DOI: 10.15514/ISPRAS–2020–32(6)–8

Практическая абстрактная интерпретация бинарного кода

[1,2] *М.А. Соловьев, ORCID: 0000-0002-0530-6442 <icee@ispras.ru>*
[1] *М.Г. Бакулин, ORCID: 0000-0002-8569-7382 <bakulinm@ispras.ru>*
[2] *С.С. Макаров, ORCID: 0000-0003-0077-237X <smakarov@ispras.ru>*
[2] *Д.В. Манушин, ORCID: 0000-0001-8985-4114 <dman95@ispras.ru>*
[1,2] *В.А. Падарян, ORCID: 0000-0001-7962-9677 <vartan@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1*

Аннотация. Математический аппарат абстрактной интерпретации предоставляет универсальный способ формализации и изучения алгоритмов анализа программ для самого широкого спектра прикладных задач. Однако его применение для практически значимых задач анализа бинарного кода связано с большим числом вызовов, как научных, так и инженерных. В настоящей работе предложены подходы к преодолению части этих трудностей. Описано промежуточное представление, учитывающее особенности бинарного кода. Предложена инфраструктура абстрактной интерпретации с возможностью выполнения интерпретации как вдоль отдельного пути в программе, так и статически до достижения неподвижной точки. В совокупности промежуточное представление и инфраструктура абстрактной интерпретации позволяют задать модель конвейера процессора, что позволяет проводить анализ бинарного кода для различных архитектур. В работе также представлены предварительные эксперименты и указаны дальнейшие направления развития проекта.

Ключевые слова: абстрактная интерпретация; анализ бинарного кода; динамический анализ; символьное выполнение; статический анализ

Для цитирования: Соловьев М.А., Бакулин М.Г., Макаров С.С., Манушин Д.В., Падарян В.А. Практическая абстрактная интерпретация бинарного кода. Труды ИСП РАН, том 32, вып. 6, 2020 г., стр. 101-110. DOI: 10.15514/ISPRAS–2020–32(6)–8

Благодарности: работа поддержана грантом РФФИ № 18-07-01256.

Practical Abstract Interpretation of Binary Code

[1,2] *M.A. Solovev, ORCID: 0000-0002-0530-6442 <icee @ispras.ru>*
[1] *M.G. Bakulin, ORCID: 0000-0002-8569-7382 <bakulinm @ispras.ru>*
[2] *S.S. Makarov, ORCID: 0000-0003-0077-237X <smakarov @ispras.ru>*
[2] *D.V. Manushin, ORCID: 0000-0001-8985-4114 <dman95 @ispras.ru>*
[1,2] *V.A. Padaryan, ORCID: 0000-0001-7962-9677 <vartan @ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. The mathematical foundations of abstract interpretation provide a unified method of formalization and research of program analysis algorithms for a broad spectrum of practical problems. However, its practical usage for binary code analysis faces several challenges, of both scientific and engineering nature. In this paper we address some of those challenges. We describe an intermediate representation that is tailored to binary code analysis; unlike some other IRs it is still useable in system code analysis. To achieve this, we take into account the low-level specifics of how CPUs work; on the IR level this mostly pertains to modeling main memory in that accesses can fail, and addresses can alias. Further, we propose an infrastructure for carrying out abstract interpretation on top of the IR. The user needs to implement the abstract state and the transfer functions, and the infrastructure handles the rest: two executors are currently implemented, one for analysis of a single path, and one for fixed point analysis. Both executors handle interprocedural analysis internally, via inlining or using summaries, so the interpretations only consider only procedure at a time, which greatly simplifies implementation. The IR and the abstract interpretation framework are used together to define a model pipeline for a target instruction set architecture, consisting of a fetch stage, a decode stage, and an execute stage. A distinct fetch stage allows to model delay slots, hardware loops, etc. We currently have limited implementations for RISC-V and x86. The x86 implementation is evaluated in two experiments where concolic execution is used to automatically analyze a «crackme» program, both in dynamic (execution trace) and static (executable image) setting. In conclusion, we outline the future directions of our project.

