DOI: 10.15514/ISPRAS-2025-37(4)-30

Sydr-Fuzz: непрерывный гибридный фаззинг и динамический анализ для жизненного цикла безопасной разработки

А.В. Вишняков, ORCID: 0000-0003-1819-220X <pmvishnya@gmail.com>
[1] *Д.О. Куц, ORCID: 0000-0002-0060-8062 <kutz@ispras.ru>*
[1,2] *В.И. Логунова, ORCID: 0000-0002-3877-1906 <vlada@ispras.ru>*
[1,3] *Д.А. Парыгина, ORCID: 0000-0002-4029-0853 <pa_darochek@ispras.ru>*
И.А. Кобрин, ORCID: 0000-0002-6035-0577 <ilayko3110@gmail.com>
Г.А. Савидов, ORCID: 0000-0000-0000-0000 <gsavidov@gmail.com>
А.Н. Федотов, ORCID: 0000-0002-8838-471X <splashgitar@gmail.com>

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

[2] *Московский физико-технический институт, Россия, 141701, Московская область, г. Долгопрудный, Институтский переулок, д. 9.*

[3] *Московский государственный университет имени М.В. Ломоносова, Россия, 119991, Москва, Ленинские горы, д. 1.*

Аннотация. Сегодня автоматизированные фреймворки динамического анализа для непрерывного тестирования востребованы как никогда – они обеспечивают безопасность программного обеспечения и соответствие требованиям Security Development Lifecycle (SDL). Эффективность поиска уязвимостей с помощью современных гибридных методов фаззинга превосходит традиционный фаззинг, основанный на покрытии кода. Мы предлагаем улучшенный конвейер динамического анализа для повышения результативности автоматизированного поиска ошибок, основанный на гибридном фаззинге. Его реализация – инструмент Sydr-Fuzz, в котором наш инструмент символьного выполнения Sydr интегрирован с libFuzzer и AFL++. В Sydr-Fuzz также входят проверка предикатов безопасности, инструмент triage для крашей Casr, а также утилиты минимизации корпуса и сбора покрытия. Бенчмаркинг показал, что Sydr-Fuzz превосходит фаззеры, ориентированные на покрытие, и сопоставим с современными гибридными фаззерами. В рамках проекта OSS-Sydr-Fuzz мы обнаружили 85 новых уязвимостей в реальных программах. Кроме того, мы открыли исходный код Casr для сообщества.

Ключевые слова: динамический анализ; гибридный фаззинг; непрерывный фаззинг; сортировка аварийных завершений; динамическая символьная интерпретация; ДСИ; поиск ошибок; цикл разработки безопасного кода; РБПО; информационная безопасность.

Для цитирования: Вишняков А.В., Куц Д.О., Логунова В.И., Парыгина Д.А., Кобрин И.А., Савидов Г.А., Федотов А.Н. Sydr-Fuzz: непрерывный гибридный фаззинг и динамический анализ для жизненного цикла безопасной разработки. Труды ИСП РАН, том 37, вып. 4, часть 2, 2025 г., стр. 251–270. DOI: 10.15514/ISPRAS–2025–37(4)–30.

Vishnyakov A.V., Kuts D.O., Logunova V.I., Parygina D.A., Kobrin E.A., Savidov G.A., Fedotov A.N. Sydr-Fuzz: Continuous Hybrid Fuzzing and Dynamic Analysis for Security Development Lifecycle. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 4, part 2, 2025. pp. 251-270.

Sydr-Fuzz: Continuous Hybrid Fuzzing and Dynamic Analysis for Security Development Lifecycle

A.V. Vishnyakov, ORCID: 0000-0003-1819-220X <pmvishnya@gmail.com>
[1] *D.O. Kuts, ORCID: 0000-0002-0060-8062 <kutz@ispras.ru>*
[1,2] *V.I. Logunova, ORCID: 0000-0002-3877-1906 <vlada@ispras.ru>*
[1,3] *D.A. Parygina, ORCID: 0000-0002-4029-0853 <pa_darochek@ispras.ru>*
E.A. Kobrin, ORCID: 0000-0002-6035-0577 <ilayko3110@gmail.com>
G.A. Savidov, ORCID: 0000-0000-0000-0000 <gsavidov@gmail.com>
A.N. Fedotov, ORCID: 0000-0002-8838-471X <splashgitar@gmail.com>

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*
[2] *Moscow Institute of Physics and Technology, Institutskiy Pereulok, 9, Dolgoprudny, Moscow Oblast, 141701, Russia.*
[3] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia.*

Abstract. Nowadays automated dynamic analysis frameworks for continuous testing are in high demand to ensure software safety and satisfy the security development lifecycle (SDL) requirements. The security bug hunting efficiency of cutting-edge hybrid fuzzing techniques outperforms widely utilized coverage-guided fuzzing. We propose an enhanced dynamic analysis pipeline to leverage productivity of automated bug detection based on hybrid fuzzing. We implement the proposed pipeline in the continuous fuzzing toolset Sydr-Fuzz which is powered by hybrid fuzzing orchestrator, integrating our DSE tool Sydr with libFuzzer and AFL++. Sydr-Fuzz also incorporates security predicate checkers, crash triaging tool Casr, and utilities for corpus minimization and coverage gathering. The benchmarking of our hybrid fuzzer against alternative state-of-the-art solutions demonstrates its superiority over coverage-guided fuzzers while remaining on the same level with advanced hybrid fuzzers. Furthermore, we approve the relevance of our approach by discovering 85 new real-world software flaws within the OSS-Sydr-Fuzz project. Finally, we open Casr source code to the community to facilitate examination of the existing crashes.

Keywords: dynamic analysis; hybrid fuzzing; continuous fuzzing; crash triage; dynamic symbolic execution; DSE; error detection; security development lifecycle; SDL; computer security.

For citation: Vishnyakov A.V., Kuts D.O., Logunova V.I., Parygina D.A., Kobrin E.A., Savidov G.A., Fedotov A.N. Sydr-Fuzz: Continuous Hybrid Fuzzing and Dynamic Analysis for Security Development Lifecycle. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 2, 2025, pp. 251-270 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-30.

1. Введение

Современное промышленное программное обеспечение (ПО) становится всё сложнее и охватывает практически все сферы жизни – от незначительных до критически важных. Введение процесса разработки безопасного ПО (РБПО) [1-3] стало естественной практикой для большинства компаний-разработчиков. Одним из широко применяемых методов анализа является гибридный фаззинг [4-12], сочетающий фаззинг и динамическое символьное выполнение (ДСИ). Фаззинг быстро расширяет покрытие кода, преодолевая простые ограничения, но плохо исследует сложные участки. ДСИ, наоборот, хорошо справляется с нетривиальными ветвлениями, но работает медленнее. Совместный запуск фаззинга и ДСИ позволяет достичь лучших результатов, чем фаззинг, основанный только на покрытии [13].

Мы разработали новый инструмент гибридного фаззинга, основанный на нашем символьном исполнителе Sydr [14], интегрировав его с AFL++ [15] и libFuzzer [16]. Такой подход позволяет повысить эффективность анализа по сравнению с существующими решениями.

В современном быстроразвивающемся мире важно не только находить ошибки, но и своевременно их исправлять. Интеграция различных инструментов в единый комплекс позволяет значительно повысить продуктивность. На этой основе мы создали набор инструментов Sydr-Fuzz, объединяющий гибридный фаззинг, минимизацию корпуса, проверку предикатов безопасности, сортировку аварийных завершений и сбор покрытия. Sydr-fuzz реализует удобный и продуктивный конвейер динамического анализа. Последовательный запуск этапов конвейера sydr-fuzz позволяет максимально эффективно использовать возможности динамического анализа на основе гибридного фаззинга. Гибридный фаззинг выполняется на начальном этапе конвейера. Его результатом становится корпус тестовых входных данных, которые обеспечивают новое покрытие кода и потенциально могут привести к обнаружению ранее неизвестных ошибок. Так как корпус может содержать множество входных данных, открывающих одно и то же покрытие или ошибки, полезно удалить избыточные входные данные из дальнейшего анализа. Минимизация корпуса стремится удалить как можно больше тестов, сохранив при этом то же покрытие кода, чтобы оставить лишь наиболее ценные входные данные.

