DOI: 10.15514/ISPRAS-2023-35(4)-1

Обзор методов динамического анализа программного обеспечения

[1,2] *В.В. Кулямин*, ORCID: 0000-0003-3439-9534 <*kuliamin@ispras.ru*>

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

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

Аннотация. Данная статья представляет собой обзор методов динамического анализа программного обеспечения (ПО), в котором основное внимание уделено методам, имеющим инструментальную поддержку, нацеленным на проверку безопасности и защищенности и применимым к системному ПО. Подробно рассмотрены техники фаззинга, верификационного мониторинга и динамической символьной интерпретации. Методы и средства динамического анализа помеченных данных исключены из обзора из-за трудностей сбора технической информации о них. При рассмотрении фаззинга и динамической символьной интерпретации больше внимания уделено не отдельным инструментам, которых известно уже более 100, а техникам решения различных задач, возникающих при их работе. Также рассмотрены техники снижения эффективности фаззинга.

Ключевые слова: динамический анализ программного обеспечения; верификация; фаззинг; динамическая символьная интерпретация; верификационный мониторинг; противодействие фаззингу.

Для цитирования: Кулямин В.В. Обзор методов динамического анализа программного обеспечения. Труды ИСП РАН, том 35, вып. 4, 2023 г., стр. 7–44. DOI: 10.15514/ISPRAS–2023–35(4)–1.

Survey of Software Dynamic Analysis Methods

[1,2] *V.V. Kuliamin* ORCID: 0000-0003-3439-9534 <*kuliamin@ispras.ru*>

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

[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. The article presents a survey of software dynamic analysis methods. The main focus of the survey is on methods supported by tools, targeted on software security verification and applicable to system software. The survey examines in detail fuzzing and dynamic symbolic execution techniques. Dynamic taint data analysis is excluded due to difficulty of gathering technical details of its implementation. Review of fuzzing and dynamic symbolic execution is focused mostly on the techniques used in supporting tools, not on tools themselves, because their number exceeds 100 already. Also, the techniques of fuzzing counteraction are surveyed.

Keywords: software dynamic analysis; verification; fuzzing; dynamic symbolic execution; runtime verification; fuzzing counteraction.

For citation: Kuliamin V.V. Survey of Software Dynamic Analysis Methods. *Trudy ISP RAN/Proc. ISP RAS,* vol. 35, issue 4, 2023. pp. 7-44 (in Russian). DOI: 10.15514/ISPRAS-2023-35(4)-1.

1. Введение

Данный обзор посвящен методам и инструментам динамического анализа программного обеспечения (ПО), нацеленным, чаще всего, хотя и не исключительно, на обеспечение его защищенности и безопасности. Известно очень много различных видов и техник динамического анализа, а основным фокусом этого обзора являются методы, обладающие следующими характеристиками.

- Имеющие существенную инструментальную поддержку, позволяющую значительно снизить трудоемкость применения метода по сравнению с его выполнением вручную.
- Позволяющие оценивать защищенность и безопасность ПО. Техники анализа, не оказывающие существенного влияние на защищенность ПО и не способствующие систематическому выявлению ошибок и уязвимостей, не рассматриваются.
- Пригодные для оценки ПО как изолированного продукта. Например, методы обнаружения и анализа вторжений или аномального поведения [1,2], широко используемые для оценки защищенности крупномасштабных систем, в которых совместно работает множество различного программного обеспечения, тоже далее не рассматриваются.
- Достаточно широко применимые (или имеющие значимый потенциал применения) к системному ПО: к библиотекам, драйверам и ядрам операционных систем (ОС), к реализациям телекоммуникационных протоколов, к системам управления базами данных и ПО промежуточного уровня, к инструментам разработки и исполнения ПО (компиляторам, интерпретаторам и пр.). Специализированным методам анализа прикладного ПО уделяется меньше внимания.

Кроме того, методы динамического анализа помеченных данных исключены из обзора, помимо упоминания тех случаев, когда они используются в рамках инструментов, фокусирующихся на реализации других методов. Это сделано в связи с большим разнообразием таких методов и поддерживающих их инструментов, для которого, однако, в доступных источниках обычно имеются лишь частичные и не обладающие нужной систематичностью описания, опускающие множество важных технических деталей.

Далее изложение организовано следующим образом. В разделе «Основные понятия» определены основные рассматриваемые виды динамической верификации и динамического анализа. В следующих разделах последовательно рассматриваются техники и инструменты фаззинга, включая также техники противодействия ему, верификационного мониторинга и динамической символьной интерпретации. Заключение завершает статью.

2. Основные понятия

Динамический анализ (dynamic analysis, runtime analysis) объединяет все методы верификации и анализа программного обеспечения, использующие результаты его выполнения, включая исполнение отдельных модулей или групп модулей в специально созданных окружениях. При этом методы верификации, в качестве результата выдающие оценку качества ПО или, несколько уже, оценку его соответствия требованиям, делятся на две большие группы: тестирование и верификационный мониторинг. Остальные методы динамического анализа, которые достаточно разнообразны, часто служат вспомогательными техниками для методов этих двух типов.

У большинства рассмотренных далее видов динамического анализа есть соответствующие разновидности статического анализа, которые иногда также задействуются в качестве вспомогательных техник в инструментах, основной решаемой задачей которых является динамический анализ и/или верификация. С точки зрения эффективности поиска ошибок и контроля качества динамический анализ обладает недостатком, связанным с тем, что он

выполняется лишь для тех путей исполнения, которые задействуются в реальном выполнении ПО (или его компонентов). В то же время, достоинством динамических техник анализа является очень малое количество ложных срабатываний, при которых инструмент создает сообщение об возможных ошибках, которые в реальной работе никогда не происходят. Чаще всего, все ошибки, о которых сообщают инструменты динамического анализа, достижимы на каком-либо реальном сценарии работы проверяемого ПО.

Тестирование (testing) проверяет соответствие работы ПО требованиям на наборе заранее выбранных ситуаций.

Одной из основных задач при подготовке тестирования является выбор достаточно представительного набора ситуаций, чтобы по результатам работы в них можно было судить о соответствии ПО требованиям в целом, и построение на их основе ***тестов*** (tests). Каждый тест включает в себя описание используемой ситуации в виде некоторого сценария работы тестируемой системы (последовательности обращений к ее операциям/функциям, выполнения команд, использования элементов интерфейса, отсылок сообщений и т.д., с передачей сопутствующих данных) и набор проверок, выполняемых, чтобы убедиться, что система ведет себя корректно, в соответствии с требованиями. Ситуации, создаваемые в ходе выполнения тестов, называются ***тестовыми ситуациями***.

Для того чтобы оценивать представительность набора тестовых ситуаций, т.е. значимость набора ситуаций, в которые система попадает при выполнении тестов по сравнению со всеми возможными при ее работе ситуациями, и возможность судить на основе поведения системы в тестах о ее поведении вообще, вводятся ***критерии полноты тестирования*** (test adequacy criteria, test completeness criteria) [3] или ***критерии покрытия*** (test coverage criteria). Они обычно основаны на разбиении всех ситуаций, возможных при работе данного ПО, на некоторый набор классов или типов (не обязательно непересекающихся) и определении доли задействованных при выполнении тестов из этих классов ситуаций.

Типовые критерии покрытия основываются на задействованных при работе в некоторой ситуации элементах самого ПО, элементах требований к нему или гипотезах о возможных значимых ошибках, которые в данной ситуации могут произойти. Например, ***покрытие инструкций кода*** (code statement coverage) означает долю/процент инструкций, которые были выполнены в ходе работы тестов. ***Покрытие ветвлений кода*** (code branch coverage) означает процент веток в коде (возможных выполнений then и else в условных операторах if или различных случаев case, выполненных в операторах выбора switch, выполнений, проходящих внутрь цикла while и обходящих его, и пр.), которые выполнялись в ходе работы тестов. ***Покрытие требований*** (requirements coverage) означает процент требований (от всех сформулированных), которые должны были выполняться в затронутых тестами ситуациях. ***Покрытие состояний*** (state coverage) или ***покрытие переходов*** (transition coverage) в некоторой автоматной модели, описывающей поведение тестируемого ПО, означают, соответственно, процент состояний, в которых система побывала во время выполнения тестов, от всех в принципе достижимых состояний, и процент выполненных во время тестирования переходов из всех возможных.

При автоматизации тестирования стараются отслеживать достигнутое на каждом исполнении тестов покрытие, и определять повышение или снижение покрытия при очередном выполнении тестов по сравнению с предшествующими, чтобы иметь представление о возможном повышении или снижении значимости оценки качества ПО в целом по результатам тестирования. По этой причине важно, чтобы измерение покрытия по выбранному критерию легко автоматизировалось. Покрытия, основанные на простых элементах кода (покрытие инструкций или ветвлений), легче всего поддаются измерению за счет достаточно простой инструментации исходного или исполнимого кода и потому чаще всего используется на практике.

Для тестирования, нацеленного на оценку защищенности проверяемой системы, критически важным становится достижение как можно более высокого покрытия, чтобы значительно

снизить риски необнаружения возможных уязвимостей. При этом существенно возрастают усилия, необходимые для подготовки тестов, а также возрастает значимость автоматизации тестирования, означающей возможность автоматического исполнения тестов и как можно более детерминированного воспроизведения их результатов. В силу этих факторов в качестве одного из основных методов тестирования защищенности широкое распространение получил фаззинг.

Фаззинг (fuzzing) [4] или ***фаззинг-тестирование*** является частным случаем тестирования. В его рамках тестовые ситуации массово генерируются псевдослучайным и/или нацеленным образом, возможно, путем внесения псевдослучайных или нацеленных модификаций (мутаций) в небольшой заранее подготовленный набор исходных ситуаций (входных данных или сценариев). Проверяться в таких тестах может просто работоспособность ПО в построенных ситуациях (то, что оно не падает, не создает ошибок в системе или исключений) или более сложные свойства (корректность управления памятью, отсутствие обращений за границами буферов и пр.), обычно фиксируемые с помощью инструментации исходного или бинарного кода некоторыми проверками в тех местах, где такие свойства могут быть нарушены (см. далее о мониторинге).

Обычно, более эффективным вариантом фаззинга является ***фаззинг с обратной связью по покрытию*** (coverage feedback fuzzing), при котором сгенерированные тестовые данные прогоняются через инструмент, измеряющий покрытие, и из них отбираются лишь те, которые добавляют новые, не покрытые имеющимися тестами ситуации. Отобранные данные часто сохраняются в корпусе входных данных, чтобы использовать их мутации для дальнейшей генерации тестовых данных.

Другим продвинутым вариантом является ***фаззинг с использованием динамической символьной интерпретации***, в рамках которого тестовые данные генерируются не только случайным образом, а иногда вычисляются как решения систем символьных ограничений, получаемых как условия попадания в ту или иную ситуацию (выполнения заданной ветви или нарушения заданного ограничения в коде).

Использование обратной связи по покрытию, некоторых эвристик внесения мутаций и динамической символьной интерпретации иногда позволяет достаточно эффективно (с точки зрения быстрого получения высоких уровней покрытия) генерировать тесты для достаточно объемного кода, вскрывающие редко встречаемые ошибки. Бурное развитие техник и инструментов фаззинга за последние 15 лет и достигнутый в нем высокий уровень автоматизации сделали фаззинг неотъемлемым инструментом обеспечения защищенности и безопасности ПО.

Верификационный мониторинг или ***мониторинг утверждений/свойств*** (runtime verification) — это метод верификации ПО, при котором исходный или бинарный код проверяемого ПО подвергается инструментации, вставляющей в некоторые места код, проверяющий заданные свойства (в виде проверяемых утверждений, assertions) и либо заносящий записи о найденных нарушениях в некоторый журнал, либо просто прерывающий работу ПО при нарушения утверждения, после чего инструментированный код исполняется, чаще всего в обычном эксплуатационном режиме. Иногда такой инструментированный код исполняется в рамках тестов, соответственно, в этих тестах можно не проверять отдельно требования, связанные с зашитыми в инструментированный код утверждениями.

Динамическая символьная интерпретация или ***динамическое символическое выполнение*** (dynamic symbolic execution, DSE) является методом динамического анализа ПО, обычно используемым как составляющая часть других методов. *Символическая интерпретация* или *символическое выполнение* (symbolic execution) состоит в том, что код (исходный или бинарный) подвергается анализу, при котором входные, выходные и внутренние данные (переменные, объекты, регистры процессора или участки памяти) рассматриваются как символьные переменные, между которыми устанавливаются связи в виде символьных выражений (выражающих одни данные через другие) в ходе интерпретации кода как средства

построения и трансформации этих символьных выражений. *Динамическая символьная интерпретация* является символической интерпретацией, выполняемой в динамике, при исполнении анализируемого ПО, и поэтому, в качестве исходной точки обычно имеет некоторое конкретное выполнение ПО. При этом часть данных представляется одновременно конкретными значениями и символическими выражениями, что позволяет упрощать ряд получаемых ограничений и проводить анализ более эффективно. Динамическая символьная интерпретация используется как при фаззинге или обычной генерации тестов для извлечения символьных ограничений, решая которые можно получить тестовые данные, обеспечивающие попадание в нужную ситуацию или проявление некоторой ошибки, так при других видах анализа: выявлении дубликатов кода, выявлении обхода механизмов защиты и др.

Обычный ***анализ помеченных данных*** (taint data analysis) является методом анализа, нацеленным на выявление информационных потоков, порождаемых некоторой выделенной частью входных или выходных данных при работе ПО. Иногда такой анализ используется для выявления распространения влияния некоторой части входных данных (например, возможно, содержащих злонамеренно искаженные данные или эксплойт с целью запуска постороннего кода), иногда — для выявления влияния различных данных на выводимые результаты (например, возможность проявления в выходных результатах каких-либо непредназначенных для посторонних лиц данных: ключей, паролей, секретных данных и пр.). ***Динамический анализ помеченных данных*** (dynamic taint data analysis, DTA) [5,6] отличается от статического использованием результатов работы анализируемого ПО на некоторых наборах входных данных.

