Анализ характера изменений программ и поиск неисправленных фрагментов кода¹

М.С. Арутюнян <arutunian@ispras.ru>

Г.С. Иванов <gregory@ispras.ru>

В.Г. Варданын <vaag@ispras.ru>

А.К. Асланиян <hayk@ispras.ru>

А.И. Аветисян <arut@ispras.ru>

Ш.Ф. Курмангалеев <kursh@ispras.ru>

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

Аннотация. Разработчики программного обеспечения часто прибегают к заимствованию кода – как внутри одного проекта, так и из других проектов. Ввиду возможного содержания ошибки в исходном фрагменте это может привести к её дальнейшему распространению по коду ПО. Используемые библиотеки без исходного кода также могут содержать потенциальные ошибки. Целью данной работы является разработка методов анализа характера изменений между версиями компонентов ПО, для которых отсутствует исходный код. А для изменений потенциально относящихся к исправлению дефектов поиск подобных, но не исправленных дефектов при помощи методов поиска клонов кода. Внедрение предлагаемого подхода к анализу используемых компонентов при разработке ПО позволит обеспечить оценку качества предлагаемых программных «заплаток». Поскольку реализованный метод независим от архитектуры операционной системы, а также работает с исполняемым кодом ПО это позволяет применять его как для анализа сторонних компонентов, так и для анализа бинарных сборок собственного программного обеспечения. Средний процент истинных срабатываний на тестовом наборе CoreBench составляет ~73%.

Ключевые слова: статический анализ кода; клоны кода; анализ исполняемого кода; анализ изменений

DOI: 10.15514/ISPRAS-2019-31(1)-3

Для цитирования: Арутюнян М.С., Иванов Г.С., Варданын В.Г., Асланиян А.К., Аветисян А.И., Курмангалеев Ш.Ф. Анализ характера изменений программ и поиск неисправленных фрагментов кода. Труды ИСП РАН, том 31, вып. 1, 2019 г., стр. 49-58. DOI: 10.15514/ISPRAS-2019-31(1)-3

1. Введение

Во время написания программы программисты часто используют готовые фрагменты кода, которые могут содержать ошибки [3]. Такими фрагментами могут быть как статически линкуемые библиотеки, так и фрагменты исходного кода, скопированные из другого проекта (например, с открытым исходным кодом). Ошибка может содержаться и в статически линкуемой библиотеке, и в заимствованном фрагменте. Затем такая ошибка может дублироваться в другом месте кода (в случае полного или частичного копирования фрагмента с ошибкой). Последующие исправление ошибки, в силу разных причин, не гарантирует исправления во всех клонах фрагмента [25].

Поиск таких ошибок может быть выполнен путём поиска клонов кода и анализа внесённых изменений в скопированный фрагмент кода. Более того, базовая информация об исправлениях

может использоваться для создания автоматических инструментов исправления ошибок [6] [7] [8].

В случаях полного или частичного отсутствия исходного кода проекта поиск ошибок такого рода усложняется. В то же время, большинство существующих средств анализа изменений работают на основе исходного кода [9] [10].

В статье описан метод сопоставления функций на базе поиска клонов кода [11] и анализа изменений на исполняемом коде, реализованный в рамках инструмента Binside в Институте системного программирования им. В.П. Иванникова Российской академии наук, на базе Binnavi [14]

Дальнейшее изложение построено следующим образом. В разд. 2 рассмотрен алгоритм сопоставления функций на основе анализ клонов кода, а также проблема поддержки анализа программного кода для различных архитектур. Разд. 3 посвящён анализу характера изменений, разд. 4 – поиску неисправленных ошибок. Разд. 5 описывает реализацию метода в инструменте. Разд. 6 обзору аналогичных работ. Разд. 7 завершает статью, описывая результаты работы.

