpora in well-studied languages, this was a rather successful experiment. This study showcased the potential of neural networks in the field of comparative-historical linguistics. This paper has been extensively cited over the past fifteen years, particularly in studies focused on developing neural networks for historical-comparative linguistic research. (see [5-7]).

In [5], the authors applied neural network technology to five genetically related languages – Romanian, French, Spanish, Portuguese, and Italian – and to one unrelated language, Turkish. Their goal was to create, based on existing etymological research on Romanian, a neural network capable of distinguishing cognate and non-cognate words across the five languages. The accuracy of their method reached 87% when using a Support Vector Machine (SVM) with alignment-based features.

In [6], the author sought to move beyond language groups with well-established etymologies, although, in our opinion, the data used were not entirely reliable. The primary source was the Austronesian cognate database, which was available online in 2015 at http://language.psy.auckland.ac.nz/austronesian/ (now inactive). It is well known that no generally accepted system of regular phonetic correspondences exists for the Austronesian family. The second source was the Indo-European etymological database at http://ielex.mpi.nl/ (also currently inaccessible). The printed version of the corresponding monograph [8] shows that it includes only basic vocabulary etymologies, not full dictionaries. A third source was an etymological database of Mesoamerican Mayan languages described in [9]. However, as the authors themselves note, these etymologies were not the result of detailed linguistic work but were obtained automatically using the Levenshtein distance metric (which measures the difference between two sequences of characters). From studies of Indo-European, Uralic, and Altaic languages, we know that two words can have a minimal Levenshtein distance yet not be cognate according to regular phonetic correspondences. Thus, it seems that only the Indo-European etymologies could be considered somewhat reliable – and even they are currently unavailable for verification.

For training, the dataset included 167,676 word pairs (cognates and non-cognates) from Austronesian languages, 83,403 from Indo-European, and 63,028 from Mayan – meaning that only 26% of the material came from a language family with a well-established system of regular correspondences. In our view, such unreliable training data makes further use of this neural network impractical. The network itself is available online at GitHub – PhyloStar/SiameseConvNet: Performs cognate identification using Siamese Convolutional Networks, but the author has not updated it in the last eight years, and we have found no references to its use by other researchers.

In terms of dataset scale, a major breakthrough was achieved with CogNet, developed jointly by scholars from Australia, France, and Italy (see [10]). This resource combines material on eight

million cognates across 338 languages (see CogNet – UKC – Universal Knowledge Core for details [11]. According to its creators, the accuracy of automatic cognate detection reaches 94%. However, a closer examination reveals that for the languages of the Russian Federation, the database primarily relies on 100-word lists, many of which are extremely incomplete – for example, at the time of access, only 17 words were recorded for Even and 15 for Dolgan. In many cases, translations are erroneous, or only one of several synonyms is given, often an archaic or obsolete one – such as the Russian word lik instead of the common word litso ('face') (see Universal Knowledge Core | .:DataScientia: [11]. Even the examples displayed on the project's main page (GitHub – kbatsuren/CogNet: CogNet: [12] a large-scale, high-quality cognate database for 338 languages, 1.07M words, and 8.1 million cognates) contain errors in Altaic etymologies, while Uralic data is barely represented.

Nevertheless, the merits of this resource cannot be denied: it successfully integrates dozens of large online explanatory dictionaries for various world languages and, in many cases, constructs quite accurate etymologies while offering excellent visualizations of the results on a map.

In this concise overview, it is impractical to enumerate all neural network projects dedicated to cognate detection. However, it is noteworthy that, in our opinion, this research direction currently stands as one of the leading trends in comparative-historical linguistics globally. Developments in this field are not confined to Europe, America, and Australia. Notably, in 2017, a paper was published on the identification of etymologies in Arabic using neural networks (refer to [13]). Subsequently, in 2020, similar research emerged for the languages of India (refer to [14]). Furthermore, in 2024, research was conducted on Chinese (refer to [15]).

In contrast, the impact of this process on the languages of the peoples of the Russian Federation has been minimal to date. Indeed, CogNet query maps indicate that the territory of Russia remains largely devoid of linguistic data (see Universal Knowledge Core | .:DataScientia: [11]).

From this overview, it becomes clear that existing neural network models achieve very high accuracy in cognate detection (87–95%) when applied to languages included in their training data, provided that the quality of that data is high. We believe it is crucial that, for the languages of the Russian Federation, training data be based on reliable sources – compiled from the most authoritative existing etymological dictionaries and supplemented with modern research conducted by professional linguists – rather than on the kinds of crowdsourced or automatically generated materials used in many lesser-known neural networks. Furthermore, it is essential that neural network outputs undergo expert evaluation by specialists, as is standard practice in medicine, rather than being published in raw form as in CogNet, since such practices devalue genuine scientific achievements and often propagate inaccurate information.

Regrettably, the code for all known neural networks specifically designed for etymological research is inaccessible to users, and there are no publicly available datasets that facilitate the verification of their accuracy. Furthermore, it is noteworthy that foreign projects developing neural networks for etymology are predominantly led by programmers, and their primary objective is not the discovery of novel etymologies but rather the assessment of network performance on existing datasets. Given that these neural networks are not open-access, the research conducted abroad has yielded limited value from a comparative-historical linguistic perspective.

At present, the LingvoDoc platform [16] hosts dictionaries representing more than two thousand dialects of the languages of the Russian Federation. These dictionaries have been partially interconnected through etymological links. This was accomplished manually, utilizing LingvoDoc tools that rely on phonetic and semantic regular correspondences. Consequently, over 1.5 million words were connected through etymological relationships. This material formed the basis for training the neural network developed by the authors.

In the first part of the present article, we describe the technical features of this neural network; in the second, we present an overview of the types of etymological proposals generated by processing the Finnish and Erzya dictionaries using the network.

2. Neural network principles

In the present study, a two-stage approach is proposed for the task of automatic cognate identification in Uralic language corpora. At the first stage, a Siamese neural network was implemented. It takes into account graphical (orthographic) information from the input examples. Evaluation on the validation set demonstrated an average accuracy of approximately 78%.

At the second stage, the model architecture was expanded with an additional processing path for word translations and several heuristic techniques (a Boolean feature **exact_match** with a fixed correction factor, learnable weight coefficients α/β, and a threshold τ = 0.9), which allowed the classification accuracy to be increased to 92%.

The following sections describe the data used, the model architecture, and key training and tuning parameters that ensure network convergence and generalization capacity.

2.1 Dataset

The training data for the model were sourced from the LingvoDoc platform [16], which offers tools for the creation, annotation, and storage of electronic dictionaries, corpora, and concordances for diverse languages, encompassing phonetic and etymological analysis.

The initial corpus contained approximately 98,000 unique lexical entries from dictionaries of Uralic languages, each entry manually verified and supplied with a list of cognates. The lists were converted into all possible "cognate–cognate" combinations, after which duplicate pairs were removed; the final collection of positive examples contained about 1 million records.

A balanced negative class was generated using random negative sampling, excluding any pairs that overlapped with the positive examples.

For both stages, the training set has the general form

$$D = \{(x_i, y_i)\}_{i=1}^{N},\ y_i \in \{0,1\},$$

where the structure differs by stage:

- **Stage 1:** $x_i = \left(w_i^{(1)}, w_i^{(2)}\right)$
- **Stage 2:** $x_i = \left(w_i^{(1)}, t_i, w_i^{(2)}, t_i\right)$,

where $w_i^{(j)}$ is a word from language *j*, and t_i is its translation into Russian.

$$y_i \begin{cases} 1, \text{если } w_i^{(1)} \text{ и } w_i^{(2)} \text{когнаты}, \\ 0, \text{если они не когнаты } \wedge. \end{cases}$$

In the testing phase, the task for each pair is to predict the label y ^∈{0,1}.

Example of a training record:

Ала́ ала́-мъ 1 (for positive example)

kvɛ'ʃkəldʑəgu simy 0 (for negative example)

Initially, the model included 16-dimensional binary vectors of phonetic features [17]. However, subsequent analysis revealed that the lack of unified transcription standards in the source corpora introduced considerable noise. As a result, the phonetic vectors were excluded, and the focus shifted to orthographic features and the semantic representations of translations.

For the fine-tuning stage, an additional corpus of 700,000 translated pairs was prepared, symmetrically distributed between the two categories – cognates and non-cognates.

Example of a fine-tuning record:

ала́ город ала́ − мъ город 1 $(for\ positive\ example)$

$kvɛ'ʃkəldʑəgu$ отдохнуть $simy$ глаз 0 $(for\ negative\ example)$.

2.2. Model Architecture Description

At the first stage, a Siamese neural network was used, consisting of two identical branches, each processing one of the words in the analysed pair $w_i^{(1)}$ and $w_i^{(2)}$.

Each branch takes as input a sequence of characters, represented by two types of embeddings with dimensionality d = 128:

- **Character embedding** $E_{char}(X) \in R^{L\times D}$, which maps each character into a continuous vector space of dimension *D*;
- **Positional embedding** $E_{pos}(X) \in R^{L\times D}$, encoding the position of each character within the sequence (where *L* is the word length).

As a result, an input representation matrix is formed: $X^{(0)}=E_{char}(X) + E_{pos}(X)$.

To prevent overfitting, SpatialDropout1D (probability = 0.2) was applied, implemented as **Dropout2d** on a tensor of shape (*batch*, *d*, *L*), which zeroes out entire embedding channels [18].

Sequence processing is performed by a bidirectional LSTM (BiLSTM) with a hidden state size of 64, capturing contextual dependencies in both forward and backward directions [19]. The resulting features are passed through four transformer blocks, each implementing a multi-head attention mechanism that models complex nonlinear dependencies between characters:

$$Attention(Q,K,V) = softmax\left(\frac{QK^t}{\sqrt{d_K}}\right)V,$$

where *Q*, *K*, and *V* are projections of the input data, with the dimensionality of the keys being (*num_heads = 4*, key dimension per head = 128 / 4 = 32) [6].

After the attention layer, a position-independent feed-forward network (FFN) with 128 neurons and ReLU activation is applied, along with normalization and dropout layers for training stability.

The features from the transformer blocks are averaged across the temporal axis (*torch.mean*):

$$u = \frac{1}{L}\sum_{i=1}^{L} h_{j,}u \in R^{2h},$$

which aggregates the entire sequence into a single fixed-size vector. The similarity between words is then computed via cosine similarity between their hidden representations:

$$s = \frac{\langle u_1, u_2\rangle}{\|u_1\|\|u_2\|} \in [-1,1].$$

All resulting values are concatenated into a single vector:

$$z = [u_1; u_2; s] \in R^{4h+1},$$

which serves as the input to the final classifier. The classification module is a three-layer fully connected perceptron. The first layer applies *LayerNorm* and *ReLU* to an affine transformation of *z*; the second hidden layer is processed similarly, and a linear output followed by a sigmoid σ yields the predicted probability $\hat{y}$:

$$h_1 = ReLu\big(LayerNorm(W_1 z + b_1)\big),$$

$$h_2 = ReLu\big(LayerNorm(W_2 h_1 + b_2)\big),$$

$$\hat{y} = \sigma(W_3 h_2 + b_3).$$

At the second stage, the Siamese network was fine-tuned on a dataset that included word translations, and several heuristic strategies were introduced to further improve accuracy.

The base architecture was expanded to handle dual pathways:

- The word_encoder consists of a BiLSTM layer (*hidden_size = 64, bidirectional*) that extracts contextual features, followed by two transformer blocks with multi-head attention ($h = 4$) and an FFN with 128 neurons to model global word dependencies.
- The translation_encoder uses an analogous BiLSTM layer, followed by four transformer blocks to process translations.

Features from both pathways are combined with trainable coefficients:

$$p_i = \alpha t_i + \beta w_{i,} \alpha + \beta = 1,$$

where α and β are parameters for translations and words, initialized as (0.7, 0.3) and trained jointly with the other network weights.

Additionally, a Boolean feature **exact_match** was introduced, equal to 1 if the first four characters of the translations (considering padding) coincide, and 0 otherwise:

$$m = 1\left\{\tau_{1:4}^{(1)} = \tau_{1:4}^{(2)}\right\}.$$

At the classification stage, vectors and are p_1 and p_2 concatenated together with their element-wise difference and product:

$$v = [p_1; p_2; |p_1, p_2|; p_1 \times p_2]$$

The standard MLP then outputs a base logit l_{base}, which is adjusted by a heuristic contribution:

which is combined with the heuristic term

$$l = l_{base} + \gamma m,$$

where the coefficient $\gamma(match_{coef})$ is fixed and is not updated during fine-tuning. For example, if the first four characters of the translations are identical (e.g., *lesnoy* → *lesnye*), the model adds a fixed correction *match_coef = 0.8* to the classifier output.

To convert logits into binary labels, a threshold τ (*threshold*) [2] was applied. Initially, the validation threshold was determined dynamically: the ROC curve on a held-out set yielded τ values in the range [0.70, 0.78], representing an optimal trade-off between true positives (TP) and false positives (FP), accounting for penalties on false positives [1].

However, when working with real data, a large number of pairs were incorrectly identified as cognates. To reduce false positives, a fixed threshold $\tau = 0.90$ was set – meaning that the model must be highly "confident" (probability ≥ 90%) before classifying a pair as cognate. In practice, this reduced false positives by approximately 2–3 times while maintaining acceptable recall, as most true cognates received high scores $p \geq 0.90$.

Training examples for both stages were randomly divided into training (90%) and validation (10%) sets. Optimization used the AdamW algorithm with initial parameters learning rate = 10^{-4} (and weight decay = (10^{-4}.The binary cross-entropy loss (BCEWithLogits) function [20] was employed.

3. Types of New Etymologies Obtained using the Neural Network

In the present article, we provide examples of types of etymological proposals generated by the neural network, identified during the processing of two dictionaries:

1. the Finnish–Russian dictionary, which contains 17,212 lexemes, available online [21];
2. the Erzya–Russian dictionary, which contains 8,512 lexemes [22].

The Finnish words were taken into new etymologies were checked by [23], Erzya words were checked by [25]. As a result of the neural network's operation, 146,474,120 possible pairings were analyzed, and 16,055 etymological proposals were presented to the user; see Fig. 1 for the output format.

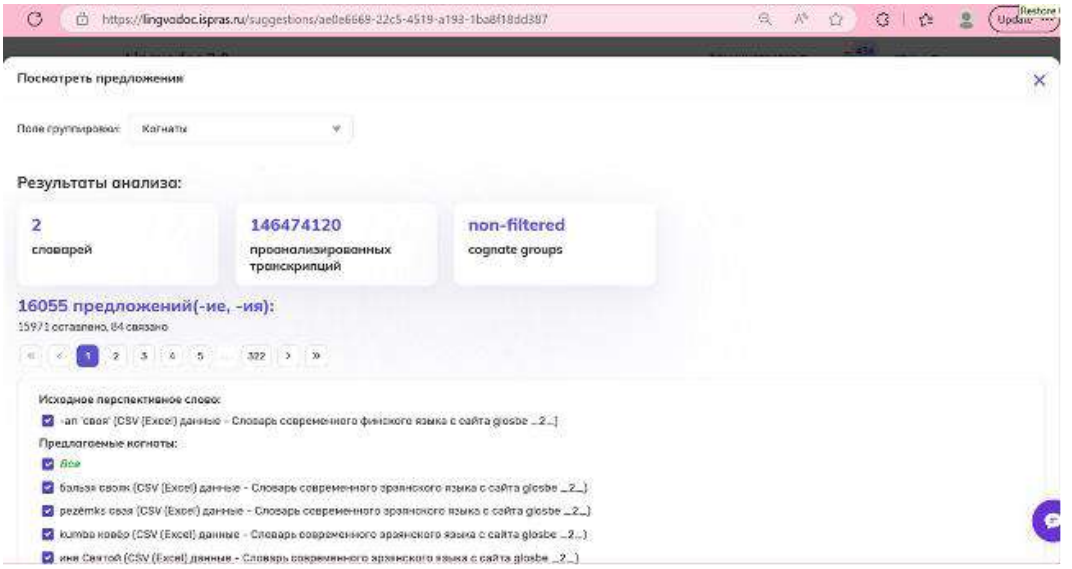


Fig. 1. Interface for displaying etymological proposals after dictionary processing by the neural network.

The user of the neural network must evaluate all of the network's proposals and highlight those deemed convincing, then each time click the "link" checkbox. As a result of our evaluation, 805 of the 16,055 etymological proposals were recognized as correct; these can be viewed in the dictionaries under the "cognates" column. Most of these proposals replicate existing etymologies reflected in the etymological dictionaries [23, 25]. However, more than 100 etymologies proposed by the neural network turned out to be new. They can be classified into three types:

1. Erzya additions to the existing "major" etymologies in [25], which have cognates in Finnish and several other Uralic languages but whose Erzya reflexes were previously unknown;
2. New distinct Finnish–Erzya etymological groups have been identified, comprising words that, according to [23], previously lacked cognates outside the Finnic (Baltic-Finnic) language family.
3. new Erzya parallels for Finnish words that, according to [23], were previously considered to be loanwords from Baltic or Germanic languages. In these cases, it is possible to demonstrate an earlier time of borrowing, already in Proto-Finno-Volgaic (with a split approximately 4,000 years ago), or to abandon the borrowing hypothesis and assume a native Uralic origin of the word.

It is evident that assessing the quality and novelty of the neural network's etymological proposals requires the user to undertake substantial research: verifying all hypotheses against existing etymological dictionaries and evaluating them in terms of the system of regular correspondences that underlies the etymologies accepted in [25; 23]. Below, as an illustration, we present 15 new etymological proposals across these groups.

3.1 Erzya Additions to Existing Etymologies

1. For the etymology FU **aja*- 'to hunt' > Fin. *aja*- 'to drive (game), to hunt, to lead', Est. *aja*- 'to move forward, to watch, to chase', Saami *vuoggje- -j*- 'to ride (a horse, reindeer), to lead' (N), *vuodjē*- 'to drive a car' (L), *vȳjje*- (T), *vujje*- (Kld), *vuajje*- (Not.) 'to drive a car, to direct', Udm. *uj*-, *uĺ*- (S), *ujî*- (G) 'to chase', Komi *voj*- (Skr) 'to be headstrong, to ignore the reins, to descend very fast from a mountain', *vojlį*- (Vm. I) 'to run, gallop, race', *vojed*- 'to run, to start running', *vojledlį*- 'to hunt', Mansi *wujt*- (K), *wojt*- (KM), *vujt*- (KO) 'to hunt' [14:4], we propose to add the comparison with Erz. *ajdäms* 'to drive, roll, rock',

where *-d-* is a suffix forming verbs with the meaning "single occurrence of an action", see [26].

2. For the etymology FU **tärmä* 'strength, strong' > Fin. *tarmo* 'energy, strength', Saami *dar'bmo -bm-* (N) 'energy, strength', *dar'bme -rbm-* (L), *tar'mē* 'strength, energy', Khanty *tärəm* (V), *tə̀rəm* (DN), *tarəm* (Kaz.) 'strong, energetic'?, Mansi *tēriŋ* (P), *tērəŋ* (K) 'agile, quick, wild', *tēr* (K) 'giftedness' ?, [25: 517], we propose to add the comparison with Erz. *tarmo* 'energy'. The Erzya forms provide evidence for the reconstruction of the back-vowel series. It may be necessary to separate the Ob-Ugric forms from this etymology, as the authors of [25: 517] also mark them with a question mark. Alternatively, it may be assumed that the same alternation of front and back vowels as in PU **kakta* ~ **käktä* 'two' applies to these forms.
3. For the etymology PF **kansa* 'people, folk, friends' > Fin. *kansa* 'people, nation, folk', Est. *kaasa* 'friend, companion, spouse', Saami *gaʒ'ʒe -ʒʒ-* (N) 'household, community', (T) *kaince*, (Kld) *kāinc*, 'companion', Udm. *kuz* (S) 'pair', (S G) *kuzo*, (K) *kuzo* 'pair', Komi *goz* (S P) 'pair', *guz* (PO) 'pair', *gozja* (S P) 'paired, married couple', [25: 645], we propose adding Erz. *kan* 'clan (line of generations), tribe'. In [25] there are no other etymologies with the cluster **-ns-* that have a reflex in Erzya, and therefore such development cannot be excluded.
4. For the etymology FU **kira* 'oath, curse, swearing' > Fin., Karel. *kiro* 'curse', Est. *kiruda* 'cursed', Veps *kirota* 'cursed', Votic *tširota*, Saami (N) *gârro* 'curse', Mansi (L) *kor-* 'cursed', Khanty (I) *korəm*, (E) *χurəm* 'to get angry', we propose adding Erz. *krojams* 'to swear', *krojsema* 'obscenity, foul language', where *-j(a)-* is a frequent verbal suffix, see [26: 92]. The loss of **i* in the Mordvinic languages is typical when the stress in Proto-Finno-Volgaic fell on the ending; see [16: 33-45], cf. FV **šišna* 'belt' > Mord. *šna* (E, M), *kšna* (E) [25: 786, 28: 909].
5. In FU **aŋa* 'to open, to release' > Fin. *avaa-* 'to open, to widen', *avanto* 'ice hole', Est. *ava-* 'to open', Mord. *ankšima, avśima* (E), *ańćema* (M) 'ice hole', Khanty *aŋə-* (V) 'to untie (a knot), to open', *eŋ-* (O) 'to take off clothes, shoes', Mansi *ę̄ŋk-* (TJ, LU), *āŋoχo-* (So.) 'to take off a dress', *ōŋkws-* (P) 'to skin a bear', Hung. *old-* (Old Hung. *ód-*) 'to release; to untie (knots)' [25: 11], it seems to us that Fin. *avanto* 'ice hole' and Mord. *ankšima, avśima* (E), *ańćema* (M) 'ice hole' do not fully match the semantics of the other forms. This is also noted by the authors of [23: I 92], who mark the connection between Fin. *avanto* 'ice hole' and *avaa-* 'to open, to widen' with a question mark. We propose comparing Fin. *avaa-* 'to open, to widen' with Erz. *avtems* 'to open up, to open', where *-t-* is a verbal suffix; for its meaning and frequency see [26: 169]. It should be noted that Erz. *-v-* is a frequent reflex of FU *-ŋ-* in the position after back vowels, cf. FU *čaŋV-* 'to beat' > Erz. *Čavo-* [25: 53].
6. In FU **ćärke-* 'to break, to hurt' > Fin. *särke-* 'to break, to hurt', Saami *čērgiidi-* (N) 'to go numb (of limbs)', *tjär'ka* (U) 'strong tingling in limbs fallen asleep after sleep', *íšĕär̄Ḡa-* (Ko. P) 'to hurt (of a wound)', Mari *šäryе-* (W) 'to open, to destroy', Khanty *t̓erəɣ-* (Trj.), *śarĭj-* (Ni.), *śarĭ-* (Kaz.) 'to hurt', Mansi *ćärk-* (TJ), *śə̄rɣ-* (KU), *śarr-* (P), *śārɣ-* (So.) 'bedauern', *śāriɣ-* (So.), *śārɣ-* (N) 'weh tun', Hung. *sér-* 'Schmerzen haben, weh tun'.

3.2 New Finnish–Erzya Comparisons and Their Additional Cognates According to LingvoDoc Dictionaries

7. To the etymological group Fin. *nikka* 'hiccup', Est. *nigu* 'hiccup', Karel. *nikko* 'hiccup', Veps *niki-* 'hiccup', Votic *nikottaa* 'hiccup', Saami (N) *njâkkâstit* 'hiccup' [23: II 220], we propose adding Erz. *niktnetems* 'to hiccup, to sob, to belch'. As noted in [26: 50, 26: 169], *-tńe-* is an unusual but known verbal suffix, whereas *-t-* is a typical verbal suffix. In [29],

Komi *ńiktiniі* 'to choke, to sputter (from coughing, tears, etc.)', *ńiktini* (Der., P.) [31] are also compared with the Finnish and Saami forms, this comparison also supports the Proto-Finno-Permic origin of these forms and allows to reconstruct the PFPerm **niktV-* 'hiccup'.