Помимо инструментов фаззинга, такой анализ часто используется для выявления подверженных атакам извне компонентов и интерфейсов ПО, возможности проникновения в систему специально испорченных данных и их влияния на сохраняемую информацию, информационных потоков, образуемых от конфиденциальной информации, возможностей компрометации или подмены паролей и ключей, для восстановления используемых алгоритмов, выявления компонентов, которые могут быть подвергнуты атаке извне, и пр.

В рамках самого анализа определяются помеченные элементы данных (чаще всего «пометка» означает просто принадлежность к множеству выделенных данных, т.е. может быть выражена одним битом, но иногда используются техники анализа, в которых метки могут быть сложными, как, например, в фаззере TaintScope [7]) и правила распространения пометок при выполнении различных инструкций в коде (какие операции могут создавать помеченные данные, какие могут снимать пометки, как пометки переносятся инструкциями, как сложные метки трансформируются инструкциями и т.д.). Анализ может быть построен на распространение меток вперед, в соответствии с последовательностью выполнения инструкций в коде, или может быть обратным, выполняемым в противоположном направлении.

Инструменты DTA достаточно разнообразны по конечному назначению: от общих фреймворков, поддерживающих проведение широкого множества видов анализа, до специальных инструментов, нацеленных на решение конкретных задач, например, проверки наличия конкретных видов уязвимостей в ПО, или использующих конкретный вид DTA в качестве вспомогательной задачи, например, для определения элементов входных данных, влияющих на проявление специфической ошибки.

Далее методы и инструменты динамического анализа помеченных данных отдельно не рассматриваются. Несмотря на их значительное разнообразие и долгую историю развития (достаточно зрелые для промышленного использования инструменты подобного типа появились еще в 2005 г.), для большинства из них довольно тяжело найти в открытых публикациях значимую информацию об используемых в рамках проводимого анализа технических решениях.

3. Фаззинг

Фаззинг [4] является разновидностью тестирования, в рамках которой тестовые ситуации массово генерируются псевдослучайным или нацеленным образом и тут же используются для выполнения тестов. Генерируемые тестовые ситуации при этом могут не являться ожидаемыми или валидными с точки зрения документации на тестируемое ПО и соображений «здравого смысла» по поводу его использования. Целью такого массового тестирования является выявление ошибок, в частном случае, уязвимостей, с помощью которых можно переключить исполнение на сторонний код. При невозможности выявить ошибки мерой успешности фаззинга служит высокий уровень покрытия проверяемого кода. Индикатором ошибки служит падение ПО или, при использовании дополнительных средств мониторинга, попадание в ситуацию, где мониторинг сообщает о некорректном поведении.

Тестовые ситуации могут генерироваться в общем случае в виде сценариев выполнения проверяемого ПО, но достаточно часто на практике используются только входные данные для одной или нескольких вызываемых функций или операций, иногда в виде файла или байтового массива (полноценные сценарии чаще возникают при тестировании протоколов или библиотек компонентов со сложным внутренним состоянием, существенно влияющим на выполняемые функции). Генерация может осуществляться псевдослучайным образом, может быть нацеленной с помощью заданных форматов или грамматик входных данных, может выполняться с помощью внесения случайных или нацеленных модификаций (мутаций) в ранее набранном корпусе исходных ситуаций (входных данных) или сценариев.

Поскольку критически важной для успешности фаззинга является массовость генерируемых тестов, он всегда выполняется с помощью инструментов, фаззеров.

Сам термин «фаззинг» (fuzz) был введен на семинаре Б. Миллера в университете Висконсина в 1988 г. Первый инструмент фаззинга [8] был создан на основе результатов работы этого семинара в 1990 г. Этот инструмент случайным образом генерировал строковые входные данные, что позволило обнаружить достаточно много ошибок в стандартных утилитах Unix. С тех пор фаззеры значительно эволюционировали, превратившись в одну из необходимых составляющих процесса разработки безопасного ПО. Агентство DARPA активно содействовало их развитию [9-11], организовывая соревнования между ними по эффективности выявления ошибок и уязвимостей. Крупные компании-разработчики ПО, такие как Cisco [12], Google [13-15], Microsoft [16,17], создали собственные линейки фаззеров, внедрили их в свои процессы разработки и тратят значительные усилия на их развитие и поддержание на современном уровне.

За прошедшие 30 с лишним лет было создано более сотни инструментов фаззинга. Оценить их количество и граф генеалогических связей между ними можно на сайте [18], где для большинства представленных инструментов есть ссылки на описывающие их статьи и Web-странички самих фаззеров. Опубликовано, как минимум, 3 книги [4,19,20], рассказывающие как о техниках фаззинга, так и о доступных инструментах. Однако, про некоторые инструменты крайне тяжело найти хоть какую-то значимую информацию. Ряд инструментов имеют только репозиторий с кодом и обрывочные инструкции по их использованию. Еще часть инструментов известны лишь по небольшим, в пределах 10 страниц, статьям, в которых авторам удается изложить только базовую идею инструмента (часто, совсем не оригинальную) и, иногда, какие-то данные о результатах его применения. Понимание специфики различных методов фаззинга затрудняют также использование разнородной терминологии в разных статьях (некоторые техники называются по-разному разработчиками различных инструментов, иногда, наоборот, один термин используется для достаточно сильно отличающихся техник или явлений) и ошибки в обзорах [21], иногда вводящие читателя в заблуждение о свойствах рассматриваемых инструментов.

В данном обзоре использован материал нескольких достаточно недавних обзоров инструментов фаззинга [22-24], обзор из работы [25] и дополнительная информация из описаний самих инструментов. Целью являлось рассмотрение основных техник реализации

компонентов инструментов фаззинга, обзор наиболее характерных примеров таких инструментов, обладающих значимой спецификой по сравнению с другими.

3.1 Структура обобщенного фаззера

Для более ясного понимания проблем создания эффективного фаззера необходимо иметь представление о решаемых им частных задачах и общих подходах к проектированию подобного инструмента. Обобщенный фаззер может быть представлен состоящим из компонентов, представленных на Рис. 1. В каждом конкретном инструменте некоторые из указанных компонентов могут отсутствовать, некоторые могут быть объединены или, наоборот, разбиты на более мелкие модули.

- *Препроцессор* получает на вход *тестируемое ПО* (system under test, SUT) в виде исходного кода или в виде исполнимого файла. Он может выполнять статический анализ кода SUT (полученная информация сохраняется в конфигурации, обычно в виде данных о возможных элементах покрытия, о выполняемых SUT проверках входных данных и пр.), его инструментацию (вставляя в код SUT конструкции, выполняющие функции монитора и тестового оракула, иногда также устраняя из кода несущественные для тестирования основной его функциональности элементы), и, если нужно, сборку, готовя ее к выполнению. В тех случаях, когда он предварительно инструментирует код, результатом его работы является *инструментированная SUT*. Иногда препроцессор предоставляет некоторую часть среды выполнения, осуществляющую инструментацию SUT на лету.
- *Обработчик конфигураций* получает на вход *конфигурации* инструмента, которые представляют собой корпус ранее отобранных или полученных наборов тестовых данных или сценариев, размеченный дополнительной информацией, используемой для их приоритезации и выбора алгоритма обработки, информацию о возможных и достигнутых элементах покрытия, а также, возможно, включают историю использованных ранее параметров работы инструмента. В его задачи входит первичная обработка входящих в конфигурацию исходных данных, чтобы выделить информацию, используемую далее для нацеленной генерации подходящих входных данных. Он также выполняет выбор конфигурации, а также набора значений параметров, управляющих работой инструмента, для текущего прохода фаззинга. Он может также модифицировать корпус данных, переупорядочивая их и меняя разметку, для повышения эффективности следующих проходов фаззинга.
- *Генератор данных* создает на основе обработанной конфигурации новые тестовые данные для очередного запуска SUT (отметим, что здесь мы генерацией называем любой способ построения данных, часто в работах по фаззингу термин «генерация» применяется только к методам, не использующим другие, ранее подготовленные входные данные). Генератор данных может использовать техники псевдослучайной генерации данных, небольшие изменения (мутации) в известных данных, эвристики поиска наиболее удачных таких мутаций (с точки зрения повышения вероятности выявления ошибки или покрытия ранее непокрытых участков кода), нацеленные техники генерации данных в соответствии с заданными или выявленными форматами входных данных SUT или грамматиками, получение данных с помощью разрешения ограничений, выявленных статическим анализом или динамической символьной интерпретацией, и т.д.
- *Монитор* отслеживает информацию об очередном исполнении тестируемого ПО, и передает ее для обработки. Монитор работает в рамках инструментированной SUT или может быть частью среды выполнения SUT.
- *Обработчик прохода* обрабатывает информацию, полученную монитором, и сохраняет часть результатов в конфигурации для более эффективного нацеливания дальнейшей

генерации. В его рамках может выполняться сбор данных о достигнутом покрытии, динамическое символьное выполнение, анализ помеченных данных и пр.

- *Тестовый оракул* выявляет ошибки, ситуации некорректного поведения тестируемого ПО. Тестовый оракул может быть обработчиком аварийного завершения процесса, в котором работает SUT, или он может быть частью инструментированной SUT, представляющей собой встроенный код обработки ошибок.
  Достаточно часто монитор и тестовый оракул представляют собой с трудом разделимые части инструментации, но для удобства понимания общих задач фаззинга мы рассматриваем их отдельно.
- *Обработчик ошибок* получает информацию об ошибке и, возможно, преобразует ее для дальнейшего использования, в частности, для использования в виде уязвимости (для этого может использоваться информация от обработчика прохода). Часть информации об ошибках может вноситься в конфигурацию.

Стоит отметить, что представленное здесь разбиение фаззера на компоненты предназначено для анализа различных используемых техник, для практической реализации инструментов оно может оказаться неудобным. Разбиение фаззера на компоненты с точки зрения эффективной и конфигурируемой реализации таких инструментов рассмотрено в работе [26].

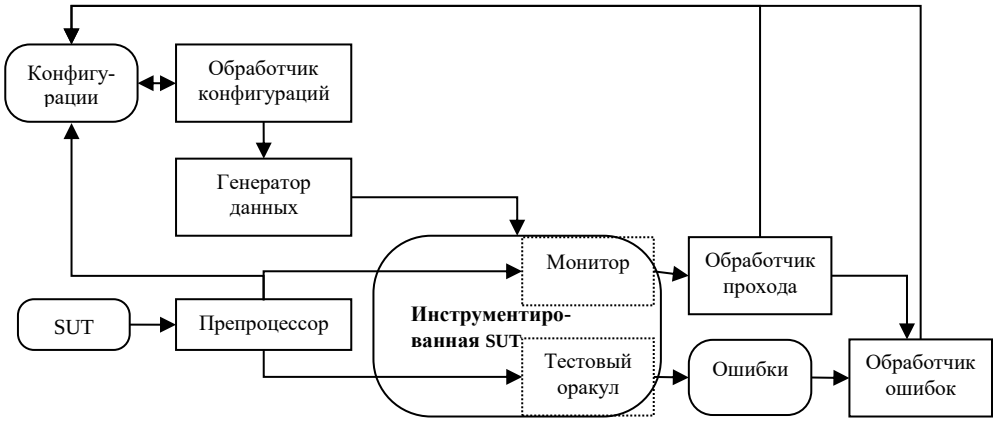

Рис. 1. Структура обобщенного инструмента фаззинга
Fig. 1. Generic fuzzer structure

3.2 Техники, используемые фаззерами

Часто используется классификация фаззеров на фаззеры черного ящика, белого ящика и серого ящика. *Фаззером черного ящика* (black box fuzzer) обычно называют инструмент фаззинга, не имеющий монитора и обработчика прохода, а также не имеющий передачи данных о структуре кода SUT из препроцессора в конфигурацию. При этом информация о найденных ошибках может использоваться для приоретизации входных данных для фаззинга. *Фаззером белого ящика* (white box fuzzer) называют инструмент, активно использующий информацию о структуре кода SUT (получаемую из монитора или из статического анализе в препроцессоре) для построения тестовых данных, чаще всего с помощью ресурсоемких методов, таких как динамическая символьная интерпретация. *Фаззером серого ящика* (grey box fuzzer) обычно называют фаззер, в котором монитор собирает лишь небольшую часть информации о коде SUT, например, только данные о достигаемом покрытии.

К сожалению, из приведенных выше определений (которым следует большинство источников) границы между этими типами фаззеров не выявляются достаточно четко.

Неясно, как именно различать фаззеры белого и серого ящиков, когда речь идет не об информации, указанной в определениях выше, а занимающей промежуточное положение между данными об элементах покрытия и полной информацией об условиях всех ветвлений. Например, фаззеры, использующие анализ помеченных данных, поскольку он менее ресурсоемкий и требует лишь часть информации о структуре кода, разные авторы относят иногда к белому ящику, иногда к серому. Более того, можно представить фаззер, основанный на случайной генерации и мутациях входных данных, как стандартные фаззеры черного ящика, но, использующий дополнительно информацию о коде разбора данных в SUT (и только ее), чтобы эффективно преодолевать места, где используются проверочные суммы, хеш-коды, магические числа (для контроля форматов и целостности входных данных). Такой инструмент не относится к фаззерам черного ящика, так как использует статический анализ кода SUT, однако почти все решаемые им задачи и соответствующие проектные решения, будут полностью аналогичны задачам и решениям в рамках обычных фаззеров черного ящика (поскольку он использует дополнительную информацию только чтобы успешно пройти код разбора входных данных в SUT).

В силу указанных причин далее эта классификация используется только для тех случаев, где имеется консенсус относительно ее применения. В спорных и промежуточных случаях она не приносит хорошего понимания возникающих проблем, там нужны более информативные описания.

Далее мы рассматриваем различные методы решения задач, возникающих при работе компонентов обобщенного фаззера.

3.2.1 Предобработка

Предобработка выполняется препроцессором. Она может использовать следующие техники.