2. Алгоритм сопоставления функций

Рассматриваются два исполняемых файла [11]. Они дизассемблируются и транслируются в промежуточное представление REIL [23], являющееся языком низкого уровня. Оно состоит из 17 инструкций, каждая из которых совершает одно элементарное действие с явно указанными в операндах переменными. Для ассемблерных инструкций, которые, к примеру, записывают во флаги результаты вычислений, при трансляции в REIL создаются явные инструкции работы с флагами. Использование представления REIL позволяет сделать анализ архитектурно независимым.

На основе указанного представления происходит генерация графа зависимостей программы (ГЗП) [22], графа вызовов функций (ГЗФ), графа потока управления и графа потока данных для каждой функции из обоих исполняемых файлов. Узлам ГЗП соответствуют инструкции REIL, а рёбрам – зависимости по данным и по управлению. При построении ГЗП используются графы потока управления и USE-DEF [21] цепочки (используется в качестве дополнительного модуля в Binside).

Процесс сопоставления состоит из двух этапов. Сначала сопоставление происходит на основе эвристик: для каждой функции рассчитывается ряд эвристик. Если результаты конкретной эвристики для некоторой функции из первого исполняемого файла совпадают с результатами эвристической оценки функции из второго исполняемого файла и не совпадают с оценками других функций, то две функции сопоставляются. Ниже приведены применяемые эвристики [12].

1. Сопоставление функций на основе хеширования ассемблерного кода.
2. Сопоставление рёбер графа вызовов на основе хеширования ГЗП по MD-индексу [28].
3. Сопоставление рёбер графа вызовов на основе хеширования MD-индекса соседних вершин начальной и конечной вершин ребра.
4. Сопоставление функций на основе хеширования вершин ГЗП, считанных на ГЗП на основе MD-индекса.
5. Сопоставление функций на основе хеширования вершин ГЗП. Эвристика считает хеш на основе зависимостей по данным между инструкциями, группирует их и считает конечный хеш.
6. Сопоставление функций на основе хеша ГЗП. Каждому коду операции сопоставляется простое число. После этого вычисляется произведение этих чисел для всех инструкций. Чтобы избежать переполнения, после каждого произведения учитывается только остаток деления на 2^{64} .

¹ Работа поддержана грантом РФФИ 18-07-01153А.

На следующем этапе происходит сопоставление тех функции, которые не сопоставились на первом этапе [11].

1. Сопоставляются все пары вершин графов вызовов функций из первого и второго исполняемого файла. Если их ГЗП изоморфны, то вершины графов вызовов функций сопоставляются. Если ни одна пара не сопоставилась, то сопоставляется пара вершин, функции которых похожи больше, чем функции других пар.
2. Рассматриваются предшественники (преемники) для каждой сопоставленной пары вершин: P1 и P2 (S1 и S2). Для всех пар вершин из P1 и P2 (S1 и S2) вычисляется наибольший общий подграф для их ГЗП и строится матрица из сопоставленных частей. Процесс обнаружения общего подграфа распараллеливается для каждой пары ГЗП, чтобы обеспечить масштабируемость.
3. Применяется венгерский алгоритм [24] для нахождения лучшего соответствия ГЗП функций из P1(S1) и ГЗП функций из P2(S2).
4. Шаги 2-4 повторяются до рассмотрения всех сопоставленных функций.

3. Анализ характера изменений в новых версиях исполняемых файлов

Набор сопоставленных функций и инструкций в них позволяет понять, какой фрагмент кода был добавлен или удален. Цель алгоритма – обнаружить важные изменения кода, являющиеся потенциальным исправлением ошибки. Список изменений, рассматриваемых инструментом, представлен ниже.