После первых двух шагов мы получаем корпус адекватного размера, содержащий набор тестов, который можно затем обработать разными способами. Sydr-fuzz предлагает три стратегии для извлечения информации из результатов гибридного фаззинга. Первая – обнаружение ошибок с помощью символьных предикатов безопасности. Эта техника основана на динамическом символьном исполнении с дополнительными ограничениями, нацеленными на выявление четырёх видов ошибок: разыменование нулевого указателя, деление на ноль, переполнение целого числа и выход за границы массива. Некоторые из этих видов ошибок связаны с несколькими категориями CWE. Вторая стратегия – сбор покрытия кода, обеспечиваемого корпусом. И третья включает сортировку аварийных завершений с использованием инструментов Casr [17]. Эти инструменты позволяют генерировать отчёты о сбоях, вызванных входными данными из корпуса, затем выполнять дедупликацию и кластеризацию этих отчётов. На выходе результаты представляются в виде кластеров, соответствующих потенциально разным багам, дополненным соответствующими тестовыми случаями.

Данная работа содержит следующие основные результаты:

- Мы разработали инфраструктуру непрерывного гибридного фаззинга для эффективного динамического анализа программ. Мы объединили этапы гибридного фаззинга [4–12], минимизации корпуса, сбора покрытия и классификации сбоев [17], применения символьных предикатов безопасности [18] в инструменте sydr-fuzz и предлагаем конвейер динамического анализа для максимизации его эффективности. Мы представляем репозиторий OSS-Sydr-Fuzz [19], созданный по образцу OSS-Fuzz [20] и адаптированный для гибридного фаззинга с использованием sydr-fuzz.
- Мы разработали новый гибридный фаззер на основе инструмента ДСИ Sydr [14]. Мы интегрировали Sydr с AFL++ [15] и libFuzzer [16] (что является первой интеграцией между инструментом ДСИ и libFuzzer). Мы объединили возможности нашего динамического символьного исполнителя Sydr с современными фаззерами. Мы реализовали в Sydr ряд полезных функций, которые способствуют достижению лучших результатов оценки. Мы внедрили анализ символьных указателей в процессе гибридного фаззинга [21].
- Мы провели оценку sydr-fuzz на платформе Google FuzzBench [22] в сравнении с современными фаззерами с покрытием и гибридными фаззерами. Мы демонстрируем, что sydr-fuzz превосходит фаззеры с покрытием и оказывается сопоставим с гибридными фаззерами, получая значительную пользу от своего символьного исполнителя Sydr [23].

- Мы открыли исходный код инструмента Casr [17] для кластеризации отчетов о сбоях, дедупликации и оценки серьезности [24].

В основном данная статья является переводом доклада Vishnyakov, A. Sydr-Fuzz: Continuous Hybrid Fuzzing and Dynamic Analysis for Security Development Lifecycle [Текст] / A. Vishnyakov, D. Kuts, V. Logunova, D. Parygina, E. Kobrin, G. Savidov, A. Fedotov // 2022 Ivannikov ISPRAS Open Conference (ISPRAS). IEEE, 2022. — С. 111–123. При переводе были устранены некоторые неточности.

2. Обзор существующих решений

2.1 Гибридный фаззинг

Гибридный фаззинг является современной методикой обнаружения программных ошибок. Его сила заключается в легковесном и быстром фаззинге и точном символьном выполнении. Фаззинг помогает быстро обнаруживать новые пути, в то время как ДСИ отвечает за систематическое исследование кода.

2.1.1 QSYM

Гибридный фаззер QSYM [7] стал одним из первых инструментов, показавших эффективность гибридного фаззинга. Yun и др. реализовали гибридный фаззер, достаточно легковесный, чтобы позволить ДСИ и фаззеру работать параллельно (в то время как в Driller [6] ДСИ запускается на короткий период для помощи фаззеру, когда тот перестает открывать новое покрытие). QSYM использует динамическую бинарную трансляцию для сокращения количества символьных эмулируемых инструкций и отказывается от использования промежуточного представления, чтобы исключить дополнительные накладные расходы. Авторы предложили две техники оптимизации для повышения эффективности анализа. Во-первых, отсечение базовых блоков позволяет пропускать эмуляцию некоторых блоков, если они выполнялись слишком часто. Во-вторых, оптимистичное решение предполагает решение только целевого ограничения, если весь предикат пути невыполним. Эти техники не обеспечивают полный анализ, но помогают инвертировать больше символических ветвей за единицу времени. Другая полезная техника, предложенная QSYM, – это кэш для инвертированных ветвей с двумя режимами кэширования. Статический режим означает, что каждая символическая ветвь инвертируется только один раз, в то время как контекстный режим позволяет учитывать глубину и набор ранее выполненных символических ветвей. QSYM также предлагает обработку символических адресов [4] путем их простого фаззинга. Он ищет минимальные и максимальные значения адресов с помощью SMT-решателя и генерирует новые входные данные при каждом вызове. Что касается гибридного фаззинга, QSYM запускает фаззер параллельно с символьным исполнителем и позволяет им обмениваться новыми тестовыми входными данными. ДСИ предпочитает такие входные файлы, которые недавно обнаружили новое покрытие и одновременно имеют меньший размер.

2.1.2 SymCC, SymQEMU

Авторы SymCC [8] разработали новую технику символьного выполнения на основе компиляции. Инструментирующий код для символьного выполнения вставляется в код целевого приложения, поэтому скомпилированный бинарный файл может быть выполнен без переключения между выполнением программного кода и интерпретатором. Код для обновления символьного состояния и обработки символьных вычислений генерируется только один раз во время компиляции. Этот метод также выигрывает от возможности применять все оптимизации LLVM IR и CPU. SymCC включает символьный бэкенд в

библиотеки, используемые целевой программой. Проверки конкретности позволяют значительно сократить количество символьных обрабатываемых вычислений. Как продолжение SymCC, авторы представили SymQEMU [9], который предложил применять символьное выполнение на основе компиляции к бинарным файлам при отсутствии исходного кода. Такой метод сочетает высокую скорость анализа с архитектурной независимостью. SymQEMU был построен на основе QEMU путем расширения его компонента TCG так, что код символьной обработки вставляется в IR TCG ops перед компиляцией в машинный код хоста. Символьный анализ останавливается на границе системного вызова, что позволяет достичь лучшей производительности. SymQEMU также реализует сборщик мусора для символьных выражений для эффективного управления памятью. Как SymCC, так и SymQEMU поддерживают интеграцию с AFL++ [15] способом, аналогичным QSYM.

2.1.3 FUZZOLIC

Borzacchiello и др. [10] предложили подход, схожий с SymQEMU. Они значительно повысили эффективность символьного выполнения благодаря быстрому анализу, построенному на основе QEMU, и новому решателю Fuzzy-Sat [25], который использует фаззинг для решения запросов. FUZZOLIC выполняет JIT-компиляцию для добавления инструментирующего кода во время выполнения. Он компилирует каждый базовый блок только один раз и выигрывает от вставки инструментирующего кода в код целевой программы. FUZZOLIC состоит из трассировщика и решателя, которые работают в отдельных параллельных процессах, и это первое важное отличие от SymQEMU. Трассировщик выполняет программу и генерирует символьные выражения, которые отправляются решателю, ответственному за решение запросов. Процессы общаются через разделяемую память. Также, в отличие от SymQEMU, FUZZOLIC внедряет символьную инструментацию в целевой код после того, как QEMU сгенерировал TCG базового блока. Это позволяет воспользоваться преимуществами оптимизаций TCG на уровне всего базового блока. Последнее существенное отличие заключается в том, что SymQEMU обрабатывает только архитектурно-независимые TCG-инструкции, в то время как FUZZOLIC может вставлять символьную инструментацию для большого количества архитектурно-зависимых TCG-нативных хелперов для платформ x86 и x86-64. Схема гибридного фаззинга в FUZZOLIC аналогична QSYM. Он принимает входные данные, мутированные AFL++, в то время как фаззер принимает интересные входные данные, предоставляемые символьным исполнителем.

2.1.4 PASTIS

David и др. [12] представили инфраструктуру автоматизированного тестирования, сочетающую фаззинг и ДСИ для проверки предупреждений, полученных от некоторого инструмента статического анализа, и выявления ошибок, если это возможно. Информация, предоставляемая SAST-инструментом, используется для добавления встроенных функций в целевой код. Все варианты кода компилируются и отправляются в брокер PASTIS, который выполняет всю коммуникацию между механизмами тестирования. С одной стороны, Honggfuzz [26] представляет набор инструментов для фаззинга, а с другой стороны, фреймворк TritonDSE [27] отвечает за символьное выполнение.