8. To the etymological group Fin. *sirkka* 'grasshopper, cricket', Karel. *sirk* 'grasshopper', Karel. *tširkka* 'grasshopper, cricket', Veps *tširk* 'grasshopper, cricket', Votic *tširk, širk* 'grasshopper, cricket' [23: III 186], we propose adding Erz. *ciŕkun* 'grasshopper, cricket', where *-un* is an unusual nominal suffix; see [30: 210] for details. On the LingvoDoc platform, other cognates of this proto-form can also be found: Moksha (Koldais) *ćiŕkun*, (Mordovskie Yurtkuli) *ćerku·n*, (Uryum) *ćiŕkun* 'grasshopper'. In [29], Mari *č8rkiem* 'to chirp, to twitter' and Komi *ćirk* 'grasshopper' are also compared to this etymological group, allowing the reconstruction FPerm *ćirkV 'grasshopper, cricket'*.
9. To the etymological group Fin. *kierittä* 'to roll', *kiero* 'bent, twisted', Est. *keer* 'bent, twisted', Karel. *kiero* 'bent, twisted', Veps *keŕ* 'roll', Votic *tšērtā* 'wheel', Saami (N) *gierre* 'thread, cord' [23: I 354], we propose adding Erz. *keverdems* 'to roll' and Hung. *kever* 'to spin, to stir', which according to [32: 746] also lacks an etymology. In Finnish, the change **w > 0* is frequent in intervocalic position, cf. *lewe* > *lyö-* 'to strike' [25: 247], *luwe* > *luu* 'bone' [25: 254], *puwe* > *puu* 'tree' [25: 410]. Thus, comparison of Finnic, Erzya, and Hungarian forms allows the reconstruction FU **kiwer- 'to roll, to turn'*.
10. The forms Fin., Karel. *vain* 'only', Est. *vaid*, Veps *vaiše*, Votic *vaitas* 'only' [23: III 392] can be compared with Erz. *vańks* 'only, merely, pure', where *-ks* is an Erzya nominal suffix; for its meaning, see [33]. One can reconstruct FV **wajn* '*only'*. In [25] there are no words with intervocalic **-jn-* that have Erzya reflexes, but the cluster **-jm-* > Erz. *-m-*, cf. FU **śajma* 'wooden vessel, boat' > Erz. *śuma, śima* 'log' [25: 456], while the cluster *-jn-* > Komi, Udm. *ń*. Therefore, a similar change in Erzya seems possible.

3.3 Erzya Parallels for Loanwords in (Baltic-)Finnic (and Saami) Languages

3.3.1 Examples of Germanic Loanwords

11. Fin. *tuoni* 'death', *tuonela* 'hell', Karel. *tuoni* 'death', Est. dial. *toonekurg, toonkurg*, Saami (N) *duodnâ* 'poor thing, death, hell'. To this etymological group we propose adding Erz. *tonači* 'hell', where *-či* is a typical affix for forming abstract nouns, cf. *мазы-чи* 'beauty', *сюпав-чи* 'prosperity, wealth'; see [34] for details. In [23: III 330] it is considered that the Finnic words are borrowed from PGmc. **dawīni > Old Norse dán* 'death', Norw. *dån* 'weariness' [23: III 330]. If this hypothesis is accepted, then a Proto-Germanic borrowing into Proto-Finno-Volgaic should be posited.
12. Fin. *kampa* 'comb' < Old Norse *kambr*, Norw. *kam* [23: I 294]. To this etymological group we propose adding Erz. *kaba* 'comb'. The Erzya reflex *-b-* < **mp* is not typical, but there are examples of **mp* > *-p-* in native vocabulary, cf. *kumpa* 'wave' > Erz. *kopildi-* 'to move in a wave-like motion'. It may be assumed that in borrowing, voicing could have occurred, which sporadically appears in Volgaic and Permic languages.
13. Fin. *avioliittovälittäjä* 'matchmaker', *avioliitto* 'marriage', *avio* 'spouse, marital partner, marriage', Est. *abielu* 'marriage' [23: I 92], we propose to compare with Erz. *avakuda* 'matchmaker', *ava* 'woman', Moksha *ava* 'woman', Mari *ava* 'mother'. In [23: I 92] a possible loan of the Finnic forms from Proto-Germanic **aiwō, *aiwa* 'law, norm' is noted as questionable; the closest semantic reflex to the Finnish is Old English *ǣw* 'law, marriage, spouse'. Note that the Mordvinic and Mari words may also be reflexes of PU **apV 'older female relative, aunt, elder sister'* [25: 15], although these parallels are absent in [25: 15], possibly due to semantic differences, since the standard phonetic reflex of **-p-* is Mord., Mari -v-. Phonetically similar words for 'woman' are also found in Turkic languages, cf.

PTurk. **apa* ‘mother, elder sister, aunt’, which in Tuvan has the reflex *ava*, see [35: 158-159]. Thus, in this case the hypothesis of Finnic forms borrowed from Germanic seems unlikely.

14. Fin. *viltti* ‘blanket’, Karel. *vilti* we propose to compare with Erz. *veĺtävks* ‘blanket’, *veĺtäms* ‘to cover with a blanket’, Moksha (Koldais) *véĺʹtə̂rda* ‘luxuriously embroidered blanket’. Comparing these forms allows for the reconstruction of PFV **wiltV ‘blanket’*. In [23: III 450] the Finnic forms are considered to be borrowed from Germanic, cf. Norw. *filt* ‘blanket’. This borrowing hypothesis appears quite plausible and, if accepted, would imply Germanic borrowings into Proto-Finno-Volgaic.

3.3.2 Examples of Baltic Loanwords

15. Fin. *vuode* ‘mattress’, Est. *voodi* ‘mattress’, which according to [23: III 472] is derived from *vuota* ‘skin flayed from a large animal’, we propose to compare with the Baltic-Finnic forms: Erz. *vatola* ‘mattress’, where *-la* is a suffix forming nouns, cf. *чова-ля* ‘bead’. The correspondence Erz. *a* – Fin. *uo* is also typical, cf. Erz. *kaŕ* – Fin. *kuori* < FU **kore* (**kōre)* ‘bark’ [25: 184]. According to [23: III 476], the Finnic words are considered borrowings from Baltic, cf. Lith. *óda* ‘skin’, Latv. *āda* ‘skin’. Without pausing here to evaluate the likelihood of borrowing these words from Baltic, we note that if the comparison with Erzya is accepted, then borrowing into Proto-Finno-Volgaic must be assumed.

16. Fin., Karel. *härkä* ‘ox’, Est., Veps *härg* ‘ox’, Votic *ärtšä* ‘ox’, we propose to compare with Erz. *šeŕge* ‘ox’, which go back to FV **čärkV*. In [23: I 210] the Finnic words are considered Baltic borrowings, cf. Lith. *žìrgas*, Latv. *zirgs* ‘horse’. Here, as in the previous etymology, the phonetics and semantics do not precisely correspond to the Finno-Volgaic, but if the borrowing hypothesis is accepted, they again indicate the greater intensity of early contacts between speakers of Baltic and Finno-Volgaic languages.

4. Conclusion

It may be concluded that the neural network we developed is sufficiently effective for identifying new etymologies even in languages as well studied as Finnish and Erzya. The analysis of sizable dictionaries (several thousand lexemes) using neural networks makes it possible to supplement existing etymologies, propose new ones, and refine the chronology of borrowings. From a linguistic perspective, the most intriguing result is that some words previously regarded as separate Baltic or Germanic borrowings into the Finnic (Baltic-Finnic) and Saami languages have parallels in the Mordvinic languages, whose phonetic correspondences in several cases (for example, Fin. *härkä* ‘ox’ – Erz. *šeŕge* ‘ox’) indicate a common and ancient source of borrowing. In such cases, we must assume either that these are not borrowings at all but inherited words in the Finno-Volgaic–and possibly Finno-Ugric–languages, and that the similarity to Germanic or Baltic is coincidental or points to a shared Nostratic origin; or else we must posit direct contacts between speakers of Proto-Finno-Volgaic and Baltic and Germanic groups.

While the hypothesis of contacts with the Balts and/or Balto-Slavs was proposed already in [36: 191] – even though that study identified no more than ten lexemes that could be regarded as borrowings into Proto-Finno-Volgaic–and is widely accepted and further substantiated in [37], to our knowledge there has previously been no evidence adduced for contacts between Germanic speakers and speakers of Proto-Finno-Volgaic. It has generally been assumed that borrowings from Germanic could have entered the Volgaic languages only via Russian; see [38]. However, among the four Germanic borrowings into Finno-Volgaic considered here, there is only one case in which a similar word appears in Tambov Russian dialects–*ava* ‘woman’ [39: I 196] – and that item is regarded as a borrowing from the Mordvinic languages, given that it is attested solely within the contact area.

List of abbreviatons

E. – Erzya
Est. – Estonian
PF – Finno-Permic
Fin. – Finnish
FU. – Proto-Finno-Ugric
PFV. – Proto-Finno-Volgaic
G. – Glazov Udmurt dialect
Hung. – Hungarian
I. – Irtysh Khanty dialect / Izhma Komi dialect (context-dependent)
K. – Kondinsk Mansi dialect
Kaz. – Kazym Khanty dialect
Karel. – Karelian
Khanty – Khanty
Kld. – Kildin Saami
Ko. – Skolt Saami
Komi – Komi
KM. – Middle Kondinsk Mansi dialect
KU. – Lower Kondinsk Mansi dialect
Latv. – Latvian
Lith. – Lithuanian
L. – Lozva Mansi dialect / Lule Saami (context-dependent)
LU. – Lower Lozva Mansi dialect
M. – Moksha
Mansi – Mansi
Mari – Mari
Moksha – Moksha
Mord. – Mordvinic
Ni. – Nizyam Khanty dialect
Norw. – Norwegian
Not. – Notozero Saami
O. – Obdorsk Khanty dialect
Old Hung. – Old Hungarian
Old Norw. – Old Norse
P. – Pechora Komi dialect / Pelym Mansi dialect / Pite Saami (context-dependent)
PFV – Proto-Finno-Volgaic
PF – Proto-Finno-Permic
PFPerm – Proto-Finno-Permic (if retained)
PGmc. – Proto-Germanic
PO. – East Permyak dialect
PU. – Proto-Uralic
S. – Middle Sysola Komi dialect / Sarapul Udmurt dialect (context-dependent)
Saami – Saami
Skr. – Sysola (Syktyvkar) Komi dialect
So. – Sosva Mansi dialect
TJ. – Tavda Mansi dialect
T. – Ter Saami
Trj. – Tremjugan Khanty dialect
U. – Ume Saami
Udm. – Udmurt
V. – Vakh Khanty dialect

Veps. – Veps
Vm. – Vym Komi dialect
Vot. – Votic
W. – Western (Forest) Mari dialect

References

[1]. Bergsma Sh., Kondrak G. Alignment-based discriminative string similarity. In Proc. ACL. 2007
[2]. Johann-Mattis List. 2012. LexStat: Automatic Detection of Cognates in Multilingual Wordlists // Proceedings of the EACL 2012 Joint Workshop of LINGVIS & UNCLH, pages 117–125, Avignon, France. Association for Computational Linguistics.
[3]. Jäger G., List J.-M., Sofroniev P. Using support vector machines and state-of-the-art algorithms for phonetic alignment to identify cognates in multi-lingual wordlists // Conference: Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, 2017, Long Papers.
[4]. Mitkov R., Pekar V., Blagoev, D. et al. Methods for extracting and classifying pairs of cognates and false friends. Machine Translation 21, 29–53 (2007).
[5]. Dinu L.P., Ciobanu A.M. Building a Dataset of Multilingual Cognates for the Romanian Lexicon // Proceedings of the Ninth International Conference on Language Resources and Evaluation LREC 2014, p. 3313-3318.
[6]. Rama T. Siamese Convolutional Networks for Cognate Identification // Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers, p. 1018–1027.
[7]. Fourrier C., Sagot B. Probing Multilingual Cognate Prediction Models // Findings of the Association for Computational Linguistics: ACL 2022. p. 3786-3801.
[8]. Dyen I., Kruskal J. B., Black P. An Indo-European classification: A lexicostatistical experiment. Transactions of the American Philosophical Society 1992, 82(5), p. 1–132.
[9]. Wichmann S., Holman E.W. Languages with longer words have more lexical change // Approaches to Measuring Linguistic Differences, 2013, p. 249–281.
[10]. Batsuren Kh., Bella G., Giunchiglia F. A large and evolving cognate database // Language Resources and Evalution, vol. 56, 2022, p. 1-25.
[11]. https://ukc.datascientia.eu/concept, дата обращения 26.11.2025.
[12]. https://datascientiafoundation.github.io/LiveLanguage/datasets/cognet/, дата обращения 26.11.2025.
[13]. Alreshidi H., Aldhlan K. Auto-Extracting Method of Cognates Words in Arabic and English Languages // International journal of advanced studies in Computer Science and Engineering (IJASCSE), vol. 6, issue 01, 2017.
[14]. Kanojia D., Bhattacharyya P., Kulkarni M., Haffari G. Challenge Dataset of Cognates and False Friend Pairs from Indian Languages // LREC 2020, p. 1-12.
[15]. Pulini M., List J.-M. Finding language-internal cognates in Old Chinese // Bulletin of Chinese Linguistics 2024, 17(1), p. 53–72.
[16]. https://lingvodoc.ispras.ru/, дата обращения 26.11.2025.
[17]. Fawcett, T. (2006). An introduction to ROC analysis. Pattern Recognition Letters, 27(8), 861–874.
[18]. Tompson, J., Jain, A., LeCun, Y., & Bregler, C. (2015). Efficient Object Localization Using Convolutional Networks. Proceedings of CVPR. https://arxiv.org/pdf/1411.4280**Ошибка! Недопустимый объект гиперссылки.**
[19]. Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11), 2673–2681.
[20]. Loshchilov, I., & Hutter, F. (2019). Decoupled Weight Decay Regularization. ICLR 2019.
[21]. https://lingvodoc.ispras.ru/dictionary/11457/163277/perspective/11457/163278/view, дата обращения 26.11.2025.
[22]. https://lingvodoc.ispras.ru/dictionary/11459/62466/perspective/11459/62467/view, дата обращения 26.11.2025.
[23]. Suomen sanojen alkuperä, ed. by Forsberg U.-M., Itkonen E. Helsinki, 1992-2000.
[24]. Erzya-Russian dictionary: 27000 lexems, ed. By Serebrennikova B. A. , Buzakovoj R. N., Mosina M. V. M., 1993.
[25]. Uralisches etymologisches Wörterbuch, ed. by K.Rédei. Budapest, 1986 – 1988.
[26]. Mesarosh E. Verb-forming suffixes in the Erzya language. Studia Uralo-Altaica 42. Seged, 1999.

[27]. Normanskaya Ju.V. Reconstruction of the Proto-Uralic paradigmatic stress and its influence on the development of the vocalism system. M., 2018.
[28]. Paasonen H. Mordwinisches Wörterbuch. II Band. Helsinki, 1992.
[29]. Lytkin V.I., Gulyaev V.G. A short etymological dictionary of the Komi language. Moscow, 1970.
[30]. Ariskina T.P. Suffixed nouns in the Erzya language: semantics and functioning // Vestnik ugrovedeniya № 2(8), 2018.
[31]. Comparative dictionary of Komi-Zyr'an dialects, complied by Zhilina T. I., Saxarova M. A., Sorvacheva V. A. Syktyvkar, 1961.
[32]. Etymologisches Wörterbuch des Ungarischen, ed. by Loránd Benkő. Budapest, 1993.
[33]. Ryabov I.N. Word-formation relations between parts of speech in the Erzya language. Saransk, 2000.
[34]. Vodyasova L.P. Ways of expressing grammatical meanings in the morphology of the Erzya language // Lingvistika №30, 2016 (https://sci-article.ru/stat.php?i=1455731730).
[35]. Etymological dictionary of Turkic language, ed. by Sevortyan E.V., t. 1, M., 1974.
[36]. Kalima J. Itämerensuomalaisten kielten balttilaiset lainasanat. Helsinki.
[37]. Napol'skix V.V. The Balto-Slavic language component in the Lower Kama region in the middle of the 1st millennium AD // Slavyanovedenie, 2006, 2, 3-19.
[38]. Butylov N.V. Foreign language vocabulary in Mordovian languages (Indo-European borrowings). Saransk, 2006.
[39]. Dictionary of Russian dialects, ed. by F.P.Filin, vol. I. Leningrad, 1965.

Информация об авторах / Information about the author

Юлия Викторовна НОРМАНСКАЯ – доктор филологических наук, главный научный сотрудник, заведующая лабораторией «Лингвистические платформы» Института системного программирования им. В.П. Иванникова РАН, ведущий научный сотрудник отдела Урало-алтайских языков Института языкознания РАН.

Yulia Viktorovna NORMANSKAYA – Dr. Sci. (Philology), Chief Researcher, Head of the Laboratory "Linguistic Platforms" at Ivannikov Institute for System Programming of the Russian Academy of Sciences; Leading Researcher, of the Department of the Ural-Altaic Languages at the Institute of Linguistics of the Russian Academy of Sciences.

Оксана Владимировна ГОНЧАРОВА – кандидат филологических наук, старший научный сотрудник лаборатории «Лингвистические платформы» Института системного программирования им. В.П. Иванникова РАН.

Oxana Vladimirovna GONCHAROVA – Cand. Sci. (Philology), Senior Researcher of the Laboratory "Linguistic Platforms" at Ivannikov Institute for System Programming of the Russian Academy of Sciences.

DOI: 10.15514/ISPRAS-2025-37(6)-43

CC BY

Сопоставление номенклатур товаров ресторанов и поставщиков с помощью LLM – Case Study для ресторанного холдинга

С. Джин, ORCID: 0000-0002-8242-6157 <sedzin@hse.ru>
П.Б. Панфилов, ORCID: 0000-0001-6567-6309 <ppanfilov@hse.ru>
А.С. Сулейкин, ORCID: 0000-0003-2294-6449 <aless.sull@mail.ru>

Национальный исследовательский университет «Высшая школа экономики», Россия, 101000, г. Москва, ул. Мясницкая, д. 20.

Аннотация. В современном ресторанном бизнесе точное сопоставление номенклатуры продуктов между ресторанами и поставщиками является критически важной задачей. Эффективное управление запасами и оптимизация закупок напрямую влияют на прибыльность бизнеса. С ростом числа поставщиков и ассортимента продукции традиционные методы сопоставления становятся менее эффективными. В данном исследовании предлагается использовать большие языковые модели (LLM) для автоматизации и повышения точности сопоставления продуктов. В рамках пилотного проекта для ресторанного холдинга мы протестировали пять групп продуктов (креветки, угорь, сыр пармезан, творог, сливочное масло), достигнув средней точности тестирования 83,8%. Архитектура решения использует быстрое проектирование, платформы с низким уровнем кода, такие как Flowise, и интеграцию с Telegram для удобной обработки данных. Ключевые проблемы, включая семантическую неоднозначность и галлюцинации моделей, были решены с помощью предметно-ориентированных словарей и валидации. Такой подход сокращает ручную работу примерно на 90%, что позволяет создавать масштабируемые решения для цепочки поставок, применимые не только в ресторанах, но и в розничной торговле и электронной коммерции.

Ключевые слова: большие языковые модели; управление цепями поставок; сопоставление номенклатуры продуктов; автоматизация; оптимизация запасов.

Для цитирования: Джин С., Панфилов П.Б., Сулейкин А.С. Сопоставление номенклатур товаров ресторанов и поставщиков с помощью LLM – Case Study для ресторанного холдинга. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 163–176. DOI: 10.15514/ISPRAS–2025–37(6)–43.

Благодарности: Работа выполнена в рамках программы фундаментальных исследований НИУ ВШЭ.

Mapping Restaurant and Supplier Product Nomenclatures Using LLM – Case Study for a Restaurant Holding

S. Jin, ORCID: 0000-0002-8242-6157 <sedzin@hse.ru>
P.B. Panfilov, ORCID: 0000-0001-6567-6309 <ppanfilov@hse.ru>
A.S. Suleykin, ORCID: 0000-0003-2294-6449 <aless.sull@mail.ru>
National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000, Russia.

Abstract. In the modern restaurant business, accurate mapping of product nomenclatures between restaurants and suppliers is a critical task. Effective inventory management and procurement optimization directly impact business profitability. With the increase in suppliers and product variety, traditional mapping methods become less efficient. This study proposes using large language models (LLM) to automate and improve the accuracy of product matching. Through a pilot project for a restaurant holding, we tested five product groups (shrimp, eel, parmesan cheese, cottage cheese, butter), achieving an average test accuracy of 83.8%. The solution architecture leverages prompt engineering, low-code platforms like Flowise, and Telegram integration for user-friendly processing. Key challenges, including semantic ambiguity and model hallucinations, were addressed via domain-specific dictionaries and validation. This approach reduces manual effort by approximately 90%, enabling scalable supply chain solutions applicable beyond restaurants to retail and e-commerce.

Keywords: Large Language Models; Supply Chain Management; Product Mapping; Automation; Inventory Optimization.

For citation: Jin S., Panfilov P.B., Suleykin A.S. Mapping Restaurant and Supplier Product Nomenclatures Using LLM – Case Study for a Restaurant Holding. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 3, 2025, pp. 163-176 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-43.

Acknowledgements. The work was carried out within the framework of the fundamental research program of the National Research University Higher School of Economics.

1. Введение

1.1 Эволюция применения LLM в управлении цепочками поставок

Большие языковые модели (Large Language Model, LLM) стали революционным инструментом в управлении цепочками поставок (Supply Chains Management, SCM), позволяя бизнесу оптимизировать операции, улучшать принятие решений и автоматизировать процессы. Традиционные подходы, такие как системы на основе правил или моделей машинного обучения (Machine Learning, ML), часто не справляются с обработкой неструктурированных данных, многоязычных контекстов и быстро меняющихся сред. Модели LLM преодолевают эти ограничения за счет расширенных возможностей понимания естественного языка и контекстного анализа, что делает их высоко адаптивными к динамичным потребностям цепочек поставок [1-2].

1.1.1 Прогнозирование спроса и оптимизация запасов

Модели LLM продемонстрировали значительный потенциал в прогнозировании спроса за счет анализа исторических данных о продажах, рыночных тенденций и внешних факторов, таких как погода или геополитические события. Например, компания Coca-Cola использует прогнозирование на основе искусственного интеллекта (ИИ) для оптимизации графиков производства и уровней запасов, сокращая перепроизводство при удовлетворении спроса клиентов [1]. Аналитика на основе модели LLM ChatGPT применяется в розничной торговле для выявления тенденций спроса и улучшения управления запасами, что приводит к сокращению дефицита и повышению операционной эффективности [3].

1.1.2 Управление взаимоотношениями с поставщиками

Модели LLM улучшают управление взаимоотношениями с поставщиками, автоматизируя такие задачи, как составление контрактов, обработка счетов и отслеживание эффективности. Например, платформа Verusen на основе ИИ улучшает взаимодействие между покупателями и поставщиками за счет повышения качества данных и содействия возможностям вверх и вниз по цепочке [4]. Использование моделей LLM компанией Microsoft в управлении цепочками поставок показало значительное сокращение времени подключения поставщиков и мониторинга соблюдения контрактов [5].

1.1.3 Оптимизация логистики

В логистике модели LLM играют ключевую роль в оптимизации маршрутов и управлении сбоями. Компания Amazon использует ИИ для анализа моделей покупок и сезонных тенденций, чтобы заранее корректировать распределение запасов по своей сети. Это приводит к сокращению сроков доставки и повышению общей эффективности [2]. Компания DHL интегрирует робототехнику на основе ИИ в складские операции для задач сортировки и упаковки, значительно сокращая время обработки и позволяя сотрудникам сосредоточиться на более сложных задачах [1].

