DOI 10.24412/1829-0450-fm-2026-1-68-75
УДК 519.688

Поступила: 09.03.2026г.
Сдана на рецензию: 10.03.2026г.
Подписана к печати: 19.03.2026г.

ИССЛЕДОВАНИЕ И РАЗРАБОТКА МЕТОДОВ СОПОСТАВЛЕНИЯ ИСХОДНОГО И БИНАРНОГО КОДА

О.М. Мовсисян, А.К. Асланян

Российско-Армянский университет
hovhannes.movsisyan@rau.am, hayk.aslanyan@rau.am
ORCID: 0000-0002-7582-7948, 0000-0002-7320-4835

АННОТАЦИЯ

В данной статье рассматривается проблема сопоставления исходного и исполняемого кода функций в условиях отсутствия отладочной и символьной информации. Представлен инструмент Pigaios++, являющийся развитием инструмента Pigaios. Основные улучшения включают внедрение механизмов предварительного сопоставления библиотечных функций, а также использование новых метрик, таких как сравнение сигнатур функций и анализ вызовов библиотечных функций. Предложен метод повышения точности сопоставления за счет анализа контекста расположения функций относительно уже идентифицированных пар. Для оценки эффективности разработан алгоритм автоматической генерации тестов. Экспериментальные результаты на базе пакета GNU Coreutils показали, что внесенные модификации позволили повысить показатель точности на 14.15% и полноты на 10.3% по сравнению с базовым решением.

Ключевые слова: сопоставление кода, статическое сопоставление, анализ уязвимостей.

Введение

В современном мире требования к программному обеспечению непрерывно растут. В условиях постоянно усиливающейся конкуренции велико стремление к быстрому получению результатов. В связи с этим часто используются уже существующие инструменты и библиотеки, которые могут применяться как в исходном виде, так и с определенными изменениями. Все это может привести к нарушениям авторских прав и условий лицензирования программного обеспечения с открытым исходным кодом. Например, если какая-либо программа имеет лицензию GNU General Public License (GPL) [1], то любой, кто изменяет или использует эту программу, обязан распространить исходный код под той же лицензией..

Программисты и компании продают пользователю в основном исполняемый код, а не исходный. Следовательно, для обнаружения вышеупомянутых

нарушений часто необходимо понять, был ли данный исполняемый код получен из данного исходного кода.

Вместе с постоянно растущими объемами кода увеличивается и количество содержащихся в них уязвимостей. Разрабатываются различные инструменты для автоматического обнаружения ошибок как в исходном, так и в исполняемом коде. И если известно о наличии уязвимости в некотором фрагменте исходного кода, то, найдя подобный участок кода в исполняемом файле, можно предположить, что такая же уязвимость присутствует и в исполняемом коде. Верно и обратное [7, 8].

Таким образом, сопоставление исходного и исполняемого кода имеет ряд применений:

- **Выявление нарушений авторских прав и проверка лицензии GNU GPL (General Public License)**. Ставится задача проверить, из какого исходного кода был получен данный исполняемый код.
- **Обнаружение уязвимостей**. Ставится задача найти в исполняемом (исходном) коде участок, аналогичный уязвимому фрагменту, присутствующему в исходном (исполняемом) коде.
- **Задачи обратной разработки (Reverse engineering)**. Ставится задача восстановления исходного кода, соответствующего данному исполняемому коду.

Сходство исходного и исполняемого кода

Известно, что из одного и того же исходного кода можно получить различный исполняемый код. Это обусловлено тем, что разные компиляторы могут генерировать разный объектный код, а также применять различные оптимизации. Исполняемые файлы также различаются в зависимости от архитектуры целевой машины.

Будем считать, что данный исполняемый код подобен исходному коду, если он может быть получен в результате компиляции этого исходного кода.

Постановка задачи

Разработать и реализовать инструмент сопоставления исходного и исполняемого кода, который обеспечивает соответствие между функциями исходного и исполняемого кода. Рассматривается случай, когда предоставленный исходный код может не компилироваться (например, из-за отсутствия некоторых файлов), но при этом является синтаксически анализируемым.