2.1.4 SymSan, Jigsaw

Основная идея SymSan [11] заключается в построении конколического исполнителя как особой формы анализа динамических потоков данных. SymSan выполняет инструментирование кода на этапе компиляции в LLVM IR [28]. Использование

высокооптимизированной инфраструктуры из DFSAN [29] помогает снизить накладные расходы на хранение и извлечение символьных выражений, форма которых также оптимизирована. SymSan предлагает использовать таблицу AST вместе со специальным дизайном узлов AST для хранения символьных выражений. В сочетании с простым прямым выделением новых узлов это позволяет значительно снизить потери производительности при обработке символьных выражений. Дополнительно SymSan реализует дедупликацию хранимых узлов AST и упрощение операций загрузки и сохранения. Авторы реализовали гибридный фаззер на основе SymSan и Angora [30]. Они также предложили новую конструкцию для повышения пропускной способности поиска и внедрили ее в прототип Jigsaw [31], который используется в качестве решателя для гибридного фаззера. Суть подхода заключается в оценке вновь сгенерированных входных данных с помощью JIT-скомпилированных ограничений пути. Jigsaw компилирует предобработанные подзадачи AST в функции LLVM IR, использует JIT-движок LLVM для компиляции IR в нативную функцию и ищет удовлетворяющее решение с помощью градиентно-управляемого поиска.

2.2 Непрерывный фаззинг

Сближение безопасной разработки и фаззинга становится отраслевым стандартом [2, 3], подобно обязательному модульному тестированию. Непрерывный фаззинг – это подход к организации автоматизированного фаззинг-тестирования как рутинной процедуры, например, путем его интеграции в конвейер CI/CD. Инфраструктура непрерывного фаззинга может быть частью внутреннего рабочего процесса организации [32, 33] или существовать в форме инструмента «фаззинг-как-услуга» [19, 34–37]. Существующие решения включают как простые запускатели фаззинг-задач, так и крупные фреймворки, охватывающие расширенную функциональность для масштабируемого и ансамблевого [12, 38, 39] фаззинга, анализа и отчетности по обнаруженным уязвимостям, бисекции коммитов, вызвавших регрессию, и т.д.

1) *OSS-Fuzz & ClusterFuzz*: Хотя автоматизированное фаззинг-тестирование не является серебряной пулей, оно достаточно эффективно для обнаружения ошибок и сокращения требуемых человеческих усилий аналитиков. Например, ClusterFuzz [40] уже нашел более 25 000 ошибок в продуктах, разработанных Google, и более 43 500 ошибок [41] в программном обеспечении с открытым исходным кодом, включенном в OSS-Fuzz. Проект OSS-Fuzz [34] выборочно предоставляет сообществу открытого исходного кода доступ к масштабируемым мощностям инфраструктуры ClusterFuzz, работающей на Google Cloud Platform [42]. ClusterFuzz предназначен для автоматического выполнения любой задачи в рамках жизненного цикла фаззинга, за исключением написания фаззинг-целей и исправления ошибок. Это включает планирование и запуск заданий фаззинга, сбор статистики, дедупликацию и классификацию новых сбоев, минимизацию тестовых случаев, бисекцию коммита, вызвавшего регрессию, и проверку исправления ошибки. ClusterFuzz поддерживает несколько механизмов фаззинга с покрытием [15, 16, 26], черный ящик фаззинга и ряд санитайзеров. Поскольку ClusterFuzz не обладает собственной инфраструктурой сборки, типичный набор проекта для участия в OSS-Fuzz включает образ Docker, конфигурацию сборки и как минимум одну фаззинг-цель. На данный момент OSS-Fuzz непрерывно проводит фаззинг более 650 критически важных и широко используемых проектов с открытым исходным кодом.
2) *OneFuzz*: Другой фреймворк непрерывного фаззинга, OneFuzz [32], был открыт Microsoft. Подобно ClusterFuzz, OneFuzz в настоящее время применяется к программному обеспечению Microsoft (Windows, Edge и др.) и привязан к

корпоративной облачной среде Azure [43]. Поддерживаемые механизмы фаззинга – это libFuzzer [16], AFL++ [15] и Radamsa [44]. OneFuzz предоставляет возможность воспользоваться встроенными шаблонами, а также позволяет создавать настраиваемые конвейеры фаззинга. Обнаруженные уязвимости классифицируются по стабильности воспроизведения и дедуплицируются. Кроме того, OneFuzz позволяет проводить live-отладку сбоев и мониторинг рабочего процесса фаззинга с помощью настраиваемых веб-хуков.

3) *Grizzly*: FirefoxCI TaskCluster использует Grizzly [33] – масштабируемый фреймворк для фаззинга, специфичный для браузеров. Grizzly независимо запускает браузер и механизм фаззинга, управляет передачей данных между ними во время анализа и выполняет сокращение тестовых случаев. Два основных интерфейса, Target и Adapter, отвечают за развертывание желаемой комбинации браузера и фаззера, но фреймворк в первую очередь ориентирован на методы фаззинга черного ящика. Основной размер шрифта – 10 пт.;
4) *Fuzzit*: Интегрированный в сервис Gitlab [45] Fuzzit [35] был разработан для интеграции фаззинга в систему непрерывной сборки проекта. Он представляет собой раннеры для набора механизмов фаззинга с покрытием, позволяющие проводить фаззинг для разных языков программирования (например, в случае C/C++ возможны варианты libFuzzer и AFL++), и способен проводить регрессионное тестирование.
5) *syzbot*: Система непрерывного фаззинга ядра Linux syzbot [46] последовательно генерирует структурированные отчеты об обнаруженных сбоях ядра. Помимо поиска ошибок, система отслеживает устаревание отчетов об ошибках и проверяет исправленные проблемы. После проведения тестирования патча syzbot проверяет, что соответствующий коммит попал в сборки ядра для всех отслеживаемых ветвей, чтобы закрыть отчет об ошибке. Схожие по проявлениям сбои могут быть сообщены отдельно, вместо того чтобы объединяться с существующей проблемой.

3. Гибридный фаззинг

Статистика показывает, что гибридные фаззинговые механизмы, сочетающие мощь символьного выполнения и фаззинга, могут достигать большего покрытия, чем два одновременно работающих фаззера [13]. Основываясь на этой идее, мы реализовали наш гибридный фаззинговый инструмент, объединив Sydr [14] с двумя наиболее популярными и мощными фаззерами с открытым исходным кодом: libFuzzer [16] и AFL++ [15]. Для достижения цели создания эффективного гибридного фаззера необходимо решить две основные задачи. Во-первых, мы разрабатываем различные эвристики, чтобы заставить Sydr работать максимально эффективно в сочетании с фаззером. Во-вторых, мы реализуем интеграцию Sydr и фаззеров, такую как межпроцессное взаимодействие и планирование входных данных для Sydr.

Для запуска sydr-fuzz в режиме гибридного фаззинга необходимо собрать целевой бинарный файл в двух версиях. Первая должна быть собрана с санитайзерами и фаззинговой инструментацией для использования соответствующим фаззером. Второй бинарный файл должен быть собран без какой-либо инструментации – он будет использоваться Sydr.

Мы реализовали различные эвристики и функции, чтобы Sydr находил новые интересные входные данные быстрее и работал эффективнее в сочетании с фаззерами. Например, мы реализовали кэш в Sydr, работающий аналогично кэшу QSYM, чтобы экономить время на попытках инвертировать одни и те же символические ветви. Основная идея заключается в том, что для каждой ветви вычисляется уникальный хэш, который используется как индекс

в битовой карте. Каждая отдельная ветвь соответствует байту, который является счетчиком, показывающим, сколько раз предпринималась попытка инвертировать ветвь. Более того, мы сохраняем контекст выполнения для каждой ветви, который отражает, какие ветви встречались во время выполнения до рассматриваемой. Если контекст изменяется (т.е. Sydr достигает ветви по другому пути выполнения), мы пытаемся инвертировать ветвь. В противном случае мы пытаемся инвертировать отдельную ветвь, когда соответствующий счетчик равен 0, 2, 4, 8 и так далее (степени двойки), пока он не достигнет 255, после чего мы прекращаем инверсию данной конкретной ветви.

