DOI: 10.15514/ISPRAS-2026-38(6)-32



Выбор метрик для ранжирования функций из поверхности атаки

[1] *М.А. Костин, ORCID: 0009-0002-1464-8302 <maksim.kostin@ispras.ru>*
[1,2] *П.М. Довгалюк, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>*
[1] *Н.И. Фурсова, ORCID: 0000-0001-6817-4670 <natalia.fursova@ispras.ru>*
[1] *В.М. Степанов, ORCID: 0000-0003-2141-3333 <vladislav.stepanov@ispras.ru>*

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

[2] *Новгородский государственный университет им. Ярослава Мудрого, Россия, 173003, Великий Новгород, ул. Большая Санкт-Петербургская, д. 41.*

Аннотация. При анализе проекта с большой кодовой базой на предмет уязвимостей не всегда очевидно, откуда следует начать поиск, особенно если проект незнакомый. Чтобы тестирование было эффективным, необходимо выделить потенциально уязвимые функции и модули, обрабатывающие недоверенные или чувствительные данные. Полученный список называется поверхностью атаки. Функции из поверхности атаки могут быть причиной уязвимостей, поэтому им следует уделять особое внимание. Однако не все функции, взаимодействовавшие с помеченными данными, одинаково интересны для исследования, поэтому возникает потребность в ранжировании функций. Существующие работы в основном посвящены напрямую поиску уязвимостей, при этом большая часть из них основывается исключительно на статическом анализе. На данный момент методы, использующие простые метрики исходного кода, могут быть эффективнее более сложных эвристических методов. В данной работе был взят за основу уже существующий набор статических метрик и дополнен новыми метриками, получаемыми в результате динамического анализа. Для того, чтобы определить влияние новых метрик на качество ранжирования был проведён эксперимент на нескольких проектах. Он показал, что новые метрики значительно улучшают качество ранжирования.

Ключевые слова: метрики кода; уязвимости; динамический анализ; анализ помеченных данных.

Для цитирования: Костин М.А., Довгалюк П.М., Фурсова Н.И., Степанов В.М. Выбор метрик для ранжирования функций из поверхности атаки. Труды ИСП РАН, том 38, вып. 6, 2026 г., стр. 187–198. DOI: 10.15514/ISPRAS–2026–38(6)–32.

Благодарности: Исследование выполнено при поддержке Российского научного фонда, проект № 24-11-20022.

Metric selection for ranking functions from attack surface

[1] *M.A. Kostin, ORCID: 0009-0002-1464-8302 <maksim.kostin@ispras.ru>*
[1,2] *P.M. Dovgalyuk, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>*
[1] *N.I. Fursova, ORCID: 0000-0001-6817-4670 <natalia.fursova@ispras.ru>*
[1] *V.M. Stepanov, ORCID: 0000-0003-2141-3333 <vladislav.stepanov@ispras.ru>*

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

[2] *Yaroslav-the-Wise Novgorod State University, 41, B. Sankt- Peterburgskaya st., Veliky Novgorod, 173003, Russia.*

Abstract. When analyzing a large codebase for vulnerabilities, it is often difficult to decide where to start, especially when the project is unfamiliar. To make testing effective, it is necessary to identify potentially vulnerable functions and modules that handle untrusted or sensitive data. The resulting list is called the attack surface. Functions in the attack surface can be a source of vulnerabilities and therefore require special attention. However, not all functions that interact with tainted data are equally interesting for analysis, raising the problem of function ranking. Existing solutions mainly focus on direct vulnerability detection, and most of them relying solely on static analysis. Currently, methods that use simple source code metrics can be more effective than more complex heuristic methods. In this work, we take an existing set of static metrics and extend it with new metrics derived from dynamic analysis. To evaluate the impact of the new metrics, we conducted an experiment on several projects. The results show that the new metrics significantly improve the ranking quality.

Keywords: code metrics; vulnerabilities; dynamic analysis; taint analysis.

For citation: Kostin M.A., Dovgalyuk P.M., Fursova N.I., Stepanov V.M. Metric selection for ranking functions from attack surface. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 6, part 2, 2026, pp. 187-198 (in Russian). DOI: 10.15514/ISPRAS-2026-38(6)-32.