Keywords: abstract interpretation; binary code analysis; dynamic analysis; static analysis; symbolic execution

For citation: Solovev M.A., Bakulin M.G., Makarov S.S., Manushin D.V., Padaryan V.A. Practical abstract interpretation of binary code. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 6, 2020, pp. 101-110 (in Russian). DOI: 10.15514/ISPRAS-2020-32(6)-8

Acknowledgements: this work was supported by RFBR grant no. 18-07-01256.

1. Введение

Предложенный Патриком и Радией Кузо (Patrick Cousot, Radhia Cousot) аппарат абстрактной интерпретации [1] позволяет проводить анализ программ с целью исследования определенных свойств. Основная идея абстрактной интерпретации заключается в аппроксимации возможных состояний в точках программы с использованием абстрактного домена, где каждое абстрактное состояние соответствует множеству конкретных состояний, эквивалентных с точки зрения изучаемых свойств. В классическом варианте абстрактная интерпретация проводится по статическому представлению программы в внутрипроцедурном режиме: фиксируется требование монотонности общей передаточной функции подпрограммы, действующей над абстрактными состояниями, а итоговое решение строится как наименьшая или наибольшая неподвижная точка с применением итеративного алгоритма (например, алгоритма Килдалла (Gary Arlen Kildall) [2]). На практике требование монотонности, как правило, выполняется за счет того, что абстрактный домен представляет собой полурешетку, а обновление состояний происходит с применением монотонной относительно частичного порядка полурешетки операции «сбор» или «объединение».

С ходом времени были реализованы системы, воплощающие идею абстрактной интерпретации на практике, в первую очередь с целью поиска ошибок и верификации ПО (например, система Astrée [3]). Однако по большей части такие системы работают на уровне исходных текстов программ и во многом опираются на структурный анализ графа потока управления. В то же время, анализ бинарного кода представляет на настоящий момент не менее, а в каких-то случаях и более востребованное направление. Например, при поиске программных дефектов на уровне исходных текстов Си-программ нет возможности оценить критичность дефекта переполнения буфера, так как то, каким образом компилятор сгенерирует код в таких случаях, не специфицируется стандартом языка (неопределенное поведение).

Экспертиза, накопленная в рамках реализации инструментов анализа полносистемных трасс исполняемого кода, позволяет с уверенностью утверждать, что возможность формализации такого рода задач как задач абстрактной интерпретации позволила бы, как минимум, значительно сократить время экспериментальной фазы исследования новых алгоритмов анализа: без наличия инфраструктуры абстрактной интерпретации невозможно быстро прототипировать такие алгоритмы, так как требуется «с нуля» создавать новый инструмент.

Таким образом, задача построения практически применимой программной системы для проведения произвольно заданной абстрактной интерпретации бинарного кода является актуальной и востребованной. Основные требования к такой системе изложены нами ранее в работе [4]: необходимо (1) промежуточное представление, над которым и будет проводиться абстрактная интерпретация; (2) транслятор бинарного кода в промежуточное представление и (3) сама инфраструктура абстрактной интерпретации. Следует учитывать особенности работы конвейера процессора (слоты задержки, повторяющиеся команды, обработка исключений), наличие механизмов виртуальной памяти (сегментная и страничная трансляция) и т.п. По мнению авторов, одним из наиболее важных практических требований к такой системе становится использование внешних спецификаций для описания семантики выполнения команд и работы конвейера для каждого поддерживаемого процессора.

Работы последних лет [5, 6] в основном сфокусированы на промежуточном представлении и отчасти способе трансляции, сколь-либо выразительная инфраструктура для дальнейшего анализа не предлагается. Во всех случаях предполагается анализ пользовательского кода, работающего в одном пространстве виртуальной памяти. Зачастую рассматриваются только арифметико-логические команды и команды передачи управления, а трансляция происходит программным способом: промежуточное представление строится в коде явно для каждой поддерживаемой команды.

