DOI: 10.15514/ISPRAS-2026-38(4)-5

Идентификация статически связанных библиотек в исполняемых файлах с использованием сопоставления графов зависимостей программы

М.С. Арутюнян, ORCID: 0000-0002-2420-7968 <mariam.arutunian@rau.am>
А.К. Асланян, ORCID: 0000-0002-7320-4835 <hayk.aslanyan@rau.am>

Российско-Армянский университет,
Армения, 0051, г. Ереван, ул. О. Эмина, д. 123.

Аннотация. Статически связанные библиотеки упрощают распространение и использование программного обеспечения, но усложняют анализ бинарных файлов, поскольку код библиотеки становится неотличимым от кода программы после компиляции. Идентификация встроенных библиотек и их версий имеет важное значение для анализа состава программного обеспечения, оценки уязвимостей и соблюдения лицензионных требований. Несмотря на значительный прогресс существующих решений, достижение точной идентификации библиотек при различных оптимизациях компилятора и нескольких версиях библиотек остается сложной задачей. В данной статье представляется метод нахождения статически связанных библиотек в исполняемых файлах. Предложенный подход строит графы зависимостей программ и графы вызовов функций как для анализируемого исполняемого файла, так и для версий-кандидатов библиотек, после чего сопоставляет соответствующие функции с помощью масштабируемого алгоритма сравнения графов. Метод был реализован и протестирован на исполняемых файлах, содержащих различные версии широко используемых библиотек. Результаты экспериментов показывают, что предложенный подход обеспечивает в среднем 91%-ную точность и 99%-ную полноту идентификации встроенных библиотек, сохраняя при этом практическую масштабируемость.

Ключевые слова: идентификация библиотеки; бинарный анализ; граф зависимостей программы; сопоставление графов; сопоставление функций.

Для цитирования: Арутюнян М.С., Асланян А.К. Идентификация статически связанных библиотек в исполняемых файлах с использованием сопоставления графов зависимостей программы. Труды ИСП РАН, 2026, том 38, вып. 4, часть 1, стр. 89–100. DOI: 10.15514/ISPRAS-2026-38(4)-5.

Identification of Statically Linked Libraries in Executables Using Program Dependence Graph Matching

M.S. Arutunian, ORCID: 0000-0002-2420-7968 <mariam.arutunian@rau.am>
H.K. Aslanyan, ORCID: 0000-0002-7320-4835 <hayk.aslanyan@rau.am>

Russian-Armenian University,
123, Hovsep Emin st., Yerevan, 0051, Armenia.

Abstract. Statically linked libraries simplify software distribution and deployment but complicate binary analysis, as the library code becomes indistinguishable from the program code after compilation. Identifying embedded libraries and their versions is essential for software composition analysis, vulnerability assessment, and license compliance. Despite significant progress in existing solutions, achieving accurate library identification under different compiler optimizations and multiple library versions remains a challenging task. This paper presents a method for identifying statically linked libraries in executable files. The proposed approach constructs program dependence graphs and function call graphs for both the analyzed executable and candidate library versions and then matches corresponding functions using a scalable graph matching algorithm. The method has been implemented and evaluated on executables containing multiple versions of widely used libraries. Experimental results demonstrate that the proposed approach achieves an average of 91% precision and 99% recall in identifying embedded libraries, while maintaining practical scalability.

Keywords: library identification; binary analysis; program dependence graph; graph matching; function matching.

For citation: Arutunian M.S., Aslanyan H.K. Identification of Statically Linked Libraries in Executables Using Program Dependence Graph Matching. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 4, part 1, pp. 89-100 (in Russian). DOI: 10.15514/ISPRAS-2026-38(4)-5.

1. Введение

Разработка программного обеспечения все чаще опирается на многократно используемые библиотеки, предоставляющие реализации распространенных алгоритмов и системных функций. Современные программные проекты обычно включают десятки сторонних библиотек, что снижает трудозатраты на разработку, одновременно повышая надежность и удобство сопровождения [1]. Эти библиотеки могут быть связаны как динамически, так и статически. При статической компоновке необходимый код библиотеки копируется непосредственно в исполняемый файл во время компиляции, создавая самодостаточный бинарный файл без внешних зависимостей от среды выполнения.