Слайсинг [14] – это еще одна эвристика для ускорения символьного выполнения. Каждый раз, когда мы пытаемся инвертировать ветвь, мы применяем алгоритм слайсинга к ее предикату пути. Идея алгоритма заключается в том, что мы оставляем в предикате пути только те ограничения пути, которые зависят по данным от целевой ветви. Остальные ограничения пути устраняются, а входные данные для них берутся из исходного набора входных данных программы.

Мы реализовали оптимистичное и строгое оптимистичное решение [47], чтобы избежать неполных и излишне перегруженных формул. Таким образом, Sydr инвертирует больше ветвей и генерирует больше входных данных. Основная идея следующая. Когда исходный урезанный (sliced) предикат невыполним, мы строим оптимистичный предикат, состоящий только из ограничения целевой ветви. Если он выполним, мы строим строго оптимистичный предикат, который получается из исходного урезанного предиката путем исключения некоторых нерелевантных ограничений пути на основе анализа стека вызовов программы и управляющих зависимостей ветвей. Если строго оптимистичный предикат совпадает с оптимистичным или невыполним, мы сохраняем входной файл для оптимистичного решения. Если они не совпадают и строго оптимистичный предикат выполним, мы сохраняем оба сгенерированных набора входных данных.

Мы реализовали семантику функций для ускорения символьного выполнения функций стандартной библиотеки. Вместо того чтобы входить в тела функций стандартной библиотеки (таких как strto*l, *alloc, strcmp и т.д.) и выполнять их символьно, мы строим символьные формулы для их возвращаемых значений, что помогает ускорить символьное выполнение и избежать переполнения ограничений.

Мы ограничиваем время, которое отводится процессу Sydr на решение одного запроса, и общее время, которое он может потратить на решение всех запросов, чтобы избежать ситуаций, когда SMT-решатель застревает на сложных запросах. Более того, мы устанавливаем таймаут на общее время выполнения одного процесса Sydr, чтобы избежать зависания программы.

Также стоит упомянуть, что Sydr имеет функциональность для инверсии таблиц переходов (операторов switch) и обработки символьных адресов. Существуют два способа обработки символических адресов в Sydr: полная обработка символических указателей и фаззинг символических адресов. Полная поддержка символических указателей [21] резко перегружает символьный движок, что приводит к ухудшению результатов фаззинга. Однако этот режим все же позволяет символьному движку успешно обнаруживать новое уникальное покрытие. Следовательно, оптимальная стратегия – включать этот режим периодически, а не использовать при каждом запуске Sydr. Когда Sydr работает без обработки символических указателей, он выполняет SMT-фаззинг для всех символических адресов. Это легковесный, но менее точный метод, который перебирает возможные значения символьного адреса с помощью SMT-решателя.

Перед началом процесса гибридного фаззинга исходный корпус автоматически минимизируется, чтобы оставить только те входные данные, которые приносят какое-либо новое покрытие.

4. Предикаты безопасности

Мы предлагаем метод предикатов безопасности – метод точного обнаружения ошибок на основе динамического символьного выполнения – как часть sydr-fuzz. Мы реализуем автоматическую проверку предикатов безопасности с последующей верификацией и дедупликацией входных данных, сгенерированных предикатами безопасности, для выявления ошибок.

Предикат безопасности – это булевый предикат, который принимает значение истина, если инструкция (или функция) программы вызывает ошибку [18]. Реализация предикатов безопасности является частью Sydr [14] и основана на следующей идее. Мы выполняем программу символьно с использованием набора входных данных, который не приводит к ошибке. Каждый раз, когда выполняется инструкция, оперирующая символьными данными, мы строим соответствующий предикат безопасности для проверки на определенный тип ошибки. Затем мы объединяем предикат безопасности с "нарезанными" ограничениями ветвей из предиката пути и передаем результирующий предикат SMT-решателю, такому как Bitwuzla [48], чтобы сгенерировать входные данные, которые воспроизведут ошибку. Если предикат выполним, мы сохраняем вход и сообщаем об ошибке. Предикаты безопасности могут обнаруживать несколько типов слабостей: разыменование нулевого указателя, деление на ноль, целочисленное переполнение и выход за границы буфера.

Предикаты безопасности для разыменования нулевого указателя и деления на ноль работают схожим образом. В первом случае мы строим предикат безопасности, чтобы проверить, может ли символьный адрес быть равен нулю, каждый раз при выполнении инструкции доступа к памяти. Во втором случае мы строим предикат безопасности, чтобы проверить, может ли символьный делитель быть равен нулю, каждый раз при выполнении инструкции деления, такой как div или idiv, во время символьного выполнения.

Для ошибки выхода за границы мы строим предикат безопасности при каждой инструкции доступа к памяти. Этот предикат истинен, если символьный адрес может быть меньше нижней границы массива или больше верхней границы. Для определения границ массива мы ведем теневой стек и теневую кучу во время символьного выполнения. Мы сохраняем информацию о границах массивов, выделенных в куче, при каждом вызове функции *alloc и удаляем ее при вызове free. Мы изменяем теневой стек в соответствии со стеком вызовов, поскольку рассматриваем верхнюю границу массива в стеке как адрес начала фрейма функции. Для массивов в куче мы получаем их границы из теневой кучи. Для массивов в стеке мы считаем верхней границей точку вызова текущей функции и эвристически вычисляем нижнюю границу.

При каждой арифметической инструкции во время символьного выполнения мы строим предикат безопасности для ошибки целочисленного переполнения. Этот предикат истинен, если процессорные флаги переноса CF (Carry Flag) или переполнения OF (Overflow Flag) равны 1 после выполнения инструкции. Если результат знаковый, мы проверяем только флаг OF, иначе – CF. Знаковость определяется по ранее встреченным условным инструкциям, которые используют хотя бы один операнд, совпадающий с используемым анализируемой инструкцией. Например, JL указывает, что результат знаковый, и мы должны проверять только флаг OF. Ключевое отличие этого предиката от других в том, что мы разделяем понятия источника ошибки (арифметическая инструкция, где может возникнуть ошибка) и приемника ошибки (место, где ошибка может быть использована), и проверяем предикат безопасности для источника только когда найден приемник, использующий этот источник. Мы выделяем следующие типы приемников: инструкции ветвления, инструкции разыменования адреса и аргументы функций [49].

Мы реализуем автоматическую проверку предикатов безопасности в sydr-fuzz. Предикаты безопасности ищут ошибки только на пути, заданном исходным набором входных данных.

Следовательно, нам нужно достичь максимального покрытия минимальным количеством входных данных. Сначала мы запускаем гибридный фаззинг для достижения высокого покрытия. Затем мы выполняем минимизацию корпуса, чтобы оставить минимальное количество файлов, обеспечивающих то же покрытие, что и до минимизации. После минимизации запускается проверка предикатов безопасности. Результаты предикатов безопасности часто оказываются ложными срабатываниями, поэтому мы реализуем их автоматическую верификацию. Мы запускаем целевой бинарный исполняемый файл, собранный с санитайзерами (автоматические проверки на наличие ошибок, встраиваемые компилятором), на входном файле, сгенерированном предикатами безопасности. Если какой-либо санитайзер сообщает об ошибке в месте, указанном предикатами как источник, мы считаем входной файл верифицированным. В противном случае мы запускаем исполняемый файл с санитайзерами на исходном входном файле из корпуса, которое использовалось для проверки предикатов, и, если входной файл от предикатов вызывает новые предупреждения санитайзеров, мы считаем его верифицированным. Sydr-fuzz выполняет дедупликацию верифицированных результатов предикатов безопасности на основе исходного файла, строки и столбца кода, где обнаружена ошибка, чтобы упростить анализ результатов.

5. Сортировка аварийных завершений

Классификация аварийных завершений является важным этапом динамического анализа, включая фаззинг или гибридный фаззинг. Количество сбоев, которые генерирует фаззер, может быть значительным. На то, чтобы определить, какие сбои представляют одну и ту же ошибку, можно потратить много времени. Мы предлагаем подход, который должен помочь разработчикам тратить меньше времени на анализ и исправление различных ошибок и проблем. Наши инструменты Casr [17] позволяют создавать отчеты о сбоях, дедуплицировать и кластеризовать их. Основные этапы работы sydr-fuzz casr следующие:

1) casr-san запускает инструментированный бинарный файл на всех входных файлах, которые потенциально вызывают сбои, и генерирует отчеты о сбоях на основе отчетов санитайзеров (при необходимости с помощью gdb).
2) casr-cluster запускает алгоритм дедупликации для отчетов Casr, полученных от casr-san. Дедупликация основана на трассе стека: каждый фрейм хэшируется, затем хэш всей трассы стека добавляется в хэш-набор. В результате в хэш-наборе останутся только уникальные отчеты (подробности [17]), остальные будут удалены из каталога casr.
3) casr-cluster запускает иерархическую кластеризацию отчетов Casr. Расстояние между сбоями вычисляется на основе трассы стека [17].
4) casr-gdb генерирует отчеты о сбоях для неинструментированных целевых бинарных файлов на основе кластеров, полученных на третьем шаге.