Если в исполняемом коде присутствует отладочная информация, решение задачи тривиально, так как она содержит прямые связи между строками исходного кода и инструкциями исполняемого кода. Однако следует отметить, что из исполняемого кода, предоставляемого конечному пользователю, отладочная и символьная информация обычно удаляется [2] (связь между строками исходного кода и инструкциями разрывается, а имена функций заменяются или удаляются).

В данной работе мы будем исходить из предположения, что в исполняемом коде, сравниваемом с исходным, отладочная и символьная информация отсутствует.

Существующие инструменты

Существует несколько инструментов, предназначенных для сопоставления исходного и исполняемого кода:

- **Pigaios** [3] – принимает на вход исполняемый и исходный код и находит соответствующие друг другу функции. Инструмент выполняет сопоставление на основе имен функций, строковых констант, количества циклов и условий, вызовов функций, глобальных переменных, количества операторов *switch* и их вариантов (*case*), а также рекурсивности функций. Инструмент не требует компиляции исходного кода – достаточно лишь успешного прохождения синтаксического анализа.
- **BinPro** [4] – принимает на вход исполняемый и исходный код и вычисляет коэффициент сходства, который представляет собой процент функций исходного кода, нашедших соответствие в исполняемом файле. Чем выше этот коэффициент, тем больше вероятность того, что исполняемый код был скомпилирован из данного исходного кода. Алгоритм сопоставления устойчив к таким изменениям, как правка комментариев, переименование функций или изменение порядка объявления переменных. Более существенное влияние на коэффициент сходства оказывают структурные изменения программы и изменение значений констант. На основе предварительно собранных данных инструмент пытается предугадать, какие функции исходного кода будут встроены (*inline*). Сопоставление выполняется на основе строковых и числовых констант, вызовов библиотечных функций, графа вызовов и количества аргументов. Требуется, чтобы исходный код был компилируемым с помощью GCC или Intel C++ Compiler (ICC). Исполняемый код должен быть скомпилирован под архитектуру x86–64 с заранее известным уровнем оптимизации.
- **Re-Source** [5] – получает на вход только исполняемый код и ищет совпадения в базах данных, содержащих исходный код. Инструмент выполняет сопоставление на основе константных значений операндов, вызовов функций и строковых констант. Базы исходного кода, использовавшиеся инструментом, в настоящее время недоступны.
- **CodeBin** [6] – принимает на вход только исполняемый код и ищет соответствия в онлайн-базе исходных кодов. Сопоставление основано на количестве операндов, вызовах пользовательских и библиотечных функций, строковых и числовых константах, а также сложности графа потока управления.

Алгоритмы сопоставления вышеуказанных инструментов учитывают лишь часть программных инструкций (вызовы функций, количество циклов) и строковые константы, что зачастую приводит к низкой точности. При этом только инструмент Pigaios способен работать с некомпилируемым исходным кодом.

Инструмент Pigaios

Инструмент Pigaios принимает на вход исходный и исполняемый код, а на выходе выдает пары сопоставленных функций. Работа инструмента состоит из трех этапов.

На первом этапе из исходного и исполняемого кода извлекаются определенные свойства (имена функций, строковые и числовые константы и др.) с использованием “Clang bindings” [9] и IDA Pro [10], соответственно.

На втором этапе на основе выбранных метрик из исходного и исполняемого кода отбираются потенциально схожие функции. Используются следующие метрики:

- Идентичность имен функций – выбираются функции с одинаковыми именами.
- Идентичность констант – выбираются функции, содержащие одни и те же строковые или числовые константы.
- Идентичность исходного файла – выбираются функции, принадлежащие исходным файлам с одинаковыми именами.

На третьем этапе для каждой пары на основе извлеченных характеристик вычисляется коэффициент сходства. К этим характеристикам относятся:

- имя функции;
- количество условий;
- константы;
- количество циклов;
- количество операторов switch;
- вызовы функций;
- глобальные и локальные переменные;
- рекурсивность.

Чем больше характеристик совпадает, тем выше вероятность того, что функции исходного и исполняемого кода идентичны. Пары, чей коэффициент сходства превышает заранее заданный порог, сохраняются как схожие функции. На третьем этапе, основываясь на парах функций, отобранных на предыдущем этапе, выбираются новые кандидаты с использованием структурных метрик:

1. «Соседние» функции. Если функции f_s (исходный код) и f_b (исполняемый код) уже сопоставлены, и в исходном коде есть функция g_s, которая предшествует или следует за f_s, а в исполняемом коде есть функция g_b, предшествующая или следующая за f_b, то пара (g_s, g_b) выбирается для сравнения.