Хотя статическая компоновка упрощает развертывание программного обеспечения, она создает ряд проблем для анализа программного обеспечения и безопасности. После включения в исполняемый файл функции библиотеки становятся неотличимыми от кода, специфичного для приложения. Следовательно, потребители программного обеспечения, аналитики безопасности часто не могут определить, какие библиотеки встроены в бинарный файл, или идентифицировать их версии. Это ограничение усложняет анализ состава программного обеспечения, оценку уязвимостей, проверку соответствия лицензиям, анализ вредоносного программного обеспечения (ПО) и реверс-инжиниринг.

Проблема особенно важна с точки зрения безопасности. Уязвимости, обнаруженные в широко используемых библиотеках, могут оставаться в статически связанных исполняемых файлах еще долго после того, как станут доступны исправления. Поскольку обновление разделяемой библиотеки не влияет на статически связанные программы, разработчикам приходится пересобирать и распространять всё приложение целиком. Поэтому идентификация библиотек, включенных в исполняемые файлы, является необходимым условием для определения наличия известных уязвимостей [2, 3] и необходимости замены устаревших версий библиотек.

Существующие подходы к идентификации статических библиотек [4] в основном основаны на сопоставлении сигнатур, сравнении байтовых шаблонов, хешировании инструкций или эвристических алгоритмах, разработанных вручную. Хотя эти методы, как правило, эффективны, они часто чувствительны к оптимизациям компилятора, преобразованиям кода, планированию инструкций, распределению регистров и различиям, вносимым различными версиями компилятора. В результате их точность может значительно снижаться при анализе оптимизированных исполняемых файлов или бинарных файлов, созданных различными инструментальными средствами.

Сравнение бинарных файлов на основе графов продемонстрировало высокую устойчивость при идентификации соответствующих компонентов программы, несмотря на синтаксические различия, возникающие во время компиляции. В нашей предыдущей работе мы предложили метод идентификации изменений между версиями программы на основе сопоставления графов зависимостей программ [5] и анализа графов вызовов функций. Предложенный метод устанавливает соответствия между функциями в двух исполняемых версиях, допуская при этом преобразования, вызванные компилятором, и другие низкоуровневые модификации кода. В данной работе этот подход распространяется на другую область применения: идентификацию статически связанных библиотек. В отличие от сравнения версий программ, идентификация библиотек требует сопоставления исполняемого файла с несколькими кандидатами на версии библиотек и определения того, какие из них присутствуют в анализируемом бинарном файле. Кроме того, код одной и той же библиотеки может изменяться при статической компоновке.

В данной статье представлен метод идентификации статически связанных библиотек в исполняемых файлах на основе алгоритма сопоставления графов. Исполняемые файлы и кандидаты на версии библиотек преобразуются в промежуточное представление, из которого строятся графы зависимостей программ и графы вызовов функций. Алгоритм сопоставления графов устанавливает соответствия между исполняемыми функциями и кандидатами на версии библиотек, в то время как процедура принятия решений на уровне библиотеки определяет наличие отдельных версий библиотек на основе полученных результатов сопоставления.

По сравнению с традиционными подходами, основанными на сигнатурах, предлагаемый метод использует структурные и семантические свойства бинарного кода, а не полагается исключительно на последовательности инструкций или байтовые шаблоны. Следовательно, он значительно более устойчив к оптимизациям компилятора и преобразованиям кода, оставаясь при этом применимым к реальным исполняемым файлам.

Остальная часть статьи организована следующим образом. В разделе 2 рассматриваются существующие подходы к статической идентификации библиотек. В разделе 3 представлен предлагаемый метод идентификации библиотек. В разделе 4 описывается реализация. В разделе 5 проводится экспериментальная оценка предлагаемого подхода. Наконец, в разделе 6 подводятся итоги работы.