1.1.4 Интеграция знаний для поддержки решений

Большие языковые модели LLM способствуют улучшению принятия решений за счет интеграции разрозненных источников данных в связные графы знаний. Эти графы фиксируют сложные взаимозависимости между объектами цепочки поставок, обеспечивая получение информации о рисках и возможностях в реальном времени. Например, сервис Microsoft Dynamics 365 Copilot использует службы Azure OpenAI для прогнозирования сбоев у поставщиков или в регионах с одновременной генерацией практических рекомендаций для менеджеров цепочек поставок [5].

1.2 Ключевые проблемы использования LLM

Большие языковые модели LLM обладают преобразующим потенциалом для управления цепочками поставок, но их внедрение сопряжено с трудностями. В этом разделе рассматриваются основные препятствия, возникающие при внедрении LLM в контексте цепочки поставок в ресторанном бизнесе, с акцентом на семантическую неоднозначность, риски галлюцинаций, уязвимости системы безопасности и адаптацию персонала.

1.2.1 Семантическая неоднозначность

Серьезной проблемой, с которой, в частности, столкнулись разработчики в ходе реализации и тестирования системы, описываемой в данной работе, является том, что модели LLM ошибаются в специфичных контекстах (например, при сопоставлении продуктов “Easy Peel Shrimp” и “Pre-Cleaned Shrimp”). Эта проблема усугубляется еще больше в многоязычной среде [7-8].

Следствием проявления этой проблемы являются 23% ошибок сопоставления в тестах [1].

Предлагаемые возможные решения для этой проблемы включают в себя:

- Предметные онтологии (5k+ терминов), которые помогают снизить число ошибок на 19% [1];
- Ручная валидация спорных случаев [5].

1.2.2 "Галлюцинации" и объяснимость

Проблема «галлюцинаций» моделей LLM заключается в генерация моделью генеративного ИИ ложных данных из-за вероятностной природы работы модели LLM [8-9].

Практическими последствиями проявления «галлюцинаций» LLM в задачах управления цепочками поставок могут стать, например, ошибочные рекомендации по поставщикам.

Предлагаемые возможные решения для проблемы «галлюцинаций» LLM:

- Обучение моделей LLM на предметно-ориентированных (domain-specific) данных для снижения «галлюцинаций» LLM на 15% [8];
- Резервные механизмы (fallback) для неподдерживаемых запросов [5].

1.2.3 Уязвимости безопасности

Распространение решений на базе моделей машинного обучения вызвало к жизни и появление нового типа атак на сервисы и приложения на основе машинного обучения, такие как «вредоносное машинное обучение» (Adversarial Machine Learning, AML), одним из проявлений которого является отравление данных (poisoning attack) или манипуляции с обучающими данными (пример: утечка данных OpenAI в 2023 году) [11-12]. Также цепочки поставок могут страдать и от атак, связанных с распространением уязвимостей через зависимости (подробнее можно ознакомиться, например, в [13]).

Решения по борьбе с атаками отравления и уязвимостями зависимостей включают в себя:

- Дифференциальная приватность при обучении [11];
- Регулярные аудиты безопасности обеспечивают снижение уязвимостей на 30% [12].

1.2.4 Адаптация персонала

Проблема готовности персонала к использованию и доверия в отношении решений на основе генеративного ИИ обычно выражается напрямую в затратах на переподготовку сотрудников компании ($3800/сотрудника) [7] и снижении продуктивности на 22% [5, 9].

Решается эта проблема обычно двумя путями:

- Поэтапное обучение сотрудников (формулировка запросов + таксономия) для увеличения эффективности на +35% [5];
- Сотрудничество ИИ-специалистов и экспертов предметной области [10].

1.3 Критические пробелы в существующих исследованиях

Несмотря на явный преобразующий потенциал, внедрение решений на базе моделей LLM сталкивается с рядом ограничений, которые необходимо устранить для более широкого использования генеративного ИИ в управлении цепями поставок SCM (табл. 1).

Серьёзной проблемой применения моделей LLM остается «семантический дрейф». Например, при классификации продуктов в случае рассматриваемого нами ресторанного бизнес-кейса несогласованная терминология вроде "Jumbo Shrimp" вместо "Colossal Shrimp" составляет ~34% ошибок [6].

Кроме того, затраты на переподготовку персонала для внедрения моделей LLM и решений на их основе значительны. Так по данным McKinsey, средняя стоимость переподготовки сотрудников закупочных подразделений для эффективного использования инструментов ИИ составляет $3800 на сотрудника [7].

Табл. 1. Ограничения применения различных подходов.
Table 1. Limitations of different approaches.

Проблема	Традиционные подходы	Ограничения LLM
Адаптация к предметной области	Требует размеченных данных	Ограниченная точность «с нуля» (~82%) [2]
Доверие к операциям	Ручная проверка (5-7 дней)	«Черный ящик» логики решений [6]
Многоязычное выравнивание	Перевод на основе правил (~68% покрытия)	Уровень ошибок ~19% (на азиатских рынках [1])

1.4 Постановка задачи разработки и описание исследования

В данной работе представлен пример решения одной из центральных задач в управлении цепочками поставок, а именно: задачи установления корректного соответствия между товарной номенклатурой поставщиков и потребителей, которая напрямую связана с задачами корректного планирования запасов, управления взаимоотношениями с поставщиками, автоматизации таких бизнес-процессов, как составление контрактов, обработка счетов и отслеживание эффективности. Данная задача связана с рутинной обработкой больших объемов текстовых данных, что идеально подходит к ее автоматизации и реализации на основе решений генеративного ИИ. В работе рассмотрены архитектурные и технологические особенности предлагаемого решения прикладной задачи и проанализированы результаты опытной эксплуатации разработанного пилотного варианта системы, протестированного на реальных данных бизнеса ресторанного холдинга.

Структура работы включает в себя введение, разделы с описанием предлагаемого подхода к построению решения, результатов реализации и тестирования прототипа системы, обсуждения результатов опытной эксплуатации системы и заключения. Во введении рассматриваются возможности, которые технологии генеративного ИИ предлагают для автоматизации и улучшения управления цепочками поставок SCM, а также те проблемы и препятствия, с которыми сталкиваются разработчики решений на основе моделей LLM, на основе анализа примеров решения практических задач управления цепочками поставок. В разделе с описанием подхода к решению практической задачи представлена архитектура решения на основе собственных цифровых платформ и технологий с открытым кодом. Результаты прогонов прототипа системы на данных реального бизнес-кейса показаны в разделе, посвященном результатам практического внедрения системы с представлением м анализом количественных показателей функционирования системы в процессе ее опытной эксплуатации. В заключении подведены итоги реализации проекта и обсуждаются перспективы масштабирования решения, а также возможности адаптации и применения предлагаемого подхода к реализации аналогичных проектов как в рамках задач управления цепочками поставок, так и за пределами данной прикладной области.

2. Подход к решению практической задачи

2.1 Описание бизнес-кейса

Наша команда разработала пилотный проект для крупного ресторанного холдинга с целью автоматизации процесса сопоставления товарных номенклатур.

2.1.1 Описание целевой задачи

Для заданного наименования товара от заказчика (ресторана) необходимо выполнить автоматический поиск аналогов в общей базе товаров поставщиков.

Отбор аналогов должен осуществляться по заданным критериям разделения товаров из базы на три категории:

- Категория 1: точно совпадающие позиции;
- Категория 2: потенциально подходящие позиции;
- Категория 3: неподходящие позиции.

Система должна генерировать собственные критерии категоризации на основе полученного наименования товара и правил, учитывающих товарную группу и требования ресторана.

В рамках пилотной разработки были введены ограничения для реализации ранжирования товарных номенклатур по критериям от заказчика для пяти товарных групп: креветки, угорь, сыр пармезан, творог и сливочное масло. В табл. 2 представлен пример заданных критериев для категоризации номенклатур в списке креветок.

Табл. 2. Пример критериев отнесения номенклатур списка креветок к одной из трёх категорий.
Table 2. An example of criteria for classifying shrimp list items into one of three categories.

Категория	Критерии
Категория 1 (точно совпадающие позиции) *выделено зеленым*	Бренд Parmente
Категория 2 (потенциально подходящие позиции) *выделено желтым*	• Все позиции без слов «мягкий сыр» в названии; • Все позиции без упоминания стран (Швейцария, Италия); • Все позиции без слова «голова» в названии ИЛИ весом ≥4,5 кг • Все позиции без слов «Premium» / «Exclusive» в названии
Категория 3 (неподходящие позиции) *выделено красным*	Все остальные позиции

2.2 Архитектура решения на базе DUC SmartSearch и модели LLM

Для разработки использовалась собственное цифровое решение DUC SmartSearch [14] на базе Danswer/Flowise, где DUC SmartSearch и Danswer (ныне Onyx [15]) – это платформы для создания ИИ-ассистентов и RAG-конвейеров, а Flowise – это платформа с минимальным ручным кодированием (low-code) для создания ИИ-ассистентов [16]. Весь конвейер обработки данных был реализован на Flowise. Альтернативно решение может быть разработано на других аналогичных (low-code) платформах (например, n8n, Langflow) или с использованием библиотек Python (Langchain и др.) с открытым исходным кодом. В процессе запуска и выполнения конвейеров обработки данных система DUC SmartSearch использует модель LLM GigaChat Max [17]. Применение других моделей LLM для решения прикладной задачи также возможно, хотя для этого потребуется дополнительная оптимизация текстовых запросов (prompt engineering).

2.2.1 Конвейер обработки данных

Прототип системы ранжирования списка продуктов строился по схеме обработки данных, как это представлено на рис. 1.

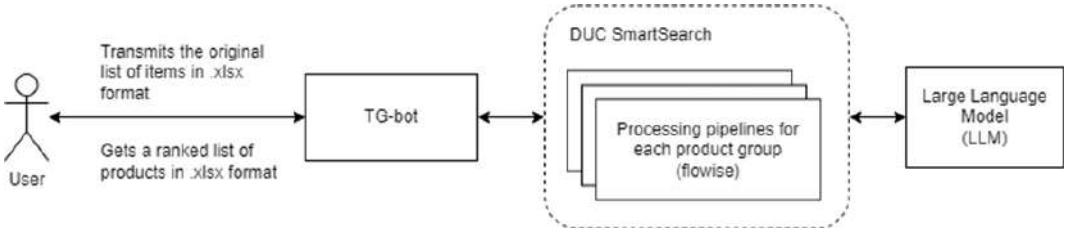


Рис. 1. Прототип системы ранжирования списка продуктов. Схема высокоуровневой архитектуры.
Fig. 1. Product List Ranking System Pilot. High-Level Architecture Diagram.

Пользовательский интерфейс системы реализован через Telegram. Здесь TG-бот принимает от пользователя исходные списки товаров в виде файлов Excel (в формате .xlsx) для последующей обработки. При этом список номенклатур должен находиться на первом листе, в первом столбце, начиная с первой строки.

Далее списки товаров поступают на свои конвейеры обработки в системе DUC SmartSearch, где происходит их классификация, в соответствии с укрупненным алгоритмом, представленным на рис. 3.

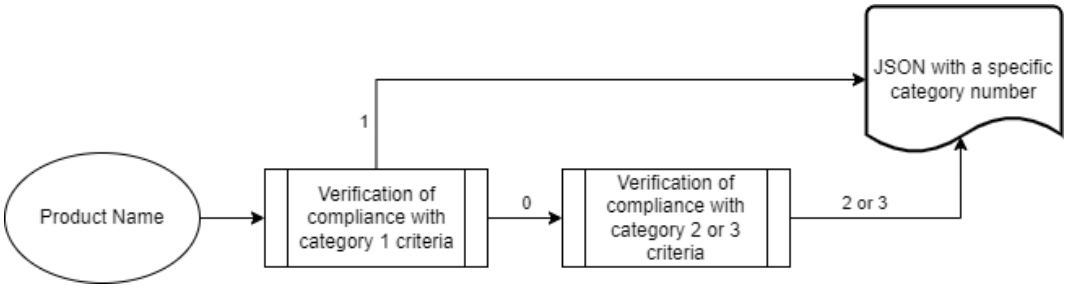


Рис. 3. Общий алгоритм классификации наименований товаров в конвейерах обработки.
Fig. 3. General algorithm for classifying product names in processing pipelines.

В процессе запуска и выполнения конвейеров система DUC SmartSearch использует поддержку со стороны LLM для реализации двухэтапной проверки соответствия критериям отнесения номенклатур списка товаров к одной из трёх категорий. Табл. 3 показывает запросы, которые используются системой для активации модели LLM GigaChat.

2.2.2 Результат обработки данных

В результате был разработан прототип системы, который получает на вход от пользователя список номенклатур определённой товарной группы и название этой группы, обрабатывает список по заданным критериям для каждой группы и предоставляет на выходе ранжированный список с указанием определённой категории в каждой строке.

Входной интерфейс системы реализован через мессенджер Telegram, где пользователь загружает Excel-файл со списком продуктов. Далее специально разработанный чат-бот обрабатывает этот Excel-файл, и результатом работы системы является ранжированный список всех продуктов-аналогов из списка поставщиков. Для реализации «умной обработки» списков поставщиков используется модель LLM GigaChat Max, которая обеспечивает достаточно высокую точность обработки списков, благодаря специально настроенному под прикладную задачу промт-инжинирингу, как это представлено в табл. 3.

Табл. 3. Пример запросов для двухэтапной проверки соответствия критериям.
Table 3. Example prompts for 2-step mapping verification.

Этап	Запрос
Проверка соответствия Категории 1	**Роль:** Вы – цифровой ассистент менеджера по снабжению ресторанного холдинга. Вы получаете название продукта (креветки), которое нужно проверить по критериям и вывести результат. **Словарь:** s/m – свежемороженые; s/hв – с хвостом; b/g – без головы. **Задача:** Выведите 1, если название соответствует ВСЕМ критериям: 1) размер из списка ["21/25", "26/30"] + 2) текст из списка ["очищенные", "чищеные", "чищен."] + указан 3) Текст из списка ["с хвостом", "на хвосте", "s/hв", "s/hв", "n/hв", "с хв."] указан. Критерии могут быть в любом порядке. ВАЖНО: название должно соответствовать ВСЕМ критериям. Выведите 0, если название не соответствует ХОТЯ БЫ ОДНОМУ критерию. Если продукт не относится к креветкам – выведите пустую строку "". Не выдумывайте. **Название продукта:** {input}
Проверка соответствия Категории 2 или 3	**Роль:** Вы – цифровой ассистент ресторанного холдинга. Вы получаете название креветок, которое нужно проверить по критериям и вывести результат. **Словарь:** s/m – свежемороженые; s/hв – с хвостом; b/c – без головы; b/g – без головы. **Задача:** Выведите 2, если название соответствует ВСЕМ критериям: 1) размер из списка ["16/20", "21/25", "26/30"] + указан 2) текст из списка "очищенные с хвостом", "чищеные", "чищен.", "чищеные, с хв.", "очищенные с хв.", "чищенные с хв.", "без головы", "b/g."] + 3) без текста из списка "в (кокосовой) панировке", "w/o", "в панцире", "панцирь", "пресноводные"]. Выведите 3, если название не соответствует ХОТЯ БЫ ОДНОМУ критерию. Если продукт не относится к креветкам – выведите пустую строку "". Не выдумывайте. **Название продукта:** {input}

Результат обработки списка товаров (рис. 4) демонстрирует ранжированный список с цветовой разметкой. Исходные данные идентичны показанным на рис. 4, но во входном файле позиции были представлены без цветов и в произвольном порядке. Разработанная система-прототип добавляет номер категории (1, 2 или 3) для каждой номенклатуры во второй столбец файла. Дополнительно список продуктов сортируется по возрастанию категории с цветовой разметкой «светофор»:

- Категория 1: зеленый;
- Категория 2: желтый;
- Категория 3: красный.

	A	B	C	D
1	Shrimp (vannamei) 21/25 peeled n/t	1	1	
2	Shrimp Wanamei peeled s/m 21/25 with tail	1	1	
3	Shrimp Wanamei 26/30/30 peeled with tail	1	1	
4	Shrimp peeled. with tail. c/m. 26/30 1kg	1	1	
5	Shrimp (Litopenaeus vannamei) peeled with tail (21/25) c/m 1kg – India	1	1	
6	Shrimp b/g c/m peeled. with tail 26/30 (1,000 kg) 30% cr. 10 pcs. India	1	1	
7	Shrimp 16/20 peeled with tail c/m India	2	2	
8	Shrimp 21/25 b/g c/m, block 1.8 kg, 1/6, pcs.	2	2	
9	Shrimp 21/25 b/g c/m, block 1.8 kg, 1/6, pcs.	2	2	
10	Shrimp 21/25 b/g c/m, block 1.8 kg, 1/6, pc	2	2	
11	Shrimp Vanamei in breading, peeled s/m 21/25. fac.1 kg	3	3	
12	Shrimp Wanamei peeled s/m 21/25 without tail	3	3	
13	Shrimp Wanamei s/m b/g 21/25 eyes.7%	3	2	
14	Shrimp Wanamei 16/20, in the shell	3	3	
15	Shrimp peeled. without tail s/m. 26/30 1kg	3	3	
16				

Рис. 4. Пример результатов обработки списка товаров.
Fig. 4. Example of results of processing a list of products.

Используемые в экспериментах наборы данных состояли из Excel-файлов того же типа, как это представлено на рис. 4. Для тестирования мы использовали около 30 файлов для каждой категории (группы), а для проверки-валидации результатов – около 10 файлов. Результаты количественной проверки полученного набора данных представлены в следующем разделе.

3. Обсуждение результатов проекта

3.1 Количественная оценка результатов внедрения решения

В качестве основной метрики качества категоризации/классификации продуктов в списке использовалась метрика Accuracy, которая рассчитывалась как доля правильно классифицированных объектов среди всех объектов в каждой товарной группе. Оценка качества классификации по товарным группам представлена в табл. 4.

Табл. 4. Точность классификации.
Table 4. Accuracy of product classification.

Товарная группа	Метрика Accuracy (Валидация)	Метрика Accuracy (Тест)
Креветки	81%	80%
Угорь	88%	85%
Творог	99%	96%
Сыр «Пармезан»	96%	77%
Сливочное масло	90%	81%
Среднее арифметическое	90,8%	83,8%

Точность решения оказалась вполне достаточной для практического использования ассистентом поставщика ресторана. Система экономит значительное время на ручном сопоставлении номенклатур товаров: достигнутая метрика Accuracy позволяет сокращать первоначальные временные затраты поставщика на ~90% (3 минуты вместо 30). Оставшиеся 10% времени расходуются на валидацию и исправление редких ошибок при появлении новых товаров.

3.2 Основные сложности использования LLM в рамках бизнес-кейса

Несмотря на то, что опытная эксплуатация системы показала многообещающие результаты использования моделей LLM и решений на их основе для проблематики сопоставления номенклатур товаров, в работе системы и ее эксплуатации проявился ряд проблем, которые можно считать общими для всего класса подобных приложений, а именно:

- Качество данных: неполные/некорректные данные влияют на результат;
- Обучение модели: требуется большой объём обучающих данных;
- Интерпретируемость: результаты LLM могут быть неочевидны для пользователей;
- Контекстная зависимость: модели не всегда верно понимают контекст.

Более конкретно, если рассматривать детали реализации и тестирования предложенного решения, то можно выделить в качестве примера несколько показательных конкретных проблем при работе с номенклатурой товаров, возникших на стороне модели LLM:

- Креветки: Термин “Easy Peel” модель понимает, как «очищенные», что некорректно;
- Пармезан: Термин “Palermo” (город в Италии) приводит к ошибочному отнесению товара к импорту;
- Указание “целая голова” вместо “1/2 головы” влияет на категорию товара;
- Модель ассоциирует “Deluxe” с “Premium/Exclusive”, что не верно;
- Масло: Термин “Ranferley” (город) ошибочно ассоциируется с Новой Зеландией.

Обычно подобные проблемы решаются через словари и промпт-инжиниринг, но требуют тщательного тестирования.

3.3 Масштабирование решения до целевого

Целевое решение предлагает использовать LLM для определения критериев ранжирования списка товаров поставщика из полученного наименования товара от заказчика (ресторана).

Для составления критериев на вход модели подается вся необходимая информация:

- Наименование товара;
- Правила составления критериев для определенной группы товаров;
- Дополнительные требования от заказчика.

Выход модели – это критерии ранжирования, которые на следующем шаге используются другой моделью для классификации списка товаров по категориям (эта модель как раз и тестируется в пилоте).

При необходимости модели LLM могут быть дополнительно обучены на примерах генерации и классификации критериев (обычно мы это делаем только тогда, когда другие методы не срабатывают).

Предлагаемое нами решение следующее:

- Некоторые элементы загружаются в систему;
- Выполняется согласование загруженных элементов: все символы опускаются, очищаются все знаки препинания и т.д. – в соответствии с типом входного текста;

- LLM используется для декодирования всех сокращений во входном тексте с использованием встроенного справочника (подготовленного заранее);
- Идентифицируется группа продуктов и клиент, затем фильтруется по необходимой группе. Названия продуктов затем обрабатываются на последнем этапе вместе с текущим списком продуктов
- Параллельно, когда сокращения декодируются, поток переходит к составлению критериев для категоризации с использованием модели LLM и на основе подготовленных правил (подсказок) для составления критериев
- В конце формируется рейтинг списка продуктов, рейтинговый список аналоговых продуктов

5. Заключение

В данном исследовании мы разработали, проанализировали и представили применение больших языковых моделей (LLM) к решению ключевой задачи сопоставления товарных номенклатур между ресторанами и их поставщиками. Предложенное решение использует расширенные возможности LLM для автоматизации и повышения точности сопоставления продуктов, оптимизируя таким образом управление запасами и процессы закупок. Ключевые выводы и вклад работы можно обобщить следующим образом:

1) Эффективность и автоматизация. Использование LLM значительно сокращает время и усилия, необходимые для ручного сопоставления товаров. Автоматизируя классификацию продуктов по категориям (точные совпадения, потенциальные совпадения, несовпадения), система оптимизирует процесс закупок, позволяя менеджерам сосредоточиться на стратегических решениях вместо рутинных задач.
2) Повышенная точность. Традиционные методы сопоставления часто не справляются с растущим разнообразием наименований товаров и сложностью предложений поставщиков. Наш подход на основе LLM демонстрирует высокую точность в идентификации релевантных соответствий, даже в случаях неоднозначных или нестандартно оформленных наименований. Это достигается благодаря способности модели понимать контекст и эффективно применять предопределённые критерии.
3) Масштабируемость и адаптивность. Предложенная архитектура разработана для обработки больших и разнородных наборов данных, что делает её применимой для бизнесов различных масштабов и отраслей. Гибкость системы обеспечивает лёгкую интеграцию с существующими системами управления запасами и закупок, гарантируя беспрепятственное внедрение и масштабирование.
4) Практическое применение. На практическом кейсе классификации креветочной продукции мы продемонстрировали способность системы обрабатывать и категоризировать наименования товаров с высокой точностью. Результаты подчеркивают потенциал LLM для трансформации управления цепочками поставок за счёт предоставления практических инсайтов и снижения операционных неэффективностей.
5) Широкое применение. Решение применимо не только в ресторанном бизнесе, но и в отраслях, зависящих от эффективного управления запасами: ритейле, производстве и e-commerce, где существуют аналогичные проблемы.
6) Проблемы и перспективы. Несмотря на значительные преимущества, сохраняются следующие вызовы: требуется повышение способности модели обрабатывать неоднозначные/неполные наименования, разработка механизмов обратной связи для постоянного улучшения системы, а также интеграция дополнительных источников данных (каталоги поставщиков, рыночные тренды).

Данное исследование демонстрирует преобразующий потенциал LLM в решении сложной задачи сопоставления товарных номенклатур. Автоматизируя этот процесс, компании достигают значительной экономии, повышают операционную эффективность и улучшают принятие решений. По мере развития технологии LLM, её применение в SCM будет расширяться, открывая путь для более интеллектуальных систем. Дальнейшие исследования должны сфокусироваться на решении выявленных проблем и поиске новых возможностей использования LLM в бизнес-операциях.