- Инструментация исходного или исполнимого кода SUT.
  Инструментация исходного кода (source code instrumentation) используется при желании иметь меньше накладных расходов во время проведения фаззинга, однако, она не позволяет задействовать информацию из динамически загружаемых библиотек. *Инструментация исполнимого кода* (binary code instrumentation) обычно рассматривается как более ресурсоемкая, но более гибкая. Она может быть как статической, так и динамической, т.е. использующей специфическую среду выполнения для мониторинга определенных инструкций, вызовов или событий в исполняемом коде. Значительное число фаззеров используют в качестве средств динамической инструментации Pin [27] или QEMU [28], часто также используются Dyninst [29,30] или DynamoRIO [31,32]. Один фаззер может поддерживать разные техники. Например, AFL [33,34] может использовать статическую инструментацию исходного кода или динамическую с помощью QEMU, которая также может быть нацелена только на код SUT или на этот код вместе с используемыми библиотеками.
- Построение оберток.
  В некоторых случаях (библиотеки, драйвера устройств и пр.) непосредственный фаззинг самой SUT существенно затруднен, и для его выполнения необходим некоторый оберточный код, который обращается к функциям SUT, являющимся целью фаззинга. Такой код называется *оберткой* (английский термин — driver). Достаточно часто обертка разрабатывается вручную, поскольку ее можно дальше использовать без изменений, пока неизменным остается интерфейс SUT. Ряд фаззеров (IOCTL Fuzzer [35], IoTFuzzer [36]) могут генерировать простые обертки. Имеется ряд дополнительных инструментов (FUDGE [37], FuzzGen [38], IntelliGen [39]), позволяющих генерировать иногда достаточно сложные обертки, учитывающие ограничения на обращения к целевым функциям. Результаты их работы могут использоваться совместно со многими разными фаззерами.

- Сокращение исполняемого кода.
  При тестировании объемного ПО иногда имеет смысл подвергать фаззингу только небольшую важную его часть и, по возможности, экономить ресурсы на инициализации остальной части системы при каждом проходе фаазинга. Такая техника — восстановление процесса, в котором работает SUT, для очередного прохода фаззинга в некотором промежуточном состоянии, где сам фаззинг может быть повторен, но не нужно повторять действия по инициализации процесса и проходу в это состояние, называется *сокращением исполняемого кода* (соответствующий английский термин — in-memory fuzzing). Поскольку эта техника связана с существенным преобразованием исполняемой части SUT, она отнесена к предобработке.
  Инструмент Cyberdyne [11], наряду с фаззером, содержит вспомогательный инструмент для сброса и восстановления состояния процесса, GRR [40], который может быть использован и другими фаззерами. AFL [33,34], libFuzzer [41] и honggfuzz [42] сами поддерживают такой режим работы. В AFL он назван persistent mode, при нем проходы фаззинга выполняются, за счет вызова fork, начиная с некоторого состояния процесса SUT, без остановки и реинициализации. При этом за то же время можно выполнить больше проходов фаззинга, однако возможны ситуации, где на возникающие ошибки (или выполняемые ветвления в коде) оказывают влияние побочные эффекты от предшествующих проходов, из-за чего эти ошибки (или покрытие соответствующих ветвей) крайне тяжело воспроизводить.
- Инструментация для (де)рандомизации параллелизма.
  В нескольких работах [43-47] рассматривается инструментация (и использующие ее фаззеры), позволяющая сделать детерминированным выполнение параллельных нитей (threads) в SUT или, наоборот, внести больше случайности в переключения между ними. Первое позволяет сделать воспроизводимыми ошибки, связанные с различным порядком выполнения нитей, второе обеспечивает покрытие при фаззинге большего количества таких порядков, что позволяет найти больше ошибок.
- Статический анализ кода SUT на этапе предобработки.
  Статический анализ кода SUT (все равно, исходного или исполнимого) на этапе предобработки может быть нацелен на выявление элементов покрытия и условий их достижения, выявление возможных мест возникновения ошибок, определение несущественных ветвлений, уводящих от основного пути выполнения SUT (вычисление хеш-кодов, проверка магических чисел и пр.), а также определение влияния элементов входных данных на исполняемые пути в коде.
  BuzzFuzz [48] анализирует исходный код SUT и на основе указанного ему списка функций, вызов которых может привести к ошибке, выявляет места их вызова и проводит анализ помеченных данных, нацеленный на построение входных данных, приводящих к вызову этих функций с некорректными значениями параметров.
  Dowser [49] с помощью статического анализа выявляет места возможных переполнений буфера – обращения к указателям и массивам внутри циклов, – чтобы затем с помощью динамического анализа помеченных данных и символического исполнения подобрать данные, приводящие к некорректному обращению.
  GRT [50] и VUzzer [51] оба используют как статический, так и динамический анализ (последний - при обработке данных мониторинга на проходах фаззинга, а не в рамках предобработки). Статический анализ, в частности, выявляет константы, которые используются как элементы входных данных, и, в случае GRT, информацию о возможной модификации данных внутри методов.
- Динамический анализ кода SUT на этапе предобработки.
  Этот анализ может быть нацелен на решение тех же задач, что и статический. Например, TaintScope [7] помощью бинарной инструментации выполняет два вида анализа: эвристическое выделение несущественных ветвлений, которые отделяют

некорректные входные данные, и анализ помеченных данных, нацеленный на выявление элементов входных данных, влияющих на попадание в место возможной ошибки в коде.

- Модификация SUT для облегчения нахождения входных данных, позволяющих добраться до выполнения ее основной функциональности. То есть, для обхода несущественных для основной функциональности ветвлений, связанные с проверкой целостности данных, магических чисел, вычислением хеш-кодов и пр. В литературе пути исполнения кода, связанные с основной функциональностью, иногда называются hot paths, а несущественные ветвления и связанные с ними пути — cold paths.
  Примером является используемая в TaintScope [7] и T-Fuzz [52] инструментация. В первом случае несущественные ветвления определяются препроцессором на основе эвристик и обходятся в динамике, пока не обнаруживаются (частичные) входные данные, приводящие к ошибке. Далее эти данные дополняются до полного набора входных данных, обеспечивающего проход до места проявления ошибки. Во втором случае несущественные ветви обнаруживаются во время выполнения фаззинга за счет того, что они тормозят рост покрытия ветвей на генерируемых за счет простых мутаций данных, и модификация SUT происходит прямо во время фаззинга.

3.2.2 Упорядочивание и выбор исходных данных

В рамках обработчика конфигураций выполняется выбор конфигурации (scheduling) для текущего прохода фаззинга, а также упорядочивание и модификация данных конфигураций для повышения эффективности дальнейших проходов. При этом используются следующие техники.

- Выбор пула исходных данных.
  Исходные данные обычно выбираются из набора тестов, сопровождающих SUT. Достаточно часто для получения необходимого разнообразия используют поиск данных в нужном формате в Интернет, рассматривая отдельно имеющиеся известные открытые наборы данных и репозитории открытых проектов (например, для формата видео mpeg есть репозиторий проекта FFmpeg [53]). Иногда применяется конвертация данных в необходимый формат из других форматов.
- Упорядочивание и выбор исходных данных на основе информации о ранее найденных ошибках и времени выполнения фаззинга. Такие методы используются в фаззерах черного ящика, не имеющих доступа к другой информации о выбираемых данных.
  Одним из наиболее развитых примеров фаззеров, использующих подобные техники, является CERT BFF [54], сами техники описаны в работах [55,56].
  Наиболее простая техника состоит в том, что выбор исходных данных для генерации из них новых выполняется чаще, если ранее выбор этих же данных чаще приводил к обнаружению ошибки. Также вероятность выбора может определяться ранее измеренным временем прохода фаззинга на этих данных, так, чтобы фаззер, по возможности, тратил некоторое фиксированное время на обработку заданного набора данных (учитывая, что обработка одного набора может содержать несколько проходов). Еще один учитываемый фактор — количество использований набора.
- Упорядочивание и выбор исходных данных на основе информации о достигаемом на них покрытии.
  AFL [33,34] является одним из первых примеров, использующих подобные техники. Он поддерживает пул исходных данных, который при генерации новых подвергается модификации в соответствии с эволюционным алгоритмом внесения мутаций, скрещивания и отбора. При отборе функция приспособленности учитывает покрытие ветвей в коде, размер самих данных и время прохождения фаззинга для них. Далее AFL отбирает некоторое количество наиболее приспособленных наборов, и для каждого из них выполняется некоторое фиксированное количество проходов фаззинга.
  AFLFast [57] добавляет еще несколько эвристик: среди наборов данных, покрывающих

одну и ту же ветвь, выбирается использовавшийся реже, среди одинаково редко используемых — тот, который покрывает набор ветвей, использовавшийся реже. Кроме того, количество проходов, использующих один набор может изменяться, так, чтобы реже использовавшиеся наборы имели больше проходов фаззинга на них.

- Повышение эффективности пула данных (повышения вероятности обнаружения ошибок или повышения покрытия) на основе эволюционной стратегии.
  Несколько фаззеров используют эволюционные алгоритмы, чтобы в ходе фаззинга модифицировать пул исходных данных. Помимо ранее указанных AFL и AFLFast, это syzkaller [58] и рассматриваемые далее. libFuzzer [41], honggfuzz [42], go-fuzz [59] и Steelix [60] учитывают в функции приспособленности количество операций сравнения, которое смог пройти набор данных. Angora [61] вносит в функцию приспособленности на основе покрытия ветвей в коде зависимости от контекста (места, из которого была вызвана функция, содержащая эти ветви). В VUzzer [51] функция приспособленности учитывает покрываемые ветви с весами, отражающими вероятность того, что ветвь является ответвлением от пути, на котором выполняется основная функциональность, на обработку какой-либо ошибки, чтобы снизить вес несущественных ветвей.
- Сокращение пула данных на основе получаемого покрытия.
  AFL может выбрасывать некоторые наборы данных из пула, если на них достигаются те же ветви, что и на меньшем количестве из оставшихся наборов. Cyberdyne [11] пытается поддерживать минимальный пул, обеспечивающий максимальное покрытие.
- Нацеливание на более частое использование данных, покрывающих определенные ветви или участки кода с помощью оптимизационных алгоритмов. Такие техники применены, например, в AFLGo [62] и QTEP [63]. В последнем для определения проблемных участков кода, которые нужно покрывать чаще, используется статический анализ и статистическая модель, оценивающая вероятность ошибки на участке кода. Для итеративного приближения к покрытию заданных элементов кода используется, например, алгоритм симуляции отжига.

3.2.3 Генерация входных данных

В данном разделе рассматриваются методы генерации данных для очередного прохода фаззинга. Обычно все такие методы делят на *генеративные*, каким-либо образом строящие данные заданной структуры без опоры на имеющиеся примеры таких же данных, и *мутационные*, строящие данные из некоторых заданных при помощи внесения в них небольших модификаций, или при помощи разбиения их на блоки и комбинирования этих блоков. При генерации очередных тестовых данных в фаззерах применяются следующие методы.

- Использование моделей входных данных или сценариев обращений к SUT, определяющих структуру и правила их построения, вместе с возможными связями и ограничениями.
  - Использование явно описанной модели.
    Такая модель может быть задана в виде грамматики (Peach [64], Nautilus [65]), может быть описана в виде структур данных на основе некоторого API (SPIKE [66,67], Sulley [68]). Фаззеры, предназначенные для тестирования протоколов, могут использовать спецификации протоколов в виде описания структуры сообщений и автоматной модели, задающей их возможные последовательности (PROTOS [69], SNOOZE [70], KiF [71], T-Fuzz [72], *не путать с другим* T-Fuzz [52]!). Фаззеры для API ядра ОС (Trinity [73], KernelFuzzer [74], syzkaller [58]) используют шаблоны обращений к системным вызовам.
    Большинство использующих модели входных данных инструментов относятся к фаззерам черного ящика, однако, есть фаззеры [75-77], совмещающие использование

грамматик и разрешение ограничений на основе результатов символической интерпретации.

Некоторые фаззеры, нацеленные на тестирование обработчиков специфических языков, имеют встроенные модели в виде грамматик этих языков (DOMFuzz [78] для тестирования, работы с DOM, jsfunfuzz [79] для тестирования обработки JavaScript). Также встроенные модели протоколов иногда используются в фаззерах, нацеленных на специфические протоколы ([80], tlsfuzzer [81], TLS-Attacker [82]).

- Построение модели в ходе работы.

  Модель структуры входных данных или сценариев вызовов может строиться в самом начале работы, до фаззинга, или достаиваться и модифицироваться в процессе фаззинга. Обработка входных данных в начале работы относится к задачам обработчика конфигурации, но из-за близости с другими техниками использования модели входных данных мы рассматриваем ее в этом разделе.

  Техника первого типа реализована в следующих инструментах. Skyfire [83] получает на вход исходную грамматику и корпус данных, удовлетворяющих ей, после чего строит вероятностную контекстно-зависимую грамматику, учитывающую как синтаксис данных, так и выявляемые на корпусе ограничения на них. После этого уже эта модель используется для генерации данных фаззинга. TestMiner [84] анализирует большой корпус данных для выделения литералов, с использованием которых затем генерируются новые тестовые данные. CodeAlchemist [85] выделяет из кода на JavaScript такие блоки, чтобы затем, комбинируя их, получать семантически корректные программы. IMF [86] с помощью анализа логов системных вызовов строит модель возможных последовательностей обращений к ним. Learn&Fuzz [87] строит модель входных данных с помощью нейросети. Еще одна техника использования нейросетей для моделирования текстовых входных данных описана в [88].

  Построение модели входных данных в виде грамматики во время работы реализовано в [89] и в GLADE [90]. Техники построения модели протокола в процессе работы использованы в работе [91] и в фаззере PULSAR [92].

- Техники построения входных данных
  - Псевдослучайная генерация данных заданного типа.

    Эта техника обычно используется лишь на числовых данных, небольших строках и байтовых массивах.
  - Использование экстремальных данных заданного типа.

    Экстремальными могут считаться самые маленькие или самые большие представимые в рамках типа числа, байтовые массивы, целиком заполненные нулями или единицами, даты и времена, связанные с особенностями календарей (например, 29.02 и 31.12 в високосные года) или полным исчерпанием используемой для хранения даты памяти, и пр.
  - Получение новых данных с помощью небольших модификаций (мутаций) имеющихся.