2. Обзор существующих работ

Идентификация статически связанных библиотек является важной задачей анализа исполняемых файлов, поскольку позволяет определять состав программного обеспечения, выявлять известные уязвимости, контролировать соблюдение лицензионных требований и выполнять анализ вредоносного программного обеспечения. Существующие методы можно условно разделить на три группы: методы, основанные на сигнатурах, методы, использующие различные признаки функций, и методы, основанные на структурном анализе программ.

Наиболее распространенными являются сигнатурные методы. Классическим представителем данного подхода является технология IDA FLIRT (Fast Library Identification and Recognition

Technology) [4], основанная на сопоставлении сигнатур, построенных по начальным байтам библиотечных функций с учетом перемещаемых адресов. Благодаря высокой скорости работы FLIRT широко применяется на практике, однако его эффективность существенно снижается при использовании различных компиляторов, уровней оптимизации или модификации исходного кода.

Развитием сигнатурного подхода являются методы, использующие более информативные характеристики функций. К ним относится метод KISS [6], выполняющий идентификацию различных версий библиотеки *libc* путем сравнения байтовых последовательностей функций с использованием сжатой базы данных библиотек. Однако данный инструмент ориентирован исключительно на библиотеку *libc*.

Инструмент IdenLib [7] формирует сигнатуры из объектных и библиотечных файлов и выполняет их точное сравнение либо использует индекс Жаккарда для поиска похожих функций. В работе [8] идентификация библиотек выполняется с использованием правил YARA [9], представляющих собой сигнатуры машинного кода с маскированием перемещаемых адресов. Несмотря на простоту реализации и высокую скорость, подобные методы по-прежнему основаны преимущественно на синтаксическом сходстве и остаются чувствительными к изменениям, возникающим в результате оптимизации компилятора.

Более современные методы используют расширенные представления бинарного кода. FunctionSimSearch [10] извлекает из функций набор структурных признаков, включающий подграфы графа потока управления, последовательности инструкций и константы, после чего вычисляет компактные хеш-представления с использованием алгоритма SimHash [11]. Для повышения качества сопоставления веса признаков определяются методами машинного обучения. Аналогичного направления придерживается проект Karta [12], который комбинирует анализ сходства функций с анализом графа вызовов, что позволяет успешно сопоставлять библиотеки даже при изменении отдельных функций.

В последние годы активно развиваются методы, использующие машинное обучение и более богатые семантические представления бинарного кода. Так, инструмент BinCoFer [13] применяет многоэтапную фильтрацию признаков для повышения точности идентификации библиотек при частичном повторном использовании кода. Метод MANTILLA [14] сочетает статический анализ бинарного кода с алгоритмами контролируемого машинного обучения для идентификации библиотек на различных архитектурах. Несмотря на высокую точность, подобные методы требуют формирования репрезентативных обучающих выборок и могут испытывать трудности при анализе ранее неизвестных библиотек или новых конфигураций компиляторов.

Отдельное направление исследований связано с использованием графовых представлений программ. Графы потока управления, графы вызовов функций и графы зависимостей программы широко применяются при поиске бинарных клонов, обнаружении уязвимостей, анализе вредоносного программного обеспечения и сравнении различных версий программ. По сравнению с сигнатурными методами графовые модели отражают не только синтаксическую структуру, но и семантические связи между элементами программы, благодаря чему обладают большей устойчивостью к оптимизациям компилятора и другим преобразованиям бинарного кода.

Несмотря на значительный прогресс в области идентификации статически связанных библиотек, большинство существующих методов основано на сигнатурах, специально разработанных признаках или моделях машинного обучения. Такие подходы могут быть чувствительны к изменениям бинарного кода, вызванным различными компиляторами и уровнями оптимизации, а методы машинного обучения требуют значительных затрат на формирование обучающих данных.

В данной работе предлагается метод идентификации статически связанных библиотек, основанный на совместном использовании графов зависимостей программы и графов

вызовов функций. В отличие от существующих подходов, предлагаемый метод использует структурные и семантические свойства бинарного кода, что позволяет повысить устойчивость идентификации к преобразованиям, выполняемым компилятором. Подробное описание предлагаемого алгоритма приведено в следующем разделе.