1. Добавление нового базового блока в исполняемом коде. Если все инструкции базового блока в первом исполняемом файле не сопоставились с инструкциями сопоставленного ему базового блока из второго исполняемого файла версии, считается, что добавился новый блок. Со стороны исполняемого кода это может означать добавление проверки или цикла.
2. Добавление новой инструкции возврата из функции. Если добавленный участок кода добавляет новый путь в конец функции, то считается, что добавилась инструкция возврата из функции. В этом случае выдётся предупреждение.
3. Изменение аргументов функции. Предупреждение выдётся, если от добавленных инструкций существует путь по потоку данных до инструкций, которые обрабатывают аргументы функции.
4. Изменение вызова функции. Предупреждение выдётся, если в сопоставленных инструкциях вызова функций во втором исполняемом файле вызывается другая функция. В ходе рефакторинга имена функций могут меняться, поэтому алгоритм указывает изменение вызова функции с учётом сопоставленных функций.
5. Добавление новой инструкции выхода из цикла.
6. Добавление новой инструкции для перехода в начало цикла.

4. Поиск неисправленных ошибок

В этом разделе рассматривается случай, когда фрагмент кода с ошибкой был скопирован в несколько частей программы, причём в новой версии программы исправление затронуло только один клон кода. Поиск ошибок, оставшихся неисправленными основан на поиске клонов кода [15].

Изменения в новой версии исполняемого файла, описанные в разделе 3, рассматриваются как исправления. После обнаружения исправлений алгоритм находит клоны неисправленного фрагмента в новой версии исполняемого файла. Такие клоны могут нуждаться в исправлении.

Как показывает практика, эти фрагменты очень часто содержат всего несколько ассемблерных инструкций. Поиск клонов таких фрагментов приводит к многочисленным ложным срабатываниям. Для решения данной проблемы были разработаны три разных алгоритма расширения неисправленных участков кода.

1. Фрагмент неисправленного кода расширяется до уровня функции, в которой он находится. Для неисправленной функции находятся все клоны в новой версии (аналитиком задаётся минимальный процент схожести).
2. Фрагмент неисправленного кода расширяется по базовым блокам по следующему принципу: в множество вставляется базовый блок, в котором найдено исправление, затем рассматриваются соседние базовые блоки по потоку управления. Количество базовых блоков ограничивается аналитиком. В новой версии исполняемого файла ищется полученное множество базовых блоков с учётом кодов операций и потока управления между базовыми блоками.
3. Фрагмент неисправленного кода расширяется по инструкциям по следующему принципу: в множество вставляются инструкции базового блока, в котором найдено исправление, потом рассматриваются соседние базовые блоки по потоку данных. Далее делается поиск полученного множества инструкций с учётом потока данных в новой версии исполняемого файла.

5. Структура инструмента

На рис. 1 представлена общая архитектура инструмента. На вход инструмента подаются два исполняемых файла. Сначала исполняемые файлы дизассемблируются с помощью IDA Pro [13], результаты выгружаются через инструмент Binexport [27] в базу данных PostgreSQL [26]. Ассемблерное представление в базе данных транслируется в представление REIL.

Рис 1. Архитектура инструмента
Fig. 1. Architecture of the tool

После трансляции ассемблерного представления в REIL происходит формирование информации необходимой для дальнейшей работы и возврат управления в основной инструмент, где выполняется сопоставление функций, анализ характера изменений и поиск неисправленных фрагментов

6. Обзор аналогичных работ

SPAIN [16] представляет собой масштабируемую систему анализа изменений, которая автоматически идентифицирует изменения безопасности и суммирует шаблоны изменений, а также соответствующие им шаблоны дефектов. В частности, если даны исходные и исправленные версии исполняемого кода программы, SPAIN находит функции, которые были изменены. Затем система обнаруживает изменённые трассы (то есть, последовательность базовых блоков) для каждой исправленной функции в целях описания изменений на уровне функции. Эти трассы могут содержать безопасные или небезопасные изменения. Посредством семантического анализа определяется, какие изменения безопасны, а какие нет. Затем с помощью анализа помеченных данных на исправленных функциях суммируются образцы небезопасных изменений и соответствующие им дефекты.