    Одна из наиболее широко используемых техник мутации — обращение некоторого количества бит в исходных данных. Она используется, например, в AFL [33,34], honggfuzz [42], BFF [54], radamsa [93], zzuf [94]. В некоторых случаях количество обращаемых бит выбирается случайно. В SymFuzz [95] некоторые обращаемые биты вычисляются на основе анализа кода, общее их количество выбирается в зависимости от SUT и изменяемых данных.

    Другая техника внесения мутаций — интерпретация блока данных как целого числа и прибавление к нему или вычитание из него небольшого числа. Использована в AFL и honggfuzz.

    Часто используются блоковые мутации — вставка или замена случайно выбранного

блока случайно сгенерированным или выбранным из другого набора данных, удаление случайно выбранного блока, перестановка местами двух случайно выбранных блоков. Такие операции используются в AFL, radamsa, honggfuzz, libFuzzer [41].

- Внесение мутаций в программу генерации данных и генерация входных данных с помощью полученных мутантов. Такая техника использована в MutaGen [96].
- Разрешение ограничений (constraint solving).
  Символьная интерпретация (DSE) позволяет выявлять ограничения на входные данные, удовлетворение которых приводит к выполнению определенного элемента покрытия (инструкции или ветви) или к нарушению условия корректности в некотором месте кода, которое соответствует определенной ошибке. Чтобы получить входные данные, удовлетворяющие выявленному набору ограничений, фаззер, использующий DSE, обычно обращается к SMT-решателю [97]. SMT-решатели (SMT-solvers) способны достаточно эффективно находить данные, удовлетворяющие систему ограничений над некоторой теорией (обычно, набор поддерживаемых теорий включает целочисленную арифметику, сравнения чисел, операции над битовыми векторами, возможность указывать объекты по ссылкам и пр.) или показывать, что заданная система неразрешима. В некоторых фаззерах для ограничений специфических видов используются внутренние реализации решателей.

3.2.4 Обработка данных о проходе

Рассматриваемые в этом разделе техники выполняются при обработке данных из монитора о текущем проходе фаззинга.

- Сбор данных о покрытии.
  Фаззер, использующий данные о достигнутом покрытии для его расширения, в рамках монитора отслеживает, какие элементы целевого критерия покрытия достигаются на данном проходе, и сохраняет эту информацию в конфигурации.
  AFL [33,34] имеет в качестве целевого покрытие ветвей в коде, однако, для идентификации ветвей использует некоторые случайно сгенерированные числа, из-за чего время от времени могут происходит коллизии (идентификаторы разных ветвей могут совпадать). В рамках CollAFL [98] предложено решение, позволяющее избавиться от коллизий без значительного дополнительного расхода ресурсов.
  libFuzzer [41] и syzkaller [58] в качестве целевого используют покрытие базовых блоков (что эквивалентно покрытию инструкций). honggfuzz [42] позволяет выбирать между покрытиями базовых блоков и ветвлений.
  Angora [61] может использовать в качестве критерия покрытия покрытие ветвей с учетом стека (т.е. дополнительной информации о вызывающих функциях).
  Обычно, более сложные и детальные критерии покрытия позволяют выявлять более «хитрые» ошибки, однако их отслеживание и хранение информации о покрытых ситуациях существенно усложняется при росте детализации.
- Динамическая символьная интерпретация (DSE).
  Динамическая символьная интерпретация используется в некоторых фаззерах на этапе обработки прохода для сбора информации о покрытых ветвлениях, их условиях, и об условиях возможного проявления ошибок, которые далее используются для генерации тестовых данных, обеспечивающих покрытие непокрытых ветвлений или попадание в ситуацию возможной ошибки. Иногда фаззинг, использующий комбинацию из DSE и других техник, называют гибридным (hybrid fuzzing) [99].
  Одним из первых фаззеров, использующих DSE, стал CUTE [100], лежащая в его основе идея применения DSE для генерации тестов описана немного раньше в работе [101].
  Наиболее важные примеры фаззеров, использующих DSE, включают KLEE [102], SAGE [103,104], GWF [75], S^2E [105], Mayhem [106], Dowser [49], BORG [107], MoWF [76],

Driller [10], QSYM [108] (хотя разработчики части из них не используют термин «фаззинг» при описании своих инструментов). Можно также упомянуть ИСП Crusher [109,110], способный использовать DSE с помощью среды Sydr [25,111]. TaintScope [7] использует DSE только для того, чтобы выявить полный набор входных данных, необходимый для достижения ошибки, после того как был выявлен частичный набор данных, приводящей к ней при игнорировании несущественных ветвлений.

Более подробно различные проблемы и техники реализации DSE рассмотрены в посвященном ей разделе ниже.

- Динамический анализ помеченных данных.

  Динамический анализ помеченных данных используется на этапе обработки прохода для выявления элементов входных данных, влияющих на выполнение ветвления в коде или на возникновение ошибки.

  Выше в разделе о предобработке есть примеры использования DTA в препроцессоре.

  Dowser [49] использует DTA для выявления элементов входных данных, которые влияют на возможные операции разыменования указателей, затем применяет DSE для вычисления ограничений на эти элементы и их разрешение для получения соответствующих входных данных.

  GRT [50], SYMFuzz [95], BORG [107], VUzzer [51] и Angora [61] используют DTA для выявления элементов входных данных, влияющих на выполнение ветвей в коде.

  REDQUEEN [112] с помощью DTA выявляет константы (магические числа и др.), использование которых во входных данных позволяет избежать выполнения несущественных ветвей.

3.2.5 Обработка ошибок

В этом разделе рассматриваются техники построения тестовых оракулов и техники обработки ошибок, используемые в инструментах фаззинга.

- Обнаружение ошибок.

  Обнаружение ошибок часто происходит за счет фиксации падения тестируемого ПО. Однако, для многих ошибок само исполнение кода, их содержащего, может не приводить к разрушению выполняющегося процесса, поэтому часто прибегают к дополнительной инструментации (исходного или бинарного кода), позволяющей выявить некорректное поведение. Такая инструментация внедряет в код разного рода утверждения (assertions) и проверки, которые демонстрируют ошибочную ситуацию.

  Более подробно разновидности проверок и средств для выявления ошибок при выполнении рассмотрены ниже в разделе о верификационном мониторинге.

- Обработка ошибок.

  В рамках фаззинга обработка данных об ошибках (англоязычный термин, triage, означает также сортировку раненых или пострадавших по степени полученного ими ущерба) иногда включает модификацию разметки в конфигурации (чтобы отразить риск возникновения ошибки для использованных входных данных — см. выше раздел об упорядочении и выборе исходных данных), выявление дубликатов (deduplication), минимизацию данных для выявления ошибки, оценку возможности использования выявленной уязвимости (exploitability assessment). Для решения этих задач фаззеры также могут использовать специализированные инструменты, такие как CASR [113,114].

  - Выявление дубликатов.

    Для выявления дубликатов ошибок, т.е. для определения того, что на разных проходах фаззинга фиксируется одна и та же ошибка, используются 3 техники: хеширование стека (stack backtrace hashing), выявление дубликатов на основе покрытия (coverage-based deduplication) и семантическое выявление дубликатов (semantic-aware deduplication).

    Хеширование стека [115] использует эвристику, утверждающую, что если несколько

последних вызванных функций в стеке перед возникновением двух ошибок одинаковы, то это одна и та же ошибка. Глубина обрезания стека (сколько последних вызванных функций сравнивать) может быть разной: в [56,115] используется 3, CERT BFF [54] использует 5, а MutaGen [96] — разные числа в разных случаях. Ясно, однако, что эта эвристика не аккуратна, она может определять разные ошибки как одну и наоборот, одну как различные.

Выявление дубликатов на основе покрытия используется в AFL [33,34]. Он считает ошибку уникальной, если при ее возникновении была покрыта ветвь, которая не покрывается в других случаях, или, наоборот, не покрывается ветвь, покрытая в других случаях.

Семантическое выявление дубликатов реализовано в специализированном инструменте RETracer [116], который использует обратный анализ помеченных данных от дампа памяти, полученного при возникновении ошибки.

- Минимизация данных для выявления ошибки.
  Такая техника реализована в AFL и BFF. В первом случае фаззер пытается уменьшить объем исходных данных и обнулить как можно большую их часть, при этом сохраняя выявленную ошибку. Во втором случае минимизируется разница в битах между выявляющими ошибку данными и теми валидными исходными данными, из которых была получен набор, выявивший ее.
  Иногда можно использовать специализированный инструмент, например, [117], для минимизации данных, выявляющих ошибку, после ее получения.
- Оценка возможности использования выявленной уязвимости.
  Такая оценка может выносится на основе эвристик [118], считающих что деление на 0 трудно превратить в программу, исполняющую посторонний код, а переполнение буфера — достаточно легко. Mayhem [106] в некоторых случаях позволяет по найденной ошибке сгенерировать такую программу (эксплойт).

3.3 Примеры фаззеров по областям применения

В данном разделе рассматриваются важные примеры фаззеров, сгруппированные по областям их применения.

- Фаззеры общего назначения. Они не имеют выраженной специфичной области использования, чаще используются для тестирования общих приложений.
  - Peach [64] (2004). Один из первых широко известных фаззеров черного ящика. Применялся для тестирования общего ПО, протоколов, встроенных устройств. Использует грамматики в специализированном формате (Peach Pits) для описания структуры корректных входных данных. Для генерации данных используются псевдослучайные генераторы и мутации. Алгоритмы генерации данных и внесения мутаций могут быть настроены. Является основой для нескольких других фаззеров, включая honggfuzz [42].
    Код открыт, с 2013 г. не развивается.
  - KLEE [102] (2008). Один из первых промышленно применимых фаззеров на основе DSE. Разработан в университете Стэнфорда, является развитием более раннего инструмента EXE [119]. Использует инструментацию исходного кода на уровне промежуточного представления LLVM. Последовательно симулирует выполнение обнаруживаемых путей в коде, используя fork для снижения накладных расходов при переключении на выполнение нового пути.
    Код открыт [120], продолжает развиваться.
  - SAGE [103,104] (2008). Тоже один из первых промышленно применимых фаззеров на основе динамической символьной интерпретации. Использует инструментацию бинарного кода платформы x86. Разработан в Microsoft, стал стандартной частью

процесса разработки для ряда приложений этой компании. Комбинирует случайную генерацию входных данных и эвристики поиска данных, нацеленные на повышение покрытия.

- American Fuzzy Lop, AFL [33,34] (2013). Широко известный фаззер, использующий покрытие ветвей для нацеливания, эволюционный алгоритм и мутации исходных данных для генерации новых данных на базе стартового пула. Реализует сокращение исполняемого кода за счет запуска новых проходов фаззинга с помощью fork. Является основой для большого числа других фаззеров.
  Код открыт [33], с 2017 г. не развивается, поддержка передана в Google [34], там нет развития с 2021 г. Активно развивается более продвинутая версия AFL++ [121,122], тоже с открытым кодом.
  Часто используемыми фаззерами, развивающими идеи AFL, являются также libFuzzer [41] и honggfuzz [42].

- Фаззеры для компонентов ядер операционных систем.
  - Trinity [73] (2004). Первый фаззер черного ящика для системных вызовов ядра Linux. Использует генерацию данных по спецификации их типов (можно указать базовые типы C, числовые интервалы, наборы флагов и пр.).
    Код открыт, продолжает развиваться, поддерживая совместимость с текущим состоянием ядра Linux.
  - IOCTL fuzzer [36] (2009). Фаззер для библиотек ядра и драйверов ОС Windows.
    Код открыт, с 2011 г. не развивается.
  - syzkaller [58] (2016). Реализовал возможность нацеливания на покрытие при фаззинге ядра ОС. Использует шаблоны обращений к системным вызовам в качестве исходного набора данных.
    Код открыт, продолжает развиваться.
  - kAFL [123] (2017). Фаззер для компонентов ядра ОС, в определенной степени независимый от ОС. Использует виртуализацию и встроенную технику трассировки инструкций процессоров Intel.
  - CAB-Fuzz [77] (2017). Фаззер для компонентов ядра ОС, использующий динамическое выполнение.

- Фаззеры для реализаций телекоммуникационных протоколов.
  - SPIKE [66,67] (2001). Фреймворк для построения генераторов данных. Содержит набор API (application program interface) для описания структур генерируемых данных, включая разнообразные зависимости и вычислимые поля.
    Код открыт, с 2017 г. не развивается.
  - Sulley [68] (2016). Фреймворк для фаззинга протоколов. Поддерживает использование случайных генераторов и мутаций, а также обработку ошибок в виде их классификации и генерации трасс, повторяющих ошибку.
    Код открыт, с 2017 г. не развивается (в 2019 г. изменен README.md).
    Развитием Sulley является boofuzz [124], его код также открыт и продолжает развиваться.
  - Можно также отметить фреймворк Defensics [125], также предназначенный для фаззинга реализаций протоколов и содержащий значительный корпус тестов для более чем 150 разных протоколов.
  - Имеется достаточно много фаззеров, способных генерировать тесты для одного заданного протокола. Можно отметить среди них SNOOZE [70] и KiF [71] для VOIP/SIP, TLS-Attacker [82] для TLS, Secfuzz [126] для IKE.
  - Обзор специфики фаззинга реализаций протоколов можно найти в [127].

- Фаззеры для компиляторов и интерпретаторов.
  - jsfunfuzz [79] (2007). Основанный на грамматике фаззер черного ящика для обработчиков JavaScript.
    Код открыт, продолжает развиваться.
  - Csmith [128,129] (2011). Поддерживает генерацию случайных программ на языке C, удовлетворяющих стандарту C99.
    Код открыт, продолжает развиваться.
  - LangFuzz [130] (2012). Развитие jsfunfuzz, может поддерживать несколько языков, но использовался только для JavaScript и PHP. Использует случайную генерацию программ по блокам и мутации.
  - Обзор специфики фаззинга для компиляторов можно найти в [131].
- Фаззеры для встроенных систем и устройств.
  - Можно отметить VDF [132] (2017), основанный на эволюционной стратегии фаззер для виртуальных устройств, а также IoTFuzzer [36] (2018).
  - Область фаззинга встроенных систем и устройств активно развивается в последние годы, обзоры специфичных для этой области инструментов и решений можно найти в [133-135].