3. Предложенный метод

3.1 Обзор

Предложенный метод идентифицирует статически связанные библиотеки путем сравнения исполняемого файла с набором версий библиотек-кандидатов. Общий рабочий процесс показан на рис. 1.

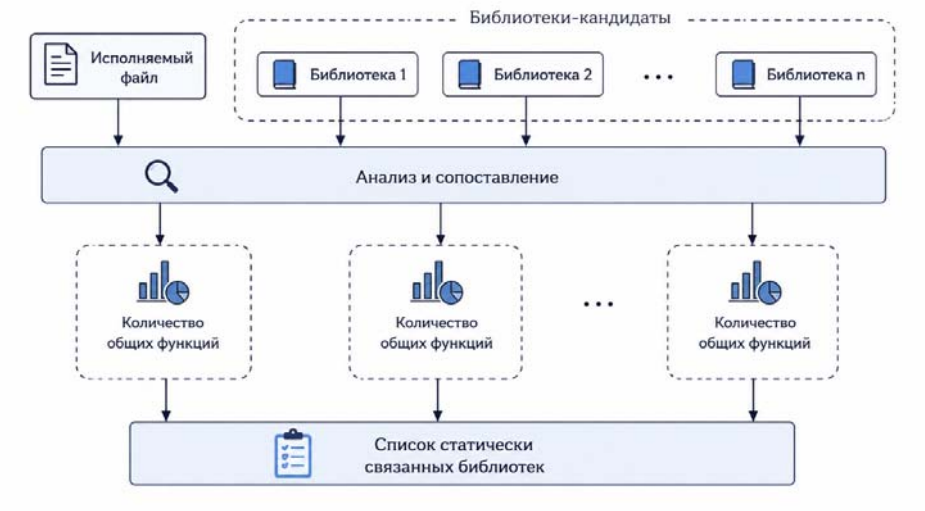

Рис. 1. Схема идентификации статически связанных библиотек.
Fig. 1. Workflow of Statically Linked Library Identification.

В отличие от традиционных подходов, основанных на сигнатурах, предложенный метод рассматривает идентификацию библиотек как задачу бинарного сравнения. Исполняемый файл независимо сравнивается с каждой библиотекой-кандидатом, после чего результаты всех сравнений объединяются для определения наиболее вероятных библиотек и их версий.

Метод включает три основных этапа:

1. предварительную обработку исполняемого файла и библиотек-кандидатов;
2. сопоставление функций;
3. идентификацию библиотек.

Этап сопоставления функций основан на методе сравнения программ, представленном в нашей предыдущей работе [15], тогда как остальные этапы разработаны специально для задачи идентификации статически связанных библиотек.

3.2. Предварительная обработка

Входными данными метода являются анализируемый исполняемый файл и репозиторий, содержащий различные версии библиотек-кандидатов. Для каждой пары «исполняемый файл – библиотека» процедура сравнения выполняется независимо. Общая схема сравнения одной пары представлена на рис. 2.

Рис. 2. Схема сравнения исполняемого файла с одной библиотекой.
Fig. 2. Flowchart of executable file comparison with a single library.

Для анализируемого исполняемого файла сначала выполняется дизассемблирование с последующим преобразованием в промежуточное представление. На его основе для каждой функции строится граф зависимостей программы [5], а для всего файла – граф вызовов функций.

Обработка библиотеки зависит от ее типа. Для динамических библиотек применяется та же процедура, что и для исполняемого файла. Статические библиотеки представляют собой архивы объектных файлов, поэтому каждый объектный файл обрабатывается отдельно. Для содержащихся в нем функций строятся графы зависимостей программы, после чего повторяющиеся представления одинаковых функций удаляются. Затем графы вызовов отдельных объектных файлов объединяются в единый граф вызовов, представляющий библиотеку целиком.

3.3 Сопоставление функций

После завершения предварительной обработки (рис. 2) выполняется сопоставление функций анализируемого исполняемого файла с функциями библиотеки-кандидата.