2. Промежуточное представление

Разработанное авторами промежуточное представление Pivot 2 подробно описано в [4]. Его наиболее крупной единицей является модуль, состоящий из следующих элементов.

- **Адресные пространства** единообразно описывают регистровые файлы, память, порты ввода-вывода и т.п. Адресные пространства подразделяются на локальные и удаленные. Локальные адресные пространства локальны для процессорного ядра, в них не возникают исключения при доступе, и тем самым они могут быть представлены как массивы. В удаленных адресных пространствах чтения и записи не имеют таких ограничений, что позволяет моделировать отображенные на память устройства, виртуальную память и т.д.
- **Фрагменты** являются подпрограммами с единственным входом и выходом. Как входными, так и выходными аргументами фрагментов являются битовые векторы, длины которых фиксированы в сигнатурах фрагментов.
- **Базовые блоки** составляют тела фрагментов. Каждый базовый блок состоит из последовательности операторов, а также имеет входные и выходные переменные. При

передаче управления по ребру соответствующие выходные переменные блока-предка копируются во входные переменные блока-потомка. Этот подход является альтернативой использованию φ-функций и позволяет естественным образом задать входы и выходы для фрагментов.

- **Операторы** в базовых блоках описывают выполняемые действия над локальными переменными фрагмента (находящимися в SSA-форме и являющимися битовыми векторами), либо над адресными пространствами. Существует 6 операторов:
  - *APPLY* — применение операции из SMT-логики BV;
  - *CALL* — вызов фрагмента;
  - *INIT* — инициализация переменной константным битовым вектором;
  - *LOAD* — загрузка значения переменной из адресного пространства;
  - *SLICE* — конкатенация и выделение подвекторов из переменных;
  - *STORE* — выгрузка значения переменной в адресное пространство.
- **Разборщики** представляют собой особый вид функций, которые по входному битовому вектору произвольной длины проводят его «лексический разбор», т.е. выдают набор лексем в виде аннотированных битовых векторов. Однако, в отличие от императивно описываемых фрагментов, разборщики задаются декларативно и работают на основе сопоставления образцов (pattern matching).

Операторы *LOAD* и *STORE* учитывают возможность различного порядка байтов в адресном пространстве при доступе, а также предполагают возможность ошибки доступа в случае удаленного адресного пространства.

```
// Адресное пространство регистров, размер адреса — 16 битов.
space regs('16) { x0: '64, x1: '64, … }
// Лексема «мнемоника ADD».
struct add;
// Лексема «операнд-регистр», содержащая номер регистра.
struct reg { rid: '5 }
// Разборщик для команды ADD.
tk rv64i {
    |0000000_nnnnn_mmmmm_000_ddddd_0110011| =>
        add, reg { rid: d }, reg { rid: m }, reg { rid: n };
}
// Семантика команды ADD (для упрощения опущены особенности
// обработки регистра с номером 0).
case fn sema(add: add, rd: reg, rs1: reg, rs2: reg) {
    write_reg(rd, 'add(read_reg(rs1), read_reg(rs2)));
}
fn read_reg(r: reg) -> '64 {
    regs('add(&regs.x0, <r.rid, 0'6>))
}
fn write_reg(r: reg, v: '64) {
    regs('add(&regs.x0, <r.rid, 0'6>)) = v;
}
```

Рис. 1. Пример спецификации машинной команды
Fig. 1. Machine instruction specification example

Кодировки машинных команд, их семантика, а также особенности конвейера могут быть описаны в рамках предложенного представления. Для облегчения задачи написания таких спецификаций был разработан предметно-ориентированный язык и реализован транслятор в промежуточное представление. На рис. 1 представлен фрагмент спецификации команды ADD регистрового сложения 64-разрядного процессора RISC-V. Здесь задается адресное