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

1. Введение

При анализе проекта с большой кодовой базой на предмет уязвимостей не всегда очевидно, откуда следует начать поиск, особенно если проект незнакомый. Чтобы тестирование было эффективным, необходимо выделить потенциально уязвимые функции и модули, обрабатывающие недоверенные или чувствительные данные. Полученный список называется поверхностью атаки. Поверхность атаки может быть определена при помощи средств статического и динамического анализа [1-2] или методом ручного аудита.

Функции из поверхности атаки могут быть причиной уязвимостей, поэтому им следует уделять особое внимание. Однако не все функции, взаимодействовавшие с помеченными данными, одинаково интересны для исследования, поэтому возникает потребность в ранжировании функций. Например, в Fuzz-Introspector [3] для функций рассчитывается определённый набор метрик, по которым эксперт самостоятельно ранжирует функции и выбирает новые цели для фаззинга.

Существующие работы преимущественно направлены на непосредственный поиск уязвимостей, а эта задача отличается от ранжирования: если при поиске уязвимостей важно отличать уязвимую функцию от безопасной, то при ранжировании она сохранит свою релевантность даже после устранения дефекта. Подходы, применяемые в упомянутых выше работах, делятся на две категории: методы, основанные на метриках, и эвристические методы. В работе [4] было выяснено, что на данный момент методы, использующие простые метрики исходного кода, могут быть эффективнее более сложных эвристических методов. Это связано с тем, что в отличие от вторых, первые не полагаются на конкретные шаблоны уязвимого кода. Сравнение проводилось в контексте задачи выбора целей для направленного

фаззинга. Поскольку эта задача близка к рассматриваемой в данной статье, было принято решение начать исследование именно с простых числовых метрик.

В настоящее время подавляющее большинство исследований основывается только на статическом анализе, однако данный подход имеет ряд недостатков. Во-первых, статический анализ обычно проводится по всему исходному коду приложения, что при большой кодовой базе приводит к значительному росту вычислительной сложности. Во-вторых, несмотря на то, что статические анализаторы эффективно выявляют определённые классы уязвимостей, количество их срабатываний на проект может исчисляться тысячами. В-третьих, статический анализ не позволяет точно выделить те функции, которые действительно исполняются во время работы программы и обрабатывают внешние данные.

Для решения вышеперечисленных проблем мы предлагаем подход, объединяющий статический и динамический анализ. Наибольший интерес представляет анализ помеченных данных [5]. Динамический анализ потоков помеченных данных заключается в отслеживании операций с помеченными данными во время выполнения программы. Это позволяет проследить путь данных по всем модулям и функциям, поэтому данный метод анализа применяется в инструментах определения поверхности атаки [1].

Таким образом, план работы заключается в следующем: взять за основу существующий набор статических метрик и дополнить его новыми метриками, получаемыми в результате динамического анализа, а затем оценить их вклад в ранжирование.

Работа предполагает наличие исходного кода исследуемых программ и ограничивается языками C/C++.

Дальнейшая структура статьи такова: раздел 2 посвящён формированию набора метрик, раздел 3 – сбору и разметке экспериментальных данных, в разделе 4 приводятся результаты эксперимента и их анализ, а в разделе 5 обсуждаются ограничения исследования.

2. Выбор метрик

За основу был взят набор метрик, используемый во фреймворке Leopard [2], так как он показал наилучший результат в сравнении методов выбора целей для направленного фаззинга [4]. Данные метрики приведены в табл. 1, они делятся на две категории. Метрики сложности разработаны исходя из того, что чем сложнее логика функции, тем сложнее её анализировать и тестировать. Метрики уязвимости, в свою очередь, сформированы из основных причин возникновения уязвимостей.

Что касается новых метрик, то, как было сказано ранее, наибольший интерес представляет анализ помеченных данных. Его результаты напрямую зависят от того, какие данные были помечены. Обычно это недоверенные данные, поступающие из внешних источников: файлы, сетевой трафик и т.д. В исследуемой программе должна присутствовать валидация таких данных, поэтому функции, которые взаимодействуют с помеченными данными, представляют повышенный интерес для исследования. Таким образом, новые метрики должны характеризовать то, как функция взаимодействует с помеченными данными, однако далеко не все уязвимые функции осуществляют это напрямую, что также нужно учесть.