На данном этапе используется ранее опубликованный нами метод сравнения программ [15]. Метод использует графы зависимостей программы для оценки семантического сходства функций, а граф вызовов функций – для учета их структурного контекста, что позволяет повысить точность сопоставления.

Поскольку алгоритм подробно описан в [15], в настоящей работе он рассматривается как один из компонентов предлагаемого метода. Результатом его работы является множество сопоставленных пар функций с оценками их сходства.

3.4 Идентификация библиотеки

Результаты сопоставления функций используются для определения наличия библиотеки в анализируемом исполняемом файле.

Для повышения точности учитываются только пары функций со 100%-м сходством. Такие совпадения рассматриваются как надежное свидетельство того, что соответствующая библиотека присутствует в исполняемом файле, тогда как частичные совпадения исключаются из дальнейшего анализа.

Процедура сравнения выполняется для всех доступных версий каждой библиотеки-кандидата. Для каждой версии определяется количество полностью совпавших функций, на основе которого вычисляется коэффициент сходства библиотеки. Затем версии ранжируются по значению этого коэффициента, а версии с наибольшими значениями рассматриваются как наиболее вероятные кандидаты.

Таким образом, результатом работы метода является ранжированный список обнаруженных библиотек с указанием наиболее вероятных версий и соответствующих коэффициентов сходства.

4. Экспериментальная оценка

4.1. Экспериментальная установка

Предложенный метод был оценен с использованием нескольких версий программных пакетов Binutils [16] и Coreutils [17], скомпилированных с различными настройками компиляции. Для оценки надежности предложенного подхода исполняемые файлы были сгенерированы с использованием компиляторов Clang [18] и GCC [19] с различными уровнями оптимизации.

В репозитории библиотек-кандидатов содержались следующие статические библиотеки: libc.a, libdl.a, libgcc.a, libgcc_eh.a, libgmp.a, libpthread.a, librt.a, libbfd.a, libctf.a, libctf-nobfd.a, libiberty.a, libopcodes.a, libz.a, libcoreutils.a и libver.a.

Для каждого исполняемого файла предложенный метод идентифицировал встроенные библиотеки путем сравнения исполняемого файла с каждой версией библиотеки-кандидата. Для каждого анализируемого исполняемого файла выполнялось сравнение с каждой библиотекой-кандидатом из репозитория. Алгоритм сопоставления графов вычислял коэффициент сходства между исполняемым файлом и каждой версией библиотеки. Библиотека считалась идентифицированной, если вычисленное значение сходства превышало заданный пользователем порог. Во всех экспериментах использовалось пороговое значение 100%.

Качество идентификации оценивалось с использованием метрик точности, полноты и F1-меры [20, 21]. Кроме того, анализировались значения сходства, вычисляемые алгоритмом сопоставления графов, что позволило оценить надежность идентификации различных библиотек и их версий.

4.2 Результаты

4.2.1. Оценка качества идентификации

Результаты оценки качества идентификации представлены в табл. 1.

Как видно из табл. 1, предложенный метод демонстрирует стабильно высокое качество идентификации независимо от используемого компилятора и уровня оптимизации. Значение полноты составляет 98–100%, что свидетельствует о способности метода обнаруживать практически все статически связанные библиотеки, присутствующие в анализируемых исполняемых файлах.

Значение точности находится в диапазоне 89–91%. Незначительное снижение точности наблюдается для исполняемых файлов, скомпилированных с использованием оптимизации GCC O2, что можно объяснить более агрессивными преобразованиями компилятора, изменяющими структуру функций. Однако влияние этих преобразований на качество идентификации остается незначительным.

Табл. 1. Результаты идентификации статически связанных библиотек.
Table 1. Results of static library identification.