PVDF [17] вычисляет семантику исправлений для дефектов. Затем семантика изменений используется для обнаружения новых дефектов в исполняемых файлах. PVDF берёт исполняемый файл с дефектом и соответствующее изменение в качестве входных данных, а затем извлекает семантику. Данный инструмент похож на SPAIN, однако он предполагает наличие изменений и фокусируется только на одном конкретном типе дефекта.

Другие исследования основаны на символьном выполнении. BinHunt [18] пытается найти семантические различия между версиями. Чтобы определить, насколько похожи два базовых блока, инструмент проводит проверку равенства на равенство всех возможных пар уравнений из обоих наборов с использованием SMT-решателей. На основе идентифицированных семантически равных базовых блоков алгоритм обратного отслеживания находит наибольший общий подграф между двумя функциями и получает сходство двух функций. После нахождения изменённых базовых блоков информация передается аналитику. iBinHunt [19] похож на BinHunt, однако использует межпроцедурный анализ и анализ помеченных данных. Эти инструменты рассматривают семантику программы, но они очень трудоёмки.

7. Результаты

В табл. 1 приведены результаты тестов по нахождению клонов неисправленного фрагмента в новых версиях. Такие фрагменты были исправлены в последующих версиях.

Табл. 1. Результаты работы
Table 1. Work results

Проект	Коммиты git		Имя функции с исправленной ошибкой	Имя функции с неисправленной ошибкой
	Старая версия	Новая версия		
Tcpdump	B534e304	d3aae719	juniper_monitor_print	juniper_mlft_print
Tcpdump	C2ef6938	50a44b6b	ikev1_nonce_print	1.ikev1_hash_print 2.ikev1_sig_print 3.ikev1_ke_print 4.ikev1_vid_print
libosip	79240bdd	a54f15b8	osip_www_authenticate_init	1.sdp_connection_init 2.osip_authorization_init 3.osip_authentication_info_init
Libosip	80a955e7	03fe3a1c	osip_negotiation_sdp_build_offer	__osip_negotiation_sdp_build_offer

В табл. 2 приведены результаты анализа изменений на тестовом наборе Corebench [20].

Табл. 2. Результаты на тестах из набора Corebench
Tab. 2. Results on tests from the Corebench suite

Проект	Коммиты git		Количество найденных изменений	Количество правильных срабатываний	Процент правильных срабатываний
	старая	новая			
find	244453b8	f7197f3a	4	2	50%
find	756b47b1	24e2271e	3	2	67%
find	aca12907	b445af98	1	1	100%
grep	02f1daa1	074842d3	2	2	100%
grep	c1cb19fe	8f0b8d8e2	2	1	50%
grep	c2b9a4fe	6d952bee	6	6	100%
make	87ac68fe	40a49f24	5	3	60%
make	97f106fa	fc644b4c	9	7	77%
make	c3188c6f	3b1432d8	4	3	75%

В среднем процент правильных срабатываний на тестовом наборе Corebench составляет 73.3%.

8. Выводы

В данной статье был предложен метод анализа характера изменений программ и поиска неисправленных фрагментов между версиями компонентов ПО, для которых отсутствует исходный код. Были сформирован список изменений, наиболее часто используемых для исправления ошибок. А также метод поиска фрагментов кода оставшихся неисправленными в новых версиях исполняемого файла при помощи методов поиска клонов кода. Доля корректных срабатываний на тестовом наборе Corebench составляет 73%. Разработанный метод позволит обеспечить оценку качества предлагаемых программных «заплаток».

Список литературы

[1] S. Ducasse, M. Rieger, S. Demeyer. A language independent approach for detecting duplicated code. In Proc. of the 15th International Conference on Software Maintenance, 1999, pp. 109-118.

[2] T. Kamiya, S. Kusumoto, K. Inoue. CCFinder: A multilinguistic tokenbased code clone detection system for large scale source code. IEEE Transactions on Software Engineering, vol. 28, issue 7, 2002, pp. 654-670.