3.4 Техники снижения эффективности фаззинга

Бурное развитие инструментов фаззинга привело к тому, что во второй половине 2010-х годов они стали активно использоваться хакерами для выявления ранее неизвестных уязвимостей в широко используемом ПО с целью дальнейшего проведения атак. В связи с этим встала задача борьбы с подобным использованием фаззеров одновременно с сохранением, по возможности, эффективности фаззинга при его использовании разработчиками. Эффективность фаззинга при этом понимается как количество обнаруживаемых ошибок и достигаемых элементов покрытия за единицу времени выполнения и/или на определенном числе его прогонов.

Обычно предполагается, что злоумышленнику, использующему фаззер для создания атак, целевое ПО доступно только в виде предназначенного для эксплуатации исполнимого кода, в то время как разработчики имеют доступ к исходному коду и могут собирать специализированные исполнимые файлы, используемые для «хорошего» фаззинга с целью поиска и дальнейшего исправления ошибок и уязвимостей. Во втором случае эффективность фаззинга не должна заметно снижаться, поэтому изменения, вносимые в код рассматриваемыми техниками решения этой задачи, обычно отключаются. Использование же предназначенного для эксплуатации исполнимого кода должно существенным образом снижать эффективность фаззинга, незначительно влияя при этом на эффективность обычного выполнения, чтобы не сказаться на производительности работы пользователей.

Также, использование техник противодействия фаззингу должно отслеживаться (и отключаться) при проведении независимого фаззинга, например, при сертификации безопасности ПО.

Одними из первых работ, описывающих практически значимые техники противодействия фаззингу, являются [136,137]. Многие такие техники реализованы в виде инструментов трансформации кода или библиотек-фреймворков, подключаемых при сборке.

Методы снижения эффективности фаззинга обычно используют комбинации следующих техник.

- Обфускация и сокрытие кода.
  Обычные техники обфускации кода в виде рандомизированных перестроений потока данных, замены констант, шифрования части данных и пр. [138,139], сами по себе не слишком подходят для решения данной задачи. Они, во-первых, значительно снижают

производительность кода при обычной работе, а, во-вторых, не слишком влияют на количество достигаемых элементов покрытия на заданном числе прогонов фаззинга (хотя и значительно повышают время их выполнения). Однако обфускация иногда может использоваться в комбинации с другими техниками для решения отдельных подзадач.

К этой же группе можно отнести и различные техники сокрытия кода (шифрование, динамическая загрузка, динамическая модификация и пр.), которые предназначены для затруднения его анализа, и часто используются в злонамеренном ПО, чтобы избежать его обнаружения. Например, SAFTE [140] использует для противодействия фаззингу динамическую загрузку кода программы из данных.

- Детектирование работы в режиме фаззинга.
  Некоторые авторы предлагают определять использование ПО в режиме фаззинга, чтобы включить техники противодействия ему. Для этого используются техники определения режима отладки [137], например, детектирование использования ptrace на Linux, а также техники, основанные на статистических шаблонах использования функций и создания событий [141].
  No-Fuzz [142,143] использует создание временных файлов в несущественном коде, чтобы по большому числу таких файлов детектировать применение фаззинга.

- Замедление выполнения несущественных ветвлений.
  Основано на автоматическом выделении несущественных ветвлений (см. выше, раздел про предобработку). Реализуется в виде добавления значительно замедляющих выполнение инструкций на несущественных ветвлениях, не связанных с выполнением основных функций. При этом обычная работа, в ходе которой эти ветвления почти никогда не используются, не замедляется, а фаззинг, при котором эти ветвления исполняются довольно часто, замедляется существенно. На количество прогонов фаззинга для достижения нужного покрытия эта техника не сильно влияет.
  В рамках [144] несущественные блоки кода выделяются на основе статистического анализа частоты их выполнения. ANTIFUZZ [145,146] предлагает вставлять вручную замедляющие вызовы в код обработки некорректных данных.

- Вставка многочисленных ветвлений, зависящих от входных данных.
  Реализуется за счет генерации большого числа ветвлений, зависящих от входных данных (вплоть до того, что почти каждый байт входных данных влияет на исполнение какого-нибудь ветвления), но не добавляющих никакой функциональности. Могут также использоваться многочисленные рандомизированные переходы на сгенерированный компилятором несущественный код. При анализе такого кода фаззер тратит значительные усилия на то, чтобы попасть в каждое ветвление. Дополнительно, само количество возникающих ветвей (могут добавляться десятки тысяч) может переполнять используемые фаззером хранилища для данных о покрытии. Используется в [142-146], а также в VALL-NUT [147].

- Вставка ложных ошибок или маскировка ошибок.
  Некоторые методы усложняют выполнение фаззинга с помощью затруднения определения ошибок. Для этого используется вставка многочисленных «ложных» ошибок, которые, однако, нельзя использовать для построения атак, в коде, связанном с выполнением несущественных ветвлений [145,146,148] или маскировка ошибок с помощью перехвата сигналов об ошибочном завершении работы ПО и обычного завершения работы в рамках кода-перехватчика [137,145,146]. Для маскировки может использоваться и перехват работы ptrace, поскольку некоторые фаззеры (например, honggfuzz) используют и этот механизм.

- Усложнение предикатов ветвлений и потоков данных.
  Для снижения эффективности фаззинга, использующего символическую интерпретацию, выполняется усложнение предикатов ветвлений, за счет чего значительная их часть

становится трудноразрешимой. Для противодействия анализу помеченных данных, аналогично, используется запутывание потоков данных, введение в них дополнительных (часто ложных) зависимостей.

В качестве такого усложнения используется замена сравнения строк или байтовых массивов на сравнение их хеш-кодов [144-146], шифрование, а затем расшифрование части входных данных [145,146], замена некоторых строк или массивов данных на их копии, полученные с помощью нетривиального копирования буферов (каждый элемент получается равным соответствующему элементу исходного буфера, но при этом он проходит через несколько преобразований) [144]. Могут использоваться замены в выражениях арифметических операций на гомоморфно преобразованные результаты операций над обратно преобразованными аргументами.

No-Fuzz [142,143] использует замену использования переменных на обращения к функциям, вычисляющим значения этих переменных сложным для анализа способом (например, числовая переменная *y*, имеющая значение больше 1, заменяется на вызов функции $f(1-1/y)$, где функция $f(x) = 1/(1-x)$ вычисляется при помощи неполного суммирования ряда $\sum_{i=0}^{N} x^i$). Также No-Fuzz использует результаты итеративных преобразований, приводящих к неподвижным точкам, несмотря на исходные данные (постоянная Капрекара [149]), в качестве констант. Такие константы выглядят для символического анализа как числа, сложным образом зависящие от (произвольно выбираемых) исходных данных.

4. Верификационный мониторинг

Верификационный мониторинг [150] (runtime verification) или ***мониторинг утверждений/свойств*** является методом верификации ПО, при котором исходный или бинарный код проверяемого ПО подвергается инструментации, вставляющей в некоторые места код проверки некоторых заданных свойства (в виде утверждений, assertions, или в другом), либо записывающий обнаруженные нарушения в некоторый журнал/трассу, либо просто прерывающий работу ПО при нарушении проверяемого свойства, после чего инструментированный код исполняется, чаще всего в обычном эксплуатационном режиме. Такой инструментированный код может выполняться в рамках тестов, при этом тесты могут не проверять отдельно требования, связанные с записанными в инструментированный код свойствами.

Верификационный мониторинг стал полноценной областью исследований в последние 25 лет, до этого спорадически выполнялись работы, близкие по тематике, но их авторы были разрознены и не пытались обобщить полученный опыт. Первые работы по инструментам верификационного мониторинга появились в 2000-2001 гг. [151,152]. Сейчас это достаточно развитая область, есть книги по верификационному мониторингу [150], несколько достаточно аккуратных обзоров [153-155].

Есть работы схожей тематики, которые относят себя к области *пассивного тестирования* [156,157] (passive testing). Они являются переносом техник формального тестирования протоколов, использующих автоматные модели, на случай мониторинга, т.е. пытаются на основе некоторого набора выполнений SUT оценить, насколько аккуратно проверены разные элементы поведения, описываемого моделью.

Основные темы, рассматриваемые в этой области, таковы.

- С помощью каких формальных теорий и нотаций (языков спецификаций) можно выразить нужные свойства так, чтобы их проверка во время работы проверяемого ПО (system under test, SUT) была возможна и достаточно эффективна.
- Как обеспечить корректную инструментацию и оценку работы SUT. Какие техники инструментации (исходного или бинарного кода, предварительные или во время выполнения, с привлечением аппаратной инструментации или без нее, и пр.) и оценки

корректности (во время выполнения, это так называемая *онлайн верификация*, или по трассе, уже после выполнения, что называется *офлайн верификацией*) использовать.

- Как добиться приемлемого снижения производительности при работе инструментированной системы.

В связи с автоматизированной верификацией свойств защищенности три вида работ вносят наиболее заметный вклад в данную область.

- Работы, посвященные автоматизированному мониторингу свойств защищенности отдельных систем, описанных в виде выделенной формальной модели.
  В работах такого типа описывается некоторая техника инструментированного мониторинга (обычно на основе трассы событий, происходивших в SUT) свойств, описанных в целостной формализованной модели безопасности [158-161].
  Такая верификация весьма ресурсоемка и часто не может производиться непосредственно во время работы SUT, из-за чего выполняется офлайн, по трассе событий, связанных с вызовами определенных функций или обращением к определенным данным. Однако предлагаемые в таких работах техники дают возможность строго проверить единовременно большое количество значимых требований к безопасности SUT, которые иными путями тяжело верифицировать.
- Работы, посвященные инструментам мониторинга, проверяющим настраиваемые или описываемые в виде дополнительных входных данных свойства безопасности.
  Часто это тоже достаточно ресурсоемкие инструменты, требующие предварительной разработки конфигурации/описания проверяемых свойств, поскольку они используют языки спецификаций, не всегда близкие по синтаксису и набору базовых понятий к языку SUT. Они отличаются друг от друга формализмами и нотациями, используемыми для описания проверяемых правил, поддерживаемыми языками SUT, техниками внедрения мониторов в проверяемый код и пр.
- Работы, посвященные инструментам мониторинга, проверяющим небольшой фиксированный набор специфических свойств, которые можно верифицировать достаточно эффективно, чтобы производительность инструментированной SUT не слишком отличалась от исходной (обычно приемлемым считается снижение производительности в 2-5 раз).
  Инструменты этого типа находят широкое применение в фаззинге в качестве тестовых оракулов, компонентов фаззера, выявляющих некорректное поведение SUT.

Работы первого типа обычно жестко связаны с верифицируемой системой и требуют значительной модификации средств мониторинга для переноса на другие системы. В следующих подразделах рассматриваются инструменты второго и третьего типов.

4.1 Инструменты с настраиваемыми проверками

Для таких инструментов в 2014-2016 проводились соревнования [162-164], сравнивающие эффективность инструментов как с точки зрения выявления ошибок, так и по снижению производительности SUT.

Стоит отметить следующие инструменты верификационного мониторинга с настраиваемыми проверяемыми свойствами.

- E-ACSL [165-167] (Executable ANSI/ISO C Specification Language, 2013).
  Реализует мониторинг программ на C, проверяемые свойства описываются на подмножестве ACSL [168]. ACSL представляет собой расширение C при помощи спецификационных комментариев, позволяющих указывать инварианты для типов данных и циклов, свойства, которые должны выполняться для глобальной или локальной переменной, предусловия и постусловия функций в виде логических выражений, почти всегда имеющих C-подобный синтаксис.
  Код открыт, последние изменения в 2017 г.

- RiTHM [169] (Runtime Time-triggered Heterogeneous Monitoring, 2013).
  Реализует мониторинг программ на C, проверяемые свойства описываются на расширении LTL (Linear Temporal Logic), используемое расширение и алгоритмы мониторинга описаны в [170]. Для проведения оценки может использовать распараллеливание как на обычных процессорах, так и на графических ускорителях (GPU).
- Larva [171-173] (2009).
  Инструмент мониторинга Java программ, языком спецификаций служит DATEs [174], разновидность событийных автоматов с таймерами. Фрагменты мониторингового кода вставляются в код SUT при помощи описания аспектов и точек вставки на AcpectJ.
  Код открыт, последние изменения в 2022 г.
- RV-Monitor [175,176] (2014).
  Независимый от языка инструмент мониторинга, оптимизированный для одновременного мониторинга многих свойств. На его основе построены расширения для Java и для C. Также имеет несколько плагинов для описания спецификаций в разных формализмах (временные логики, расширенные конечные автоматы, системы переписывания термов и пр.). Еще одним его расширением является RV-Android [177], фреймворк для контроля безопасности приложений на платформе Android.
  Код открыт, последние изменения в 2020 г.
- MarQ [178] (Monitoring at runtime with Quantified Event Automata, 2015).
  Инструмент мониторинга Java программ, поддерживает как онлайн, так и офлайн верификацию. Спецификации описываются в нотации событийных автоматов с числовыми метками (QEA, Quantified Event Automata), интегрируются в SUT при помощи AspectJ.
- Mufin [179,180] (Monitoring with Union-Find, 2016).
  Инструмент мониторинга Java программ, мониторы в нем определяются при помощи специализированного API. Использует специализированные алгоритмы для мониторинга большого количества объектов.

4.2 Инструменты с фиксированными проверками

Далее рассматриваются наиболее широко используемые инструменты мониторинга, выполняющие проверку жестко зафиксированного небольшого множества свойств.