Исполняемый файл	Компилятор	Оптимизация	Точность (%)	Полнота (%)	F1-мера (%)
Binutils	Clang	O0	91	98	94
Binutils	GCC	O0	91	98	94
Binutils	GCC	O2	89	99	94
Coreutils	Clang	O0	91	100	95
Coreutils	Clang	O2	90	100	95

Как показано в табл. 1, значение F1-меры остается практически неизменным и составляет 94-95% для всех исследованных конфигураций компиляции. Это подтверждает устойчивость предложенного метода и его способность сохранять высокий баланс между точностью и полнотой при различных условиях компиляции.

4.2.2. Идентификация библиотек

Для более детального анализа были исследованы значения коэффициента сходства, вычисляемые алгоритмом сопоставления графов для различных библиотек. Полученные результаты представлены в табл. 2.

Как видно из табл. 2, среднее значение коэффициента сходства для корректно идентифицированных библиотек находится в диапазоне 96,68–99,99%. Для большинства библиотек коэффициент превышает 99%, а в ряде случаев достигает 99,99%. Например, для различных версий библиотеки zlib среднее значение сходства составляет около 99,5%, тогда как для библиотеки libtiff оно превышает 98%.

Полученные результаты показывают, что алгоритм сопоставления графов обеспечивает высокую точность сопоставления функций и позволяет корректно идентифицировать библиотеки даже в случаях, когда в исполняемый файл включена лишь часть их функций.

Табл. 2. Степень сходства найденных библиотек.
Table 2. Similarity scores of identified libraries.

Исполняемый файл	Библиотека	Доля идентифицированных функций библиотеки (%)	Среднее сходство (%)
zlibTest	libz.a	39.74	99.50
zlibTest	libz.so.1.2.11	29.95	99.59
zlibTest	libz.so.1	29.95	99.59
wlua54.2	lua 5.4.2	8.54	99.80
tiffTest	libtiff.a	52.15	98.00

Доля идентифицированных функций библиотеки характеризует отношение числа функций библиотеки, успешно идентифицированных алгоритмом, к общему числу функций данной библиотеки. Поскольку при статической компоновке в исполняемый файл включается только необходимые функции, значение данного показателя обычно значительно меньше 100 %.

4.2.3. Идентификация различных версий библиотек

Для оценки способности метода различать различные версии одной и той же библиотеки был проведен дополнительный эксперимент. Исполняемый файл, содержащий статически связанную библиотеку zlib версии 1.2.11, сравнивался с несколькими версиями библиотеки zlib. Результаты представлены в табл. 3.

Как видно из табл. 3, предложенный метод корректно идентифицирует библиотеку zlib среди нескольких ее версий. При этом практически идентичные версии библиотеки, такие как libz.so.1.2.13 и libz.so.1.2.11, демонстрируют практически одинаковые значения коэффициента сходства. Это объясняется тем, что различные версии библиотеки имеют общую кодовую базу и отличаются лишь отдельными изменениями, внесенными в процессе их развития.

Таким образом, вычисляемый коэффициент сходства является надежным критерием идентификации библиотек. Однако различение практически идентичных сборок может потребовать использования дополнительных признаков, например информации о версии библиотеки или других метаданных.

Табл. 3. Идентификация различных версий zlib.
Table 3. Identification of different zlib versions.

Библиотека/Версия	Количество найденных функций	Среднее сходство (%)
libz.so.1.2.11	58	96.66
libz.so.1.2.13	58	95.43
libz.a 1.2.11	52	95.61

4.2.4. Устойчивость к модификациям кода

Для оценки устойчивости предложенного метода были проведены дополнительные эксперименты с искусственно модифицированными исполняемыми файлами. В ходе экспериментов из функций удалялось до 15% инструкций. Полученные результаты представлены в табл. 4.

Табл. 4. Устойчивость к модификациям кода.
Table 4. Robustness to code modifications.

Удаленные инструкции	Среднее сходство (%)
0 %	98.91–99.77
1–4 %	99.63–99.66
5 %	98.22–99.64
6–8 %	99.63–99.66
15 %	99.66–99.77

Как видно из табл. 4, даже после внесения подобных изменений алгоритм успешно идентифицировал все исследуемые функции. При этом среднее значение коэффициента сходства оставалось в диапазоне 98,2–99,8%, а увеличение количества удаляемых инструкций не приводило к заметному снижению качества сопоставления.