Список литературы / References

[1]. Dhara S., Delgado Barba S. (2024). Large language models in supply chain management [Master's thesis]. POLITesi Repository (online), Available at: https://www.politesi.polimi.it/retrieve/9f1da09d-256f-428c-8bae-a94d529df3e6/2024_07_Dhara_Delgado.pdf, accessed 25.10.2025.
[2]. Li B., Mellou K., Zhang B., Pathuri J., Menache I. (2023). Large language models for supply chain optimization. arXiv preprint arXiv:2307.03875.
[3]. Kumar S., Mellou K., Pathuri J. (2023). Large language models for supply chain optimization. ArXiv Preprint. Available at: https://arxiv.org/abs/2307.03875, accessed 25.10.2025.
[4]. Bag S., Rahman M.S., Srivastava G., Shore A., Ram, P. (2023) Examining the role of virtue ethics and big data in enhancing viable, sustainable, and digital supply chain performance, Technological Forecasting and Social Change, Vol. 186, DOI: 10.1016/j.techfore.2022.122154.
[5]. HBR Editors. (2025). How generative AI improves supply chain management. Harvard Business Review (online). Available at: https://hbr.org/2025/01/how-generative-ai-improves-supply-chain-management, accessed 25.10.2025.
[6]. Aporia AI Insights. (2024). Risks of using LLMs in enterprise applications (online). Available at: https://www.aporia.com/learn/risks-of-using-llms-in-enterprise-applications/, accessed 25.10.2025.
[7]. McKinsey Global Institute. (2025). Beyond automation: How gen AI is reshaping supply chains, Available at: https://www.mckinsey.com/capabilities/operations/our-insights/beyond-automation-how-gen-ai-is-reshaping-supply-chains/, accessed 25.10.2025.
[8]. IMD. (2024). How will large language models impact supply chains? Available at: https://www.imd.org/ibyimd/supply-chain/large-language-model-impacts-on-supply-chain/, accessed 25.10.2025.
[9]. Kellton Tech Blog. (2024). Large language models (LLMs): Navigating challenges and uncovering benefits. Available at: https://www.kellton.com/kellton-tech-blog/large-language-models-challenges-benefits, accessed 25.10.2025.
[10]. Lokad TV. (2023). Large language models in supply chain [online video]. Available at: https://www.lokad.com/tv/2023/12/13/large-language-models-in-supply-chain/, accessed 25.10.2025.
[11]. Cobalt.io. (2024). LLM supply chain attack prevention strategies (online). Available at: https://www.cobalt.io/blog/llm-supply-chain-attack-prevention-strategies, accessed 25.10.2025.
[12]. OWASP Foundation. (2023). OWASP Top Ten for Large Language Model Applications (online). Available at: https://owasp.org/www-project-top-10-for-large-language-model-applications), accessed 25.10.2025.
[13]. Атаки на цепочки поставок: как уязвимости распространяются через зависимости (online). Available at: https://www.securitylab.ru/blog/personal/xiaomite-journal/355832.php, accessed 27.10.2025.
[14]. Поисковый сервис DUC SmartSearch. Available at: https://duc-technologies.ru/smartsearch, accessed 27.10.2025.
[15]. Open Source AI Platform for Work (online). Available at: https://www.onyx.app/, accessed 27.10.2025.
[16]. Open source agentic systems development platform (online). Available at: https://flowiseai.com/, accessed 27.10.2025.
[17]. Сервис GigaChat (online). Available at: https://giga.chat/, accessed 27.10.2025.

Информация об авторах / Information about authors

Сеунгмин ДЖИН – кандидат технических наук, работает в НИУ ВШЭ с 2024 года в должности доцента департамента бизнес-информатики высшей школы бизнеса НИУ ВШЭ, а также старшего научного сотрудника международной лаборатории интеллектуальных систем

и структурного анализа факультета компьютерных наук НИУ ВШЭ. Сфера научных интересов: визуальная аналитика, искусственный интеллект, нейронные сети, машинное обучение и глубинное обучение и их приложения в рекомендательных системах и других областях.

Seungmin JIN – Cand. Sci. (Tech.), has been working at HSE University since 2024 as an associate professor in the Department of Business Informatics at the HSE Graduate School of Business, as well as a senior researcher at the International Laboratory of Intelligent Systems and Structural Analysis at the HSE Faculty of Computer Science. Research interests: visual analytics, artificial intelligence, neural networks, machine learning and deep learning and their applications in recommender systems and other areas.

Петр Борисович ПАНФИЛОВ – кандидат технических наук, доцент, является профессором департамента бизнес-информатики высшей школы бизнеса НИУ ВШЭ. Его научные интересы включают компьютерные науки и инженерию, науки о данных, распределенные вычисления и обработку данных и их приложения.

Petr Borisovich PANFILOV – Cand. Sci. (Tech.), professor in the Department of Business Informatics of the Graduate School of Business at HSE University since 2014. His research interests include computer science and engineering, data science, distributed computing and data processing and their applications.

Александр Сергеевич СУЛЕЙКИН – кандидат технических наук, является научным сотрудником научно-исследовательской лаборатории процессно-ориентированных информационных систем факультета компьютерных наук НИУ ВШЭ. Его научные интересы включают компьютерные науки и инженерию, науки о данных, искусственный интеллект и решения на основе генеративного искусственного интеллекта в промышленности и бизнесе.

Aleksander Sergeevich SULEYKIN – Cand. Sci. (Tech.), a researcher at the Research Laboratory of Process-Oriented Information Systems at the HSE Faculty of Computer Science. His research interests include computer science and engineering, data science, artificial intelligence, and generative AI-based solutions in industry and business.

DOI: 10.15514/ISPRAS-2025-37(6)-44

CC BY

Векторные представления шрифтов: дополнительный признак для понимания документов

[1,2] *Д.Е. Копылов, ORCID: 0009-0000-6348-4004 <it-daniil@yandex.ru>*
[1,2] *М.В. Щурик, ORCID: 0009-0004-9144-1617 <marriiamnii@gmail.com>*

[1] *Институт динамики систем и теории управления имени В.М. Матросова СО РАН, 664033, Россия, г. Иркутск, ул. А. Лермонтова, д. 134.*

[2] *Институт математики и информационных технологий Иркутского государственного университета 664003, Россия, Иркутск, бульвар Гагарина, д. 20.*

Аннотация. В статье представлена модель на основе сверточной нейронной сети, которая ставит в соответствие изображению текста векторы, кодирующие информацию о шрифтах. Модель состоит из двух идентичных сверточных блоков, объединяющих признаки в вектор, который затем анализируется линейными слоями для поиска отличий. Обученная таким образом модель способна различать шрифты, игнорируя содержание текста, что делает ее универсальной для различных типов документов. Векторные представления шрифтов тестируются на дополнительных задачах, таких как классификация текста по жирности начертания и наклону, демонстрируя высокую точность и подтверждая их полезность для анализа стилевых особенностей. Эксперименты с вариативными и ручными шрифтами показывают универсальность модели и ее применимость для работы с разнообразными данными. Результаты сравнения с базовой моделью подтверждают эффективность предложенной архитектуры. Однако выявлены ограничения, связанные с работой на данных низкого качества и мультиязычных текстах, что открывает направления для будущих исследований. Предложенный подход представляет значительный вклад в область обработки документов, расширяя возможности анализа шрифтов и их использования в задачах классификации, поиска и выделения ключевых элементов текста. Код и модели были опубликованы на GitHub (https://github.com/YRL-AIDA/FontEmb).

Ключевые слова: сверточные нейронные сети; классификация шрифтов; нейронные сети; компьютерные шрифты.

Для цитирования: Копылов Д.Е., Щурик М.В. Векторные представления шрифтов: дополнительный признак для понимания документов. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 177–188. DOI: 10.15514/ISPRAS–2025–37(6)–44.

Благодарности: Работа выполнена в рамках государственного задания Министерства науки и высшего образования Российской Федерации (тема № 1023110300006-9).

Vector Representations of Fonts: an Additional Feature for Understanding Documents

[1] *D.E. Kopylov, ORCID: 0009-0000-6348-4004 <it-daniil@yandex.ru>*
[2] *M.V. Shchurik, ORCID: 0009-0004-9144-1617 <marriiamnii@gmail.com>*

[1] *Matrosov Institute for System Dynamics and Control Theory of the Siberian Branch of Russian Academy of Sciences (ISDCT SB RAS), 134, Lermontov st., Irkutsk, 664033, Russia.*
[2] *Irkutsk State University Institute of Mathematics and Information Technologies, 20, Gagarin Boulevard, Irkutsk, 664003, Russia.*

Abstract. The article presents a model based on a convolutional neural network that matches a vector of embeddings encoding information about fonts to a text image. The model consists of two identical convolutional blocks that combine features into a vector, which is then analyzed by linear layers to find differences. The model trained in this way is able to distinguish fonts, ignoring the text content, which makes it universal for various types of documents. Embedding vectors are tested on additional tasks, such as text classification by fatness and tilt, demonstrating high accuracy and confirming their usefulness for analyzing stylistic features. Experiments with variable and manual fonts show the versatility of the model and its applicability to work with a variety of data. The results of the comparison with the base model confirm the effectiveness of the proposed architecture. However, the limitations associated with working with low-quality data and multilingual texts have been identified. The code and models were published on GitHub (https://github.com/YRL-AIDA/FontEmb).

Keywords: convolutional neural networks; font classification; neural networks; computer fonts.

For citation: Kopylov D.E., Shchurik M.V. Vector representations of fonts: an additional feature for understanding documents. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 3, 2025, pp. 177-188 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-44.

Acknowledgements. The research was carried out within the state assignment of Ministry of Science and Higher Education of the Russian Federation (theme No. 1023110300006-9).

1. Введение

1.1 Цель генерации векторов шрифтов

Анализ растровых изображений документов (сканированные PDF, фотографии страниц) представляет собой важную задачу компьютерного зрения с приложениями в юриспруденции, делопроизводстве и цифровых архивах. Современные системы обработки документов успешно решают базовые задачи: распознавание текста (OCR), детекцию структурных элементов (таблиц, заголовков) и извлечение семантической информации. Однако эти подходы преимущественно используют текстовые данные и пространственное расположение элементов. Когда в такие модели стали подавать изображение слов [1] их качество выросло.

Между тем, важно не столько изображение, сколько информация о шрифте. Эта информация несет значительную семантическую нагрузку: различие шрифтов часто маркирует смысловые разделы документа (заголовки, цитаты, примечания), а их стилистические особенности могут указывать на тип документа (официальный, рекламный, технический). Традиционные методы обработки документов не учитывают эти особенности, хотя их интеграция могла бы улучшить качество структурного анализа и интерпретации содержимого.

Целью данной работы является разработка компактной модели для генерации векторных представлений шрифтов, независимых от конкретных символов. В отличие от существующих решений, акцент делается на эффективность обработки: малый размер как самой модели, так и выходных векторных представлений, что критически важно для массовой обработки документов, где количество текстовых элементов на порядки превышает разнообразие используемых шрифтов.

1.2 Про существующие подходы

Активные исследования шрифтов начались в конце 1980-х годов. В этих исследованиях основное внимание уделялось оптическому распознаванию символов [2]. В 2000-х годах сформировалось самостоятельное направление, посвященное анализу статистических признаков и классификации шрифтов [3-4]. Эти ранние работы заложили методологическую основу, однако использовавшиеся в них подходы требовали ручного выделения признаков. Значительный прогресс был достигнут в работе [5], где впервые применили сочетание методов глубокого обучения (SCAE) с обширным набором данных, получив в результате 768-мерные векторные представления шрифтов.

С появлением современных архитектур CNN, таких как AlexNet и ResNet, векторные представления шрифтов нашли применение в различных смежных областях: улучшении систем OCR [6], подборе гармоничных сочетаний шрифтов [7], анализе их эмоционального воздействия [8]. Однако сохранялась проблема высокой размерности получаемых представлений: даже сокращение вектора до 200–300 измерений [8-9] оставалось недостаточным для эффективной работы в системах с ограниченными вычислительными ресурсами.

Высокая размерность векторных представлений оправдана в задачах тонкого анализа и настройки шрифтов, где требуется максимальная детализация. Однако при обработке документов, где количество используемых шрифтов редко превышает десяти, а число отдельных элементов может быть крайне велико, более предпочтительными оказываются компактные модели с малым размером выходного вектора. В этом отношении представляет интерес исследование [10], где использовалась малая сеть, формирующая всего 4-мерный вектор признаков.

Отдельное направление исследований связано с адаптацией методов векторного представления для различных языковых систем. Примером могут служить работы по распознаванию арабских шрифтов [11] и созданию шрифтов по описанию и образцу на другом языке [12]. В этих задачах также традиционно применяются векторные представления большой размерности, что поднимает вопрос об оптимальном балансе между детализацией и эффективностью вычислений. С точки зрения данной работы подходы [11-12] интересны, поскольку они не обращают внимание на текстовое содержание.

1.3 Предлагаемый в работе подход

Большинство существующих методов анализа шрифтов требуют либо полного алфавита, либо целых слов для работы. В реальных документах такие идеальные условия встречаются редко. В статья предлагается более практичное решение – модель, которая определяет схожесть шрифтов по отдельным сегментам изображения слова.

Предлагаемый подход использует простую сверточную сеть, которая анализирует изображение символа и выдает компактный вектор признаков. Затем линейные слои сравнивают два таких вектора и выдают бинарный ответ: 1, если шрифты совпадают, 0, если они разные. Главное преимущество – модель работает с любыми отдельными сегментами, не требуя алфавита и даже распознанных символов.

Такой подход особенно полезен при обработке сканов и фотографий документов. Модель легко встроить в существующие системы анализа документов, чтобы учитывать шрифтовую информацию при распознавании структуры и содержания.

2. Данные

2.1 Признаки шрифтов

Для анализа и классификации шрифтов в документах используются различные визуальные и стилевые признаки. Модель должна различать эти признаки независимо от содержания текста, что особенно важно для задач, где требуется выделить ключевые элементы документа

(например, заголовки, сноски или важные фрагменты). Однако для успешного обучения модели, способной отличать шрифты друг от друга, необходимо обеспечить достаточную вариативность этих признаков в данных.

Важным аспектом при подготовке данных является их сбалансированность. Под сбалансированностью в данном контексте понимается не равное количество примеров для каждого шрифта, а баланс между различными комбинациями признаков. В наборе данных может присутствовать много примеров текста со шрифтом Times New Roman, но всего несколько примеров текста со шрифтом Comic Sans MS. Шрифт Times New Roman может быть представителем сразу нескольких признаков, а Comic Sans MS представлять только рукописные шрифты с заданной комбинацией других признаков.

Также сразу следует отметить, что в данной работе не ставится задача точной идентификации конкретного шрифта (например, определение, что перед нами именно Times New Roman). Вместо этого модель должна научиться строить векторные представления, которые отражают схожесть шрифтов на основе их визуальных и стилевых признаков. В табл. 1 приведены основные признаки шрифтов, которые могут быть использованы для обучения модели. Эти признаки не являются исчерпывающими, но позволяют выделить 144 различные комбинации свойств, что достаточно для практических задач, поскольку в документах обычно используются шрифты, которые явно отличаются друг от друга.

Табл. 1. Признаки шрифтов и примеры.
Table 1. Font signs and examples.

Тип	Вариации	Пример
По начертанию	Тонкие	Courier New, Leelawadee UI Semilight, Segoe Script
	Нормальные	Arial, Times New Roman, Calibri
	Жирные	Impact, Arial Black
По наклону	Наклонные	Monotype Corsiva, Segoe Script
	Не наклонные	Arial, Times New Roman, Calibri
Моноширинные или нет	Моноширинные	Courier New, Consolas, Cascadia Mono
	Не моноширинные	Arial, Times New Roman, Calibri
По засечкам	Гротески	Arial, Calibri, Helvetica, Verdana
	Антиквы	Times New Roman, Book Antiqua
	Брусковые	Impact, Arial Black, Constantia
По схожести с рукописными	Схожи с рукописными	Segoe Script, Comic Sans MS, Monotype Corsiva
	Печатные	Arial, Times New Roman, Calibri
По форме	Прямоугольные	Arial, Arial Black, Impact
	Круглые	Cambria, Constantia, Courier New

2.2 Вариативные шрифты

Одним из наиболее интересных и перспективных направлений развития шрифтов являются вариативные шрифты [13] (Variable Fonts). В отличие от традиционных шрифтов, где каждый стиль (например, обычный, жирный, курсив) представлен отдельным файлом, вариативные шрифты позволяют динамически изменять параметры шрифта, такие как толщина, наклон, ширина и даже форма символов, в рамках одного файла. Это достигается за счет использования осей вариативности, которые определяют, как изменяется внешний вид шрифта. Например, ось "wght" регулирует толщину символов, а ось "ital" – их наклон.

Вариативные шрифты предоставляют значительную гибкость в дизайне документов. Они позволяют не только варьировать признаками между двумя крайними значениями (например, тонкий и жирный), но и использовать промежуточные значения, что делает их особенно полезными для создания адаптивных и визуально привлекательных интерфейсов. Среди популярных примеров вариативных шрифтов можно выделить: Advent Pro, Roboto, EB Garamond, Caveat, Sofia Sans Condensed, Victor Mono, Ysabeau Infant, Finlandica, Piazzolla.

В данной работе мы рассматриваем такие свойства, как написание в верхнем регистре, подчеркивание, зачеркивание как свойства шрифта. Эти свойства также могут варьироваться, что позволяет создавать дополнительные вариации шрифтов. Например, подчеркивание может быть реализовано не только стандартным способом, но и путем отрисовки прямоугольника внизу слова, что имитирует подчеркивание.

Использование вариативных шрифтов и их свойств при обучении модели позволяет значительно увеличить разнообразие данных и улучшить способность модели к обобщению.

2.3 Генерация изображений шрифтов

Помимо выбора шрифтов, важным аспектом подготовки данных является отрисовка слов с использованием этих шрифтов. Для обучения модели, которая должна различать шрифты, не обращая внимания на текст и его позицию, необходимо обеспечить, чтобы изображения содержали случайный текст и случайное расположение символов. Это позволяет модели фокусироваться исключительно на стилевых и визуальных особенностях шрифтов, а не на семантике или структуре текста.

Другой важный момент: поскольку мы оцениваем свойство шрифта, а не символов и слов, то не обязательно, чтобы слова целиком попадали в область. Достаточно взять прямоугольник и допустить возможность, что в окно модели попадет не больше 3-х символов (в худшем случае один символ попадет целиком и два будут обрезаны по бокам). Отметим, что на практике можно разбивать изображение строки или слова на одинаковые блоки (размером совпадающим с высотой) и в качестве результата возвращать вектор со средними значениями, уже характеризующий строку или слово. Чтоб показать разнообразие символов, текст генерируется как на английском, так и на русском, но символы не смешиваются, чтоб исключить особенности стилей языков.

Изображения группируются парами с одинаковым шрифтом и разным. Последние включают еще вариант совпадения текста. В пропорции пары с одинаковым шрифтом составляют – 50%, пары с разным шрифтом и разным текстом – 25%, пары с разным шрифтом и одинаковым текстом – 25%. Пары с одинаковым шрифтом и одинаковым текстом исключены из рассмотрения, чтобы избежать ситуации, при которой модель фокусируется на содержании текста.

Также при сравнении модель не должна обращать внимание на качество, поэтому изображения уменьшаются в случайное число раз и измененные возвращаются к исходному размеру. Размер шрифта выбирается также случайно.

Такой подход к генерации изображений позволяет создать разнообразный и сбалансированный набор данных, направленный на обучение модели, способной различать шрифты на основе их визуальных и стилевых особенностей. Это особенно важно для последующего применения модели в задачах, где требуется анализ шрифтов, независящий от содержания текта.

Число всех пар равняется 1 миллиону для набора, часть которого приведена в табл. 1. Также число пар с использованием вариативных шрифтов составляет 1 миллион изображений. Размер каждого изображения из пары равняется 40x40 пикселей.

3. Архитектура модели, ее обучение и тестирование

3.1 Архитектура

Модель, предложенная в данной работе, основана на простой сверточной нейронной сети (CNN), архитектура которой была взята из работы [10].

Сверточная часть в работе [10] состоит из двух VGG-блоков (visual geometry group), каждый из которых состоит из двух последовательно идущих сверточных слоев и слоя понижения разрешения или подвыборки (pooling). За ними следуют полносвязные слои. В данной работе архитектура была видоизменена (рис. 1). Сверточная часть (CNNModel) стала состоять из одного блока. Каждое изображение из пары проходит через эту сеть, на выходе для каждого изображения строится векторное представление. Векторные представления конкатенируются. И только после этого полученный вектор передается в полносвязные слои.

VGG-блок состоит из двух сверточных слоев. Первый слой состоит из одного входного канала (изображение в градациях серого) и 16 выходных каналов, при этом; размер ядра свертки: 3x3. Второй сверточный слой имеет аналогичные параметры, но с 16 входными и 32 выходными каналами. Последним идет операция выбора максимального значения в окне 2x2 с шагом 2 (слой, известный как pooling layer), что уменьшает размерность изображения вдвое по каждому измерению. На выходе сверточных слоев и операций уменьшения размерности данные преобразуются в одномерный вектор и передаются в полносвязный слой, который имеет входной размер 3200 (что соответствует объединенному размеру карт признаков 32x10x10).

Линейная часть представляет собой два полносвязных слоя с входным размером 3200 и выходным 64 для первого и с входным размером 64 и выходным размером 8 для второго.

Для введения нелинейности после каждого сверточного и полносвязного слоя используется функция активации ReLU.

Для обучения модели вводится вспомогательная модель (на рис. 1 – DiffModel), которая принимает на вход два вектора, возвращаемых основной моделью, конкатенирует их и пропускает через два полносвязных слоя с функцией активации ReLU. Первый полносвязный слой имеет вход и выход равный 16; второй полносвязный слой имеет вход равный 16 и выход равный 1 (бинарная классификация).

3.2 Обучение

Модель обучается решать задачу по сравнению шрифтов. Основная цель обучения вспомогательной модели (DiffModel) – научиться определять, являются ли два входных вектора, возвращаемых после свертки, представлениями одного и того же шрифта или разных. Если шрифты одинаковы, модель DiffModel возвращает 1, иначе 0.

Сама по себе задача сравнения стилей шрифтов не представляет большого интереса. Наиболее важным является обучение сверточной сети CNNModel, которая формирует векторные представления для шрифтов. Эти векторы должны кодировать стилистическую информацию о шрифтах, что делает их полезными для решения других задач.

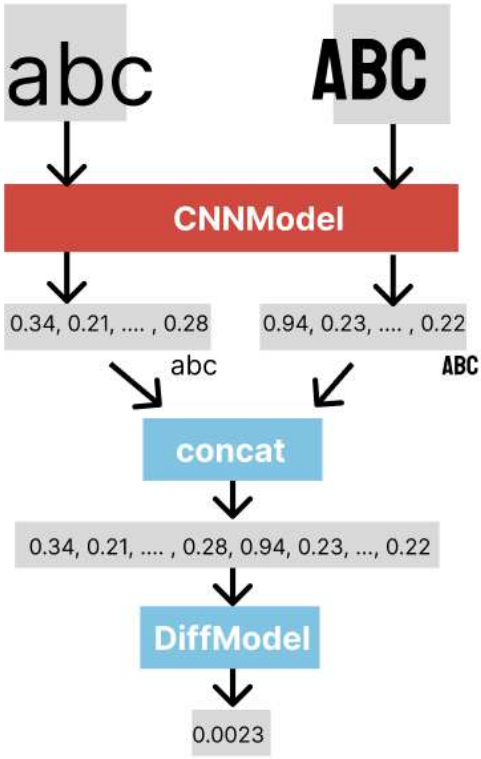


Рис. 1. Архитектура модели
Fig. 1. Model architecture.

Для обучения модели используется набор данных, состоящий из 1 миллиона изображений, для тестирования модели – набор, состоящий из 10000 изображений. Обучение проводится с использованием оптимизатора Adam с шагом обучения 0.0005. Размер порции данных за один шаг обучения составляет 256, что обеспечивает баланс между скоростью обучения и стабильностью обновления параметров модели. Модель обучается в течение 100 эпох.