[3] S. C. Misra и V. C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. Lecture Notes in Computer Science, vol. 2667, 2003, pp. 724-732.

[4] ГОСТ 19.102-77. Единая система программной документации. Стадии разработки. Москва, Стандартинформ, 2010.

[5] Security Development Lifecycle (SDL). Режим доступа: <https://www.microsoft.com/en-us/sdl>, дата обращения 10.01.2019.

[6] F. Long, M. Rinard. Automatic patch generation by learning correct code. In Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'2016), 2016, pp. 298-312.

[7] D. Kim, J. Nam, J. Song, S. Kim. Automatic patch generation learned from human-written patches, In Proc. of the 35th International Conference on Software Engineering (ICSE), 2013, pp. 802-811.

- [8] S. Mehtaev, J. Yi, A. Roychoudhury. Directfix: Looking for simple program repairs. In Proc. of the 37th IEEE International Conference on Software Engineering (ICSE), 2015, pp. 448-458.
- [9] Y. Tian, J. Lawall, D. Lo. Identifying linux bug fixing patches. In Proc. of the 34th International Conference on Software Engineering (ICSE), 2012, pp. 386-396.
- [10] C.S. Corley, N.A. Kraft, L.H. Etzkorn, S.K. Lukins. Recovering traceability links between source code and fixed bugs via patch analysis. In Proc. of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering, 2011, pp. 31-37.
- [11] H.K. Aslanyan. Effective and Accurate Binary Clone Detection. *Mathematical Problems of Computer Science*, vol. 48, 2017, pp. 64-73.
- [12] H. Aslanyan, A. Avetisyan, M. Arutunian, G. Keropyan, S. Kurmangaleev, V. Vardanyan, Scalable Framework for Accurate Binary Code Comparison. In Proc. of the 2017 Ivannikov ISPRAS Open Conference (ISPRAS), Moscow, 2017. DOI: 10.1109/ISPRAS.2017.00013
- [13] Ida Pro. Режим доступа: <https://www.hex-rays.com/products/ida>, дата обращения 10.01.2019.
- [14] Binnavi. Режим доступа: <https://www.zynamics.com/binnavi.html>, дата обращения 10.01.2019.
- [15] А.К. Аслanian, Ш.Ф. Курмангалеев, В.Г. Варданян, М.С. Арутюнян, С.С. Саргсян. Платформенно-независимый и масштабируемый инструмент поиска клонов кода в бинарных файлах. *Труды ИСП РАН*, vol. 28, вып 5, 2016 г., стр. 215-226. DOI: 10.15514/ISPRAS-2016-28(5)-13.
- [16] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, F. Song. SPAIN: Security Patch Analysis for Binaries Towards Understanding the Pain and Pills. In Proc. of the IEEE/ACM 39th International Conference on Software Engineering, 2017, pp. 462-472.
- [17] S. Letian, F. Jianming, C. Jing, P. Guojun. PVDF: An automatic Patch-based Vulnerability Description and Fuzzing method. In Proc. of the 2014 Communications Security Conference (CSC 2014), 2014, pp. 1-8.
- [18] D. Gao, M. K. Reiter, D. Song. Binhunt: Automatically finding semantic differences in binary programs. *Lecture Notes in Computer Science*, vol. 5308, 2008, pp. 238-255.
- [19] J. Ming, M. Pan, D. Gao. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In Proc. of the 15th international conference on Information Security and Cryptology, 2012, pp. 92-109
- [20] Corebench. Режим доступа: <https://www.comp.nus.edu.sg/~release/corebench/>, дата обращения 10.01.2019.
- [21] M. Jean, M. Lou. Efficient Computation of Interprocedural Definition-Use Chains. *ACM Transactions on Programming Languages and Systems*, vol. 16, issue 2, 1994, pp. 175-204.
- [22] J. Ferrante, J. Karl, D. Joe. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, vol. 9, issue 3, 1987, pp. 319-349.
- [23] T. Dullien, S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In Proc. of the CanSecWest Conference, 2009, 7 p.
- [24] Bruff Derek. The Assignment Problem and the Hungarian Method, 2005. http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf дата обращения 10.01.2019.
- [25] CVE-2017-12990/Fix printing of ISAKMPv1 Notification payload data. Режим доступа: <https://50.3.69.148/hawken/tcpdump/commit/c2ef693866beae071a24b45c49f9674af1df4028>, дата обращения 10.01.2019.
- [26] PostgreSQL. Режим доступа: <https://www.postgresql.org/>, дата обращения 10.01.2019.
- [27] Binexport. Режим доступа: <https://github.com/google/binexport/>, дата обращения 10.01.2019.
- [28] T. Dullien, E. Carrera, S. Eppler, S. Porst. Automated Attacker Correlation for Malicious Code. Technical report, DTIC Document, 2010, 9 p.