2. Вызываемые функции. Если f_s и f_b сопоставлены, и f_s вызывает g_s, а f_b вызывает g_b, то пара (g_s, g_b) выбирается для сравнения.
3. Вызывающие функции. Если f_s и f_b сопоставлены, и g_s вызывает f_s, а g_b вызывает f_b, то пара (g_s, g_b) выбирается для сравнения.

Для этих новых пар также вычисляется коэффициент сходства. Процесс поиска новых пар продолжается итеративно до тех пор, пока это возможно по указанным метрикам. Пары с высоким коэффициентом сходства добавляются к списку найденных соответствий.

Итоговым результатом работы инструмента является совокупность пар схожих функций, накопленных на втором и третьем этапах.

Инструмент Pigaios++

Для улучшения работы инструмента Pigaios были добавлены механизмы сопоставления библиотечных функций, новые метрики и дополнительные свойства сравнения.

Сопоставление библиотечных функций

На втором этапе работы инструмента, перед отбором пар на основе метрик, выполняется сопоставление библиотечных функций. Важно отметить, что имена библиотечных функций не удаляются из исполняемого кода даже в тех случаях, когда в нем отсутствует символьная информация.

Процесс организован следующим образом:

Из исполняемого кода выделяются все библиотечные функции. Их обнаружение осуществляется с помощью инструмента IDA Pro с использованием технологии F.L.I.R.T. (Fast Library Identification and Recognition Technology) [11]. Затем найденные функции ищутся в исходном коде по их именам. Если в исходном коде обнаруживается функция с идентичным именем, данная пара считается точно сопоставленной. Функции, сопоставленные на данном этапе, исключаются из дальнейшего рассмотрения как потенциальные кандидаты на других стадиях работы алгоритма. Этот шаг направлен на оптимизацию производительности инструмента, а также позволяет значительно снизить вероятность выбора ошибочных пар функций в ходе последующего анализа.

Метрики

К существующим метрикам инструмента Pigaios были добавлены новые:

- **Сигнатура функции** – для сравнения выбираются функции из исходного и исполняемого кода, имеющие идентичные сигнатуры.
- **Вызовы библиотечных функций** – для сравнения отбираются функции, которые вызывают одни и те же библиотечные функции.

- **Функции, расположенные между сопоставленными парами** – выбираются пары функций, которые находятся в интервалах между уже сопоставленными строками исходного кода и соответствующими адресами исполняемого кода.

Свойства сравнения

В перечень свойств для сравнения функций было добавлено новое свойство – **сигнатура функции**. Теперь для каждой сравниваемой пары, помимо вышеупомянутых характеристик, учитывается и соответствие сигнатур, что влияет на итоговый коэффициент сходства.

Тестирование

Для оценки точности работы инструментов была разработана автоматизированная система тестирования. Процесс тестирования состоит из двух этапов: автоматической генерации тестов и непосредственного тестирования инструментов на этих данных. Для автоматической генерации тестов этого используются две версии одной и той же программы (которые могут совпадать), обозначим их V1, V2. На первом этапе исходный код V1 компилируется с отладочной информацией, что позволяет сохранить символьную информацию, в частности, имена функций. Компиляция может выполняться с различными уровнями оптимизации. Полученный исполняемый код обозначим B1.

Затем выполняется сопоставление функций из B1 и V2, имеющих одинаковые имена. Полученные пары представляют собой «эталонные» соответствия, которые в лучшем случае должен обнаружить тестируемый инструмент при сравнении B1 и V2.

На втором этапе тестируемый инструмент применяется к сформированным тестам, после чего вычисляются показатели точности (precision) и полноты (recall).

В начале процесса сохраняется соответствие между именами функций исполняемого кода и адресами их начала. Затем из исполняемого файла удаляется вся символьная информация (включая имена функций) с помощью утилиты strip [12]. Удаление символов необходимо для максимального приближения условий теста к реальным сценариям, так как поставляемое пользователям ПО обычно не содержит такой информации. На следующем шаге исходный и «очищенный» исполняемый код подаются на вход тестируемого инструмента.