пространство регистров, лексемы (мнемоники и операнды), разборщик и операционная семантика команды. В общем случае разборщики дополнительно могут параметризоваться режимом работы процессора, если это влияет на декодирование. Фрагменты и разборщики автоматически оптимизируются при трансляции: производится продвижение и свертка констант, частичное удаление избыточности при помощи нумерации значений, частичный вынос вычислений из циклов и упрощение графа потока управления в части удаления пустых блоков и объединения линейных последовательностей блоков.

3. Инфраструктура абстрактной интерпретации

Абстрактная интерпретация проводится в рамках модуля промежуточного представления, и может быть как внутрипроцедурной, так и межпроцедурной. Для проведения абстрактной интерпретации требуется выбрать тип, который будет описывать абстрактное состояние, и задать передаточные функции. Тип абстрактного состояния должен содержать выделенный элемент, соответствующий невозможному состоянию (⊥). Передаточные функции делятся на три группы:

- функции, соответствующие операторам *APPLY*, *CALL*, *INIT*, *LOAD*, *SLICE*, *STORE*, принимают входное состояние и возвращают выходное;
- функция *CALL_RET*, обрабатывающая возврат из вызванного фрагмента;
- функция *EDGE*, обрабатывающая распространение состояний по ребрам ГПУ.

Заданная интерпретация также сообщает инфраструктуре направление обхода: прямое или обратное. Порядок вызова передаточных функций, поведение в части межпроцедурного анализа, а также остановка анализа реализуется инфраструктурной частью, общей для всех разновидностей абстрактной интерпретации, и называемой *исполнителем*. На настоящий момент реализованы два исполнителя.

Исполнитель вдоль пути позволяет применять передаточные функции вдоль некоторого одного пути в программе. При достижении ветвления исполнитель проверяет, что ровно одно из состояний целевых блоков является возможным (т.е. не имеет значение ⊥). Если это так, то исполнитель продолжает продвигаться по пути, а иначе останавливается.

Исполнитель неподвижной точки решает классическую MFP- или LFP-задачу. Он работает в рамках заданного фрагмента, поддерживая текущие состояния на входах и выходах базовых блоков, и обновляет эти состояния в соответствии с алгоритмом, похожим на алгоритм Килдалла [2] (отличия не являются существенными и связаны, в основном, с заменой φ-функций на копирования переменных на ребрах ГПУ). Для этого исполнителя требуется, чтобы состояние являлось полурешеткой с верхним и нижним элементами.

Оба исполнителя при достижении вызова фрагмента могут по желанию пользователя либо продолжить анализ внутри этого фрагмента (с подстановкой текущего состояния), либо использовать предоставленную аннотацию (summary). В первом случае также имеется возможность сохранить аннотацию, полученную при подстановке вызываемого фрагмента, для дальнейшего повторного использования. Вынос реализации межпроцедурного анализа в общую инфраструктурную часть существенно упрощает реализацию самих интерпретаций, т.к. требуется поддерживать абстрактное состояние только одного текущего фрагмента.

На базе разработанной инфраструктуры были реализованы и протестированы:

- **конкретная интерпретация**, эмулирующая выполнение Pivot-операторов и действий на ребрах в рамках фрагмента;
- **символьная интерпретация**, фиксирующая в виде SMT-выражений связи между переменными и ячейками адресных пространств фрагмента и вычисляющая предикаты пути (условия достижимости различных точек в программе);
- **смешанная интерпретация**, являющаяся композицией конкретной и символьной интерпретации, где только выбранная часть данных анализируется в символьном виде, а оставшиеся данные рассматриваются конкретно.

Кроме того, та же самая инфраструктура абстрактной интерпретации используется и в трансляторе спецификаций в части оптимизаций: например, для частичного удаления избыточности строятся множества доступных выражений в каждой точке фрагмента.