Далее приведены разработанные метрики, их сбор будет осуществляться при помощи инструмента для определения поверхности атаки Natch [1]. Natch (Network Application Tainting Can Help) – это инструмент, использующий динамический анализ, в первую очередь анализ потоков помеченных данных, для определения поверхности атаки отдельных приложений и сложных систем.

Минимальное/Среднее/Максимальное расстояние до доступа к помеченным данным. В Natch для каждого события доступа к помеченным данным сохраняется стек вызовов. Именно по нему можно рассчитывать расстояние – количество вызовов от текущей функции до помеченной. Гипотеза состоит в том, что чем ближе функция к точке доступа к

помеченным данным, тем сильнее она от них зависит. При расчёте учитываются только текущая функция и вызываемые функции, расположенные далее по стеку вызовов.

Количество путей помеченных данных – метрика, равная количеству достижимых помеченных функций из анализируемой функции. Данная метрика дополняет метрики расстояния и совместно с ними характеризует зависимость функции от помеченных данных. Так, большое число путей при большом среднем расстоянии указывает на слабую зависимость. Это позволяет отсеять функции, подобные main, которые редко представляют интерес.

Табл. 1. Набор статических метрик, используемых в Leopard.
Table 1. Static metrics used by Leopard.

Категория	Метрика
Сложность	Цикломатическая сложность
	Количество циклов
	Количество вложенных циклов
	Максимальный уровень вложенности циклов
Уязвимость	Количество параметров
	Количество аргументов вызываемых функций
	Количество арифметических операций с указателями
	Количество переменных, используемых в арифметике указателей
	Макс. количество арифметических операций с указателями на переменную
	Количество вложенных управляющих структур
	Максимальный уровень вложенности управляющих структур
	Максимальное количество управляющих структур, зависимых по контролю
	Максимальное количество управляющих структур, зависимых по данным
	Количество управляющих структур "if" без "else"
	Количество переменных в условиях управляющих структур

На рис. 1 представлен пример расчёта данных метрик.

Количество переменных в помеченных выражениях и **максимальное количество помеченных выражений на переменную** характеризуют интенсивность работы с помеченными данными. Чем больше переменных и число операций на одну переменную, тем проще допустить ошибку в обработке помеченных данных.

Максимальное количество циклов на помеченное выражение. Если обработка помеченных данных проводится внутри цикла, то это также говорит об интенсивности работы с помеченными данными. И, чем выше максимальная вложенность, тем проще допустить ошибку в такой функции.

Количество помеченных блоков, зависимых по контролю, и **максимальное количество зависимостей по контролю помеченного блока** характеризуют путь до обработки помеченных данных в границах функции. Чем больше количество зависимостей, тем сложнее валидация помеченных данных, и тем легче допустить в ней ошибку.

Количество помеченных условий в ветвлениях. Чем больше помеченных условий в ветвлениях, тем сложнее поток управления функции, поскольку он регулируется недоверенными данными.

Также полезной может быть информация о покрытии. **Доля покрытых базовых блоков** определяет долю выполненного кода функции. Например, на рис. 2 приведена полностью покрытая функция с простой логикой. Такие функции почти не представляют интереса, тогда как функции с частично покрытой, но сложной логикой, напротив, требуют большего внимания.