В качестве функции потерь используется функция бинарной кросс-энтропии (Binary Cross-Entropy, BCE), которая минимизирует ошибку при классификации пар шрифтов (одинаковые/разные). Это позволяет сети CNNModel научиться формировать такие векторы, которые сохраняют информацию о стилевых особенностях шрифтов.

3.3 Оценка качества векторов

Как неоднократно подчеркивалось, главной задачей является не сравнение шрифтов, а формирование векторов, которые могут характеризовать их стилистические особенности. Для проверки полезности этих векторов решаются две дополнительные задачи:

1. Классификация текста по жирности начертания: модель должна определить, является ли текст жирным или обычным начертанием.
2. Классификация текста по наклону: модель должна определить, является ли текст прямым или курсивным.

Для решения этих задач используются модели, аналогичные DiffModel. Они также состоят из двух полносвязных слоев с функцией активации ReLU, но число узлов на первом слое уменьшено в два раза по сравнению с моделью DiffModel, как и на последующем слое. Выходной

слой остается таким же, как и в DiffModel.

Эти задачи позволяют оценить, насколько хорошо векторы, возвращаемые моделью CNNModel, кодируют информацию о стилевых особенностях шрифтов. Если модель успешно справляется с этими задачами, это свидетельствует о том, что векторное представление содержит полезную информацию, которая может быть использована для решения других задач, таких как классификация документов, поиск похожих шрифтов или анализ стилей текста.

4. Эксперименты

В данном разделе представлены результаты экспериментов, проведенных для оценки эффективности предложенной модели. Основное внимание уделяется сравнению с моделью, состоящей из двух VGG-блоков, решению задач классификации по жирности и наклону, а также анализу влияния различных типов шрифтов на качество модели.

4.1 Оценка качества векторов

Для оценки эффективности предложенной архитектуры была проведена серия экспериментов, в которых сравнивались результаты базовой модели с одним и двумя VGG-блоками (табл. 2). Обе модели обучались на одном и том же наборе данных, содержащем изображения текста с различными шрифтами.

Результаты показывают, что в отличие от работы [10] для данной задачи достаточно одного VGG-блока.

4.2 Классификация по жирности начертания и наклону

Для проверки информативности векторных представлений шрифтов была решена задача классификации текста по жирности начертания (обычный и жирный). Новая модель (вместо DiffModel) была дообучена на 10 тысяч изображений. Аналогично была решена задача классификации текста по наклону (прямой и курсив). Результаты представлены в табл 3.

Табл. 2. Архитектура сверточной моделей.
Table 2. Architecture convolution models.

Метрика	**Два VGG-блока**	**Один VGG-блок**
точность	0.8760	**0.8954**
F1-мера	0.8849	**0.9007**
Число параметров CNNModel	837 880	**210 184**

Табл. 3. Классификация жирности начертания и курсива.
Table 3. Bold and italic classification.

		Два VGG-блока	**Один VGG-блок**
Жирность	**точность**	0.8750	**0.9060**
	F1-мера	0.8713	**0.9052**
Курсив	**точность**	0.5390	**0.5920**
	F1-мера	0.4743	**0.5920**

Результаты исследования демонстрируют, что полученные векторные представления содержат информацию о таких характеристиках шрифтов, как жирность начертания и наклон. Это делает их полезными для решения задач, связанных с анализом шрифтов.

Анализ показал, что при поиске отличий между шрифтами модель в меньшей степени ориентируется на наклон символов, в то время как жирность является ключевым признаком, который модель использует для дифференциации шрифтов.

Модель может демонстрировать ограниченную переносимость (снижение точности при работе с данными, которые отличаются от обучающей выборки) при решении подобных задач. Это обусловлено тем, что сравнение по жирности и начертанию эффективно только при анализе двух конкретных шрифтов. Анализ отдельно взятого шрифта не позволяет сделать выводы об особенностях начертания.

4.3 Варианты данных

Для анализа влияния вариативных шрифтов на качество модели был проведен эксперимент с использованием шрифтов, таких как Roboto, Oswald и других, которые позволяют варьировать толщину, наклон и другие параметры. Также были добавлены стили, собранные вручную, чтобы оценить, как модель справляется с нестандартными шрифтами.

Модель работает одинаково хорошо с различными шрифтами. Она также способна распознавать шрифты, с которыми на которых не была обучена, но при этом точность ее работы снижается на 20% (табл. 4).

Табл. 4. Вариативные и обычные шрифты.
Table 4. Variable and classic fonts.

Набор для обучения	Тестирование на обычных шрифтах	Тестирование на вариативных шрифтах	Разница в метрике правильности
Обычный + Вариативный	0.8854	0.9054	0.0200
Обычный	0.9077	0.7193	**0.1884**
Вариативный	0.7155	0.9020	**0.1865**

5. Обсуждение результатов

Предложенная модель, основанная на простой сверточной нейронной сети, демонстрирует значительный потенциал в задаче генерации векторных представлений для анализа шрифтов. Использование двух идентичных сверточных блоков с последующим объединением в вектор и линейными слоями для поиска отличий позволяет модели эффективно различать шрифты, игнорируя при этом содержание текста. Это особенно важно для задач, где стилевые особенности шрифтов играют ключевую роль, таких как классификация документов или выделение ключевых элементов текста.

Одним из наиболее интересных аспектов работы является тестирование векторных представлений шрифтов на дополнительных задачах, таких как классификация по жирности и наклону. Высокие показатели точности для жирности (90.5%) свидетельствуют о том, векторные представления содержат как минимум информацию о жирности начертания шрифтов. Такие результаты открывают возможности для использования модели в более сложных задачах, таких как анализ структуры документов.

Следует отметить, что модель все еще имеет некоторые ограничения. Например, хотя она успешно справляется с игнорированием повторяющихся символов, ее эффективность на данных с низким разрешением или значительными искажениями требует дальнейшего изучения. Как выяснилось, это особенно актуально, по крайней мере, для информации о курсиве. Это может стать направлением для будущих исследований.

6. Заключение

В данной работе представлена модель, основанная на простой сверточной нейронной сети, которая ставит в соответствие изображению текста векторные представления, кодирующие информацию о шрифтах. Основная цель модели – научиться различать шрифты, игнорируя при этом содержание текста, что делает ее универсальной для различных типов документов. Модель состоит из двух идентичных сверточных блоков, которые извлекают признаки из изображения, объединяют их в вектор и передают через линейные слои для поиска отличий. Обученная таким образом модель демонстрирует высокую точность в задаче сравнения стилей шрифтов, что подтверждается результатами экспериментов.

Важным аспектом работы является тестирование векторных представлений шрифтов на дополнительных задачах, таких как классификация текста по жирности и наклону. Результаты показывают, что полученные векторные представления содержат полезную информацию о стилевых особенностях шрифтов, что делает их применимыми для решения более сложных задач, таких как анализ структуры документов, поиск похожих шрифтов или выделение ключевых элементов текста. Кроме того, эксперименты с вариативными шрифтами и шрифтами, собранными вручную, демонстрируют универсальность модели и ее способность работать с разнообразными типами данных.

Предложенный подход оказался весьма эффективным и открыл перспективы для применения полученных результатов в различных областях. В будущих исследованиях можно сосредоточиться на оптимизации модели для работы с более сложными и разнообразными данными, а также на интеграции ее в существующие системы обработки документов.

Список литературы / References

[1]. Xu Y., Li M., Cui L. Huang S. Zhou M. LayoutLM: Pre-training of Text and Layout for Document Image Understanding. In Proc. of the 26th ACM SIGKDD, 2020, pp. 1192-1200. DOI:10.1145/3394486.3403172.

[2]. Brzakovic D., Tou J. T. An approach to computer-aided document examination. International journal of computer & information sciences, vol. 14, 1985, pp. 365-385.

[3]. Allier B., Emptoz H. Type extraction and character prototyping using Gabor filters. In Proc. of the 7th ICDAR, 2003, pp. 799-803. DOI: 10.1109/ICDAR.2003.1227772.

[4]. O'Donovan P., Lībeks J., Agarwala A., Hertzmann A. Exploratory font selection using crowdsourced attributes. ACM Transactions on Graphics, vol. 33, pp. 1–9. DOI:10.1145/2601097.2601110.

[5]. Wang Z., Yang J., Jin H., Shechtman E., Agarwala A., Brandt J., Huang, T.S. DeepFont: Identify Your Font from An Image. In Proc. of the 23rd ACM MM, 2015, pp. 813-814. DOI:10.1145/2733373.2807988.

[6]. Tensmeyer C., Saunders D., Martinez T.R. Convolutional Neural Networks for Font Classification. In Proc. of 14th IAPR ICDAR, 2017, pp. 985-990. DOI:10.1109/ICDAR.2017.164.

[7]. Jiang S., Wang Z., Hertzmann A., Jin H., Fu Y. Visual font pairing. IEEE Transactions on Multimedia, 2019, 22(8), pp. 2086-2097. DOI:10.1109/TMM.2019.2952266.

[8]. Yasukochi N., Hayashi H., Haraguchi D., Uchida S. Analyzing Font Style Usage and Contextual Factors in Real Images. In Proc. of the 17th ICDAR, 2023, pp. 331-347. DOI:10.1007/978-3-031-41682-8_21.

[9]. Kulahcioglu T., De Melo G. Fonts like this but happier: A new way to discover fonts. In Proc of the 28th ACM MM, 2020, pp. 2973-2981. DOI:10.1145/3394171.3413534.

[10]. Bychkov O., Merkulova K., Dimitrov G., Zhabska Y., Kostadinova I., Petrova P., Petrov P., Getova I., Panayotova G. Using Neural Networks Application for the Font Recognition Task Solution. In Prec. of 55th ICEST, 2020, pp. 167-170. DOI: 10.1109/ICEST49890.2020.9232788.

[11]. Slimane F., Ingold R., Hennebert J. ICDAR2017 Competition on Multi-Font and Multi-Size Digitally Represented Arabic Text. In Prec. of 14th IAPR ICDAR, 2017, vol. 1, pp. 1466-1472. DOI: 10.1109/ICDAR.2017.239.

[12]. Tatsukawa Y. et al. FontCLIP: A Semantic Typography Visual-Language Model for Multilingual Font Applications. Computer Graphics Forum, 2024, 43(2), p. e15043. DOI: 10.1111/cgf.15043.

[13]. Phinney T. Variable Fonts Are the Next Generation. Communication Arts, 2016.

Информация об авторах / Information about authors

Даниил Евгеньевич КОПЫЛОВ – магистрант направления подготовки «Прикладная математика и информатика» Иркутского государственного университета, сотрудник Института динамики систем и теории управления имени В.М. Матросова Сибирского отделения Российской академии наук. Сфера научных интересов: прикладная математика, анализ данных.

Daniil Evgenievich KOPYLOV is master's student of Irkutsk State University, employee of Matrosov Institute for System Dynamics and Control Theory of Siberian Branch of Russian Academy of Sciences. Research interests: applied mathematics, data analysis.

Мария Викторовна ЩУРИК – бакалавр направления подготовки «Прикладная математика и информатика» Иркутского государственного университета. Сфера научных интересов: прикладная математика, анализ данных, искусственный интеллект.

Maria Viktorovna SHCHURIK is a bachelor's student of Irkutsk State University. Research interests: applied mathematics, data analysis, artificial intelligence.

DOI: 10.15514/ISPRAS-2025-37(6)-45

CC BY

Алгоритм двумерной трассировки лучей как метод препроцессинга для задач с волновыми аттракторами

С.А. Елистратов, ORCID: 0000-0002-7006-6879 <sa.elist-ratov@yandex.ru>

Институт океанологии им. П.П. Ширшова РАН,
Россия, 117218, Москва, Нахимовский проспект, 36.
Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Гидродинамический расчет аттракторов внутренних волн представляет собой довольно трудоемкую задачу с вычислительной точки зрения. Особенностью волновых аттракторов является то, что для конкретной геометрии аттрактор определенного типа формируется лишь для некоторого диапазона частот, что вызывает необходимость провести необходимые процедуры препроцессинга и проверить наличие аттрактора в выбранной постановке до начала самого расчета. Одним из способов это сделать является метод трассировки лучей, который воспроизводит распространение внутренних гравитационных волн в невязком линейном приближении. В статье рассматривается алгоритм трассировки лучей для широкого спектра профилей стратификации жидкости в достаточно произвольной области. Описываются эффекты, которые необходимо учитывать при применении метода. Приводятся результаты расчетов, включая специальные типы когерентных структур. Предлагается способ оценки скорости сходимости лучей.

Ключевые слова: волновые аттракторы; препроцессинг; трассировка лучей; внутренние волны.

Для цитирования: Елистратов С.А. Алгоритм двумерной трассировки лучей как метод препроцессинга для задач с волновыми аттракторами. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 189–202. DOI: 10.15514/ISPRAS–2025–37(6)–45.

2D Ray Tracing Algorithm as a Preprocessing Method for Wave Attractor Problems

S.A. Elistratov, ORCID: 0000-0002-7006-6879 <sa.elist-ratov@yandex.ru>

Shirshov Institute of Oceanology of the Russian Academy of Sciences,
36, Nakhimovskiy ave., Moscow, 117218, Russia.
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Wave attractor numerical simulation is a costly problem that requires a precise calculation method as well as thorough setup parameters determination. These two facts require using preprocessing methods before CFD simulation. The coherent structure for a geometry and stratification selected will appear in a certain range of perturbation frequencies, which are typically unknown in advance. To check whether the attractor forms, one can run a ray tracing which represents the propagation of internal wave narrow beams in inviscid linear approximation of the Navier-Stokes equations. The current article describes the algorithm that can be used for the ray tracing on a wide class of problems. It is shown that this method is capable to detect specific forms of attractors under specific conditions. Additionally, a ray convergence measure estimation is proposed.

Keywords: wave attractors, preprocessing, ray tracing, internal waves.

For citation: Elistratov S.A. 2D ray tracing algorithm as a preprocessing method for wave attractor problems. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 3, 2025, pp. 189-202 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-45.

1. Введение

Расчет течений с волновыми аттракторами является непростой задачей и требует выбора достаточно тяжелых вычислительных инструментов для корректного воспроизведения течения. Учитывая, что аттрактор образуется лишь при определенных параметрах задачи [1] вряд ли можно себе позволить перебирать их с помощью решения задач вычислительной гидродинамики (computational fluid dynamics, CFD) – требуется механизм быстрой проверки на наличие аттрактора в системе с определенными характеристиками. В качестве такого инструмента выступает алгоритм трассировки лучей. Этот метод не является тривиальным, однако в большинстве работ [1-12] он считается известным по умолчанию, хотя используется для достаточно простых конфигураций системы. В работе [2] исследуется вопрос о сходимости пучков внутренних волн с математической точки зрения, при том, что это исследование не приближает к построению самого алгоритма. Тем не менее, последнее развитие исследований волновых аттракторов [6-8] требует возможности расширения традиционного подхода на более широкие классы постановок, включая возможность запуска в сложных областях с нелинейной стратификацией.

Отдельно упомянем аналитическое решение, представленное в [5], полученное для класса аттракторов (n,1) в трапецеидальной области с линейным профилем стратификации. К сожалению, такой подход не позволяет применять полученное решение для более сложных областей или задач с непостоянной частотой плавучести, равно как и не позволяет делать выводы о скорости сходимости лучей (что в задачах гидродинамики связано с аккумуляцией энергии и образованием неустойчивости), оставляя трассировку лучей едва ли не единственным методом препроцессинга для установления факта наличия и определения формы аттрактора.

Целью настоящей работы является подробное описание метода трассировки лучей в контексте внутренних волн. Поскольку значительная часть исследований посвящена двумерным постановкам, а задачи в неисследованных ранее постановках сначала исследуются в 2D, так как это существенно экономит вычислительные ресурсы, алгоритм трассировки будет рассмотрен также в двумерном случае.

2. Описание алгоритма

В качестве входных параметров принимается следующее:

a) Параметры постановки:
   1. Частота внешнего возбуждения ω
   2. Геометрия области
   3. Профиль стратификации $\rho(y)$

b) Параметры численного алгоритма:
   1. Начальная точка (x_0, y_0)
   2. Знаки проекций направления распространения луча на координатные оси (σ_x, σ_y)
   3. Шаг трассировки Δs
   4. Количество итераций N_r

c) Дополнительные параметры:
   1. Шаг численного дифференцирования границы δ

Технически, вместо профиля стратификации может быть сразу задана частота Брента-Вяйсяля $N(y) = \sqrt{-\frac{g}{\rho}\frac{\partial \rho}{\partial z}}$ (не путать с числом итераций N_r), поскольку профиль $\rho(y)$ в алгоритме используется исключительно для расчета частоты плавучести $N(y)$.

Итерация происходит следующим образом:

1. расчет шага (пересчет координат следующей точки (x_{i+1}, y_{i+1})
2. проверка условий (см. ниже)
3. перезапись шага, если условия не выполнены

2.1 Свободное распространение пучка

Исходя из линеаризованных уравнений Навье-Стокса с учетом плавучести, пренебрегая конвективным членом в силу его квадратичности по скорости и перемешиванием стратификатора, можно получить следующее дисперсионное соотношение для внутренних волн [13]:

$$\omega/N = \cos\theta$$

где $\theta -$ угол между направлением распространения волны (волновым вектором) и направлением гравитации, ω – частота волны. В это соотношение входит не сам волновой вектор, а лишь направление распространения волны, что и позволяет рассчитывать распространение монохроматических волновых пучков. В контексте задач с аттракторами в случае ламинарных течений за частоту принимается частота волнопродуктора [14] как частота генерируемой им внутренней волны, поступающей в систему. Отметим, что для режимов с неустойчивостью дисперсионное соотношение неприменимо для вторичных волн, поскольку было выведено в линейном приближении.

Иногда в задачах с аттракторами используется немонохроматическое внешнее воздействие (обычно оно воспроизводит приливные явления, которые имеют различные моды [15,16]) – в этом случае трассировка проводится для каждой спектральной компоненты отдельно. Это можно сделать в силу линейности приближения.

Алгоритм расчета выглядит следующим образом (рис. 1):

1. расчет угла:

$$\theta_i = \arccos\left(\frac{\omega}{N(y_i)}\right)$$

2. шаг трассировки:

$$x_{i+1} = x_i + \Delta s \cdot \sigma_x \cos\theta_i$$
$$y_{i+1} = y_i + \Delta s \cdot \sigma_y \sin\theta_i$$

Знаки коэффициентов σ_x и σ_y при свободном распространении пучка остаются неизменными.

Такой подход обеспечивает точность $O(\Delta s)$, однако этого оказывается достаточно для установления формы аттрактора. Отметим, что в CFD-расчете когерентная структура всегда будет уширена за счет вязкости [12].

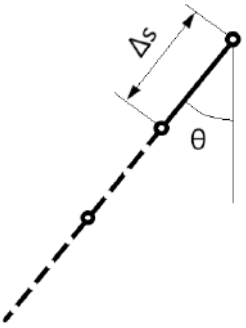


Рис. 1. Схема шага трассировки (свободное распространение луча).
Fig. 1. Ray-tracing step (free ray propagation).

2.2 Отражение

Отражение пучка происходит при невозможности дальнейшего его распространения, то есть либо при выходе за физическую границу области, либо при нарушении требования $\omega < N(y)$, которое является необходимым условием выполнения дисперсионного соотношения в силу ограниченности косинуса.

При отражении от наклонной поверхности возможны два случая в зависимости от соотношения между углом наклона поверхности α и углом распространения волны θ. Для определения угла α (рис. 2) используется численное дифференцирование границы. Если она задана неявной функцией $F(x, y) = 0$, наклон определяется по правилам дифференцирования неявной функции:

$$\alpha = \tan^{-1}\left(-\frac{\partial_y F}{\partial_x F}\right)$$

Заметим, что в числителе и знаменателе стоят именно такие производные, а не наоборот, поскольку α – это угол стенки *с вертикалью*. При этом следует отдельно проверять $\partial_x F$ на 0, и в этом случае принудительно полагать $\alpha = \pi/2$.

При программной реализации вычисление частных производных может быть заменено численным дифференцированием:

$$\partial_x F \approx \frac{F(x_i + \delta, y_i) - F(x_i - \delta, y_i)}{2\delta}$$
$$\partial_y F \approx \frac{F(x_i, y_i + \delta) - F(x_i, y_i - \delta)}{2\delta}$$

Шаг дифференцирования выбирается одинаковым для обоих направлений. Ясно, что при таком расчете производные имеют точность $O(\delta^2)$, поскольку используется схема центральных разностей; но на практике имеет смысл выбирать шаг дифференцирования δ на несколько порядков меньше шага трассировки Δs [17].

При явном задании профиля границы $y_b(x)$ в целях унификации кода будем использовать неявную функцию $F(x, y) \equiv y - y_b(x)$.

После расчета α пересчет знаков направления луча пересчитывается следующим образом (рис. 2):

1. $\alpha < \theta$: меняется знак σ_x
2. $\alpha > \theta$: меняется знак σ_y

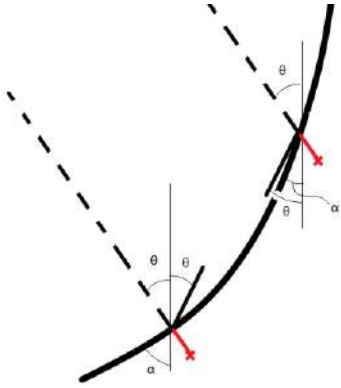


Рис. 2. Отражение луча на границе при различных соотношениях α и θ.
Fig. 2. Ray boundary reflection under different α and θ ratio.

После пересчета знаков необходимо сразу перезаписать результат для текущей точки, а не уходить на следующую итерацию. В противном случае трассировочная кривая будет рассчитана верно, однако это повлияет на оценку скорости сходимости, рассматриваемую ниже.

Кроме указанного простого отражения, возможно возникновение специфической ситуации с точечным аттрактором, когда луч «забивается» в угол и бесконечно увязает в цикле проверок (рис. 3). В этом случае его можно принудительно зафиксировать в угловой точке и прервать трассировку. Альтернативой является измельчение шага Δs, однако на результат это не повлияет и лишь увеличит время работы алгоритма.

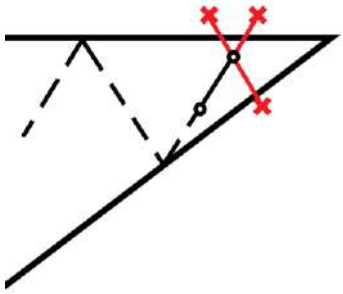


Рис. 3. Остановка трассировки в случае точечного аттрактора.
Fig. 3. Ray-tracing stop while the pointwise attractor has been reached.

Предложенный алгоритм может использоваться при расчете различных постановок. На рис. 4 представлена трассировка для достаточно распространенной постановки в трапецеидальной геометрии, заполненной однородно стратифицированной жидкостью. Трассировка позволяет получить различные типы (m, n) аттракторов на разных частотах.