В результате мы получаем кластеры, содержащие потенциально различные ошибки (в виде отчетов Casr). Вместе с каждым отчетом предоставляется соответствующий входной файл, которое приводит к этому сбою. Некоторые шаги (3, 4) можно пропустить, установив соответствующие опции.

Отчет о сбое содержит информацию о сбое, такую как версии ОС и пакетов, выполненная командная строка, трасса стека, открытые файлы и сетевые подключения, состояние регистров, часть исходного кода, вызвавшая сбой, с соответствующей строкой сбоя и т.д. Таким образом, разработчику не нужно запускать gdb и анализировать сбой вручную, вся необходимая информация уже есть в отчете.

Также наши инструменты Casr позволяют оценить серьезность сбоя. Мы делим их на три широких класса (как и в gdb exploitable [50]): эксплуатируемый (exploitable), вероятно эксплуатируемый (probably exploitable) и не эксплуатируемый (not exploitable). Классы

включают различные ошибки, которые могут возникнуть во время выполнения программы, такие как переполнение стека, двойное освобождение памяти и другие. Класс сбоя определяется на основе трассы стека, дизассемблированного участка кода, вызвавшего исключение, сигнала, поступившего в программу, и некоторой другой информации. Классифицированные сбои помогают разработчикам понять, какие из них следует анализировать и исправлять в первую очередь.

6. Инфраструктура непрерывного гибридного фаззинга

Интеграцию непрерывного фаззинга с использованием фреймворка sydr-fuzz можно продемонстрировать на примере проекта OSS-Sydr-Fuzz [19]. OSS-Sydr-Fuzz создан по образцу упомянутого выше OSS-Fuzz [34] и призван подтвердить эффективность гибридного фаззинга на реальном программном обеспечении. Соответствующий список обнаруженных уязвимостей [51] представлен в разделе 7.

Необходимые исходные коды включают код тестируемого проекта и дополнительный репозиторий фаззинга OSS-Sydr-Fuzz [19]. Репозиторий содержит набор предварительно подготовленных средств для настройки и запуска сессии гибридного фаззинга. Ключевыми задачами для запуска являются сборка кода проекта, развертывание инструментов фаззинга и подготовка фаззинг-целей. Соответственно, стандартный пакет проекта OSS-Sydr-Fuzz включает скрипт сборки, инструкции Dockerfile, составленные фаззинг-цели и файл конфигурации гибридного фаззинга для каждой цели. Кроме того, репозиторий может предоставлять дополнительные материалы, такие как начальный корпус входных данных, словари и т.д.

Изображенный на схеме (рис. 1) процесс CI фаззинга реализован на платформе GitLab [52]. Система применяет конвейер динамического анализа, состоящий из пяти этапов: гибридный фаззинг, минимизация корпуса, поиск ошибок с помощью предикатов безопасности, сортировка аварийных завершений и сбор покрытия. Фаззинг запускается при возникновении внешнего триггерного события. В настоящее время он запускается вручную для выбранного проекта и ветки. Альтернативно, он может быть организован как запланированная рутина или запускаться новыми коммитами.

Перед началом анализа выбранного проекта запускатель заданий фаззинга активирует задачу сборки Docker. В первую очередь, сборщик проверяет актуальность коммита и запрашивает у реестра контейнеров, существует ли образ Docker из предыдущей итерации и можно ли его повторно использовать. Полученный или пересобранный контейнер используется для всех фаззинг-целей проекта. Каждая фаззинг-цель имеет индивидуальный корпус и отдельное задание фаззинга, которое последовательно запускает этапы конвейера анализа. Кроме того, существует возможность расширить начальный корпус входных данных файлами, полученными в ходе предыдущих сеансов фаззинга. Помимо журналов и статистики анализа, выходные данные задания фаззинга включают результирующий корпус, информацию о покрытии, отчеты Casr и входные данные, вызывающие ошибки.

7. Экспериментальная оценка результатов

Мы постоянно используем sydr-fuzz для проведения программного анализа. Репозиторий OSS-Sydr-Fuzz [19] содержит список тестируемых проектов, инструкции по их сборке, настройки конфигурации для окружения и процесса фаззинга. За один год использования sydr-fuzz было обнаружено 85 ранее неизвестных дефектов в 22 проектах с открытым исходным кодом [51]. Из них 13 ошибок были найдены с помощью предикатов безопасности, реализованных в Sydr.

Для сравнения sydr-fuzz с современными инструментами фаззинга мы использовали фреймворк Google FuzzBench [22]. Эксперименты на FuzzBench развертывались на сервере

с 256 ГБ ОЗУ и двумя процессорами AMD EPYC 7702 (по 64 ядра каждый). Sydr-fuzz последовательно тестировался в сравнении с 4 фаззерами: libFuzzer [16], AFL++ [15], SymQEMU [9] и FUZZOLIC [10]. Мы выбрали 15 целей из FuzzBench, предназначенных для оценки покрытия. Эксперимент был настроен на проведение 10 прогонов для каждой комбинации фаззера и цели, каждый прогон выполнялся на одном ядре CPU в течение 23 часов. Из-за ограниченных ресурсов сервера мы разделили тестирование sydr-fuzz на две группы целей. Результаты наших экспериментов на FuzzBench общедоступны [23].

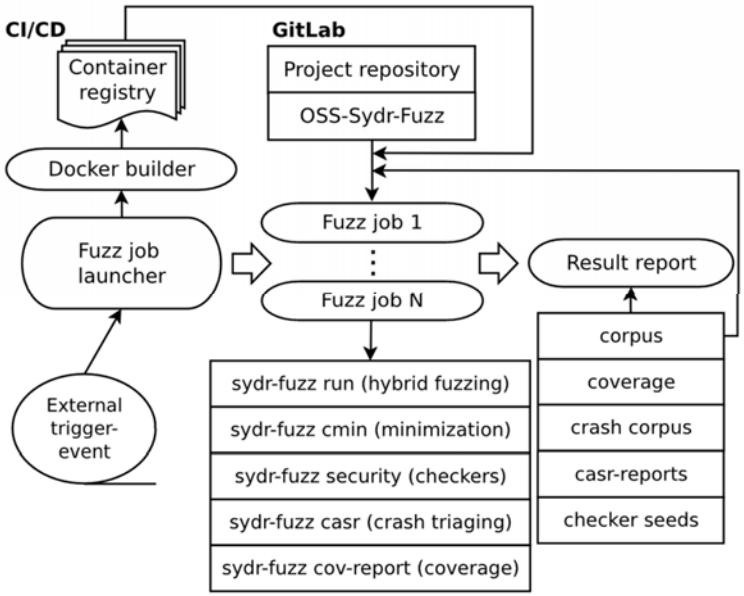

Рис. 1. CI-схема OSS-Sydr-Fuzz.
Fig. 1. CI-scheme for OSS-Sydr-Fuzz.

В ходе оценки использовалась следующая конфигурация sydr-fuzz. Гибридный фаззинг выполнялся с одним экземпляром фаззера и одним запущенным экземпляром Sydr в каждый момент времени. Sydr инвертировал ветви в прямом порядке с одним потоком решения. Ограничение в 10 секунд устанавливалось для решения одного SMT-запроса, и 60 секунд – для общего времени решения. Каждый запуск Sydr ограничивался 2 минутами. Использовался кэш для предотвращения инверсии одних и тех же ветвей Sydr. Для Sydr были включены строгое оптимистичное решение и фаззинг символических адресов. Каждый символический адрес обрабатывался фаззингом для до 10 различных моделей. Фаззинг символических адресов останавливался для текущего запуска Sydr, когда было сгенерировано 1000 таких моделей. Каждый 25-й запуск Sydr вместо фаззинга символических адресов включался режим полной обработки символических указателей. Процесс построения предиката пути приостанавливался после планирования 300 заданий на инверсию ветвей во время символьного выполнения. Использование памяти для Sydr ограничивалось 8 ГБ. При превышении этого лимита выполнение программы завершается, и продолжается только инверсия ветвей.