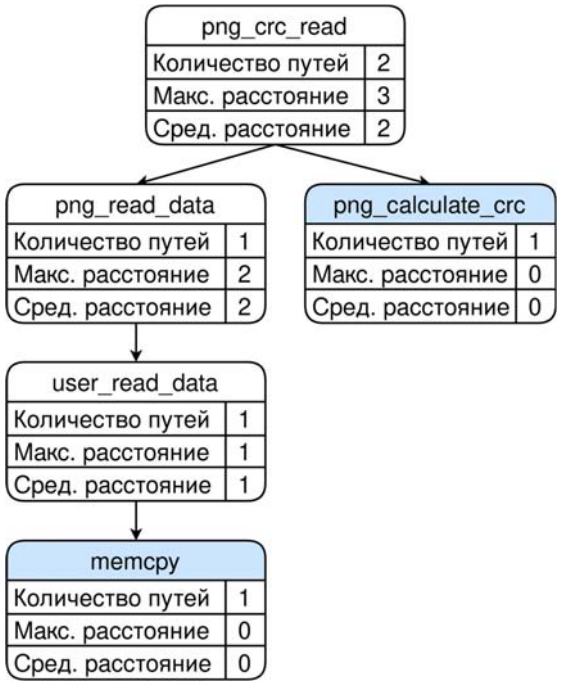


Рис. 1. Пример расчёта количества путей помеченных данных и метрик расстояния.
Fig. 1. Example of calculating number of taint paths and distance metrics.

```
void MeshGeometry::ReadLayer(const Scope& layer)
{
    const ElementCollection& LayerElement = layer.GetCollection("LayerElement");
    for (ElementMap::const_iterator eit = LayerElement.first; eit != LayerElement.second; ++eit) {
        const Scope& elayer = GetRequiredScope(*(*eit).second);

        ReadLayerElement(elayer);
    }
}
```

Рис. 2. Полностью покрытая функция с простой логикой.
Fig. 2. Function with full code coverage and simple logic.

Количество if-ветвлений с непокрытым телом. Подобные конструкции зачастую представляют собой проверки на крайние случаи, так как их тело редко исполняется. При этом многие уязвимости возникают именно из-за отсутствия таких проверок (рис. 3).

3. Экспериментальные данные

Чтобы оценить влияние новых метрик на качество ранжирования, мы сформировали выборку уязвимых функций из 9 проектов (табл. 2), опираясь на набор тестов для фаззеров Magma [6]. Эти проекты значительно отличаются по функциональности и формату входных данных. К каждой уязвимости прилагается исправление, поэтому точно известно, какая функция является её причиной, что значительно упрощает разметку.

Поскольку применяется динамический анализ, то также потребовалось подобрать программы и входные данные так, чтобы уязвимые функции не только вызывались, но и оказывались в поверхности атаки.


```
    ut32 beg = obj->page_tbl_idx - 1, end = beg + obj->page_tbl_entries;
    for (ut32 pi = beg; pi != end; pi++) {
+     if (pi >= h->mpages) {
+       RZ_LOG_ERROR("LE: object #%u page table entry index %u is out "
+              "of range.\n",
+         oi + 1, pi + 1);
+       goto fail_cleanup;
+     }
      LE_page *page = &bin->le_pages[pi];
```

Рис. 3. Исправление уязвимости добавлением проверки на крайний случай.
Fig. 3. Vulnerability fix by adding the edge case check.

Табл. 2. Количество уязвимых функций в экспериментальной выборке по проектам.
Table 2. Number of vulnerable functions in dataset per project.

Проект	# функций	Проект	# функций	Проект	# функций
libpng	6	libxml2	9	php	7
libsndfile	10	lua	4	poppler	10
libtiff	5	openssl	8	sqlite3	11
Итого	70				

Далее необходимо определить способ назначения оценок функциям. Оценки могут быть дискретными (0 или 1) или непрерывными (принимать значения на отрезке [0; 1]). В первом случае функции делятся на две категории: релевантные и нерелевантные. Однако при таком подходе разметка возможна лишь для исходного набора уязвимых функций и заведомо нерелевантных простых функций, так как об остальных информация отсутствует. Во втором случае оставшимся функциям могут быть назначены промежуточные оценки, что позволит увеличить выборку и её репрезентативность.

В итоге разметка производилась следующим образом: заведомо нерелевантным функциям (простым вспомогательным функциям, функциям-обёрткам и т. д.) назначались нули, а уязвимым функциям – единицы. Поскольку об остальных функциях информация отсутствует, для них оценка вычислялась на основе расстояния до уязвимых функций в стеке вызовов (рис. 4), которое определялось аналогично расстоянию до доступа к помеченным данным. С увеличением расстояния вес метки уменьшался путём умножения на малый ненулевой коэффициент. Эмпирическим путём было выбрано значение 0,6: при его увеличении функции, подобные main, получали слишком высокие оценки, что уравнивало их с другими более интересными функциями, а при уменьшении, напротив, метка слишком быстро угасала, создавая большой разрыв между известными уязвимыми функциями и всеми остальными.

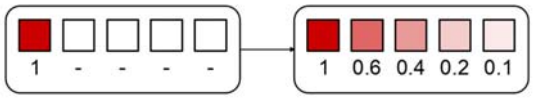


Рис. 4. Распространение метки.
Fig. 4. Label propagation.