Analysis of program changes nature and searching for unpatched code fragments

M.S. Arutunian <arutunian@ispras.ru>

G.S. Ivanov <gregory@ispras.ru>

V.G. Vardanyan <vaag@ispras.ru>

H.K. Aslanyan <hayk@ispras.ru>

A.I. Avetisyan <arut@ispras.ru>

Sh.F. Kurmangaleev <kursh@ispras.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences,

25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract: Software developers often resort to borrowing code both within one project and from another. Due to the possible content of errors in the source code snippet, this can lead to error propagation across program. Libraries used without source code may also contain potential errors. The purpose of this work is developing methods for analyzing the nature of changes between versions of software components for which source code is missing. And for changes potentially related to the correction of defects, search for similar, but not fixed defects using the code clone search methods. The implementation of the proposed approach to the analysis of the components used in software development will ensure the assessment of the quality of the proposed software patches. Since the implemented method is independent of the architecture of the operating system, and also works with executable software code, this allows it to be used for analyzing third-party components and for analyzing binary builds of proprietary software. The average percentage of true positives on the CoreBench test suite is ~ 73%.

Keywords: code static analysis; code clones; binary code analysis; unpatched code fragments

DOI: 10.15514/ISPRAS-2019-31(1)-3

For citation: Arutunian M.S., Ivanov G.S., Vardanyan V.G., Aslanyan H.K., Avetisyan A.I., Kurmangaleev Sh.F. Analysis of the nature of program changes and the search for uncorrected code fragments. *Trudy ISP RAN/Proc. ISP RAS*, vol. 31, issue 1, 2019. pp. 49-58 (in Russian). DOI: 10.15514/ISPRAS-2019-31(1)-3

References

- [1] S. Ducasse, M. Rieger, S. Demeyer. A language independent approach for detecting duplicated code. In Proc. of the 15th International Conference on Software Maintenance, 1999, pp. 109-118.
- [2] T. Kamiya, S. Kusumoto, K. Inoue. CCFinder: A multilinguistic tokenbased code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, vol. 28, issue 7, 2002, pp. 654-670.
- [3] S. C. Misra и V. C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. *Lecture Notes in Computer Science*, vol. 2667, 2003, pp. 724-732.
- [4] ГОСТ 19.102-77. Единая система программной документации. Стадии разработки. Москва, Стандартинформ, 2010.
- [5] Security Development Lifecycle (SDL). Режим доступа: <https://www.microsoft.com/en-us/sdl>, дата обращения 10.01.2019.
- [6] F. Long, M. Rinard. Automatic patch generation by learning correct code. In Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'2016), 2016, pp. 298-312.
- [7] D. Kim, J. Nam, J. Song, S. Kim. Automatic patch generation learned from human-written patches, In Proc. of the 35th International Conference on Software Engineering (ICSE), 2013, pp. 802-811.
- [8] S. Mehtaev, J. Yi, A. Roychoudhury. Directfix: Looking for simple program repairs. In Proc. of the 37th IEEE International Conference on Software Engineering (ICSE), 2015, pp. 448-458.
- [9] Y. Tian, J. Lawall, D. Lo. Identifying linux bug fixing patches. In Proc. of the 34th International Conference on Software Engineering (ICSE), 2012, pp. 386-396.
- [10] C.S. Corley, N.A. Kraft, L.H. Etzkorn, S.K. Lukins. Recovering traceability links between source code and fixed bugs via patch analysis. In Proc. of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering, 2011, pp. 31-37.