7.1 Сравнение с фаззерами

Во-первых, мы сравнили sydr-fuzz с двумя современными фаззерами, чтобы доказать преимущества использования символьного движка (рис. 2). Мы протестировали AFL++ [15] и libFuzzer [16] в сравнении с sydr-fuzz, сконфигурированного с соответствующим фаззером. Одинаковые версии libFuzzer (de5b16d) и AFL++ (8fc249d) использовались в sydr-fuzz и FuzzBench для оценки. Также во всех инструментах использовалась идентичная конфигурация фаззера. Поскольку в ходе одного испытания sydr-fuzz запускает два экземпляра (фаззер и Sydr) на одном CPU-ядре, мы сконфигурировали FuzzBench на запуск двух воркеров libFuzzer (-workers=2) и двух инстансов AFL++ (основной и вторичный узлы).

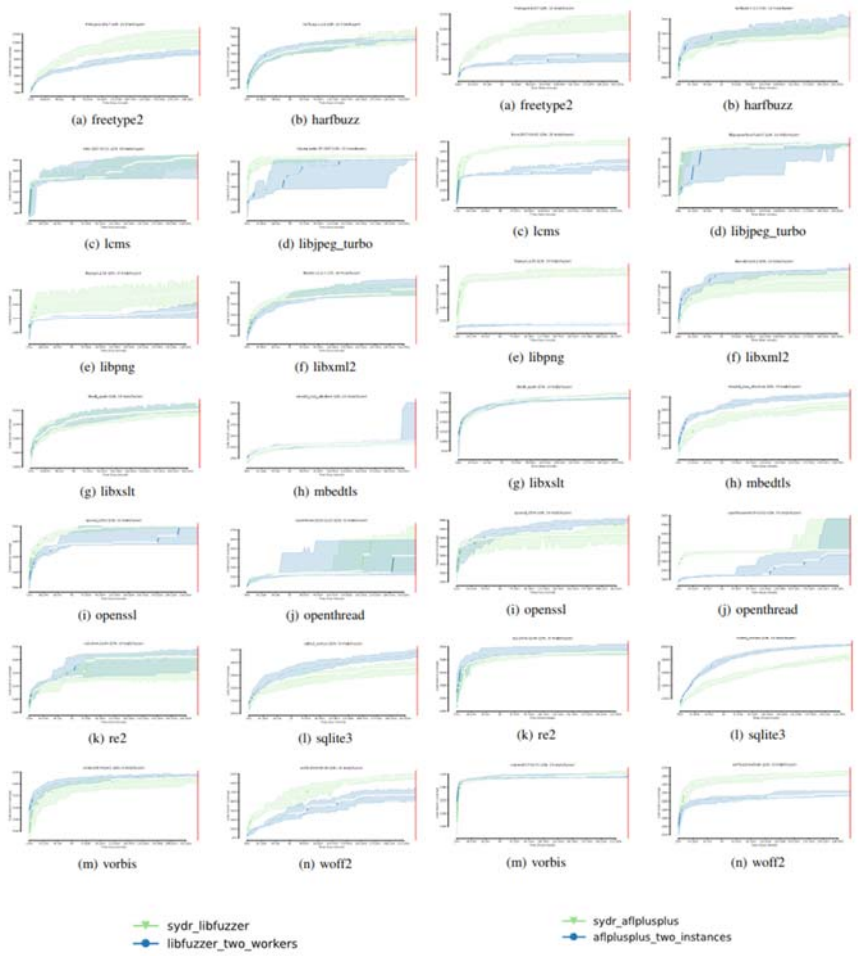

Рис. 2. Тестирование Sydr-Fuzz и фаззеров (23ч).
Fig. 2. Results of Sydr-Fuzz u fuzzer testing.

Результаты сравнения sydr-fuzz и libFuzzer показаны на рис. 2 в двух левых столбцах. Sydr-fuzz превзошел libFuzzer по достигнутому покрытию в 9 из 14 приложений. Из 5 приложений, где libFuzzer показал лучшие результаты, существенное преимущество наблюдается только для sqlite3. Для остальных 4 приложений итоговое покрытие незначительно отличается. Эксперимент проводился в двух запусках FuzzBench по 7 бенчмарков в каждом. Sydr-fuzz показал более высокий нормализованный показатель покрытия в обоих запусках: 98.67% и 99.63% для sydr-fuzz против 96.51% и 98.33% для libFuzzer соответственно.

В двух правых столбцах рис. 2 показаны результаты эксперимента sydr-fuzz в сравнении с AFL++. Sydr-fuzz также показал лучшие результаты в 9 из 14 приложений в этом эксперименте. Как видно из рис. 2, на большинстве приложений наблюдается значительный разброс результатов между sydr-fuzz и AFL++ в пользу обоих инструментов. Для двух групп целей sydr-fuzz получил более высокий средний показатель покрытия: 98.75% и 99.19% для sydr-fuzz против 94.87% и 96.70% для AFL++ соответственно.

7.2 Сравнение с гибридными фаззерами

На следующем этапе мы оценили sydr-fuzz в сравнении с современными гибридными фаззерами. Мы выбрали SymQEMU [9] и FUZZOLIC [10], поскольку эти инструменты также выполняют гибридный фаззинг на основе AFL++ [15] с символьным выполнением бинарного кода. Мы убедились, что sydr-fuzz использует ту же версию AFL++ (8fc249d) и настройки, что и эти инструменты. Набор бенчмарков незначительно отличается от использованного для оценки фаззеров из-за несовместимости тестируемых инструментов с некоторыми целями: оба инструмента не работали с libxslt и openssl, а FUZZOLIC – с woff2.

Результаты сравнения sydr-fuzz и SymQEMU показаны на рис. 3 в двух левых столбцах.

Sydr-fuzz смог превзойти SymQEMU в 7 из 13 приложений. Результаты довольно близки на большинстве бенчмарков. Из всех бенчмарков, где SymQEMU показал лучшие результаты, только для zlib_uncompress наблюдается значительное преимущество над sydr-fuzz. Для бенчмарка sqlite3 было запущено только 5 испытаний из-за нестабильности SymQEMU на этой цели. Для двух групп экспериментов sydr-fuzz показал более высокое среднее покрытие: 99.35% и 99.95% для sydr-fuzz против 97.03% и 99.67% для SymQEMU соответственно.

Результаты экспериментов с sydr-fuzz и FUZZOLIC показаны в правых двух столбцах рис. 3. Для оценки FUZZOLIC было доступно только 12 целей. Sydr-fuzz смог превзойти FUZZOLIC в 6 из 12 бенчмарков. Как и в случае с SymQEMU, результаты в этом эксперименте близки. FUZZOLIC показал значительно большее покрытие только на цели sqlite3. Sydr-fuzz существенно превзошел FUZZOLIC на целях libjpeg_turbo и openthread. На остальных бенчмарках результаты очень схожи по истечении 23 часов. Тем не менее, sydr-fuzz смог достичь немного более высокого среднего нормализованного показателя покрытия в обеих группах экспериментов: 99.1% и 99.84% для sydr-fuzz против 99.07% и 99.81% для FUZZOLIC соответственно.

Таким образом, sydr-fuzz превзошел как libFuzzer, так и AFL++ на большинстве оцениваемых бенчмарков и достиг более высокого общего покрытия. Наши результаты также показывают, что sydr-fuzz находится на том же уровне, что и мощные современные гибридные фаззеры, и в некоторых случаях может их превосходить. Кроме того, sydr-fuzz смог превзойти все протестированные coverage-guided и гибридные фаззеры на 4 целях: freetype2, openthread, libxslt (SymQEMU и FUZZOLIC не смогли выполнить libxslt) и woff2 (FUZZOLIC не удалось запустить). Также есть одна цель, sqlite3, на которой все другие инструменты показали лучшие результаты, чем sydr-fuzz. Это можно объяснить неэффективной работой Sydr в данном конкретном случае, что снижает общую производительность фаззинга.