Полученные результаты свидетельствуют о высокой устойчивости графового представления программы к небольшим изменениям бинарного кода, возникающим как вследствие оптимизаций компилятора, так и при локальных модификациях функций.

4.2.5. Обсуждение результатов

Проведенные эксперименты показывают, что предложенный метод обеспечивает надежную идентификацию статически связанных библиотек при различных условиях компиляции. Высокие значения полноты свидетельствуют о том, что метод успешно обнаруживает практически все встроенные библиотеки, тогда как значения точности подтверждают низкий уровень ложноположительных срабатываний.

Дополнительный анализ коэффициентов сходства показал, что алгоритм сопоставления графов обеспечивает высокую степень соответствия между исполняемыми файлами и библиотеками-кандидатами. Это позволяет надежно идентифицировать библиотеки даже при наличии изменений, вызванных оптимизациями компилятора, а также в случаях, когда в исполняемый файл включена только часть функций библиотеки.

Эксперименты по сравнению различных версий библиотек показали, что предложенный подход успешно определяет принадлежность исполняемого файла к соответствующей библиотеке. При этом различение практически идентичных сборок остается сложной задачей, поскольку они имеют почти одинаковую структуру графов и, как следствие, близкие значения коэффициента сходства.

Полученные результаты также показывают преимущества графового подхода по сравнению с традиционными сигнатурными методами, эффективность которых существенно зависит от компилятора и уровня оптимизации. Использование структурных и семантических характеристик программы позволяет сохранять высокое качество идентификации даже после существенных преобразований бинарного кода.

5. Заключение

В работе предложен метод идентификации статически связанных библиотек в исполняемых файлах, основанный на сопоставлении графов зависимостей программ и графов вызовов функций. В отличие от сигнатурных подходов, метод использует структурные и семантические свойства бинарного кода, что обеспечивает устойчивость к преобразованиям, выполняемым различными компиляторами и уровнями оптимизации.

Экспериментальная оценка показала, что предложенный метод обеспечивает высокое качество идентификации встроенных библиотек, достигая 89-91% точности, 98-100% полноты и F1-меры 94-95%. Проведенные эксперименты также подтвердили способность метода корректно идентифицировать различные версии библиотек и сохранять устойчивость к небольшим изменениям бинарного кода.

Полученные результаты демонстрируют, что использование графового представления программы является эффективным подходом к идентификации статически связанных библиотек и может применяться при анализе состава программного обеспечения и поиске известных уязвимостей.

В дальнейшем планируется расширить метод для поддержки дополнительных архитектур процессоров и форматов исполняемых файлов, повысить его масштабируемость за счет параллельного сопоставления графов, а также интегрировать систему идентификации библиотек со средствами автоматизированного анализа уязвимостей.

Список литературы / References

[1]. Krueger C.W. Software reuse. ACM Computing Surveys, 1992, vol. 24, no. 2, pp. 131-183. DOI: 10.1145/130844.130856.

[2]. National Institute of Standards and Technology. National Vulnerability Database (NVD). Available at: https://nvd.nist.gov/, accessed 10.03.2026.

[3]. Cybersecurity and Infrastructure Security Agency. Known Exploited Vulnerabilities Catalog. Available at: https://www.cisa.gov/known-exploited-vulnerabilities-catalog, accessed 10.03.2026.

[4]. Hex-Rays. IDA F.L.I.R.T. Technology, 27.05.2015. Available at: https://www.hex-rays.com/products/ida/tech/flirt/index.shtml, accessed 10.03.2026.

[5]. Ferrante J., Ottenstein K.J., Warren J.D. The Program Dependence Graph and Its Use in Optimization. ACM Transactions on Programming Languages and Systems, 1987, vol. 9, no. 3, pp. 319-349. DOI: 10.1145/24039.24041.