Пары функций, сопоставленные инструментом, сравниваются с эталонными парами, полученными на первом этапе. Для каждой найденной функции из исполняемого кода восстанавливается ее имя (по ранее сохраненному адресу начала функции). Если в найденной паре имена функций из исходного

и исполняемого кода совпадают, результат считается истинно положительным, в противном случае – ошибкой. На основе этих данных рассчитываются точность и полнота.

Для проведения тестирования использовался исходный код libxml2 [13] и стандартного пакета программ coreutils [14] операционной системы Linux. Пакет включает более ста программ и был выбран для обеспечения разнообразия входных данных.

Результаты

Инструмент Pigaios++ был протестирован с помощью разработанной автоматизированной системы. В Табл. 1 приведено сравнение результатов работы оригинального инструмента Pigaios и его модифицированной версии Pigaios++.

Таблица 1. Сравнение результатов Pigaios и Pigaios++

Исходный Код	Исполняемый Код (Оптимизация)	Pigaios Точность (%)	Pigaios++ Точность (%)	Pigaios Полнота (%)	Pigaios++ Полнота (%)
coreutils-8.18	coreutils-8.18 (O0)	18.7	34.7	3.1	9.2
coreutils-8.32	coreutils-8.18 (O0)	17	33.7	2.8	9.7
coreutils-8.18	coreutils-8.18 (O3)	12.1	27.7	2.6	9.4
libxml-2	libxml2.so.2(02)	22.4	30.7	25.8	36.9

Анализ сравнения показывает, что внесенные изменения положительно повлияли на итоговые показатели и повысили точность и полноту работы инструмента. В частности, показатель точности улучшился в среднем на **14.15**%, а полнота – на **10.3**%.

ЛИТЕРАТУРА

1. https://www.gnu.org/licenses/gpl-3.0.en.html
2. https://en.wikipedia.org/wiki/Debug_symbol

3. http://joxeankoret.com/blog/2018/08/12/histories-of-comparing-binaries-with-source-codes/
4. *Miyani D., Huang Z., Lie D.* Binpro: A tool for binary source code provenance [J]. arXiv preprint arXiv:1711.00830, 2017.
5. *Rahimian A., Charland Ph., Preda S. and Debbabi M*. (2012). RESource: a framework for online matching of assembly with open source code. In International Symposium on Foundations and Practice of Security. Springer, 211–226.
6. https://users.encs.concordia.ca/~mmannan/student-resources/Thesis-MASc-Shahkar-2016.pdf
7. *Keropyan G., Vardanyan V., Aslanyan H., Kurmangaleev S. and Gaissaryan S.* Multiplatform Use-After-Free and Double-Free Detection in Binaries. Mathematical Problems of Computer Science, vol. 48, 2017. PP. 50–56.
8. *Aslanyan H., Asryan S., Hakobyan J., Vardanyan V., Sargsyan S.and Kurmangaleev S.* Multiplatform Static Analysis Framework for Programs Defects Detection. In CSIT Conference 2017, Yer., Armenia, 2017.
9. https://clang.llvm.org/docs/Tooling.html
10. https://www.hex-rays.com/ida-pro/
11. https://www.hex-rays.com/products/ida/tech/flirt/in_depth/
12. https://sourceware.org/binutils/docs/binutils/strip.html
13. https://github.com/coreutils/coreutils

RESEARCH AND DEVELOPMENT OF METHODS FOR SOURCE AND BINARY CODE MATCHING

H. Movsisyan, H. Aslanyan
Russian-Armenian University

ABSTRACT

This paper addresses the problem of matching source code with its corresponding executable functions in the absence of debugging and symbol information. We present Pigaios++, an enhanced tool built upon the Pigaios framework. Key improvements include the integration of preliminary library function matching mechanisms and the introduction of new metrics, such as function signature comparison and library call analysis. Furthermore, a method is proposed to improve matching precision by analyzing the positional context of functions relative to already identified pairs.

To evaluate the effectiveness of the tool, an automated test generation algorithm was developed. Experimental results conducted on the GNU Coreutils suite demonstrate that these modifications improved the average precision by 14.15% and recall by 10.3% compared to the baseline implementation.

Keywords: code matching, static matching, vulnerability analysis.