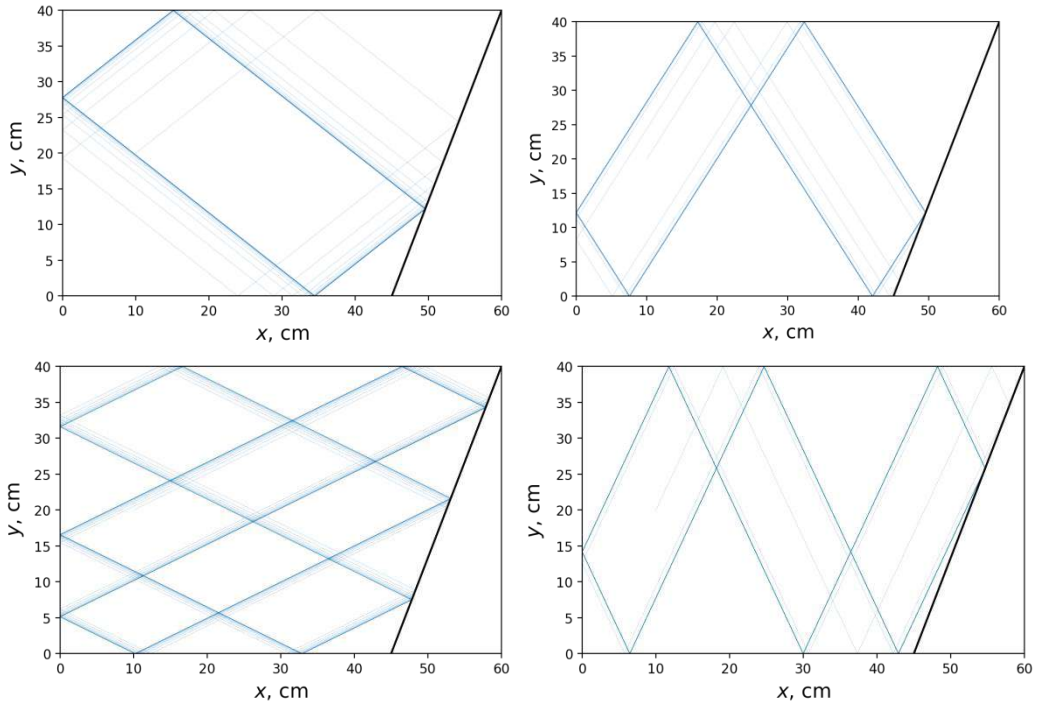


Рис. 4. Скелеты аттракторов, полученные трассировкой лучей: трапецеидальная геометрия, линейная стратификация.
Fig. 4. Ray-traced attractor skeletons: trapezoid geometry, linear stratification.

Рис. 5 демонстрирует результаты работы алгоритма в случае нелинейной стратификации. В частности, обнаружена возможность существования аттракторов с четным числом отражений по вертикали, что невозможно для линейной стратификации, где фокусировка компенсируется дефокусировкой на наклонной стенки при движении луча в другом направлении по вертикали [7].

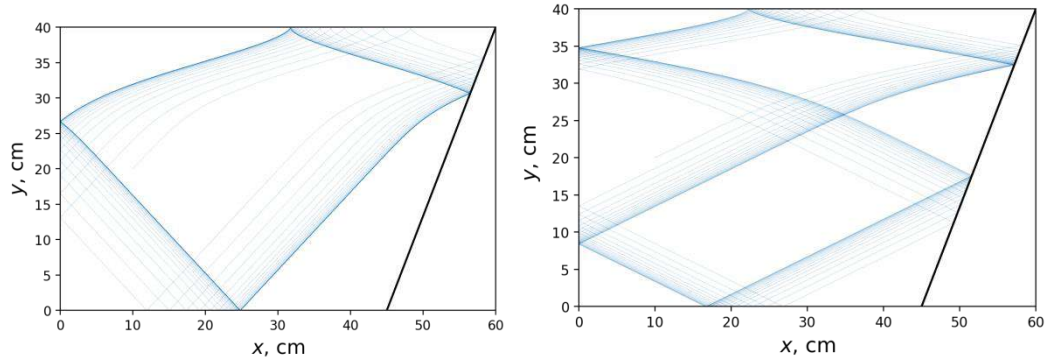


Рис. 5. Скелеты аттракторов для стратификации с галоклином.
Fig. 5. Attractor skeletons for a stratification with halocline.

Помимо этого, в средах с нелинейной стратификацией с помощью трассировки лучей обнаружена возможность существования специфического класса аттракторов, «запертых» по глубине. Его особенностью является то, что отражение луча происходит не на границах области, а на границе области условия. В этом случае формируются аттракторы с большим

числом отражений по горизонтали в области высокого градиента плотности (солености) [6,7]. Трассировка лучей для таких аттракторов показана на рис. 6.

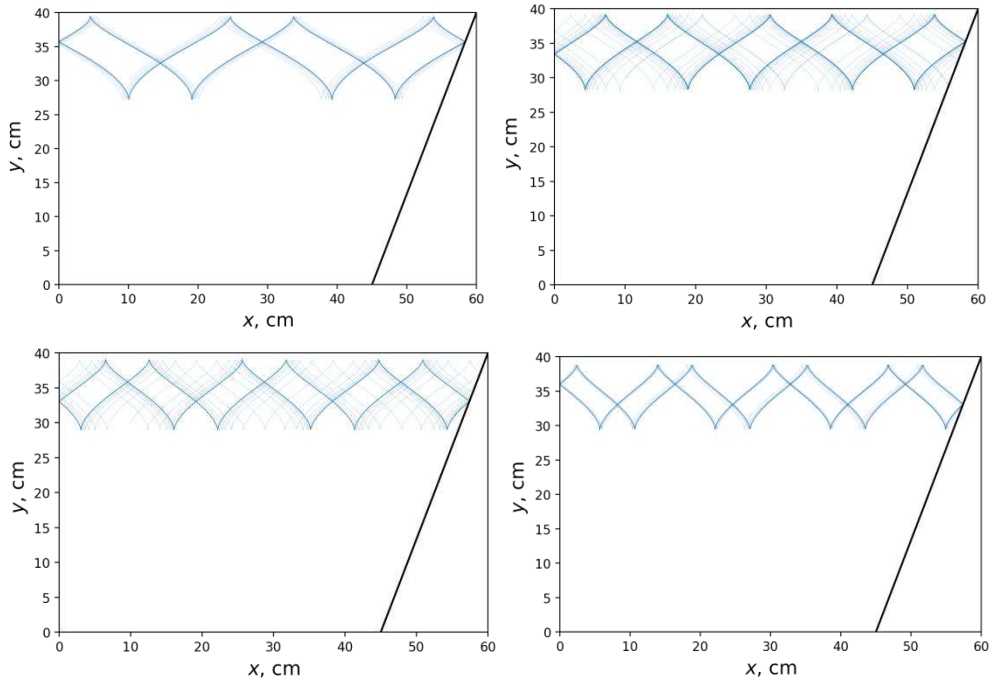
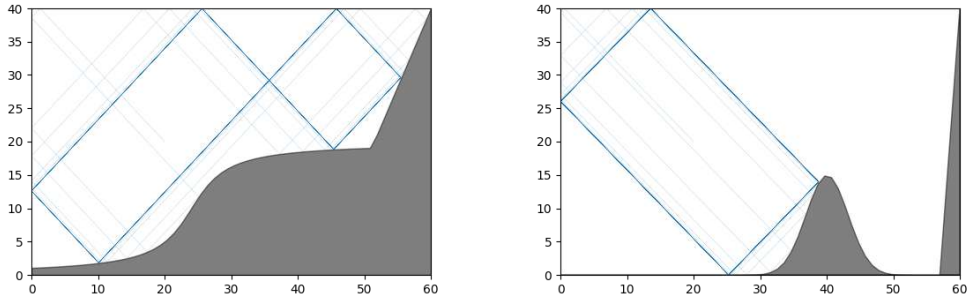


Рис. 6. Аттракторы, запертые по глубине (в области высокого градиента солености).
Fig. 6. Depth-trapped attractors (in the high salinity-gradient region).

Аттракторы в сложных областях демонстрируют специфические эффекты и также являются объектами исследования [8,18]. Предложенный алгоритм трассировки успешно справляется с выявлением аттракторов в таких постановках, что изображено на рис. 7.



a) b)

Рис. 7. Аттракторы в сложных геометриях: a) срединно-глубинное плато, b) подводный холм.
Fig. 7. Attractors in complex geometries: a) mid-depth plateau, b) underwater hill.

2.3 Оценка скорости сходимости

При проведении предварительных расчетов также важным оказывается определение скорости сходимости лучей, что в дальнейшем, при проведении моделирования, будет отражаться на степени аккумуляции энергии и, как следствие, образовании неустойчивостей. Традиционно в качестве оценки скорости сходимости лучей выступает показатель Ляпунова [1,2].

По определению это

$$\lambda_L \mid \|(x,y)-(x',y')\| = e^{\lambda_L s}\|(x_0,y_0)-(x_0{}',y_0{}')\| (\text{при } \|(x_0,y_0)-(x_0{}',y_0{}')\| \to 0)$$

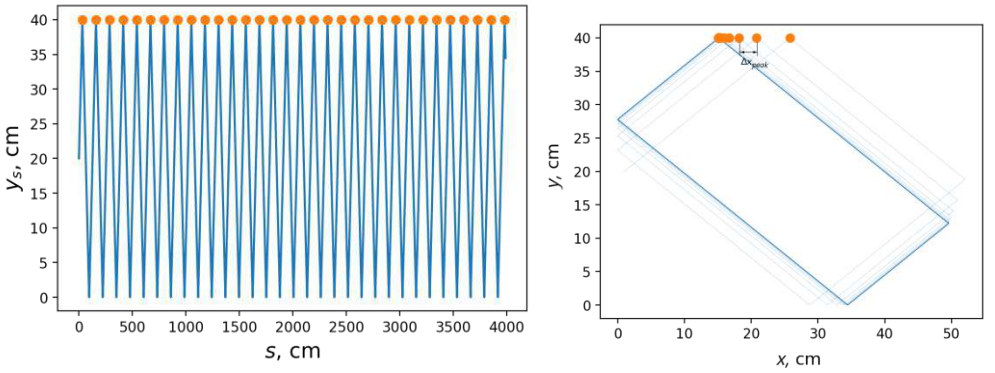
Однако следование этому определению приводит к проблемам со сдвигами, обусловленными отражениями. Рассчитать расстояния между ребрами разных периодов не представляется возможным, поскольку требует разделения полной совокупности точек луча на периоды.

Для решения этой проблемы предлагается следующая оценка:

1. Определение максимумов по одной оси
2. Исследование сходимости точек с теми же индексами по другой оси в зависимости от координаты вдоль луча s.

Фактически, это непредельная форма показателя Ляпунова, при которой два луча выпускаются не из произвольных точек, а из соседних точек отражения на верхней границе. Поскольку нас интересует лишь сам показатель, начальное расстояние между ними можно исключить из рассмотрения как мультипликативную константу.

На рис. 8 (a) показана зависимость проекции точки на луче на вертикальную ось и максимумы (оранжевые точки), определенные с помощью встроенного алгоритма *scipy.signal.find_peaks* в *python3*. Рис. 8 (b) отображает те же точки в плоскости $x-y$, видна их сходимость к вершине аттрактора.



a) b)

Рис. 8. Определение координат максимумов вертикальной координаты вдоль луча (a) и их сходимость (b).

Fig. 8. Maxima of the vertical coordinate on attractor ray determination (a) and their convergence (b).

Для определения количественной меры сходимости будем рассматривать разности горизонтальных координат определенных на предыдущем этапе точек Δx_{peak}, как показано на рис. 8 (b).

Исходя из предположения об экспоненциальном характере зависимости $\Delta x_{peak}(s)$, в качестве такой меры предлагается использовать показатель экспоненты λ:

$$\Delta x_{peak} \propto e^{\lambda s}$$

Покажем, что такая зависимость обеспечивается экспоненциальной сходимостью самих координат пиков:

$$x_{peak} = x_{peak}^{\infty} + Xe^{\lambda' s}$$

$$x_{peak}^{i+1} - x_{peak}^{i} = Xe^{\lambda' s_{i+1}} - Xe^{\lambda' s_i} = Xe^{\lambda' s_i}(e^{\lambda'(s_{i+1}-s_i)} - 1)$$

Поскольку в области экспоненциальной сходимости $s_{i+1} - s_i \approx \text{const} = L_p$, где L_p – периметр аттрактора, то

$$\Delta x_{peak} = x_{peak}^{i+1} - x_{peak}^{i} = X' e^{\lambda' s_i},$$

где $X' = X \cdot \left(e^{\lambda' L_p} - 1\right) = \text{const}(s)$. Отсюда $\lambda' = \lambda$, что означает, что декремент затухания разностей можно получить как декремент сходимости координат.

Отрицательное значение показателя λ свидетельствует об экспоненциальной сходимости лучей, что можно интерпретировать как аттрактор. Положительное значение означает репеллер.

Рис. 9 иллюстрирует эту зависимость, которая действительно оказывается экспоненциальной. Красная пунктирная линия – аппроксимационная прямая в логарифмическом масштабе (что соответствует экспоненте в линейном), проведенная методом наименьших квадратов. Ее наклон соответствует значению λ.

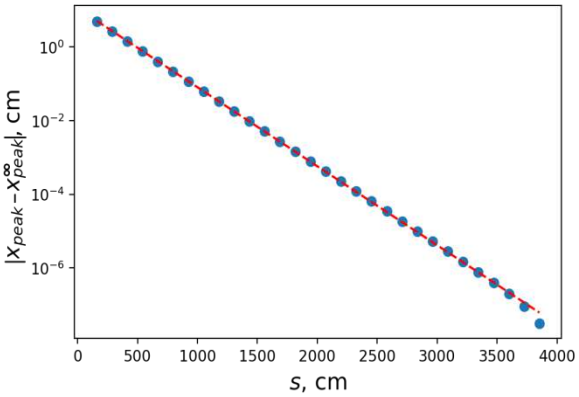


Рис. 9. Показатель сходимости.
Fig. 9. Convergence exponent.

Как можно заметить, предлагаемый подход позволяет не только решить проблему со сдвигами, но и обойтись запуском трассировки от одного начального положения вместо двух, что сокращает время работы.

Несмотря на неплохие результаты, этот метод оказывается неприменим для аттракторов с числом отражений лучей, большим четырех (иначе говоря, в настоящем виде он подходит лишь для обработки аттракторов типа (1,1) [1]). Это обусловлено тем, что на верхней границе происходит больше одного отражения за полный цикл по аттрактору – а это означает, что максимумы y_{peak} формируют несколько подпоследовательностей $x_{peak}^{i_k}$, сходящимся к разным точкам отражения. Для определения декремента λ будет достаточно определить количество отражений на верхней границе n, после чего из последовательности x_{peak} выделить каждую n-ю точку и применить к ней вышеописанные рассуждения. Это можно сделать, поскольку в каждом цикле луч проходит отражения в одной и той же последовательности.

Определить число отражений можно и вручную, однако иногда это бывает сделать невозможно (например, при построении диаграммы Мааса [1], когда требуется запуск значительного числа трассировок). В качестве автоматического способа определения числа отражений рассмотрим спектр быстрого преобразования Фурье $\text{FFT}(x_{peak})$ в зависимости от безразмерной частоты по номерам пиков. В случае аттрактора он имеет максимум в нуле, соответствующий среднему координат отражений, и один максимум, соответствующий частоте смене пределов. По нему можно определить $n = \left[\frac{1}{f_{max}}\right]$, где f_{max} – ненулевая частота спектрального максимума, а скобки $[\cdot]$ обозначают операцию взятия целой части (см. рис 10).

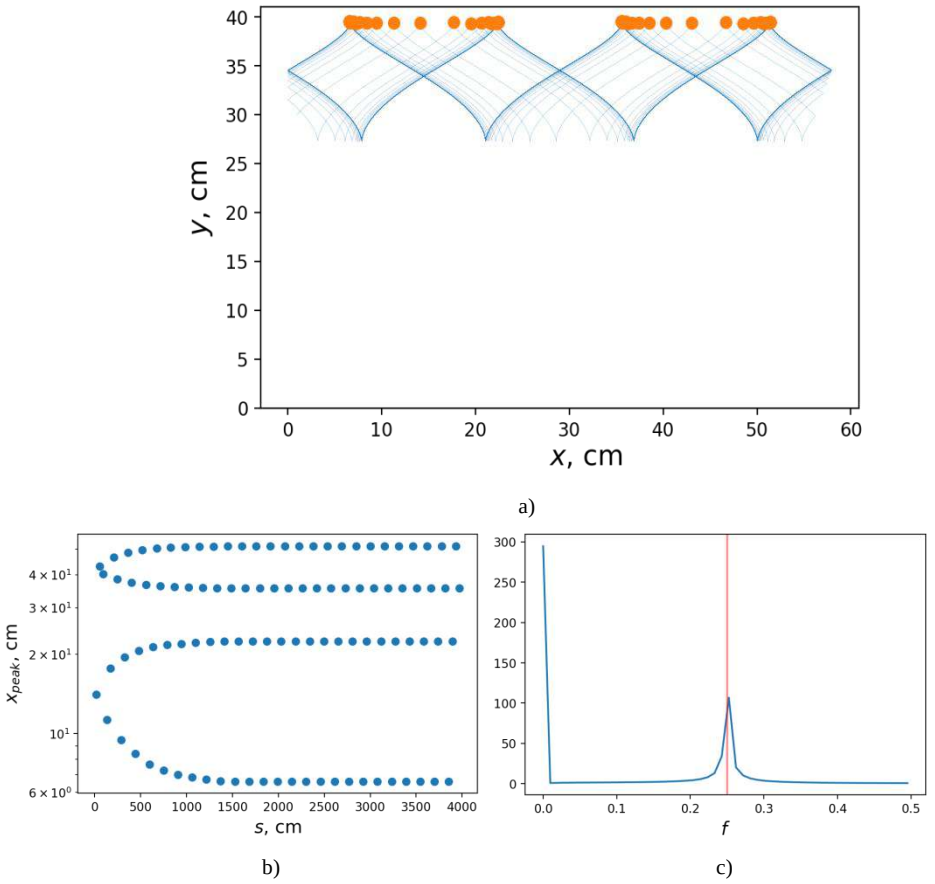


Рис. 10. Обработка сложного аттрактора:
a) выделенные точки отражения лучей на верхней границе,
b) горизонтальные координаты точек отражения, формирующие несколько сходящихся подпоследовательностей, c) спектр x_{peak}, красная линия – максимум спектра, использующийся при определении числа отражений n.
Fig. 10. Complex attractor processing: a) reflection points on the upper boundary,
b) reflection points horizontal coordinates, forming several subsequences,
c) x_{peak} spectrum, red line marks maximum for reflection number n determination.

После выделения подпоследовательности $x_{peak}^{i_k} = x^{n \cdot i}$, сходящейся к одной из точек отражения, строится экспоненциальная аппроксимация схождения (рис. 11):

$$x_{peak}^{i \cdot n} - x_{peak}^{\infty} \propto e^{\lambda s_{in}},$$

где в качестве предела x_{peak}^{∞} рассматривается последний член подпоследовательности $x_{peak}^{i \cdot n_{max}}$. Наклон аппроксимационной линии в полулогарифмическом масштабе и есть искомый показатель λ.

В табл. 1 представлены типичные значения этой величины для некоторых типов аттракторов для различных профилей солености [7].

В сложных областях возможно наличие сложного аттрактора, состоящего из нескольких компонент связности, но на одной частоте. Такие постановки делают трассировку зависимой

от начального приближения (рис. 12). Тем не менее, это не влияет на оценку скорости сходимости каждой из компонент связности, поскольку предложенная оценка использует только одну трассировочную кривую.

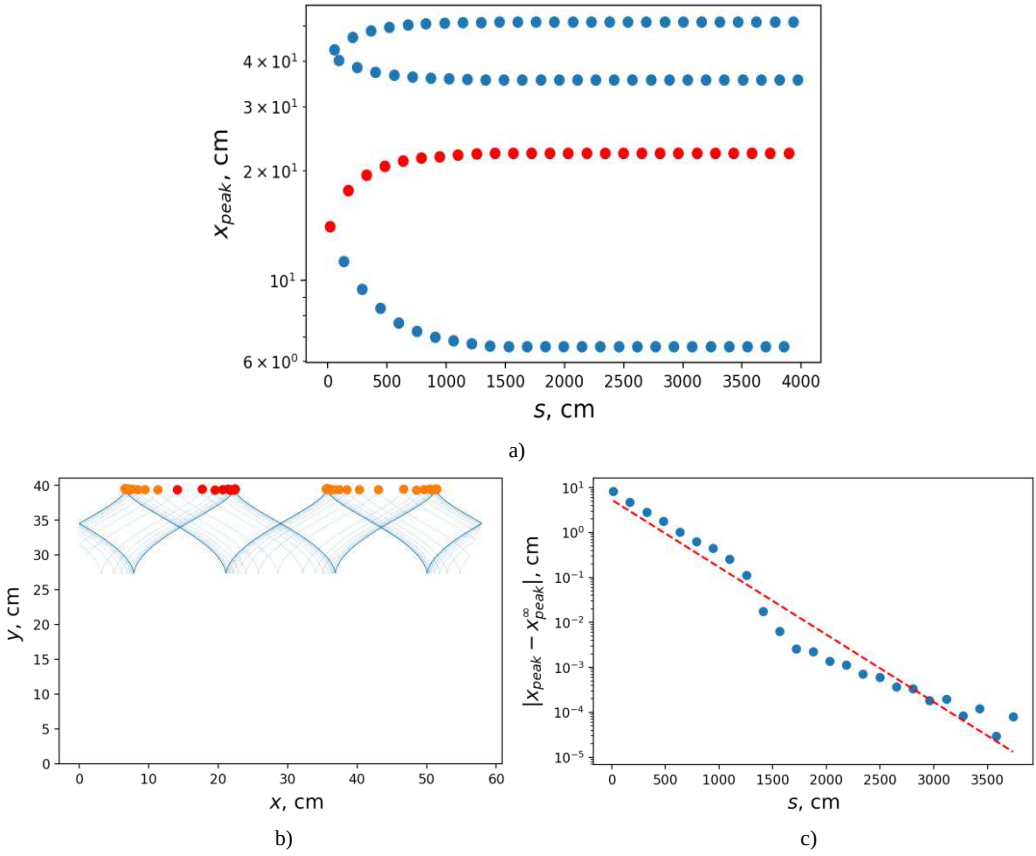


Рис. 11. a) выделенная с помощью спектрального метода сходящаяся подпоследовательность точек отражения (показана красным), b) соответствующие ей точки отражения, и c) аппроксимация схождения (x^{∞}_{peak} обозначен предел).

Fig. 11. a) converging subsequence extracted via the spectral method (red), b) corresponding reflection points, and c) convergence approximation (x^{∞}_{peak} is the limit).

Табл. 1. Типичные значения показателя сходимости λ.

Тип аттрактора	λ
(1,1), линейный профиль	-0.0021
(2,1), линейный профиль	-0.0030
(3,1), линейный профиль	-0.0038
(1,1), профиль с галоклином	-0.0040
(4,1), профиль с галоклином, аттрактор в слое	-0.0010
(3,1), профиль с галоклином, аттрактор в слое	-0.0023
(1,1), экспоненциальный профиль	-0.0065
(3,2), экспоненциальный профиль	-0.0017

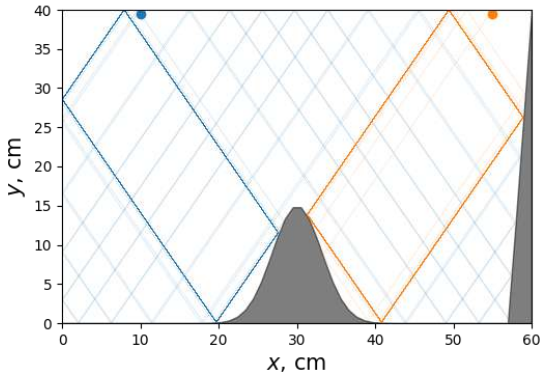


Рис. 12. Аттрактор с несколькими компонентами связности на одной частоте. Цветные точки отмечают начальные точки трассировки для соответствующих кривых.
Fig. 12. Attractor with several connectivity components at the same frequency. Colour dots mark the initial points for the corresponding curves.

3. Заключение

В статье рассмотрен алгоритм трассировки лучей применительно к задачам с волновыми аттракторами в двумерной постановке. Он позволяет рассчитывать скелеты аттракторов для широкого класса задач с различными профилями стратификации в различных геометриях. Также предлагается оценка скорости сходимости лучей, использующая всего один шаг трассировки и применимая для аттракторов с произвольным числом граничных отражений. Метод позволяет как рассчитывать скелеты аттракторов в классических постановках, так и находить новые типы когерентных структур при специальных условиях. Описанный алгоритм может использоваться для дальнейших исследований в этой области в качестве метода препроцессинга перед запуском CFD-расчета.