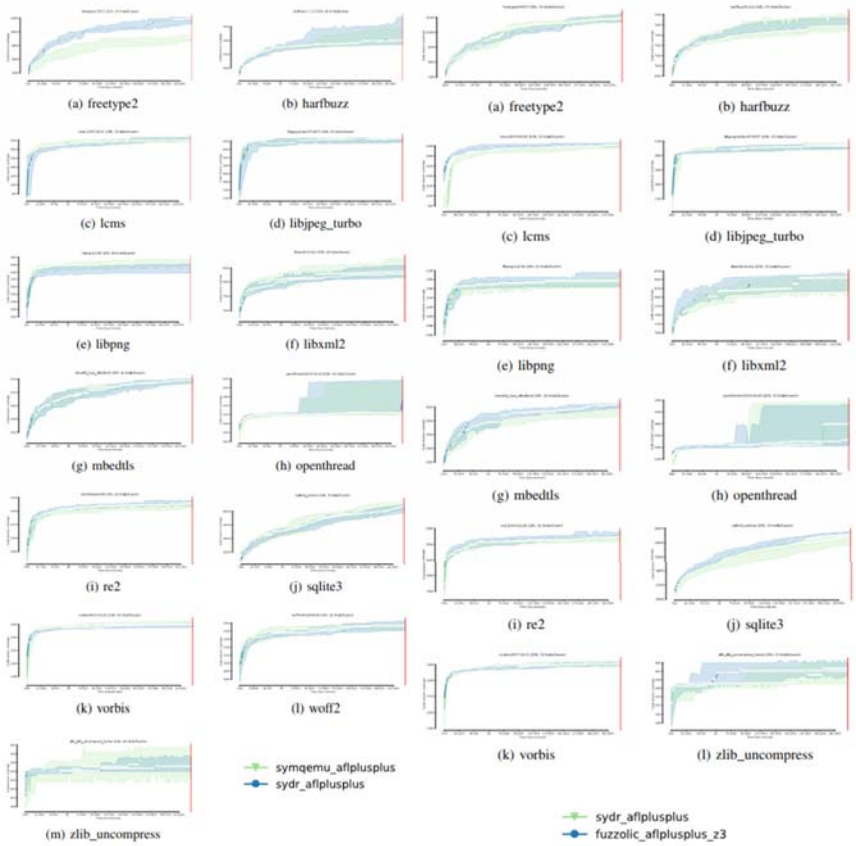

Рис. 3. Тестирование Sydr-Fuzz и гибридных фаззеров (23ч).
Fig. 3. Results of Sydr-Fuzz and hybrid fuzzers testing.

8. Заключение

Мы представили фреймворк непрерывного гибридного фаззинга Sydr-Fuzz для эффективного динамического анализа программ в рамках жизненного цикла разработки безопасных приложений. Мы объединили оркестрацию гибридного фаззинга, минимизацию корпуса, обнаружение ошибок, сбор информации о покрытии и классификацию сбоев в единый набор инструментов. Мы представили новую интеграцию гибридного фаззинга на основе символьного исполнителя Sydr [14] и популярных фаззеров с открытым исходным кодом AFL++ [15] и libFuzzer [16]. Мы создали репозиторий OSS-Sydr-Fuzz [19] с целями из ПО с открытым исходным кодом для sydr-fuzz и обнаружили 85 новых ошибок в 22 проектах [51]. Мы предложили конвейер динамического анализа для sydr-fuzz, чтобы максимизировать полезный эффект от набора инструментов. Мы открыли для использования исходный код инструмента Casr для кластеризации и дедупликации отчетов о сбоях [24].

Наша оценка показывает, что, с одной стороны, sydr-fuzz превосходит современные фаззеры с покрытием AFL++ и libFuzzer на большинстве оцениваемых целей и достигает более высокого общего покрытия. С другой стороны, sydr-fuzz оказался сопоставим с мощными современными гибридными фаззерами SymQEMU [9] и FUZZOLIC [10] и в некоторых случаях даже может их превзойти. Значительная польза, которую sydr-fuzz получает от динамического символьного исполнителя Sydr в процессе фаззинга, таким образом, демонстрирует актуальность использования передовых гибридных фаззеров в динамическом анализе.

Список литературы / References

[1]. M. Howard and S. Lipner, The security development lifecycle. Microsoft Press Redmond, 2006, vol. 8. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms995349.aspx (accessed 24.09.2025).
[2]. ISO/IEC 15408-3:2008: Information technology – Security techniques – Evaluation criteria for IT security – Part 3: Security assurance components. 2008. [Online]. Available: https://www.iso.org/standard/46413.html (accessed 24.09.2025).
[3]. GOST R 56939-2016: Information protection. Secure software development. General requirements. National Standard of Russian Federation, 2016. [Online]. Available: http://protect.gost.ru/document.aspx?control=7&id=203548 (accessed 24.09.2025).
[4]. S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, Unleashing Mayhem on binary code, in Proceedings of the 2012 IEEE Symposium on Security and Privacy, ser. SP '12, IEEE, 2012, pp. 380-394.
[5]. B. S. Pak, Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution, M.S. thesis, School of Computer Science Carnegie Mellon University, 2012.
[6]. N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, Driller: Augmenting fuzzing through selective symbolic execution, in NDSS, vol. 16, 2016, pp. 1-16.
[7]. I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, QSYM: A practical concolic execution engine tailored for hybrid fuzzing, in 27th USENIX Security Symposium, 2018, pp. 745-761.
[8]. S. Poeplau and A. Francillon, Symbolic execution with SymCC: Don't interpret, compile! In 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 181-198.
[9]. S. Poeplau and A. Francillon, SymQEMU: Compilation-based symbolic execution for binaries, in Proceedings of the 2021 Network and Distributed System Security Symposium, 2021.
[10]. L. Borzacchiello, E. Coppa, and C. Demetrescu, FUZZOLIC: Mixing fuzzing and concolic execution, Computers & Security, vol. 108, p. 102 368, 2021.
[11]. J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, SYMSAN: Time and space efficient concolic execution via dynamic data-flow analysis, in 31st USENIX Security Symposium (USENIX Security 22), USENIX Association, 2022, pp. 2531-2548.
[12]. R. David, J. Salwan, and J. Bourroux, From source code to crash test-cases through software testing automation, Proc. of the 28th C&ESAR, p. 27, 2021.
[13]. FuzzBench symbolic report, 2021. [Online]. Available: https://www.fuzzbench.com/reports/experimental/2021-07-03-symbolic/index.html (accessed 24.09.2025).
[14]. A. Vishnyakov, A. Fedotov, D. Kuts, A. Novikov, D. Parygina, E. Kobrin, V. Logunova, P. Belecky, and S. Kurmangaleev, "Sydr: Cutting edge dynamic symbolic execution," in 2020 Ivannikov ISPRAS Open Conference (ISPRAS), IEEE, 2020, pp. 46-54.
[15]. A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in 14th USENIX Workshop on Offensive Technologies (WOOT 20), 2020.
[16]. K. Serebryany, "Continuous fuzzing with libFuzzer and AddressSanitizer," in 2016 IEEE Cybersecurity Development (SecDev), IEEE, 2016, p. 157.
[17]. G. Savidov and A. Fedotov, "Casr-Cluster: Crash clustering for linux applications," in 2021 Ivannikov ISPRAS Open Conference (ISPRAS), IEEE, 2021, pp. 47-51.
[18]. A. Vishnyakov, V. Logunova, E. Kobrin, D. Kuts, D. Parygina, and A. Fedotov, "Symbolic security predicates: Hunt program weaknesses," in 2021 Ivannikov ISPRAS Open Conference, IEEE, 2021, pp. 76-85.