Применение распространения меток обосновывается следующими соображениями. Во-первых, если оставшимся функциям назначать оценки, например, на основе истории исправлений, то данные будут содержать много шума, так как многие уязвимости могут быть ещё не найдены. Во-вторых, если была найдена уязвимая функция, то также полезно исследовать и зависимые от неё функции (то есть вызывающие функции), у которых также

будет высокая оценка. В-третьих, при вызове функции с высокой оценкой, скорее всего, будет вызвана и уязвимая функция или даже несколько.

Итого было размечено 670 функций.

4. Результаты

Для оценки вклада новых метрик была проведена кросс-валидация на двух разных моделях машинного обучения по принципу *Leave One Group Out (LOGO)*: на каждом шаге модель обучалась на 8 проектах и тестировалась на оставшемся одном.

Мерой качества выступила метрика *nDCG* – популярная метрика ранжирования, которая учитывает как релевантность отдельных записей, так и их порядок. Она принимает значения от 0 до 1, и чем ближе к 1, тем лучше.

Случайный лес. Результаты представлены на рис. 5. Среднее улучшение составило 8,8%, при этом улучшение было получено на каждом из проектов.

Градиентный бустинг. Результаты представлены на рис. 6. Среднее улучшение составило 9,2%, при этом улучшение было получено почти на каждом из проектов, за исключением libpng и php, где были получены результаты, близкие к исходным, что также приемлемо.

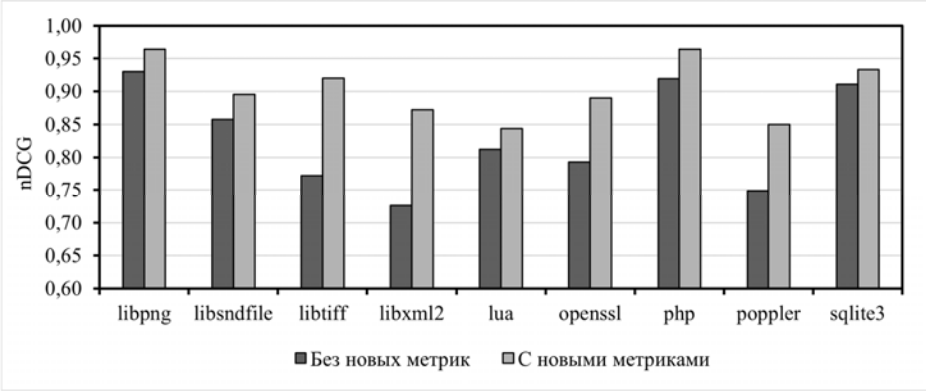


Рис. 5. Оценка качества ранжирования (Случайный Лес).
Fig. 5. Evaluation of ranking quality (Random Forest).

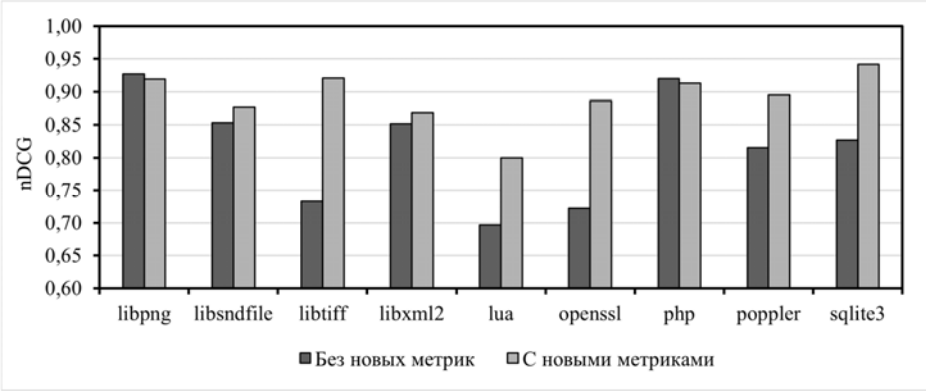


Рис. 6. Оценка качества ранжирования (Градиентный Бустинг).
Fig. 6. Evaluation of ranking quality (Gradient Boosting).