- Мониторинг ошибок использования памяти.
  Эти инструменты нацелены на выявление некорректной работы с памятью: обращений к памяти за пределами объекта или буфера (пространственная корректность, spatial safety), или обращений к неинициализированной/ освобожденной памяти (временная корректность, temporal safety).
  - AddressSanitizer [181,182] (ASan).
    Инструментирует исходный код для выявления некорректных обращений к памяти, используя теневую память, которая хранит разметку текущей памяти процесса на безопасную/опасную для обращений. Снижение производительности при его использовании обычно остается в пределах 2-3 раз.
    Имеется интегрированный в QEMU вариант, QASan [183].
    Код открыт [182], последние изменения в 2019 г.
  - MEDS [184].
    Нацелен на эффективное обнаружение некорректных обращений к памяти в крупномасштабных приложениях, с большими объемами рабочей памяти. Немного больше снижает производительность, чем ASan, но позволяет обнаруживать больше ошибок при фаззинге с помощью AFL.

- SoftBound/CETS [185,186].
  Инструментирует исходный код. Связывает границы и признак занятости с каждым указателем, поэтому теоретически способен обнаружить все ошибки некорректного использования памяти. Это обеспечивается за счет большего снижения производительности, которое, однако, в большинстве случаев на превосходит 3-х раз.

- Мониторинг ошибок приведения типов.
  Подобные ошибки возникают в C++ в связи с неаккуратным использованием конструкций static_cast и dynamic_cast для приведения типов указателя на объект к дочернему типу. В некоторых случаях могут приводить к появлению уязвимостей. К инструментам, нацеленным на обнаружение ошибок этого вида, относятся CAVER [187], TypeSan [188], HexType [189]. Они дополняют код информацией о реальных типах объектов и указателей и проверками корректности приведения типов. Приводят к снижению производительности от 1.7 до 5 раз. TypeSan и HexType позволяют выявлять подобные проблемы для локальных и глобальных переменных, и для объектов на стеке.

- Мониторинг случаев неопределенного поведения.
  В языках C/C++ в достаточно многих ситуациях стандарт не фиксирует точную семантику, определяющую поведение тех или иных конструкций языка, позволяя разработчикам компиляторов использовать такие места для более эффективной оптимизации. Однако в ряде случаев конструкции с неопределенной семантикой могут становиться источниками ошибок и уязвимостей [190].

  - Valgrind [191,192].
    Наиболее известный инструмент этого типа. Использует статическую (до начала выполнения) трансформацию бинарного кода. Одна из основных функций связана с контролем использования памяти (Memcheck) — вся память при инструментации получает метки о ее занятости или свободе, все выделяемые блоки памяти окаймляются дополнительными блоками, попадание в которые служит индикатором выхода за границы корректно используемой памяти. Помимо проверки обращений к памяти может проверять работу со стеком, искать состояния гонок (data races). Снижение производительности при использовании Valgrind достаточно велико (бывает 10-50 раз, иногда больше), поэтому он редко используется с фаззерами.
  - Dr. Memory [193].
    Выявляет ошибки, связанные с использованием неинициализированной или освобожденной памяти. По сравнению с Memcheck в Valgrind инструментированный Dr. Memory код работает в 2-3 раза быстрее, но для фаззинга это тоже оказывается часто недостаточно эффективно.
  - MemorySanitizer [194,195] (MSan).
    Еще один монитор ошибок, связанных с использованием неинициализированной памяти в C и C++. За счет оптимизированного использования теневой памяти снижает производительность в 2-4 раза. Достаточно широко используется с фаззерами.
    Код открыт, последние изменения в 2020 г.
  - UndefinedBehaviorSanitizer [196,197] (UBSan).
    Монитор большого количества типов ошибок, связанных с неопределенным поведением в C/C++. Снижение производительности при полном наборе проверок может быть большим, каждый отдельный вид проверок обычно дает снижение не более чем в 2-3 раза, но использование совместно проверок нескольких видов снижает производительность более значительно.
    Код открыт, последние изменения в 2020 г.

- ThreadSanitizer [198,199].
  Монитор условий гонок (data races) в многопоточных программах.
  Код открыт, последние изменения в 2020 г.

5. Динамическая символьная интерпретация

Символическая интерпретация или ***символическое выполнение*** (symbolic execution) состоит в том, что код (исходный или бинарный) подвергается анализу, при котором участвующие в нем данные (переменные, объекты, регистры процессора или участки памяти) рассматриваются как символьные переменные, между которыми устанавливаются связи в виде символьных выражений (выражающих одни данных через другие) в ходе интерпретации инструкций кода как трансформаций этих символьных выражений. ***Динамическая символьная интерпретация*** (dynamic symbolic execution, DSE) является символической интерпретацией, выполняемой в динамике, при исполнении анализируемого ПО, и поэтому, в качестве исходной точки проводимого анализа обычно имеет некоторое конкретное выполнение, с конкретными входными данными. Другим термином для динамической символьной интерпретации является ***конколическое выполнение*** (concolic execution) [100] или ***конколическая интерпретация*** (concolic interpretation), здесь англоязычный термин образован из склейки слов конкретный (concrete) и символический (symbolic). При таком исполнении часть данных представляется одновременно конкретными значениями и символическими выражениями, что позволяет упрощать ряд получаемых ограничений и проводить анализ с меньшими накладными расходами.

Динамическая символьная интерпретация используется при фаззинге или обычной генерации тестов для извлечения символьных ограничений, решая которые можно получить тестовые данные, обеспечивающие попадание в нужную ситуацию или проявление некоторой ошибки. Она может применяться и при других видах анализа: выявлении дубликатов кода, выявлении способов обхода механизмов защиты, формальной верификации кода и др.

Сама идея использовать символическое выполнение при тестировании для получения ограничений на входные данные, которые затем разрешаются, давая исходные данные для тестов, известна уже почти 50 лет [200-203].

Практически реализуемое применение динамической символьной интерпретации (DSE) для генерации тестов описано в работе [101], в 2005 г. Далее идеи этой работы привели к созданию фаззера, использующего DSE, которым стал CUTE [100], хотя он еще оставался на уровне исследовательского прототипа. Сам термин «конколическое выполнение» был введен в описывающей его статье. Первые промышленно применимые инструменты фаззинга на базе DSE [102,103] появились позже, в 2008 г.

Далее рассматриваются основные техники, используемые при реализации инструментов DSE, большая часть материала взята из обзоров [99,204,205] и обзора в работе [25].

Прямое символическое выполнение некоторого кода выглядит как последовательная трансформация его инструкций в символические преобразования их входных данных в результаты и последовательное же раскрытие всех встречающихся разветвлений в вычисление различных выражений на разных ветках. Применение его в таком виде к достаточно объемному реалистичному коду сталкивается со многими проблемами.

- Необходимость сокращения количества анализируемых путей.
  Простое размножение путей при выборе ветвлений даже на простых циклах быстро приводит к комбинаторному взрыву.
- Необходимость символической обработки указателей и массивов.
  В программах часто данные берутся из некоторой памяти по адресам, причем сами эти адреса вычисляются динамически. Необходимо каким-то образом описывать и анализировать возникающие при этом зависимости между данными, для этого

используется так называемая модель памяти, основная задача которой — моделирование операций с адресами.

- Необходимость обработки обращений во внешние библиотеки и системы.
  Прямая символическая интерпретация вызовов внешних функций невозможна без полного доступа к их коду, поэтому необходимо как-то моделировать работу с окружением анализируемой системы.
- Чтобы провести анализ нового пути в рамках инструмента DSE, чаще всего требуется решить соответствующие ему ограничения. Также разрешение полученных ограничений нужно и при использовании DSE в рамках фаззинга или какой-то другой деятельности. Обычно при этом возникает необходимость дополнительной обработки выявленных символических ограничений при их передаче решателю.
  При этом важно, какие именно теории и операции используются решателем, как обрабатываются инструкции, не укладывающиеся в поддерживаемые решателем теории. Может использоваться внешний решатель, собственная реализация разрешающего алгоритма, или какая-то смешанная техника. Перед передачей для обработки решателем могут выполняться оптимизирующие преобразования и сокращения ограничений.
  В инструментах поиска ошибок ограничения могут использоваться не только для описания путей, но и задания условий возникновения ошибок.

Помимо техник решения этих задач, инструменты, использующие DSE, могут различаться то применяемой инструментации кода SUT — на уровне исходного или бинарного кода, с помощью вставок в код иди за счет внешнего мониторинга выполнения определенных инструкций.

5.1 Сокращение количества и выбор анализируемых путей

Для сокращения множеств анализируемых путей и выбора очередного пути анализа применяются следующие техники.

- Выбор путей
  - Выбор путей для символического выполнения.
    В первых инструментах DSE (CUTE [100]) символически исполнялись все находимые пути. Далее были использованы разные стратегии выбора. KLEE [102,119] и SAGE [103,104] в первую очередь выполняют пути, позволяющие покрыть больше новых ветвей. S^2E [105] выбирает пути, наибольшим образом задействующие код целевых компонентов (несущественные компоненты и функции выполняются только конкретно). Driller [10] и QSYM [108] используют фаззер AFL [33] в качестве дополнительного источника для выбора путей, причем Driller использует символьное выполнение тогда, когда применение AFL уже не дает нового покрытия, QSYM использует символьное выполнение параллельно конкретному, в первую очередь для путей, обеспечивающих покрытие новых ветвлений. AEG [206] и Mayhem [106] в первую очередь обрабатывают пути, содержащие элементы кода с высокой вероятностью ошибок (циклы с обращением по указателям в них и пр.).
  - Выполнение сразу нескольких путей.
    KLEE, AEG, S^2E иногда пытаются выполнять новые пути с меньшими затратами за счет переключения на них с помощью fork или использования механизмов копирования при записи, чтобы снизить затраты на хранение состоянии анализируемой программы. Mayhem не пытается исполнять два пути одновременно, но при выполнении одного сохраняет контрольные точки для более легкого возвращения к новым путям.
- Упрощение и сокращение множества анализируемых путей
  - Сокращение путей с помощью использования условных выражений.
    Если решатель позволяет использовать выражения ite(c,x,y,), означающие, что если

выполнено условие c, нужно взять значение x, а иначе значение y, инструменты DSE активно пользуются этим. Условные выражения позволяют сокращать количество анализируемых путей и компактно представлять символически многие функции.

- Упрощение анализа с помощью анализа помеченных данных.
  Некоторые инструменты, например, LeanSym [207], используют динамический анализ помеченных данных, чтобы выделить только влияющую на проход по определенной ветви часть входных данных и использовать ограничения только на нее.
- Использование символических моделей (summary) для функций, циклов или отдельных участков кода.
  Некоторые техники DSE [208-210] позволяют во время анализа строить символические модели часто вызываемых функций и выполняемых циклов. [211,212] представляют аналогичные техники, позволяющие строить символические аннотации для отдельных ветвей. Эти аннотации и модели затем можно использовать, не выполняя повторный анализ этих элементов кода.
- Использование ограничений на входные данные.
  AEG [206] позволяет определять предусловия на входные данные, которые сужают множество анализируемых путей.
- Отождествление символических состояний.
  Для сокращения количества путей, подлежащих анализу, иногда используется отождествление состояний [213].

5.2 Моделирование работы с адресами

В инструментах DSE используются следующие техники моделирования работы с указателями, массивами и сложными структурами данных.

- В CUTE и SAGE использовались конкретные адреса, получаемые в текущем конкретном выполнении.
- Полное символическое моделирование памяти.
  Предлагалось еще в статье Кинга [202], поддерживалось во фреймворках BitBlaze [214,215] и BAP [216]. В KLEE и SAGE моделирование части массивов перекладывается на SMT решатели [97], в предположении, что они поддерживают соответствующие теории. Аналогично, S^2E [105] использует теорию массивов для моделирования участков памяти, на которые может ссылаться некоторый указатель. В Sydr [111,217] используется аппроксимация возможных адресов линейными выражениями, которые затем моделируются битовыми векторами.
- В Mayhem и angr [218] используется смешанное моделирование, для чтения адреса моделируются символически, а запись идет по конкретным адресам. В angr запись тоже может моделироваться символически, если используются небольшие буфера.

5.3 Моделирование работы с окружением

Для моделирования окружения анализируемой программы используются следующие техники.

- Пропуск функций.
  Некоторые функции (например, управления памятью, печати и пр.) можно полностью игнорировать при символическом анализе, поскольку они не оказывают влияния на его результаты.
- Моделирование семантики функций с помощью символических выражений.
  S^2E [105] и angr [218] используют символические модели для библиотечных функций, которые используются каждый раз, когда в коде эти функции вызываются. angr содержит

символические модели для довольно большого числа функций из стандартных библиотек libc и POSIX.

- Использование упрощенных реализаций или упрощенных символических моделей.
  В KLEE [102] и AEG [206] обращения к функциям стандартной библиотеки C заменяются на вызов встроенной упрощенной реализации, что упрощает их дальнейший анализ.
  S^2E [105] использует при анализе только некоторые пути из реализаций функций.
  SymCC [219] содержит для части функций стандартной библиотеки C упрощенные символические модели, которые добавляют возможные разветвления с их условиями в набор ограничений пути.
  Fuzzolic [220] может использовать три режима анализа функции: пропуск, анализ без инвертирования условий путей и полную символическую интерпретацию.

5.4 Дополнительная обработка ограничений

Следующие техники применяются для обработки ограничений перед передачей их решателю.

- Устранение избыточных ограничений с помощью выделения независимых подмножеств всего набора ограничений.
  Используется в EXE [119] и KLEE [102].
- Выбрасывание части ограничений для упрощения условия прохода по пути.
  QSYM [108] иногда выбрасывает из условий прохода по новому пути все ограничения, кроме последнего, полученного отрицанием последнего условия для только что пройденного пути. Часто это помогает решателю найти подходящие данные из-за простоты полученного ограничения (и несмотря на игнорирование сложной истории ранее пройденных условий).
  QSYM также может выкидывать из анализа некоторые блоки, которые часто исполняются на разных путях и порождают новые ограничения, усложняя тем самым условия прохождения других путей.
- Добавление ограничений, обеспечивающих проявление ошибок.
  Дополнительные ограничения представляют собой условия проявления ошибок, что при их успешном разрешении позволяет получить входные данные, демонстрирующие данную ошибку.
  KLEE добавляет ограничения, приводящие к делению на 0 в арифметических выражениях. IntScope [221] добавляет условия возникновения переполнения в операциях над целочисленными типами данных, Sydr [111] тоже может выполнять такие действия. Mayhem [106] может добавлять ограничения, приводящие к переполнению буферов или некорректной обработке форматных строк во входных данных.
  Некоторые фаззеры (SAVIOR [222], ParmeSan [223]) используют код, добавляемый санитайзерами, как источник условий, которые могут привести к возникновению ошибки.