- [11] H.K. Aslanyan. Effective and Accurate Binary Clone Detection. *Mathematical Problems of Computer Science*, vol. 48, 2017, pp. 64-73.
- [12] H. Aslanyan, A. Avetisyan, M. Arutunian, G. Keropyan, S. Kurmangaleev, V. Vardanyan, Scalable Framework for Accurate Binary Code Comparison. In *Proc. of the 2017 Ivannikov ISPRAS Open Conference (ISPRAS)*, Moscow, 2017. DOI: 10.1109/ISPRAS.2017.00013
- [13] Ida Pro. Режим доступа: <https://www.hex-rays.com/products/ida>, дата обращения 10.01.2019.
- [14] Binnavi. Режим доступа: <https://www.zynamics.com/binnavi.html>, дата обращения 10.01.2019.
- [15] H.K. Aslanyan, S.F. Kurmangaleev, V.G. Vardanyan, M.S. Arutunian, S.S. Sargsyan, Platform-independent and scalable tool for binary code clone detection. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 5, 2016, pp. 215-226 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-13.
- [16] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, F. Song. SPAIN: Security Patch Analysis for Binaries Towards Understanding the Pain and Pills. In *Proc. of the IEEE/ACM 39th International Conference on Software Engineering*, 2017, pp. 462-472.
- [17] S. Letian, F. Jianming, C. Jing, P. Guojun. PVDF: An automatic Patch-based Vulnerability Description and Fuzzing method. In *Proc. of the 2014 Communications Security Conference (CSC 2014)*, 2014, pp. 1-8.
- [18] D. Gao, M. K. Reiter, D. Song. Binhunt: Automatically finding semantic differences in binary programs. *Lecture Notes in Computer Science*, vol. 5308, 2008, pp. 238-255.
- [19] J. Ming, M. Pan, D. Gao. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *Proc. of the 15th international conference on Information Security and Cryptology*, 2012, pp. 92-109
- [20] Corebench. Режим доступа: <https://www.comp.nus.edu.sg/~release/corebench/>, дата обращения 10.01.2019.
- [21] M. Jean, M. Lou. Efficient Computation of Interprocedural Definition-Use Chains. *ACM Transactions on Programming Languages and Systems*, vol. 16, issue 2, 1994, pp. 175-204.
- [22] J. Ferrante, J. Karl, D. Joe. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, vol. 9, issue 3, 1987, pp. 319-349.
- [23] T. Dullien, S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In *Proc. of the CanSecWest Conference*, 2009, 7 p.
- [24] Bruff Derek. The Assignment Problem and the Hungarian Method, 2005.
http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf дата обращения 10.01.2019.
- [25] CVE-2017-12990/Fix printing of ISAKMPv1 Notification payload data. Режим доступа: <https://50.3.69.148/hawken/tcpdump/commit/c2ef693866beae071a24b45c49f9674af1df4028>, дата обращения 10.01.2019.
- [26] PostgreSQL. Режим доступа: <https://www.postgresql.org/>, дата обращения 10.01.2019.
- [27] Binexport. Режим доступа: <https://github.com/google/binexport/>, дата обращения 10.01.2019.
- [28] T. Dullien, E. Carrera, S. Eppler, S. Porst. Automated Attacker Correlation for Malicious Code. Technical report, DTIC Document, 2010, 9 p.