Разделение на непосредственно интерпретацию и исполнители, общие для различных интерпретаций, решает две задачи: во-первых, объем реализации интерпретаций (включая состояние), оказывается весьма мал, а, следовательно, эти реализации проще отлаживать (напомним, что быстрое проведение экспериментов — часть мотивации данной работы); а во-вторых, различные комбинации исполнителей и одних и тех же интерпретаций порождают либо хорошо известные задачи, либо близкие к ним (табл. 1). В таблице также указан объем реализации каждой из интерпретаций на языке Rust.

Табл. 1. Известные задачи как композиция интерпретации и исполнителя
Table 1. Known problems as composition of interpretation and executor

Интерпретация	**Строки кода**	**Исполнитель**	
		Вдоль пути	**Неподвижная точка**
Конкретная	350	эмуляция	свертка и продвижение констант
Символьная	500	«классическое» символьное выполнение по одному пути	слабейшие предусловия сильнейшие постусловия[1]

4. Модель процессора

Для описания поведения процессора используются разборщики и фрагменты задания операционной семантики. Их объединение в единую систему делается на базе модельного конвейера, состоящего из трех этапов: выборка, декодирование, исполнение.

На этапе **выборки** модельный конвейер выполняет действия по определению адресного пространства и адреса в нем очередной команды, а также режима процессора в части, влияющей на декодирование команд. Сюда же относится обработка исключений (в том числе внешних) и таких особенностей как слоты задержки, аппаратные циклы и т.п. Этот этап может быть задан как подпрограмма в спецификации.

На этапе **декодирования** модельный конвейер осуществляет преобразование машинного кода в последовательность лексем. При возникновении ошибки конвейер возвращается к этапу выборки. Декодирование осуществляется по заданным в спецификации разборщикам.

Наконец, этап **выполнения** соответствует анализу фрагмента, задающего семантику декодированной команды. Для этого по сигнатуре (последовательности лексем) выбирается подходящий фрагмент.

Таким образом, каждый полный цикл конвейера может рассматриваться как отображение входного состояния машины в выходное, причем то, какая команда будет выбираться следующей, также заложено в этом преобразовании. Это позволяет, в частности, проводить дизассемблирование методом рекурсивного спуска без дополнительной информации о командах, в том числе параллельно. При анализе по трассам выполнения в последних содержатся сведения об этапах выборки и декодирования, что для части задач позволяет сразу переходить к анализу семантики команд.

[1] В зависимости от выбора направления анализа и задания операций сбора/объединения.

5. Эксперименты

Экспериментальные запуски разработанной и реализованной инфраструктуры проводились на примере Cruehead [7]. Эта программа относится к тестовым программам класса crackme, т.е. ее исследование предполагает обратную инженерию бинарного кода с целью извлечения алгоритма проверки пары «имя пользователь»-«пароль» и генерации такой пары, которая пройдет проверки, заложенные в программе. В [7] описан ручной подход к решению этого примера. В проведенных экспериментах решение строилось автоматически.