Список литературы / *References*

[1]. L.R.M. Maas and F.-P. A. Lam. Geometric focusing of internal waves. Journal of Fluid Mechanics, 300:1–41, 1995.
[2]. M. Lenci *et al.* Internal-wave billiards in trapezoids and similar tables. Nonlinearity 36, 1029, 2023
[3]. L.R.M. Maas. Wave attractors: linear yet nonlinear. International Journal of Bifurcation and Chaos. 15. 2757-2782, 2005. 10.1142/S0218127405013733.
[4]. G. Pillet et al. Internal wave attractors in 3D geometries: A dynamical systems approach. European Journal of Mechanics – B/Fluids, 77, 1-16, 2019. 10.1016/j.euromechflu.2019.01.008.
[5]. I. Sibgatullin, A. Petrov, X. Xu, and L.R.M. Maas. On (n,1) wave attractors: Coordinates and saturation time. Symmetry, 14(2), 2022.
[6]. S.A. Elistratov. Halocline internal wave attractors visualization. Scientific visualization, 16(1):82–94, 2024.
[7]. S.A. Elistratov and I.I. But. Towards the salinity profile influence on an internal wave attractor formation. Water waves, 2024
[8]. S.A. Elistratov and I.I. But. A viscous effect of wave attractor in geometry with underwater peak. Intelligent Marine Technology and Systems, 2(15), 2024
[9]. Сибгатуллин И.Н., Ерманюк Е.В., Ватутин К.А., Рязанов Д.А., Сюй С., 2019, Океанологические исследования, 2019, Том 47, No 1, С. 112–115
[10]. C. Pacary et al. Observation of inertia-gravity wave attractors in an axisymmetric enclosed basin. Physical Review Fluids, 8(10), 104802:1-20, 2023. 10.1103/physrevfluids.8.104802
[11]. J. Hazewinkel et al. Observations on the robustness of internal wave attractors to perturbations. Physics of Fluids, 22., 2010. 10.1063/1.3489008.

[12]. C. Brouzet et al. Scale effects in internal wave attractors. Phys. Rev. Fluids, 2:114803, 2017.
[13]. D.E. Mowbray, B.S.H. Rarity. A theoretical and experimental investigation of the phase configuration of internal waves of small amplitude in a density stratified liquid. Journal of Fluid Mechanics. 1967;28(1):1-16. doi:10.1017/S0022112067001867.
[14]. L. Gostiaux et al. A novel internal waves generator. Experiments Fluids. 2007, vol. 42, pp. 121-130.
[15]. D.A. Ryazanov et al. Biharmonic Attractors of Internal Gravity Waves. Fluid Dynamics. 2021, 56(3), pp .403-412.
[16]. S. Elistratov, I. But. On The Visualization of Subattractor Under Mixed Tidal Forcing. Scientific Visualization, 2025, vol. 17(1), 138-149.
[17]. Н. Н. Калиткин К17. Численные методы: учеб. пособие. 2-е изд., исправленное. СПб.: БХВ-Петербург, 2011, 592 с.: ил. (Учебная литература для вузов).
[18]. S.A. Elistratov, I.I. But. Wave attractor in basin with underwater step: from discrete to continuous energy spectrum. Indian J Phys (2025). https://doi.org/10.1007/s12648-025-03560-w.

Информация об авторах / Information about authors

Степан Алексеевич ЕЛИСТРАТОВ – сотрудник Лаборатории цифрового моделирования технических систем Института системного программирования с 2021 года. Сфера научных интересов: волновые аттракторы, вычислительная гидродинамика, численное моделирование.

Stepan Alekseevich ELISTRATOV – employee of the Laboratory of digital modelling of technical systems of the Institute for System Programming of the RAS since 2021. Research interests: wave attractors, CFD, applied math.

DOI: 10.15514/ISPRAS-2025-37(6)-46

CC BY

Модель размыва правого берега переливной запруды на протоке Пемзенской

Д. И. Потапов, ORCID: 0000-0001-6394-228X <potapovdi9@mail.ru.>
И. И. Потапов, ORCID: 0000-0002-3323-2727 <potapov2i@gmail.com>
Вычислительный центр Дальневосточного отделения Российской академии наук, г. Хабаровск, Россия.

Аннотация. В работе предложена математическая модель для изучения эрозии берегового склона протоки Пемзенской (река Амур) в районе переливной запруды после формирования и расширения прорана. Проран в переливной запруде образовался из-за размыва правого берега во время половодий 2019–2022 годов. Известно, что время установления гидродинамических параметров потока значительно короче времени изменения его расхода, поэтому поток в районе запруды описывается в рамках квазистационарного приближения. Для моделирования турбулентной вязкости потока используется алгебраическая модель Лео К. ван Рейна. Изменения донных и береговых отметок русла вычисляются с помощью аналитической модели движения наносов, разработанной в трудах Петрова, Потапова (2019). Чтобы предотвратить заиливание прорана при боковом перемещении донного материала с сухого берега, в уравнение донных деформаций введен стоковый член. Этот член регулирует глубину прорана, которая асимптотически стремится к его режимной глубине. Для решения задачи численно разработан алгоритм на основе метода конечных элементов. Выполнено сравнение результатов расчетов береговых деформаций с экспериментальными данными, показавшее их хорошее качественное и количественное согласование. Экспериментальные данные получены из информационной системы «Амур» с открытым исходным кодом.

Ключевые слова: русловые процессы; деформации дна; створ русла; информационная система; открытый код; проран переливной запруды; протока Пемзенская; река Амур.

Для цитирования: Потапов И. И., Потапов Д. И. Модель размыва правого берега переливной запруды на протоке Пемзенской. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 203–216. DOI: 10.15514/ISPRAS–2025–37(6)–46.

Благодарности: Работа выполнена при поддержке гранта Российского научного фонда № 24-17-20009 и гранта Правительства Хабаровского края (соглашение № 108С/2024 от 31.07.2024).

Erosion Model of Overflow Dam Right Bank on the Pemzenskaya Bayou

I. I. Potapov, ORCID: 0000-0002-3323-2727 <potapov2i@gmail.com>
D. I. Potapov, ORCID: 0000-0001-6394-228X <potapovdi9@mail.ru>

Computing Center of the Far Eastern Branch of the Russian Academy of Sciences, Khabarovsk, Russia

Abstract. The paper proposes a mathematical model for studying the erosion of the coastal slope of the Pemzenskaya channel (Amur River) in the area of the overflow dam after the formation and expansion of the proran. The proran in the overflow dam was formed due to the erosion of the right bank during the floods of 2019-2022. It is known that the time to establish the hydrodynamic parameters of the flow is much shorter than the time to change its flow rate, therefore, the flow in the dam area is described within the quasi-stationary approximation. The algebraic model of Leo K.Van Rijn is used to model the turbulent viscosity of the flow. Changes in the bottom and shore markings of the riverbed are calculated using an analytical model of sediment movement developed in the works of Petrov and Potapov (2019). In order to prevent siltation of the proran during lateral movement of bottom material from the dry shore, a runoff term is introduced into the equation of bottom deformations. This term regulates the proran depth, which asymptotically tends to its regime depth. An algorithm based on the finite element method has been numerically developed to solve the problem. The results of calculations of coastal deformations were compared with experimental data, which showed their good qualitative and quantitative agreement. The experimental data was obtained from the Amur open source information system.

Keywords: channel processes; bottom deformations; channel cross section; information system; open source; overflow dam opening; Pemzenskaya channel; Amur River.

For citation: Potapov I.I., Potapov D.I. Erosion model of overflow dam right bank on the Pemzenskaya bayou. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, 2025. pp. 203-216 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-46.

For citation: This work was supported by the Russian Scientific Fund project No 24-17-20009, and by the Government of Khabarovsk Territory (agreement No 108C/2024 dated 2024/07/31).

1. Введение

Изучение процесса эрозии песчаных береговых склонов рек под действием протекающего речного потока является важной прикладной задачей. Первые попытки построения строгих математических моделей для определения формы поперечного сечения русла восходят к работам Гловера [1], Маккавеева [2], Кондратьева [3], в которых были предложены эвристические модели, позволившие получить качественные оценки изучаемого процесса.

Дальнейшее развитие теории было связано с большим количеством экспериментальных работ Икеды [4], Хирано [5], Паркера [6], Питлика [7]. Были предложены математические модели, учитывающие различные механизмы переноса донного материала: транзитного перемещения путем влечения по дну и взвешивания частиц донного материала в речном потоке, напорного движения, медленного гравитационного и лавинного движения донных частиц на береговых склонах Икеды [4], Хирано [5].

В обзорной работе [8] выполнено исследование текущего состояния рассматриваемого вопроса. Основная его идея заключается в том, что при изучении проранов предпочтение отдается профильным [9] или плановым математическим моделям [10] позволяющим моделировать процесс развития проранов.

Кроме того, применение феноменологических формул транспорта донного материала позволяло лишь качественно описывать процессы береговой эрозии. В работах Петрова [11] были предложены и развиты [1] аналитические формулы транспорта донного материала, не содержащие в себе феноменологических коэффициентов.

В работах Бондаренко [12], Потапова [13] предложены математические модели для изучения процессов береговых деформаций русла при постоянном и переменном расходе речного потока в трапециевидных в начальный момент времени каналах.

В данной работе предложена математическая модель задачи о развитии процесса эрозии берегового склона протоки Пемзенской (р. Амур) в створе переливной запруды после возникновения и развития в створе запруды прорана [14], связанного со смывом правого берега запруды в периоды половодий 2019-2022 гг. Поскольку характерное время установления гидродинамических параметров потока много меньше характерного времени изменения его расхода [13], гидродинамический поток в створе запруды описывается в рамках квазистационарного приближения. Турбулентная вязкость потока описывается алгебраической моделью Лео К. ван Рейна. Для описания изменения донных и береговых отметок русла используется аналитическая модель движения наносов, предложенная в работах Петрова [1, 15, 16]. Для предотвращения заиливания прорана при боковом сходе донного материала с сухого берега в уравнение донных деформаций добавлен стоковый член, который регулирует развитие глубины прорана, асимптотически стремящейся к его режимной глубине.

Для численного решения задачи предложен алгоритм, основанный на методе конечных элементов. Выполнено сравнение результатов расчетов береговых деформаций с данными полевых наблюдений, показавшее их хорошее качественное и количественное согласование.

2. Математическая постановка задачи

Рассмотрим задачу о развитии прорана в переливной запруде на протоке Пемзенской р. Амур. Отметки донной поверхности ζ и свободной поверхности потока η (определенные на 09.08.2023) для створа, проходящего по гребню запруды, представлены на рис. 1. Левый берег и центральная часть запруды (участок L_1), выполненные из крупнообломочной наброски, не подвержены размыванию речным потоком. Рассматривается эрозия правого берега протоки (участок L_2).

Предполагается, что в области размыва Ω_2 русло имеет постоянный малый продольный уклон J в направлении движения гидродинамического потока и для него выполняются условия малости в отношении глубины потока к его ширине $H/B<<1$. Геометрия расчетной области $\Omega=\Omega_1\cup\Omega_2$ и ее границы $\Gamma=\Gamma_1\cup\Gamma_2$ представлены на рис.1. Граница Γ_1 представляет собой смоченную береговую и донную поверхности протоки, граница Γ_2 определяет свободную поверхность потока. Граница Γ_3 определяет сухую береговую и пойменную поверхности правого берега протоки.

Предполагая, что установившийся поток в створе канала зависит только от координат створа y и z, получим уравнение движения потока в створе канала [3, 5, 7]:

$$\frac{\partial}{\partial y}\left(\mu_e\frac{\partial U}{\partial y}\right)+\frac{\partial}{\partial z}\left(\mu_e\frac{\partial U}{\partial z}\right)+\rho_w g J=0, \qquad (1)$$

замыкаемое граничными условиями

$$U=0,\quad (y,z)\in\Gamma_1,\quad \frac{\partial U}{\partial z}=0,\quad (y,z)\in\Gamma_2,\quad \frac{\partial U}{\partial y}=0,\quad (y,z)\in\Gamma_4. \qquad (2)$$

Здесь U – осредненная по Рейнольдсу скорость речного потока в области Ω, $\mu_e = \mu + \mu_t$, μ_t – турбулентная вязкость потока, μ - молекулярная вязкость потока, J – уклон речного русла, $H = \eta - \zeta$ – глубина потока, g – ускорение свободного падения, η и $\zeta = \zeta(r)$ – отметки свободной поверхности потока и донной поверхности русла соответственно, ρ_w – плотность воды.

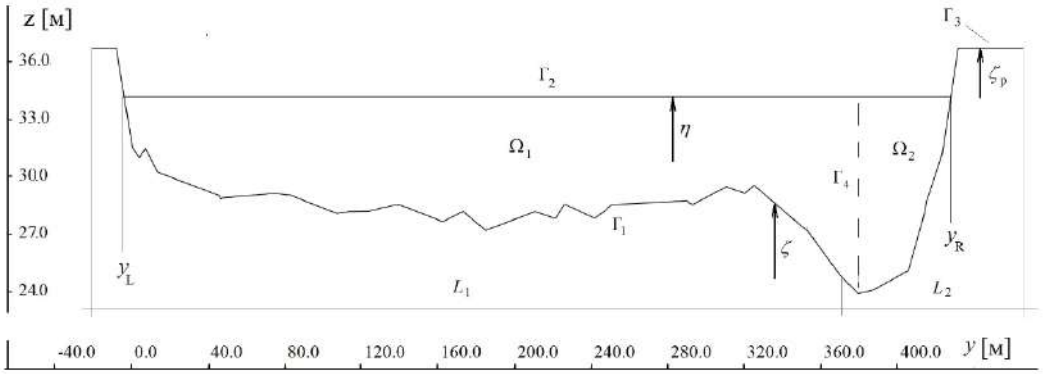


Рис.1. Геометрия расчетной области $\Omega = \Omega_1 \cup \Omega_2$ *и ее границы* $\Gamma = \Gamma_1 \cup \Gamma_2$, Γ_3 *- граница сухого берега,* Γ_4 *- граница псевдо-симметрии.*

Fig.1. Geometry of the calculated area $\Omega = \Omega_1 \cup \Omega_2$ *and its boundaries* $\Gamma = \Gamma_1 \cup \Gamma_2$, Γ_3 *- the border of the dry coast,* Γ_4 *- pseudo-symmetry boundary.*

Расчетная область Ω снизу ограничена подвижной донной поверхностью $\zeta = \zeta(t, y)$, эволюция которой описывается уравнением [1]

$$(1-\varepsilon)\frac{\partial \zeta}{\partial t} - \frac{\partial}{\partial y}\left(\Lambda_y \frac{\partial \zeta}{\partial y}\right) = -Q_x. \tag{3}$$

Уравнение (3) замыкается начальными и граничными условиями

$$\zeta(0, y) = \zeta_0(y), \qquad 0 \le y \le W, \quad t = 0, \qquad \eta = \eta(0), \tag{4}$$

$$\zeta(y_L, t) = \eta(t), \qquad \zeta(y_R, t) = \eta(t), \qquad 0 \le t \le T_S, \tag{5}$$

В уравнение (3) добавлена правая часть Q_x, ответственная за режимный баланс донного материала в деформируемом створе русла

$$Q_x = \alpha_H \frac{G_0}{H_0} \max\left(0{,}1 - \sqrt{\frac{T_c}{T}}\right)\left(\frac{(\eta_0 - \zeta) - H_0}{H_0}\right)\left(\left|1 - \frac{1}{\tan\varphi}\left|\frac{\partial \zeta}{\partial y}\right|\right|\right)^N, \tag{6}$$

$$H_0 = \frac{\alpha_\theta \rho_b \tan\varphi}{J} d_{50}, \tag{7}$$

где $\alpha_H = 0.5,\ \alpha_\theta = 1.2,\ N = 6$ – параметры модели, H_0 – режимная глубина невозмущенного запрудой русла [1]. Коэффициенты уравнения донных деформаций (3) определялись по модели [15]:

$$\Lambda = G_0 T \frac{\sqrt{T} - \sqrt{T_c}}{g\,\rho_w \sqrt{\rho_w}\cos\gamma}, \quad T_c = T_{0c}\left(\sqrt{1+\left(\frac{\partial\zeta}{\partial y}\right)^2}\right)^{-1}\sqrt{1-\left(\frac{1}{\tan\varphi}\frac{\partial\zeta}{\partial y}\right)^2},$$
$$T_{0c} = \frac{8}{9}\frac{\rho_b \tan\varphi}{c_x}\rho_w g\, d_{50},\ G_0 = \frac{16}{15}\frac{1}{\kappa_d\,\rho_b\left(\tan\phi\right)^2}, \quad \rho_b = \frac{\rho_s - \rho_w}{\rho_w}\ . \tag{8}$$

Здесь $\eta = \eta(t), \quad \zeta = \zeta(t, y)$ – отметки свободной поверхности потока и донной поверхности русла соответственно, T_{0c} – критическое придонное касательное напряжение на ровном дне, ρ_s – плотность песка, φ – угол внутреннего трения донных частиц, γ – острый угол между нормалью к поверхности дна ζ и вертикальной линией, c_x – лобовое сопротивление частиц, d_{50} – средний диаметр донных частиц, $\kappa = 0.41$ – коэффициент Кармана, κ_d – коэффициент Кармана для водогрунтовой смеси, ζ_0 – отметки дна в начальный момент времени.

Задача (1)-(5) замыкается алгебраической моделью турбулентной вязкости Лео К. ван Рейна [17]:

$$\mu_t = \rho\kappa u_* H(y)\begin{cases}\left(1-\dfrac{z}{H(y)}\right)\left(\dfrac{z}{H(y)}\right), & \dfrac{z}{H(y)} < 0.5,\\ 0.25,\end{cases} \quad u_* = \frac{\kappa U_b}{\ln(z_b / z_0)}, \tag{9}$$

где U_b – скорость U на расстоянии $z_b = 0.05H$ от дна $z_0 = 0.03k_s$, k_s – эффективная высота шероховатости дна.

Для расчета коэффициентов уравнения (3) вычислялись придонные касательные напряжения

$$T = \rho u_*^2 . \tag{10}$$

3. Слабая формулировка задачи

Рассмотрим слабую вариационную формулировку Галеркина для задачи (1)-(6) с набором пробных функций $\{N_\alpha, L_\alpha\} \in L_2(\Omega)$. В задаче необходимо найти неизвестные функции $\{U^h,\ \zeta^h\} \in L_2(\Omega)$, такие, что:

$$\int_\Omega \mu_e\left(\frac{\partial N_\alpha}{\partial y}\frac{\partial U^h}{\partial y} + \frac{\partial N_\alpha}{\partial z}\frac{\partial U^h}{\partial z}\right)d\Omega = \int_\Omega N_\alpha \rho\, g\, J\, d\Omega, \tag{11}$$

$$\int_{\Gamma_1}\left(L_\alpha\left((1-\varepsilon)\frac{\partial\zeta^h}{\partial t} + Q_x\right) + \Lambda\frac{\partial L_\alpha}{\partial y}\frac{\partial\zeta^h}{\partial y}\right)d\,y = 0 . \tag{12}$$

Уравнения (11)-(12) замыкаются краевыми условиями

$$U = 0, \qquad (r, z) \in \Gamma_1, \tag{13}$$

$$\zeta(y_L, t) = \eta(t), \qquad \zeta(y_R, t) = \eta(t), \qquad 0 \le t \le T_S. \tag{14}$$

Уравнение (12) замыкается начальными условиями

$$\zeta(0, y) = \zeta_0(y), \qquad 0 \le y \le W, \quad t = 0. \tag{15}$$

4. Дискретный аналог задачи

Используем метод конечных элементов в формулировке Галеркина. Разобьем расчетную область Ω на трехузловые конечные элементы Ω_e, $\Omega = \bigcup_e \Omega_e$. Введем на конечном элементе функции формы [18] N_α:

$$N_\alpha = \frac{1}{2\Omega_e}(a_\alpha + b_\alpha\, y + c_\alpha z), \tag{16}$$

и их градиенты

$$b_\alpha = \frac{\partial N_\alpha}{\partial y}, \quad c_\alpha = \frac{\partial N_\alpha}{\partial z},$$

здесь $\begin{cases} a_1 = y_2 z_3 - y_3 z_2, & b_1 = z_2 - z_3, & c_1 = y_3 - y_2, \\ a_2 = y_3 z_1 - y_1 z_3, & b_2 = z_3 - z_1, & c_2 = y_1 - y_3, \\ a_3 = y_1 z_2 - y_2 z_1, & b_3 = z_1 - z_2, & c_3 = y_2 - y_1, \end{cases}$ $\Omega_e = \frac{1}{2}\det\begin{bmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{bmatrix}$,

где y_k, z_k – координаты вершин (узлов) конечного элемента, Ω_e – площадь конечного элемента. Для уравнения донных деформаций расчетную область Γ_1 разобьем на двухузловые конечные элементы Γ_e, $\Gamma_1 = \bigcup_e \Gamma_e$ с функциями формы L_α:

$$L_1 = \frac{y - y_1}{h_e}, \qquad L_2 = 1 - \frac{y - y_1}{h_e}, \quad h_e = y_2 - y_1. \tag{17}$$

Определим аппроксимацию искомых функций на конечном элементе

$$U^h = N_\alpha U_\alpha, \qquad \mu_e^h = N_\alpha \mu_{e\alpha}, \qquad \alpha = 1,2,3. \tag{18}$$

$$\zeta^h = L_\alpha \zeta_\alpha \qquad \alpha = 1,2. \tag{19}$$

где $U_\alpha, \mu_{e\alpha}, \zeta_\alpha$ – значения искомых полей в α-тых узлах конечного элемента, в выражениях (18), (19) и далее ниже по тексту, где это не приводит к неоднозначностям, используется правило суммирования по «немым» узловым индексам ($N_\gamma U_\gamma \equiv \sum_{\gamma=1}^{3} N_\gamma U_\gamma$), $\mu_{e\alpha}$ – значение функции приведенной вязкости в узлах конечного элемента, $\overline{\mu}_e$ – осредненные величины функции вязкости на конечном элементе.

Используя интерполяцию (16)-(17), преобразуем интегральные тождества (11)-(12) в дискретные аналоги задачи

$$K_{\alpha\beta}U_\beta + F_\alpha = 0,\ F_\alpha = \rho_w g\, J\, \frac{\Omega_e}{3}\begin{bmatrix}1 & 1 & 1\end{bmatrix}^T,$$

$$M^z_{\alpha\beta} = \rho\frac{h_e}{6}\begin{bmatrix}2 & 1\\ 1 & 2\end{bmatrix},\ K^z_{\alpha\beta} = \frac{\overline{\Lambda}}{h_e}\begin{bmatrix}1 & -1\\ -1 & 1\end{bmatrix},\ Q_\alpha = \overline{Q}_x\frac{h_e}{2}\begin{bmatrix}1\\ 1\end{bmatrix}.$$

5. Алгоритм решения задачи

В начальный момент времени $t^0 = 0$ определяется начальная форма донно-береговой поверхности русла в створе $\zeta^0 = \zeta\left(t^0, y\right)$ и отметка свободной поверхности потока $\eta^0 = \eta\left(t^0\right)$.

Выполним расчет начальной придонной динамической скорости створа

$$u_* = \sqrt{g\, J\, R},\quad R = S_\Omega / P,\quad S_\Omega = \int_{y_L}^{y_R} H(y)dr,\quad P = \int_{y_L}^{y_R}\sqrt{1+\left(\frac{d\zeta(y)}{d\,y}\right)^2}\,dy.$$

Проведем расчет начального поля вязкости для всех i-х узлов расчетной области $\Omega^{h,0}$

$$\mu_{ei} = \mu_{ti} + \mu,\qquad \mu_{ti} = \rho\kappa u_* H(y_i)\begin{cases}\left(1-\dfrac{z_i}{H(y_i)}\right)\left(\dfrac{z_i}{H(y_i)}\right), & \dfrac{z_i}{H(y_i)} < 0.5\\ 0.25\end{cases},$$

Выполним следующие циклические действия.