[6]. von Tschirschnitz M. Library and Function Identification by Optimized Pattern Matching on Compressed Databases: A Close to Perfect Identification of Known Code Snippets. In Proc. of the 2nd Reversing and Offensive-oriented Trends Symposium (ROOTS), 2018.

[7]. idenLib. Available at: https://github.com/secrary/idenLib, accessed 10.03.2026.

[8]. Akabane S., Okamoto T. Identification of Library Functions Statically Linked to Linux Malware. Procedia Computer Science, 2020, vol. 176, pp. 3436-3445. DOI: 10.1016/j.procs.2020.09.013.

[9]. Alvarez V. YARA Documentation, 2013. Available at: https://yara.readthedocs.io/en/v4.1.0/index.html, accessed 10.03.2026.

[10]. Google Project Zero. FunctionSimSearch. Available at: https://github.com/googleprojectzero/functionsimsearch, accessed 10.03.2026.

[11]. Charikar M.S. Similarity Estimation Techniques from Rounding Algorithms. In Proc. of the 34th Annual ACM Symposium on Theory of Computing, 2002, pp. 380-388. DOI: 10.1145/509907.509965.

[12]. Itkin E. Karta – Matching Open Sources in Binaries. Available at: https://research.checkpoint.com/karta-matching-open-sources-in-binaries/, accessed 10.03.2026.

[13]. Zou Y., Zhang Y., Zhao G., Wu Y., Shen S., Fu C. BinCoFer: Three-stage Purification for Effective C/C++ Binary Third-party Library Detection. Journal of Systems and Software, 2025, vol. 229. DOI: 10.1016/j.jss.2025.112480.

[14]. Carrillo-Mondéjar J., Rodríguez R.J. Identifying Runtime Libraries in Statically Linked Linux Binaries. Future Generation Computer Systems, 2025, vol. 164, article 107602. DOI: 10.1016/j.future.2024.107602.

[15]. Arutunian M., Hovhannisyan H., Malajyan A., Avetisyan K., Aslanyan H. Method for Identifying Changes Between Program Versions. In Proc. of the 35th International Conference on Computer Theory and Applications (ICCTA), Alexandria, Egypt, 2025, pp. 107-112. DOI: 10.1109/ICCTA68914.2025.11520031.

[16]. GNU Project. GNU Binutils. Available at: https://www.gnu.org/software/binutils/, accessed 10.03.2026.

[17]. GNU Project. Coreutils – GNU Core Utilities. Available at: https://www.gnu.org/software/coreutils/, accessed 10.03.2026.

[18]. LLVM Project. Clang: A C Language Family Frontend for LLVM. Available at: https://clang.llvm.org/, accessed 10.03.2026.

[19]. Stallman R.M., GCC Developer Community. Using the GNU Compiler Collection. Boston: GNU Press, 2003.

[20]. Powers D.M.W. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation. Journal of Machine Learning Technologies, 2011, vol. 2, no. 1, pp. 37-63.

[21]. Manning C.D., Raghavan P., Schütze H. Introduction to Information Retrieval. Cambridge: Cambridge University Press, 2008. 482 p.

Информация об авторах / Information about authors

Мариам Сероповна АРУТЮНЯН – кандидат технических наук, исследователь и преподаватель Российско-Армянского университета. Сфера научных интересов: статический анализ программ, безопасность программного обеспечения и оптимизация компиляторов.

Mariam Seropovna ARUTUNIAN – Cand. Sci. (Tech.), researcher at the Center of Advanced Software Technologies and a lecturer at Russian-Armenian University, Armenia. Research interests: static program analysis, software security, and compiler optimization.

Айк Каренович АСЛАНЯН – кандидат физико-математических наук, исследователь в Центре передовых программных технологий и преподаватель Российско-Армянского университета. Его научные интересы включают статический и динамический анализ программ, безопасность программного обеспечения и оптимизацию компиляторов.

Hayk Karenovich ASLANYAN – Cand. Sci. (Phys.-Math.), researcher at the Center of Advanced Software Technologies and a lecturer at Russian-Armenian University, Armenia. His research interests include program static and dynamic analysis, software security, and compiler optimization.