Первый эксперимент проводился по предварительно полученной полносистемной трассе выполнения, где было введено произвольное имя пользователя и пароль (некорректная пара). Фрагмент трассы, соответствующий проверке пароля, был проанализирован с помощью смешанной интерпретации и исполнителя вдоль пути. Буфер с паролем был помечен как символьный, и на его байты было установлено дополнительное ограничение: все они должны быть цифрами. Все остальные данные интерпретировались конкретно, и их начальное состояние бралось из трассы. Смешанная интерпретация построила предикат пути, который соответствует корректному паролю (рис. 2). Предикат был отправлен в SMT-решатели Z3 [8] и Boolector [9], в обоих случаях система успешно решилась и был получен корректный пароль.

```
(ne (slice<0, 8> pwd) #x00)
(ne (slice<8, 16> pwd) #x00)
(ne (slice<16, 24> pwd) #x00)
(ne (slice<24, 32> pwd) #x00)
(ne (slice<32, 40> pwd) #x00)
(ne (slice<40, 48> pwd) #x00)
(eq #x000057BB (xor (add (mul (add (mul (add (mul (add (mul (add (mul
  (extu<24> (sub (slice<0, 8> pwd) #x30)) #x0000000A) (extu<24> (sub
  (slice<8, 16> pwd) #x30))) #x0000000A) (extu<24> (sub (slice<16, 24>
  pwd) #x30))) #x0000000A) (extu<24> (sub (slice<24, 32> pwd) #x30)))
  #x0000000A) (extu<24> (sub (slice<32, 40> pwd) #x30))) #x0000000A)
  (extu<24> (sub (slice<40, 48> pwd) #x30))) #x00001234))
```

Рис. 2. Предикат пути в примере Cruehead
Fig. 2. Path predicate in the Cruehead example

Важно отметить, что изображенный на рис. 2 предикат был получен как результат применения предварительных упрощений SMT-выражений. Эти упрощения автоматически производятся разработанной инфраструктурой и позволяют сократить глубину AST-дерева на один десятичный порядок, в основном за счет «склеивания» различных байтов 32-разрядных чисел в единый битовый вектор.

Второй эксперимент проводился по исполнимому образу программы (т.е. ее EXE-файлу). Здесь также применялось смешанное выполнение, но одновременно и имя пользователя, и пароль были помечены как символьные. Было дополнительно установлено символьное ограничение на длину имени пользователя и пароля, для этого в состоянии конкретной части смешанной интерпретации были заданы произвольные строки «имя пользователь» и «пароль». Поскольку в смешанной интерпретации путь определяется по конкретной части состояния, в данном примере этого было достаточно для фиксации длин строк. В данном эксперименте построенная система ограничений также была успешно решена и была получена пара «имя пользователя»-«пароль», которая была принята программой как верная.

Выполнение обоих экспериментов на машине с процессором AMD Ryzen 5 1400 3.20Hz с 32ГиБ оперативной памяти занимает менее секунды, учитывая время загрузки трассы или статического образа, трансляцию машинных команд в промежуточное представление, их анализ и работу двух решателей.

6. Заключение

Данная работа подводит промежуточные результаты продолжающегося исследования. Описаны ключевые особенности разработанной инфраструктуры, обеспечивающие на практике применимость метода абстрактной интерпретации к анализу бинарного кода различных процессорных архитектур. Реализация опробована на двух предварительных экспериментах, которые показали корректную работу.

Дальнейшие планы включают «эргономические» улучшения в предметно-ориентированном языке и его трансляторе (в частности, поддержку тип-параметрических функций и их мономорфизацию), а также в API, которое предоставляет инфраструктура. Планируется реализация в рамках инфраструктуры дизассемблера и легковесного средства трассировки отдельных путей на базе конкретной интерпретации. Набор исполнителей будет дополнен исполнителем по нескольким путям, необходимого для проведения «классического» символьного выполнения. Планируется также подготовка спецификаций для широко распространенных процессорных архитектур. После завершения данных работ планируется этап оценки и улучшения производительности всей системы на корпусе примеров.

Среди первоочередных практических приложений, которые могут быть построены на базе разработанной системы, можно отметить:

- отладчик, основанный на смешанном выполнении, что позволит анализировать отдельные фрагменты кода, когда невозможно запустить анализируемое ПО (так называемый частично работоспособный код);
- средство динамического символьного выполнения, позволяющее определять условия достижимости определенных участков кода;
- средство поиска и ранжирования ошибок и уязвимостей на уровне бинарного кода.

Список литературы / References

[1]. Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proc. of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1977, pp. 238-252.

[2]. Kildall G. A unified approach to global program optimization. In Proc. of the 1st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1973, pp. 194-206.