6. Заключение

В данном документе проведен обзор техник и инструментов динамического анализа программ, нацеленных, в первую очередь, на проверку свойств безопасности и защищенности. Подробно рассмотрены методы фаззинга, верификационного мониторинга, отдельно представлены техники динамической символьной интерпретации, которые активно используются в рамках инструментов фаззинга. Методы и средства динамического анализа помеченных данных остались за рамками обзора из-за трудностей сбора технической информации о них. При рассмотрении фаззинга и динамической символьной интерпретации больше внимания уделено не отдельным инструментам, из-за их очень большого количества,

а техникам решения различных задач, возникающих при их работе. Также представлен обзор техник снижения эффективности фаззинга.

Отдельного упоминания заслуживает линейка инструментов динамического анализа, разрабатываемая и активно развиваемая на протяжении уже долгого времени в Институте системного программирования им. В. П. Иванникова РАН, покрывающая почти все виды динамического анализа [109-111,160,161,224-226].

Список литературы / References

[1]. M. Ozkan-Okay, R. Samet, Ö. Aslan, and D. Gupta. A Comprehensive Systematic Literature Review on Intrusion Detection Systems. IEEE Access, vol. 9, pp. 157727-157760, 2021, doi: 10.1109/ACCESS.2021.3129336

[2]. L. Santos, C. Rabadao, and R. Gonçalves. Intrusion Detection Systems in Internet of Things: A literature review. Proc. of 13-th Iberian Conference on Information Systems and Technologies (CISTI), Caceres, Spain, 2018, pp. 1-7, doi: 10.23919/CISTI.2018.8399291

[3]. H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, 29(4):366-427, 1997. doi: 10.1145/267580.267590

[4]. M. Sutton, A. Greene, and P. Amini. Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley, 2007. ISBN: 9780321446114

[5]. J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. Proc. of Network and Distributed System Security Simposium, 2005. doi: 10.1184/R1/6468716.v1

[6]. E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). Proc. of IEEE Symposium on Security and Privacy, pp. 317-331, 2010. doi: 10.1109/SP.2010.26

[7]. T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: a Checksum-aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. Proc. of IEEE Symposium on Security and Privacy, pp. 497-512, 2010. doi: 10.1109/SP.2010.37

[8]. B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM, 33(12):32-44, 1990. doi: 10.1145/96267.96279

[9]. The Cyber Grand Challenge. URL: https://blogs.grammatech.com/the-cyber-grand-challenge (доступ 13.06.2023)

[10]. N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. Proc. of Network and Distributed System Security Symposium. 2016. doi: 10.14722/NDSS.2016.23368

[11]. P. Goodman and A. Dinaburg. The Past, Present, and Future of Cyberdyne. IEEE Security & Privacy, 16(2):61-69, 2018. doi: 10.1109/MSP.2018.1870859

[12]. Cisco Secure Development Lifecycle. URL: https://www.cisco.com/c/en/us/about/trust-center/technology-built-in-security.html#~trustworthysolutionsfeatures (доступ 13.06.2023)

[13]. Chromium Security. URL: https://www.chromium.org/Home/chromium-security/bugs/ (доступ 13.06.2023)

[14]. Clusterfuzz. Chrome Fuzzing Infrastructure. URL: https://code.google.com/archive/p/clusterfuzz/ (доступ 13.06.2023)

[15]. M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker. Announcing OSS-Fuzz: Continuous fuzzing for open source software. Google Open Source Blog, 2016. URL: https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html (доступ 13.06.2023)

[16]. Microsoft Security Development Lifecycle. URL: https://www.microsoft.com/en-us/securityengineering/sdl/practices (доступ 13.06.2023)

[17]. E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. Proc. of 35-th International Conference on Software Engineering (ICSE), San Francisco, USA, 2013, pp. 122-131, doi: 10.1109/ICSE.2013.6606558

[18]. Fuzzing Survey URL: https://fuzzing-survey.org/ (доступ 15.06.2023)

[19]. N. Rathaus, G. Evron. Open Source Fuzzing Tools. Syngress, 2007. ISBN: 9781597491952

[20]. A. Takanen, J. D. DeMott, C. Miller, and A. Kettunen. Fuzzing for Software Security Testing and Quality Assurance. 2-nd ed. Artech House, 2018. ISBN: 9781608078509
[21]. J. Li, B. Zhao, and C. Zhang. Fuzzing: a Survey. Cybersecurity 1, 6, 2018. doi: 10.1186/s42400-018-0002-y
[22]. C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu. A Systematic Review of Fuzzing Techniques. Computers & Security, 75:118-137, 2018. doi: 10.1016/j.cose.2018.02.002
[23]. V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The Art, Science, and Engineering of Fuzzing: A Survey. IEEE Transactions on Software Engineering, 47(11):2312-2331, 2021. doi: 10.1109/TSE.2019.2946563. URL: http://arxiv.org/abs/1812.00140
[24]. H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the Art. IEEE Transactions on Reliability, 67(3):1199-1218, 2018. doi: 10.1109/TR.2018.2834476
[25]. А. В. Вишняков. Поиск ошибок в бинарном коде методами динамической символьной интерпретации. Диссертация на соискание учёной степени к. ф.-м.н., ИСП РАН, Москва, 2022.
[26]. A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti. LibAFL: a Framework to Build Modular and Reusable Fuzzers. Proc of ACM SIGSAC Conference on Computer and Communication Security, pp. 1051-1065, 2022. doi: 10.1145/3548606.3560602
[27]. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. ACM SIGPLAN Notices, 40(6):190-200, 2005. doi: 10.1145/1064978.1065034
[28]. F. Bellard. QEMU, a Fast and Portable Dynamic Translator. Proc. of ATEC'05, USENIX Annual Technical Conference, pp. 41-46, 2005. doi: 10.5555/1247360.1247401
[29]. Dyninst. URL: https://dyninst.org/dyninst (доступ 05.12.2023)
[30]. Dyninst GitHub. URL: https://github.com/dyninst/dyninst (доступ 05.12.2023)
[31]. D. L. Bruening. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Ph.D. thesis, Massachusetts Institute of Technology, 2004.
[32]. DynamoRIO. URL: https://github.com/DynamoRIO/dynamorio (доступ 05.12.2023)
[33]. M. Zalewski. American Fuzzy Lop. URL: https://github.com/mirrorer/afl (доступ 14.06.2023)
[34]. AFL, supported by Google. URL: https://github.com/google/AFL (доступ 19.06.2023)
[35]. D. Oleksiuk. IOCTL Fuzzer. URL: https://github.com/Cr4sh/ioctlfuzzer (доступ 14.06.2023)
[36]. J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. IoTFuzzer: Discovering Memory Corruptions in IoT through App-based Fuzzing. Proc. of the Network and Distributed System Security Symposium, 2018. doi:10.14722/ndss.2018.23159
[37]. D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. FUDGE: Fuzz Driver Generation at Scale. Proc. of 27-th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 975-985, 2019. doi: 10.1145/3338906.3340456
[38]. K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer. FuzzGen: Automatic Fuzzer Generation. Proc. of 29-th USENIX Security Symposium, pp. 2271-2287, 2020. doi: 10.5555/3489212.3489340
[39]. M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang. IntelliGen: Automatic Driver Synthesis for Fuzz Testing. Proc. of IEEE/ACM 43-rd International Conference on Software Engineering: Software Engineering in Practice, pp. 318-327, 2021. doi: 10.1109/ICSE-SEIP52600.2021.00041. URL: https://arxiv.org/abs/2103.00862
[40]. GRR. URL: https://github.com/lifting-bits/grr (доступ 14.06.2023)
[41]. LibFuzzer – a Library for Coverage-guided Fuzz Testing. URL: https://llvm.org/docs/LibFuzzer.html (доступ 14.06.2023)
[42]. R. Swiecki and F. Gröbert. Honggfuzz. https://github.com/google/honggfuzz (доступ 16.06.2023)
[43]. K. Sen. Effective random testing of concurrent programs. Proc. of 22-th IEEE/ACM International Conference on Automated Software Engineering, pp. 323-332, 2007. doi: 10.1145/1321631.1321679
[44]. P. Joshi, C.-S. Park, K. Sen, and M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. ACM SIGPLAN Notices, 44(6):110-120, 2009. doi: 10.1145/1543135.1542489
[45]. Z. Lai, S. Cheung, and W. Chan. Detecting Atomic-set Serializability Violations in Multithreaded Programs through Active Randomized Testing. Proc. of 32-nd ACM/IEEE International Conference on Software Engineering, 1:235-244, 2010. doi: 10.1145/1806799.1806836

[46]. Y. Cai and W. K. Chan. MagicFuzzer: Scalable deadlock detection for large-scale applications. Proc. of 34-th International Conference on Software Engineering (ICSE), Zurich, Switzerland, pp. 606-616, 2012. doi: 10.1109/ICSE.2012.6227156

[47]. M. Samak, M. K. Ramanathan, and S. Jagannathan. Synthesizing racy tests. Proc. of 36-th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 175–185, 2015. doi: 10.1145/2737924.2737998

[48]. V. Ganesh, T. Leek, and M. Rinard. Taint-based Directed Whitebox Fuzzing. Proc. of 31-st International Conference on Software Engineering (ICSE'09), pp. 474-484, 2009. doi: 10.1109/ICSE.2009.5070546

[49]. I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for Overflows: a Guided Fuzzer to Find Buffer Boundary Violations. Proc. of 22-nd USENIX Security Symposium, pp. 49-64, 2013. doi: 10.5555/2534766.2534772

[50]. L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. GRT: Program-Analysis-Guided Random Testing. Proc. of 30-th IEEE/ACM International Conference on Automated Software Engineering, pp. 212-223, 2015. doi: 10.1109/ASE.2015.49

[51]. S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware Evolutionary Fuzzing. Proc. of Network and Distributed System Security Symposium, 2017. doi: 10.14722/NDSS.2017.23404

[52]. H. Peng, Y. Shoshitaishvili and M. Payer. T-Fuzz: Fuzzing by Program Transformation. Proc. of IEEE Symposium on Security and Privacy, pp. 697-710, 2018. doi: 10.1109/SP.2018.00056

[53]. Репозиторий FFmpeg. URL: http://samples.ffmpeg.org/ (доступ 16.06.2023)

[54]. CERT BFF. URL: https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=507974 (доступ 15.06.2023)

[55]. A. D. Householder and J. Foote. Probability-Based Parameter Selection for Black-Box Fuzz Testing. SEI Technical Note, CMU/SEI-2012-TN-019, 2012. doi:10.21236/ada610472

[56]. M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling Black-box Mutational Fuzzing. Proc. of ACM SIGSAC Conference on Computer & Communications Security (CCS '13), pp. 511-522, 2013. doi: 10.1145/2508859.2516736

[57]. M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS '16), pp. 1032-1043, 2016. doi: 10.1145/2976749.2978428

[58]. Syzkaller – kernel fuzzer. URL: https://github.com/google/syzkaller (доступ 15.06.2023)

[59]. D. Vyukov. go-fuzz. URL: https://github.com/dvyukov/go-fuzz (доступ 19.06.2023)

[60]. Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu. Steelix: Program-State Based Binary Fuzzing. Proc. of 11-th Joint Meeting on Foundations of Software Engineering, pp. 627-637, 2017. doi: 10.1145/3106237.3106295

[61]. P. Chen and H. Chen. Angora: Efficient Fuzzing by Principled Search. Proc. of IEEE Symposium on Security and Privacy, pp. 711-725, 2018. doi: 10.1109/SP.2018.00046

[62]. M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed Greybox Fuzzing. Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS '17), pp. 2329-2344, 2017. doi: 10.1145/3133956.3134020

[63]. S. Wang, J. Nam, and L. Tan. QTEP: Quality-aware Test Case Prioritization. Proc. of 11-th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017), pp. 523-534, 2017. doi: 10.1145/3106237.3106258

[64]. M. Eddington. Peach Fuzzer. URL: https://peachtech.gitlab.io/peach-fuzzer-community/ (доступ 13.06.2023)

[65]. C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi, and D. Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. Proc. of Network and Distributed System Security Symposium, 2019. doi: 10.14722/ndss.2019.23412

[66]. S. Bradshaw. Fuzzer Automation with SPIKE. URL: https://resources.infosecinstitute.com/topic/fuzzer-automation-with-spike/ (доступ 13.06.2023)

[67]. SPIKE Protocol Fuzzer Creation Kit. URL: https://github.com/guilhermeferreira/spikepp (доступ 13.06.2023)

[68]. P. Amini, A. Portnoy, and R. Sears. Sulley. URL: https://github.com/OpenRCE/sulley (доступ 15.06.2023)

[69]. R. Kaksonen, M. Laakso, and A. Takanen. Software security assessment through specification mutations and fault injection. In: R. Steinmetz, J. Dittman, M. Steinebach (eds). Communications and Multimedia Security Issues of the New Century. IFIP — The International Federation for Information Processing, vol 64. Springer, pp. 173-183, 2001. doi: 10.1007/978-0-387-35413-2_16

[70]. G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In: S. K. Katsikas, J. López, M. Backes, S. Gritzalis, and B. Preneel (eds). Information Security, ISC 2006. Lecture Notes in Computer Science, 4176, pp. 343-358. Springer, 2006. doi: 10.1007/11836810_25

[71]. H. J. Abdelnur, R. State, and O. Festor. KiF: a Stateful SIP Fuzzer. Principles, Systems and Applications of IP Telecommunications, 2007. doi: 10.1145/1326304.1326313

[72]. W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano. T-Fuzz: Model-Based Fuzzing for Robustness Testing of Telecommunication Protocols. Proc. of IEEE 7-th International Conference on Software Testing, Verification and Validation, pp. 323-332, 2014. doi: 10.1109/ICST.2014.45