На текущем шаге по времени $t^n = 0$ определяется начальная форма донных отметок русла в створе $\zeta_i = \zeta_i\left(t^n, y\right)$ и отметка свободной поверхности потока $\eta = \eta\left(t^0\right)$.

По функциям ζ_i и η определяется форма расчетной области $\Omega = \Omega\left(t^n\right)$, для которой генерируется конечно – элементная сетка $\Omega^{h,n} = \Omega^h\left(t^n\right)$.

Для каждого i-го узла донной поверхности (за исключением береговых узлов) с радиус-вектором узла $\vec{x}_i = \left(y_i, z_i\right) \in \Gamma_1^n$ выполним расчет векторов ($\vec{\tau}_i$, $\vec{n}_i$, $\vec{p}_i$ и $\vec{p}_i^{\,V}$) и поиск приграничных конечных элементов с функциями формы $N_\alpha\left(\vec{p}_i\right), N_\alpha\left(\vec{p}_i^{\,V}\right)$

$$\vec{\tau}_i = \vec{x}_{i+1} - \vec{x}_{i-1},\ \vec{n}_i = \left(-\frac{\tau_{i2}}{\sqrt{\tau_{i1}^2+\tau_{i2}^2}}, \frac{\tau_{i1}}{\sqrt{\tau_{i1}^2+\tau_{i2}^2}}\right),$$

$$\vec{p}_i = \vec{x}_i + h\vec{n}_i,\ \vec{p}_i^{\,V} = \vec{x}_i + h^V\vec{n}_i,\ h^V = 0.05H\left(y_i\right).$$

Аналогичный поиск конечных элементов с функциями формы $N_\alpha\left(\vec{p}_i\right)$ выполняется для i-х узлов свободной поверхности $\vec{x}_i \in \Gamma_2^n$ с векторами

$$\vec{n}_i = (0,-1),\ \vec{p}_i = \vec{x}_i + h\vec{n}_i,$$

Вычислим скорость U_α

$$\bigcup\left(K_{\alpha\beta}U_{\beta}+F_{\alpha}\right)=0,$$

Здесь и далее $\bigcup[*]$ обозначает операцию сборки локальных конечно – элементных дискретных аналогов задачи в глобальную систему алгебраических уравнений задачи.

Вычислим текущую придонную динамическую скорость

$$u_*\left(y_i\right)^n=\frac{\kappa N_{\alpha}\left(\vec{p}_i^{\,V}\right)U_{\alpha}}{\ln\left(h^V/z_0\right)},$$

Вычислим значения придонных сдвиговых напряжений на границе Γ_1^n

$$T=\rho u_*^2$$

Вычислим значения отметок донной поверхности ζ_{α}^{n+1} на текущем шаге по времени

$$\bigcup\left(\left[\frac{M_{\alpha\beta}}{\Delta t}+\theta K_{\alpha\beta}^z\right]^n\zeta_{\alpha}^{n+1}=\left[\frac{M_{\alpha\beta}}{\Delta t}-\left(1-\theta\right)K_{\alpha\beta}^z\right]^n\zeta_{\alpha}^{n}+\theta Q_{\alpha}^{n+1}+\left(1-\theta\right)Q_{\alpha}^{n}\right)$$

Вычислим коррекцию отметок донной поверхности ζ_i^{n+1} на текущем шаге по времени с учетом лавинного обрушения склона

$$\zeta_m^n=\zeta_m^n-\alpha_L\Delta_L,\ \ \zeta_{m-1}^n=\zeta_{m-1}^n+\alpha_L\Delta_L$$

$$\Delta_L=\begin{cases}r+\Delta\zeta_m^n, & r<-\Delta\zeta_m^n,\\ 0,\ r\geq-\Delta\zeta_m^n, & r=\Delta x\tan\varphi_m,\end{cases},\qquad \Delta\zeta_m^n=\zeta_m^n-\zeta_{m-1}^n,\ \ \alpha_L\approx 0.6.$$

Определим значение отметки свободной поверхности η^{n+1} на текущем шаге по времени.

Выполним перестроение сеточной области задачи $\Omega^{h,n+1}$.

Выполним интерполяцию $u_*\left(y_i\right)^n$ с границы Γ_1^n на границу Γ_1^{n+1}

Вычислим значения приведенной вязкости для всех узлов расчетной области $\vec{x}_i=\left(y_i,z_i\right)\in\Omega^{h,n+1}$

$$\mu_{ei}=\mu_{ti}+\mu,\qquad \mu_{ti}=\rho\kappa u_*\left(y_i\right)H\left(y_i\right)\begin{cases}\left(1-\dfrac{z_i}{H\left(y_i\right)}\right)\left(\dfrac{z_i}{H\left(y_i\right)}\right), & \dfrac{z_i}{H\left(y_i\right)}<0.5,\\ 0.25\end{cases}$$

Изменение текущего времени $t^{n+1}=t^n+\Delta t$, если $t^{n+1}<t_{end}$ переходим на пункт 1. При $t^n>t_{end}$ прекращаем расчет.

6. Результаты расчетов

Для верификации предложенной математической модели (1)-(10) был проведен ряд вычислительных экспериментов. Расчеты осуществлялись при следующих параметрах: $\kappa=0.25;\varepsilon=0.375;\ c_x=0.4;\ \rho_w=1000$ кг/м^3; $\rho_s=2650$ кг/м^3.

Параметры донного материала в забронированной части створа $k_s = 1.1,\ d_{50} = 1,\ \tan\varphi = 3$ (подобласть Ω_1) отличались от параметров в размываемой части створа (подобласть Ω_2) $k_s = 0.1,\ d_{50} = 0.0005\text{м},\ \tan\varphi = 0.5$.

Тестовые расчеты проводились на сгущающейся последовательности расчетных сеток с различным размером конечных элементов. Для самой мелкой сетки (рис. 3 в), выбранной в качестве расчетной, средний диаметр конечного элемента составлял около 0.2 м (53543 узла), погрешность по расходу по отношению к предыдущей сетке не превышала 0.0034%.

К сожалению, авторы не располагали кривой отметок уровня свободной поверхности над запрудой протоки Пемзенской. Поэтому, для ретроспективного моделирования была выбрана отметка глубины потока на момент натурных наблюдений 09.08.2023 г и сделана привязка данной глубины к уровню р. Амур по гидропосту г. Хабаровска. Уровни данного гидропоста для выбранного периода моделирования приведены на рис. 2.

Расчеты гидродинамического потока в рамках предложенной модели показали значительное занижение поля скоростей в области Ω_1 над телом переливной запруды и достаточно хорошее согласование расчетных скоростей со скоростями натурных наблюдений в области прорана Ω_2. Пример таких расчетов представлен на рис.3. Полученные результаты являются ожидаемыми, поскольку при постановке задачи в створе потока основным требованием является условие малости градиентов скоростей по направлению потока. Данное условие не выполняется в области Ω_1 и достаточно хорошо выполняется в области прорана Ω_2. Поэтому, для расчетов русловых процессов при расчетах полей скорости и турбулентной вязкости в вертикальном створе А-А рис.3 была определена «псевдо – граница русла» Γ_4, на которой задавалось условие симметрии потока (2). Такой подход возможен в силу нескольких причин,

- - дно левой части прорана не является размываемым участком тела запруды;
- - скоростной режим потока в окрестности границы Γ_4 близок к гидростатическому;
- -скоростной поток над телом запруды не является установившимся и его расчет по предлагаемой модели не будет правильным. Следовательно, введение псевдо – границы Γ_4 может считаться оправданным, поскольку различие измеренных (рис.3. кривая 1) и расчетных (рис.3. кривая 2) скоростей на свободной поверхности потока в области Ω_2 не превышает 9 %.

На рис. 4 представлены профили донных отметок створа, полученные в результате полевых исследований в период с 13 августа 2019 года по 15 июня 2025 года, обозначенные кривыми 1–5. Точность эхолота, используемого для измерений донных отметок, составляла 0.05 м, точность GPS-приемника при определении плановых координат составляла 1.5-2 м.

В период с 13 августа 2019 года по 9 августа 2023 года процесс углубления прорана в теле запруды характеризовался ярко выраженной нестационарностью (см. рис. 5), и его глубина не достигла уровня верхнего и нижнего бьефов. В связи с этим применение предложенной в данной работе модели для анализа данного процесса было невозможно. Однако в 2023 году глубины верхнего и нижнего бьефов в районе прорана сравнялись, что позволило применить разработанную модель для расчета эрозии правого берега протоки.

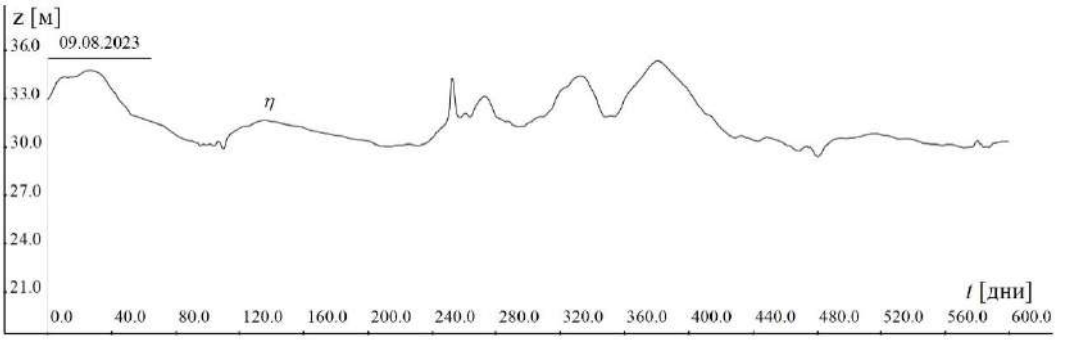


Рис.2. Отметки уровня р. Амур в Балтийской системе высот по Хабаровскому гидропосту.
Fig.2. Amur River level markers in the Baltic elevation system at the Khabarovsk Hydroelectric station.

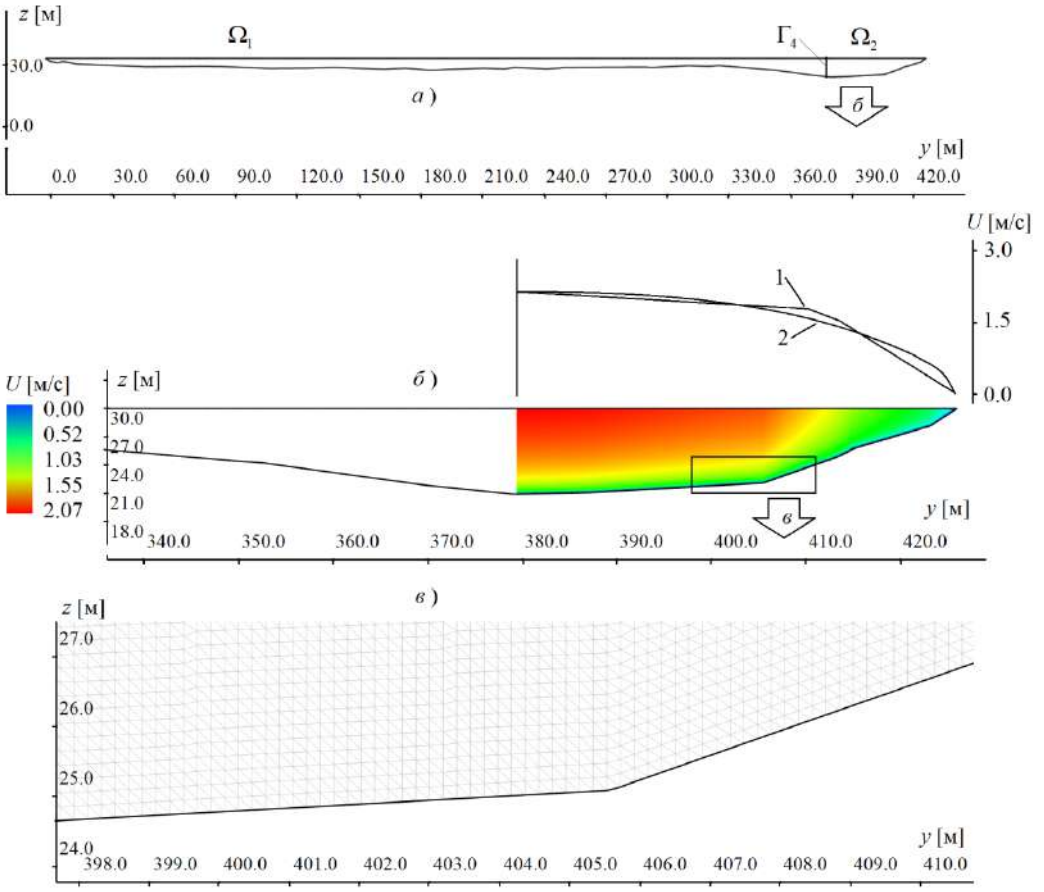


Рис.3. а) распределение расчетных скоростей в области Ω_2 створа переливной запруды, б) измеренные (кривая 1) и расчетные (кривая 2) значения поверхностных скоростей в створе, в) фрагмент расчетной сетки.

Fig.3. a) distribution of calculated velocities in the area Ω_2 of the overflow dam, b) measured (curve 1) and calculated (curve 2) values of surface velocities in the reservoir, в) a fragment of the computational grid.

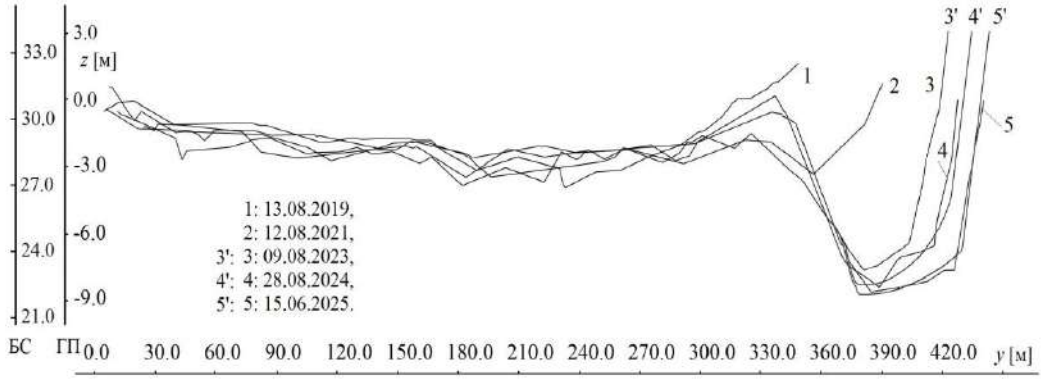


Рис.4. Размыв правого берега запруды Пемзенской.
Кривые 1-5 - натурные замеры донных отметок створа за период с 2019 по 2025 гг.
Кривые 3'-5' – результаты численного моделирования за период с 2023 по 2025 год.
Fig.4. Erosion of the right bank of the Pemzenskaya dam.
Curves 1-5 are full-scale measurements of the bottom marks of the target for the period from 2019 to 2025.
Curves 3'-5' are the results of numerical modeling for the period from 2023 to 2025.

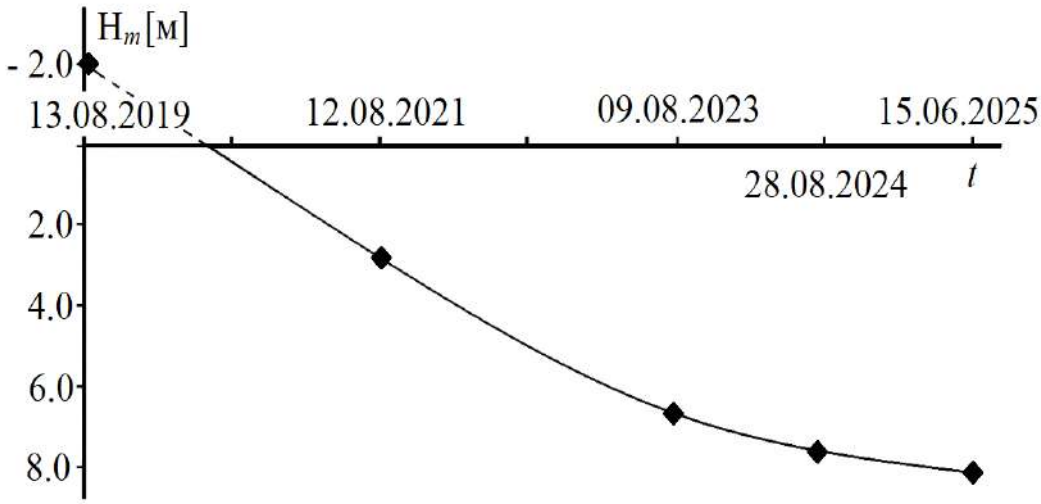


Рис.5. Изменение средней глубины прорана во времени.
Fig.5. Average depth changes of the gap over time.

Сравнительный анализ данных полевых наблюдений (линии 3-5, рис. 4) с результатами расчетов (линии 3'-5', рис.4) показывает их удовлетворительное согласование как по качественным, так и по количественным параметрам. При средней скорости эрозии береговой линии в 14.3 метра в год расхождения между расчетными и экспериментальными данными не превышают двух метров.

В заключение отметим, что прогнозируемое годовое смещение береговой линии вследствие эрозионных процессов согласуется с результатами полевых наблюдений, проведенных 28 августа 2024 года и 15 июня 2025 года. Различия между расчетными данными и фактическими измерениями не превышают 13%, что свидетельствует об адекватности предложенной русловой модели.

7. Заключение

Разработана математическая модель для анализа процессов эрозии берегового склона протоки Пемзенской при формировании прорана у правого берега переливной запруды с момента достижения прораном режимной глубины протоки.

Модель учитывает влияние изменений расхода потока в проране на эволюцию отметок свободной поверхности во времени. Ключевым компонентом предложенной модели является стоковый член в уравнении донных деформаций, который предотвращает заиление прорана и регулирует его углубление, стремясь к установившемуся значению глубины прорана. В процессе валидации модели было показано, что предложенная математическая модель, использованный метод и алгоритм расчета позволили получить решение задачи, хорошо согласующееся с данными полевых наблюдений в зоне прорана как в отношении фронтальной эрозии берегового склона, так и в отношении интенсивности врезания речного потока в русловое ложе.

Список литературы / References

[1]. Glover R. E., Florey Q. L. Stable channel profiles // U. S. Bureau of Reclamation, Washington. 1951.

[2]. Кондратьев Н.Е., Ляпин А.Н., Попов И.В., Пиньковский С.И., Федоров Н.Н. Якунин И.И. Русловой процесс. Л.: Гидрометеоиздат. 1959. 372 с./Kondratiev N.E., Lyapin A.N., Popov I.V., Pinkovsky S.I., Fedorov N.N. Yakunin I.I. Riverbed process. L.: Hydrometeoizdat. 1959. 372 p. (in Russian).

[3]. Макавеев Н.И. Русло реки и эрозия в ее бассейне. М.: Издательство АН СССР, 1955, 348 р./ [2]. Makaveev N.I. River bed and erosion in its basin. M.: Publishing House of the USSR Academy of Sciences, 1955, 348 p. (in Russian).

[4]. Ikeda S., Parker G., Saway K. Bend theory of river meanders. Part 1. Linear development // J. Fluid Mech. 1981. no. 112. P. 363–377.

[5]. Ikeda S. Stable channel cross–sections of straight sand rivers // J. Water Resources Res., 1991. Vol. 27, no. 9. P. 2429–2438.

[6]. Hirano M. River–bed variation with bank erosion // J. of Hydraulic, Coastal and Environmental Engineering. 1973. no. 210. P. 13–20 (in Japanese).

[7]. Parker G. Self–formed straight rivers with equilibrium banks and mobile bed. Part 1. The sand–silt river // J. Fluid Mech. 1978. part 1, Vol. 89. P. 109–126.

[8]. Monteiro-Alves, R.; Moran, R.; Toledo, M.A.; Jimenez-Rodriguez, R.; Picault, C.; Courivaud, J.-R. Overflow-Induced Breaching in Heterogeneous Coarse-Grained Embankment Dams and Levees‑A State of the Art Review. Appl. Sci. 2025, 15, 8808. https://doi.org/10.3390/app15168808

[9]. Visser, K.; Tejral, R.D.; Neilsen, M.L. WinDAM C Earthen Embankment Internal Erosion Analysis Software, n.d. (accessed on 19 June 2025) Available online: https://www.ars.usda.gov/research/publications/publication/?seqNo115=317437.

[10]. Dazzi, S.; Vacondio, R.; Mignosa, P. Integration of a Levee Breach Erosion Model in a GPU-Accelerated 2D Shallow Water Equations Code. Water Resour. Res. 2019, 55, 682‑702.

[11]. Петров П.Г. Движение сыпучей среды в придонном слое жидкости // ПМТФ, 1991. № 5. С. 72 — 75. / Petrov P.G. Movement of a granular medium in the bottom layer of liquid // PMTF, 1991. No. 5. P. 72 — 75. (in Russian).

[12]. Бондаренко Б.В., Потапов И.И. Моделирование эволюции поперечного сечения песчаного канала // Вычислительные технологии evolution of the cross section of the sand channel. 2009. Т.14, № 5. С. 1–14. / Bondarenko B.V., Potapov I.I. Modeling the evolution of the cross section of the sand channel // Computational technologies evolution of the cross section of the sand channel. 2009. T.14, no. 5. pp. 1–14. (in Russian).

[13]. Потапов И.И., Бондаренко Б.В. Математическое моделирование эволюции берегового склона в каналах с песчаным руслом // Вычислительные технологии. 2013. Т.18, № 4. С. 25–36./ Potapov I.I., Bondarenko B.V. Mathematical modeling of the evolution of the coastal slope in channels with a sandy bed // Computational technologies. 2013. T.18, no. 4. pp. 25–36 (in Russian).

[14]. Потапов И.И., Силакова Ю.Г. Процесс разрушения переливной запруды на Пемзенской протоке реки Амур//Гидротехническое строительство. № 2, 2025. / Potapov I.I., Silakova Yu.G. The process

of destruction of the overflow dam on the Pemzenskaya channel of the Amur River//Hydraulic engineering construction. No. 2, 2025. (in Russian).
[15]. Петров А.Г., Потапов И.И. Избранные разделы русловой динамики // М.: Ленанд. 2019. 244 с. / Petrov A.G., Potapov I.I. Selected sections of channel dynamics // M.: Lenand. 2019. 244 p. (in Russian).
[16]. Van Rijn L. C. Sediment transport, Part II: Suspended load transport // Journal of Hydraulic Engineering. – 1984. – Vol. 110, No. 11. – P. 1613–1641.
[17]. Шабров Н. Н. Метод конечных элементов в расчетах деталей тепловых двигателей. - Л.: Машиностроение, 1983. - 212 с./ Shabrov N. N. The finite element method in calculations of heat engine parts. Moscow: Mashinostroenie Publ., 1983. 212 p. (in Russian).
[18]. Potapov I.I. Source code of the Amur information system. https://github.com/PotapovII/Amur.

Информация об авторах / Information about authors

Игорь Иванович ПОТАПОВ – доктор физико-математических наук, профессор, заведующий лаборатории вычислительной механики Вычислительного центра Дальневосточного отделения Российской академии наук с 2009 года. Сфера научных интересов: численные методы, русловые и гидродинамические процессы в равнинных реках.

Igor Ivanovich POTAPOV – Dr. Sci. (Phys.-Math.), Professor, Head of the Laboratory of Computational Mechanics of the Computing Center of the Far Eastern Branch of the Russian Academy of Sciences since 2009. Area of scientific interests: numerical methods, channel and hydrodynamic processes in lowland rivers.

Дмитрий Игоревич ПОТАПОВ – младший научный сотрудник лаборатории вычислительной механики Вычислительного центра Дальневосточного отделения Российской академии наук.

Dmitry Igorevich POTAPOV – reasearcher of Laboratory of Computational Mechanics of the Computing Center of the Far Eastern Branch of the Russian Academy of Sciences.