[3]. Mauborgne L. ASTRÉE: Verification of Absence of Run-Time Error. In Building the information society, IFIP International Federation for Information Processing, vol. 156, 2004, pp. 384-392.

[4]. Соловьев М.А., Бакулин М.Г., Горбачев М.С., Манушин Д.В., Падарян В.А., Панасенко С.С. О новом поколении промежуточных представлений, применяемом для анализа бинарного кода. Труды ИСП РАН, том 30, вып. 6, 2018, стр. 39-68 / Solovev M.A., Bakulin M.G., Gorbachev M.S., Manushin D.V., Padaryan V.A., Panasenko S.S. Next generation intermediate representations for binary code analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 6, 2018, pp. 39-68 (in Russian). DOI: 10.15514/ISPRAS-2018-30(6)-3.

[5]. Dullien T., Porst S. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In Proc. of the CanSecWest Conference, 2009, 7 p.

[6]. Jung M., Kim S., Han H., Choi J., Cha S.K. B2R2: building an efficient front-end for binary analysis. In Proc. of the NDSS workshop on Binary analysis research, 2019, 10 p.

[7]. Cruehead. URL: https://ansmirnov.ru/crackme-cruehead/ (in Russian), accessed: 23.10.2020.

[8]. The Z3 Theorem Prover. URL: https://github.com/Z3Prover/z3, accessed: 23.10.2020.

[9]. Boolector. URL: https://boolector.github.io/, accessed: 23.10.2020.

Информация об авторах / Information about authors

Михаил Александрович СОЛОВЬЕВ – кандидат физико-математических наук, старший научный сотрудник отдела компиляторных технологий ИСП РАН; старший преподаватель кафедры системного программирования факультета ВМК МГУ. Его научные интересы

включают анализ бинарного и исходного кода, обратную инженерию ПО, операционные системы.

Mikhail Aleksandrovich SOLOVEV is a candidate of physical and mathematical sciences, senior researcher at the compiler technologies department of ISP RAS; senior lecturer at the system programming department of the faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. His research interests include binary and source code analysis, software reverse engineering, and operating systems.

Максим Геннадьевич БАКУЛИН – младший научный сотрудник отдела компиляторных технологий. Его научные интересы включают анализ бинарного и исходного кода, динамический анализ помеченных данных, символьное выполнение, эмуляцию и виртуализацию.

Maksim Gennadevich BAKULIN is a junior researcher at the compiler technologies department. His research interests include binary and source code analysis, dynamic taint analysis, symbolic execution, emulation, and virtualization.

Сергей Сергеевич МАКАРОВ – студент кафедры системного программирования. Его научные интересы включают анализ бинарного кода, эмуляцию и виртуализацию, операционные системы, обратную инженерию ПО.

Sergei Sergeevich MAKAROV is a student at the system programming department. His research interests include binary code analysis, emulation and virtualization, operating systems, and software reverse engineering.

Дмитрий Валерьевич МАНУШИН – аспирант кафедры системного программирования. Его научные интересы включают анализ бинарного кода, анализ исходного кода, безопасность ПО.

Dmitrii Valerevich MANUSHIN is a postgraduate at the system programming department. His research interests include binary code analysis, source code analysis and software security.

Вартан Андроникович ПАДАРЯН – кандидат физико-математических наук, ведущий научный сотрудник отдела компиляторных технологий ИСП РАН; доцент кафедры системного программирования факультета ВМК МГУ. Его научные интересы включают компиляторные технологии, безопасность ПО, анализ бинарного кода, параллельное программирование, эмуляция и виртуализация.

Vartan Andronikovich PADARYAN is a candidate of physical and mathematical sciences, leading researcher at the compiler technologies department of ISP RAS; associate professor of the system programming department of the faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. His research interests include compiler technologies, software security, binary code analysis, parallel programming, emulation, and virtualization.