При анализе большого проекта удобно рассматривать лишь небольшое количество наиболее релевантных функций, поэтому мы также оценили качество ранжирования по первым *k* отобранным функциям. Результаты представлены на рис. 7. Здесь тоже наблюдается повсеместное улучшение.

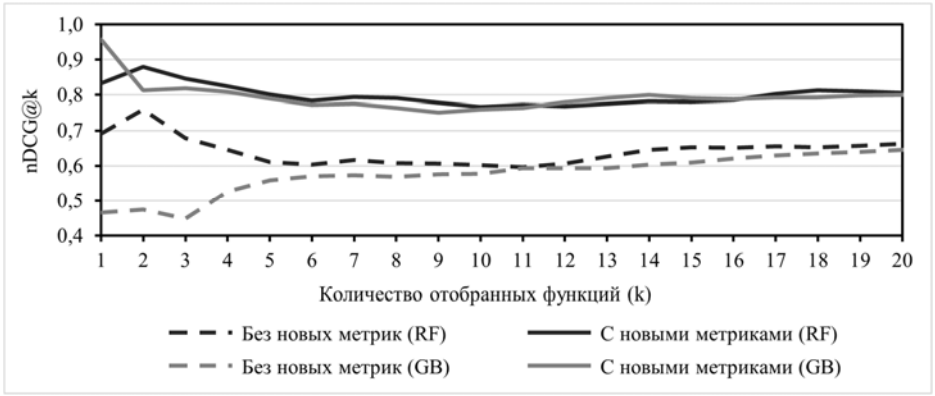


Рис. 7. Оценка качества ранжирования по первым k функциям.
Fig. 7. Evaluation of ranking quality by first k functions.

Важность признаков для каждой модели приведена в табл. 3. Основной вклад вносит группа метрик расстояния до доступа к помеченным данным. Далее идут цикломатическая сложность и группа метрик, характеризующая операции над указателями. Следует отметить, что метрики покрытия и помеченного кода почти не повлияли на результат. Тем не менее, принимать окончательное решение об удалении тех или иных признаков преждевременно, так как объём данных невелик, и некоторые признаки могут оказаться недооценёнными.

Табл. 3. Важность признаков обученных моделей.
Table 3. Feature importances of trained models.

Метрики	Случайный лес	Градиентный бустинг
Расстояние до метки	0,506	0,389
Указатели	0,189	0,187
Циклом-ая сложность	0,188	0,155
Помеченный код	0,027	0,026
Покрытие	0,001	0,009

Мы также рассмотрели случаи, когда уязвимые функции получали слишком низкие оценки. Такие оценки в основном получали относительно простые функции (рис. 8) и функции с большим расстоянием до доступа к помеченным данным (рис. 9). Возможно, с увеличением числа функций в выборке другие метрики станут более значимыми, что смягчит данную проблему.

Помимо этого, был проведён корреляционный анализ. Результаты показывают, что количество параметров функции слабо коррелирует с целевой переменной, хотя эта метрика должна достаточно хорошо отражать запутанность кода. Поскольку основной вклад вносят метрики расстояния, характеризующие не столько код функции, сколько её путь до обработки помеченных данных, то при дальнейшем исследовании полезно будет подсчитывать параметры функций таким образом, чтобы учитывался этот путь, а также, возможно, сформировать новые аналогичные метрики.

```
static inline void
d2ulaw_array (const double *ptr, int count, unsigned char *buffer, double normfact)
{ while (--count >= 0)
-   { if (ptr [count] >= 0)
+   { if (!isfinite (ptr [count]))
+       buffer [count] = 0 ;
+     else if (ptr [count] >= 0)
        buffer [count] = ulaw_encode [lrint (normfact * ptr [count])] ;
      else
        buffer [count] = 0x7F & ulaw_encode [- lrint (normfact * ptr [count])] ;
    } ;
} /* d2ulaw_array */
```