[19]. OSS-Sydr-Fuzz: Hybrid fuzzing for open source software. [Online]. Available: https://github.com/ispras/oss-sydr-fuzz (accessed 24.09.2025).
[20]. OSS-Fuzz: Continuous fuzzing for open source software. [Online]. Available: https://github.com/google/oss-fuzz (accessed 24.09.2025).
[21]. D. Kuts, "Towards symbolic pointers reasoning in dynamic symbolic execution," in 2021 Ivannikov Memorial Workshop (IVMEM), IEEE, 2021, pp. 42-49.
[22]. J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "FuzzBench: An open fuzzer benchmarking platform and service," in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1393-1403.
[23]. FuzzBench results for Sydr-Fuzz. [Online]. Available: https://sydr-fuzz.github.io/fuzzbench (accessed 24.09.2025).
[24]. CASR: Crash analysis and severity report. [Online]. Available: https://github.com/ispras/casr (accessed 24.09.2025).
[25]. L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzing symbolic expressions," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 711-722.
[26]. R. Swiecki and F. Gröbert, Honggfuzz. [Online]. Available: https://github.com/google/honggfuzz (accessed 24.09.2025).
[27]. F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in Symposium sur la sécurité des technologies de l'information et des communications, ser. SSTIC, 2015, pp. 31-54.
[28]. C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), vol. 4, 2004, p. 75.
[29]. DataFlowSanitizer design document, 2018. [Online]. Available: https://clang.llvm.org/docs/DataFlowSanitizerDesign.html (accessed 24.09.2025).
[30]. P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in 2018 IEEE Symposium on Security and Privacy (SP), IEEE, 2018, pp. 711-725.
[31]. J. Chen, J. Wang, C. Song, and H. Yin, "JIGSAW: Efficient and scalable path constraints fuzzing," in 2022 IEEE Symposium on Security and Privacy (SP), IEEE, 2022, pp. 1531-1531.
[32]. OneFuzz: A self-hosted fuzzing-as-a-service platform. [Online]. Available: https://github.com/microsoft/onefuzz (accessed 24.09.2025).
[33]. Grizzly browser fuzzing framework. [Online]. Available: https://github.com/MozillaSecurity/grizzly.
[34]. K. Serebryany, "OSS-Fuzz - Google's continuous fuzzing service for open source software," USENIX Association, 2017.
[35]. Fuzzit. [Online]. Available: https://github.com/fuzzitdev/fuzzit (accessed 24.09.2025).
[36]. cifuzz: Fuzz tests as easy as unit tests. [Online]. Available: https://github.com/CodeIntelligenceTesting/cifuzz (accessed 24.09.2025).
[37]. S. Warkentin, Getting started using Mayhem with continuous integration, 2020. [Online]. Available: https://www.brighttalk.com/webcast/17668/439580 (accessed 24.09.2025).
[38]. Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in 28th USENIX Security Symposium (USENIX Security 19), 2019, pp. 1967-1983.
[39]. S. Österlund, E. Geretto, A. Jemmett, E. Güler, P. Görz, T. Holz, C. Giuffrida, and H. Bos, "Collabfuzz: A framework for collaborative fuzzing," in Proceedings of the 14th European Workshop on Systems Security, 2021, pp. 1-7.
[40]. J. D. DeMott, R. J. Enbody, and W. F. Punch, "Towards an automatic exploit pipeline," in 2011 International Conference for Internet Technology and Secured Transactions, IEEE, 2011, pp. 323-329.
[41]. OSS-Fuzz issue report tracker. [Online]. Available: https://bugs.chromium.org/p/oss-fuzz/issues/list?q=-status%3AWontFix%2CDuplicate%20-component%3AInfra&can=1 (accessed 24.09.2025).
[42]. Google cloud platform. [Online]. Available: https://github.com/GoogleCloudPlatform (accessed 24.09.2025).
[43]. M. Copeland, J. Soh, A. Puca, M. Manning, and D. Gollob, "Microsoft azure," New York, NY, USA: Apress, pp. 3-26, 2015.
[44]. A crash course to Radamsa. [Online]. Available: https://gitlab.com/akihe/radamsa (accessed 24.09.2025).

[45]. Coverage-guided fuzz testing in GitLab. [Online]. Available: https://docs.gitlab.com/ee/user/application_security/coverage_fuzzing/ (accessed 24.09.2025).
[46]. D. Vyukov, syzbot: Automated kernel testing, 2018. [Online]. Available: https://lpc.events/event/2/contributions/237/attachments/61/71/syzbot_automated_kernel_testing.pdf (accessed 24.09.2025).
[47]. D. Parygina, A. Vishnyakov, and A. Fedotov, "Strong optimistic solving for dynamic symbolic execution," in Ivannikov Memorial Workshop (IVMEM), IEEE, 2022.
[48]. A. Niemetz and M. Preiner, "Bitwuzla at the SMT-COMP 2020," CoRR, vol. abs/2006.01621, 2020. arXiv: 2006.01621. [Online]. Available: https://arxiv.org/abs/2006.01621 (accessed 24.09.2025).
[49]. T. Wang, T. Wei, Z. Lin, and W. Zou, "IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution," in NDSS, 2009.
[50]. GDB 'exploitable' plugin. [Online]. Available: https://github.com/jfoote/exploitable (accessed 24.09.2025).
[51]. Sydr-Fuzz trophy list. [Online]. Available: https://github.com/ispras/oss-sydr-fuzz/blob/master/TROPHIES.md (accessed 24.09.2025).
[52]. GitLab: The one devops platform. [Online]. Available: https://about.gitlab.com/ (accessed 24.09.2025).
[53]. D. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

Информация об авторах / Information about authors

Алексей Вадимович ВИШНЯКОВ – кандидат физико-математических наук, закончил бакалавриат и магистратуру ВМК МГУ в 2020 году. Сфера научных интересов: компьютерная безопасность, жизненный цикл безопасной разработки (SDL), анализ бинарного кода, символьная интерпретация, фаззинг, автоматическое обнаружение ошибок и компиляторы.

Alexey Vadimovich VISHNYAKOV – Cand. Sci. (Phys.-Math.), obtained BSc degree and M.D. in the Faculty of Computational Mathematics and Cybernetics at Lomonosov Moscow State University. Research interests: computer security, security development lifecycle (SDL), binary analysis, symbolic execution, fuzzing, automatic error detection, and compilers.

Даниил Олегович КУЦ – кандидат технических наук. Сфера научных интересов: анализ бинарного кода, символьное выполнение, гибридный фаззинг.

Daniil Olegovich KUTS – Cand. Sci. (Phys.-Math.) since 2023. Research interests: binary analysis, symbolic execution, hybrid fuzzing.

Влада Игоревна ЛОГУНОВА – научный сотрудник лаборатории системного программирования и информационной безопасности Института системного программирования. Сфера научных интересов: динамический анализ, анализ бинарного кода, динамическая символьная интерпретация, гибридный фаззинг.

Vlada Igorevna LOGUNOVA – Research Fellow at the Laboratory of System Programming and Information Security of the Institute for System Programming. Research interests: dynamic analysis, binary code analysis, dynamic symbolic execution, hybrid fuzzing.

Дарья Алексеевна ПАРЫГИНА – магистр Московского государственного университета имени М.В. Ломоносова, старший лаборант Института системного программирования. Сфера научных интересов: символьное выполнение, гибридный фаззинг, направленный фаззинг.

Darya Alekseevna PARYGINA – master of Lomonosov Moscow State University, senior laborant of the Institute for System Programming of the RAS. Research interests: symbolic execution, hybrid fuzzing, directed fuzzing.

Илай Александрович КОБРИН – закончил бакалавриат и магистратуру ВМК МГУ в 2025 году. Сфера научных интересов: компьютерная безопасность, анализ бинарного кода, символьная интерпретация, фаззинг, операционные системы.

Eli Aleksandrovich KOBRIN obtained BSc degree and M.D. in the Faculty of Computational Mathematics and Cybernetics at Lomonosov Moscow State University. Research interests: computer security, binary analysis, symbolic execution, fuzzing, operating systems.

Георгий Анатольевич Савидов – закончил бакалавриат и магистратуру ВМК МГУ в 2025 году. Сфера научных интересов: компьютерная безопасность, анализ бинарного кода, символьная интерпретация, фаззинг, операционные системы.

Georgy Anatolievich Savidov obtained BSc degree and M.D. in the Faculty of Computational Mathematics and Cybernetics at Lomonosov Moscow State University. Research interests: computer security, binary analysis, symbolic execution, fuzzing, operating systems.

Андрей Николаевич ФЕДОТОВ – закончил НИЯУ МИФИ в 2013 году, кандидат технических наук с 2017 года. Сфера научных интересов: информационная безопасность, символьная интерпретация, оценка критичности ошибок, обратная инженерия, поиск ошибок, языки программирования, динамический анализ.

Andrey Nikolaevich FEDOTOV – Cand. Sci. (Tech.) since 2017, graduated from National Research Nuclear University MEPHI (Moscow Engineering Physics Institute) in 2013. Research interests: information security, symbolic execution, error severity estimation, reverse engineering, error search, programming languages, dynamic analysis.