Релевантность	**0,15**
Цикломатическая сложность	3

Рис. 8. Недооценённая функция с низкой цикломатической сложностью.
Fig. 8. Underestimated function with low cyclomatic complexity.

```
@@ -3437,6 +3437,10 @@ static int getNodeSize(
    rc = getIntFromStmt(db, zSql, &pRtree->iNodeSize);
    if( rc!=SQLITE_OK ){
      *pzErr = sqlite3_mprintf("%s", sqlite3_errmsg(db));
+   }else if( pRtree->iNodeSize<(512-64) ){
+     rc = SQLITE_CORRUPT;
+     *pzErr = sqlite3_mprintf("undersize RTree blobs in \"%q_node\"",
+                              pRtree->zName);
    }
  }
```

Релевантность	**0,08**
Среднее расстояние до метки	9,48
Максимальное расстояние до метки	18

Рис. 9. Недооценённая функция с большим расстоянием до доступа к помеченным данным.
Fig. 9. Underestimated function with a long distance to taint data access.

5. Ограничения

Несмотря на то, что методы, основанные на метриках, не полагаются на конкретные шаблоны уязвимого кода, с их помощью трудно выявлять функции, склонные к логическим ошибкам. И это не только сужает область их применения, но и усложняет обучение модели, так как метрики слабо подходят для такого рода ошибок.

То, что значительная часть признаков получается в результате динамического анализа, открывает возможность для увеличения выборки экспериментальных данных. Одну и ту же программу можно запускать с разными входными данными и получать разные значения входных признаков для одних и тех же функций, что повышает репрезентативность выборки. Однако оценки функций в разных контекстах при этом могут значительно отличаться, поэтому репрезентативность выборки сильно зависит от разнообразия входных данных.

Наконец, ранжирование лишь определяет приоритет тестирования функций, то есть высокая оценка не означает, что функция содержит уязвимость, а лишь указывает на необходимость тщательного исследования или тестирования. Таким образом, исключительно ранжирования недостаточно для поиска уязвимостей, и оно должно применяться как вспомогательный инструмент.

6. Заключение

В статье предложен набор метрик для ранжирования функций из поверхности атаки. За основу был взят существующий набор статических метрик и расширен новыми метриками, получаемыми в результате динамического анализа. Для оценки вклада новых метрик в ранжирование было выбрано и размечено 670 функций из 9 проектов, затем проведена кросс-валидация на двух моделях машинного обучения. Эксперимент показал, что новые метрики значительно улучшают качество ранжирования и являются определяющими для обеих моделей.

В дальнейшем планируется расширение выборки и её балансировка. На данный момент она включает функции из 9 проектов, различающихся по размеру и стилю кода. Также будет полезно разделить функции на категории и провести по ним балансировку, чтобы ни одна из них не имела преимущества.

Наиболее полезными признаками оказались метрики расстояния, характеризующие не столько код функции, сколько её путь до обработки помеченных данных. Возможно, будет полезно добавить метрики, такие как кумулятивная цикломатическая сложность.

Список литературы / References

[1]. Довгалюк П.М., Климушенкова М.А., Фурсова Н.И., Степанов В.М., Васильев И.А., Иванов А.А., Иванов А.В., Бакулин М.Г., Егоров Д.И. Natch: Определение поверхности атаки программ с помощью отслеживания помеченных данных и интроспекции виртуальных машин. Труды ИСП РАН, 2022, том 34, вып. 5, стр. 89-110. DOI: 10.15514/ISPRAS-2022-34(5)-6.

[2]. Du X., Chen B., Li Y., Guo J., Zhou Y., Liu Y., Jiang Y. LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment through Program Metrics. Proceedings of the 41st International Conference on Software Engineering, 2019, pp. 60-71.

[3]. Fuzzing Introspection of OSS-Fuzz projects. Доступно по ссылке: https://introspector.oss-fuzz.com, дата обращения 15.04.2026.

[4]. Weissberg F., Möller J., Ganz T., Imgrund E., Pirch L., Seidel L., Schloegel M., Eisenhofer T., Rieck K. SoK: Where to Fuzz? Assessing Target Selection Methods in Directed Fuzzing. Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, 2024, pp. 1539-1553.

[5]. Schwartz E.J., Avgerinos T., Brumley D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In Proc. of the IEEE Symposium on Security and Privacy, 2010, pp. 317-331.

[6]. Magma: A Ground-Truth Fuzzing Benchmark. Доступно по ссылке: https://hexhive.epfl.ch/magma/, дата обращения 15.04.2026.

Информация об авторах / Information about authors

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

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

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

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

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

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

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

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