[73]. Trinity: Linux System Call Fuzzer. URL: https://github.com/kernelslacker/trinity (доступ 13.06.2023)

[74]. KernelFuzzer. URL: https://github.com/FSecureLABS/KernelFuzzer (доступ 15.06.2023)

[75]. P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. Proc. of 29-th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 206–215, 2008. doi: 10.1145/1375581.1375607

[76]. V.-T. Pham, M. Böhme, and A. Roychoudhury. Model-based Whitebox Fuzzing for Program Binaries. Proc. of 31-st IEEE/ACM International Conference on Automated Software Engineering, pp. 543-553, 2016. doi: 10.1145/2970276.2970316

[77]. S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. Proc. of USENIX Annual Technical Conference, pp. 689-701, 2017. doi: 10.5555/3154690.3154755

[78]. DOMFuzz. URL: https://github.com/MozillaSecurity/domfuzz (доступ 16.06.2023)

[79]. Jzfunfuzz. URL: https://github.com/MozillaSecurity/funfuzz (доступ 16.06.2023)

[80]. C. Brubaker, S. Jana, B. Ray, S. Khurshid, V. Shmatikov. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. Proc. of IEEE Symposium on Security and Privacy, pp. 114-129, 2014. doi: 10.1109/SP.2014.15

[81]. H. Kario. Tlfuzzer. URL: https://github.com/tlsfuzzer/tlsfuzzer (доступ 16.06.2023)

[82]. J. Somorovsky. Systematic Fuzzing and Testing of TLS Libraries. Proc. of ACM SIGSAC Conference on Computer and Communications Security, pp. 1492-1504, 2016. doi: 10.1145/2976749.2978411

[83]. J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven Seed Generation for Fuzzing. Proc. of the IEEE Symposium on Security and Privacy, pp. 579-594, 2017. doi: 10.1109/SP.2017.23

[84]. L. Della Toffola, C. A. Staicu, and M. Pradel. Saying 'hi!' is not Enough: Mining Inputs for Effective Test Generation. Proc. of 32-nd IEEE/ACM International Conference on Automated Software Engineering, pp. 44-49, 2017. doi: 10.5555/3155562.3155572

[85]. H. Han, D. Oh, and S. K. Cha. CodeAlchemist: Semantics-aware Code Generation to Find Vulnerabilities in Javascript Engines. Proc. of Network and Distributed System Security Symposium, 2019. doi: 10.14722/ndss.2019.23263

[86]. H. Han and S. K. Chaю IMF: Inferred Model-based Fuzzer. Proc. of ACM SIGSAC Conference on Computer and Communications Security, pp. 2345-2358, 2017. doi: 10.1145/3133956.3134103

[87]. P. Godefroid, H. Peleg, and R. Singh. Learn&Fuzz: Machine Learning for Input Fuzzing. Proc. of 32-nd IEEE/ACM International Conference on Automated Software Engineering, pp 50-59, 2017. doi: 10.48550/arXiv.1701.07232. URL: https://arxiv.org/abs/1701.07232

[88]. P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng. Automatic Text Input Generation for Mobile Testing. Proc. of IEEE/ACM 39-th International Conference on Software Engineering (ICSE), pp. 643-653, 2017. doi: 10.1109/ICSE.2017.65

[89]. M. Höschele and A. Zeller. Mining Input Grammars from Dynamic Taints. Proc. of 31-st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 720-725, 2016. doi: 10.1145/2970276.2970321

[90]. O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing Program Input Grammars. Proc. of 38-th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 95-110, 2017. doi: 10.1145/3062341.3062349. URL: https://arxiv.org/abs/1608.01723

[91]. A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the State: a State-aware Black-box Web Vulnerability Scanner. Proc. of 21-st USENIX Security Symposium, pp. 523–538, 2012. doi: 10.5555/2362793.2362819

[92]. H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck. PULSAR: Stateful Black-box Fuzzing of Proprietary Network Protocols. Proc. of International Conference on Security and Privacy in Communication Systems, pp. 330-347, 2015. doi: 10.1007/978-3-319-28865-9_18

[93]. A. Helin. Radamsa. URL: https://gitlab.com/akihe/radamsa (доступ 16.06.2023)

[94]. S. Hocevar. Zzuf. URL: https://github.com/samhocevar/zzuf (доступ 16.06.2023)

[95]. S. K. Cha, M. Woo, and D. Brumley. Program-Adaptive Mutational Fuzzing. Proc. of IEEE Symposium on Security and Privacy, pp. 725-741, 2015. doi: 10.1109/SP.2015.50

[96]. U. Kargén and N. Shahmehri. Turning Programs Against Each Other: High Coverage Fuzz Testing Using Binary-code Mutation and Dynamic Slicing. Proc. of 10-th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015), pp. 782-792, 2015. doi: 10.1145/2786805.2786844

[97]. L. D. Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. Communications of the ACM, 54(9): 69-77, 2011. doi: 10.1145/1995376.1995394

[98]. S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. CollAFL: Path Sensitive Fuzzing. Proc. of IEEE Symposium on Security and Privacy, pp. 679-69677, 2018. doi: 10.1109/SP.2018.00040

[99]. F. Rustamov, J. Kim, J. Yu, and J. Yun. Exploratory Review of Hybrid Fuzzing for Automated Vulnerability Detection. IEEE Access, 9:131166-131190, 2021. doi: 10.1109/ACCESS.2021.3114202

[100]. K. Sen, D. Marinov, and G. Agha. CUTE: a Concolic Unit Testing Engine for C. ACM SIGSOFT Software Engineering Notes, 30(5):263–72, 2005. doi: 10.1145/1095430.1081750

[101]. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. ACM SIGPLAN Notices, 40(6): 213-223, 2005. doi: 10.1145/1064978.1065036

[102]. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. Proc. of the 8-th USENIX conference on Operating System Design and Implementation, pp. 209–224, 2008. doi: 10.5555/1855741.1855756

[103]. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. Proc. of Network and Distributed System Security Symposium, pp. 151-166, 2008.

[104]. P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. Communications of ACM, 55(3):40-44, 2012. doi: 10.1145/2093548.2093564

[105]. V. Chipounov, V. Kuznetsov, G. Candea. S2E: a Platform for In-Vivo Multi-path Analysis of Software Systems. ACM SIGARCH Computer Architecture News Notices, 46(3):265–278, 2011. doi: 10.1145/1961295.1950396

[106]. S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. Proc. of IEEE Symposium on Security and Privacy, pp. 380-394, 2012. doi: 10.1109/SP.2012.31

[107]. M. Neugschwandtner, P. M. Comparetti, I. Haller, and H. Bos. The BORG: Nanoprobing Binaries for Buffer Overreads. Proc. of 5-th ACM Conference on Data and Application Security and Privacy (CODASPY '15), pp. 87-97, 2015. doi: 10.1145/2699026.2699098

[108]. I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: a Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. Proc. of 27-th USENIX Security Symposium, pp. 745-761, 2018. doi: 10.5555/3277203.3277260

[109]. S. Sargsyan, J. Hakobyan, M. Mehrabyan, M. Mishechkin, V. Akozin, and S. Kurmangaleev. ISP-Fuzzer: Extendable Fuzzing Framework. Proc. of 2019 Ivannikov Memorial Workshop (IVMEM), pp. 68-71, 2019. doi: 10.1109/IVMEM.2019.00017

[110]. М. В. Мишечкин, В. В. Акользин, Ш. Ф. Курмангалеев. Архитектура и функциональные возможности инструмента ИСП Фаззер. Открытая конференция ИСП РАН им. В.П. Иванникова, 2020.

[111]. A. Vishnyakov, A. Fedotov, D. Kuts, A. Novikov, D. Parygina, E. Kobrin, V. Logunova, P. Belecky, S. Kurmangaleev. Sydr: Cutting Edge Dynamic Symbolic Execution. Ivannikov ISPRAS Open Conference (ISPRAS), pp. 46-54, 2020. doi: 10.1109/ISPRAS51486.2020.00014

[112]. C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. REDQUEEN: Fuzzing with Input-to-state Correspondence. Proc. of Network and Distributed System Security Symposium, 2019. doi: 10.14722/ndss.2019.23371

[113]. G. Savidov, A. Fedotov. Casr-Cluster: Crash Clustering for Linux Applications. 2021 Ivannikov ISPRAS Open Conference (ISPRAS), pp. 47-51, 2021. doi: 10.1109/ISPRAS53967.2021.00012
[114]. CASR: Crash Analysis and Severity Report. URL: https://github.com/ispras/casr (доступ 05.12.2023)
[115]. D. Molnar, X. C. Li, and D. A. Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. Proc. of 18-th USENIX Security Symposium, pp. 67-82, 2009. doi: 10.5555/1855768.1855773
[116]. W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. Proc. of 38-th International Conference on Software Engineering, pp. 820-831, 2016. doi: 10.1145/2884781.2884844
[117]. J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case Reduction for C Compiler Bugs. Proc. of ACM SIGPLAN Notices, 47(6):335-346, 2012. doi: 10.1145/2345156.2254104
[118]. J. Foote. GDB exploitable plugin. URL: https://github.com/jfoote/exploitable (доступ 19.06.2023)
[119]. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. Engler. EXE: Automatically Generating Inputs of Death. Proc. of 13-th ACM Conference on Computer and Communications Security, pp 322-335, 2006. doi: 10.1145/1180405.1180445
[120]. KLEE Symbolic Virtual Machine. URL: https://github.com/klee/klee
[121]. A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. Proc. of 14-th USENIX Conference on Offensive Technologies (WOOT'20), article 10. USENIX Association, 2020. doi: 10.5555/3488877.3488887
[122]. AFL++. URL: https://github.com/AFLplusplus/AFLplusplus (доступ 05.12.2023)
[123]. S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. Proc. of 26-th USENIX Security Symposium, pp. 167-182, 2017. doi: 10.5555/3241189.3241204
[124]. Boofuzz. URL: https://github.com/jtpereyda/boofuzz (доступ 19.06.2023)
[125]. Defensics. URL: https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html (доступ 05.12.2023)
[126]. P. Tsankov, M. T. Dashti, and D. Basin. SecFuzz: Fuzz-Testing Security Protocols. Proc. of 7-th International Workshop on Automation of Software Test (AST), pp. 1-7, 2012. doi: 10.1109/IWAST.2012.6228985
[127]. T. L. Munea, H. Lim, and T. Shon. Network Protocol Fuzz Testing for Information Systems and Applications: a Survey and Taxonomy. Multimedia Tools and Applications, 75:14745-14757, 2016. doi: 10.1007/s11042-015-2763-6
[128]. X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. ACM SIGPLAN Notices, 46(6):283-294, 2011. doi: 10.1145/1993316.1993532
[129]. Csmith. URL: https://github.com/csmith-project/csmith (доступ 20.06.2023)
[130]. C. Holler, K. Herzig, and A. Zeller. Fuzzing with Code Fragments. Proc. of 21-th USENIX Security Symposium, pp. 445-458, 2012. doi: 10.5555/2362793.2362831
[131]. H. Ma. A Survey of Modern Compiler Fuzzing. 2023. doi: 10.48550/arXiv.2306.06884. URL: https://arxiv.org/abs/2306.06884
[132]. A. Henderson, H. Yin, G. Jin, H. Han, and H. Deng. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. In: M. Dacier, M. Bailey, M. Polychronakis, M. Antonakakis (eds). Research in Attacks, Intrusions, and Defenses (RAID 2017). LNCS, 10453:3-25, Springer, 2017. doi: 10.1007/978-3-319-66332-6_1
[133]. M. Eceiza, J. L. Flores and M. Iturbe. Fuzzing the Internet of Things: a Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded Systems. IEEE Internet of Things Journal, 8(13):10390-10411, 2021. doi: 10.1109/JIOT.2021.3056179
[134]. M. Eisele, M. Maugeri, R. Shriwas, C. Huth, and G. Bella. Embedded Fuzzing: a Review of Challenges, Tools, and Solutions. Cybersecurity, 5, article 18, 2022. doi: 10.1186/s42400-022-00123-y
[135]. J. Yun, F. Rustamov, J. Kim, and Y. Shin. Fuzzing of Embedded Systems: A Survey. ACM Computing Surveys, 55(7):1-33, article 137, 2023. doi: 10.1145/3538644
[136]. O. Whitehouse. Introduction to Anti-fuzzing: a Defence in Depth Aid. 2014. URL: http://research.nccgroup.com/2014/01/02/introduction-to-anti-fuzzing-a-defence-in-depth-aid (доступ 05.12.2023)

[137]. E. Edholm, D. Göransson. Escaping the Fuzz – Evaluating Fuzzing Techniques and Fooling Them with Anti-fuzzing. M.S. thesis, Chalmers University of Technology, 2016.

[138]. C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. Proc. of 25-th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 184-196, 1998. doi: 10.1145/268946.268962

[139]. P. Junod, J. Rinaldini, J. Wehrli and J. Michielin. Obfuscator-LLVM — Software Protection for the Masses. Proc. of 2015 IEEE/ACM 1-st International Workshop on Software Protection, pp. 3-9, 2015. doi: 10.1109/SPRO.2015.10

[140]. J. Zhang, Z. Li, Y. Liu, Z. Sun, and Z. Wang. SAFTE: a Self-injection Based Anti-fuzzing Technique. Computers and Electrical Enginerring, vol. 111, part B, 108980, 2023. doi: 10.1016/j.compeleceng.2023.108980

[141]. C. CC. Cheng, L. Lin, C. Shi, Y. Guan. An Anti-fuzzing Approach for Android Apps. In G. Peterson, S. Shenoi (eds), Digital Forensics 2023: Advances in Digital Forensics XIX, IFIP Advances in Information and communication Technology, Springer, vol. 687, pp. 37-53, 2023. doi: 10.1007/978-3-031-42991-0_3

[142]. Z. Zhou, C. Wang, and Q. Zhao. No-Fuzz: Efficient Anti-fuzzing Techniques. In: F. Li, K. Liang, Z. Lin, S. K. Katsikas. (eds). Security and Privacy in Communication Networks 2022. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 462, pp. 731-751. Springer, 2023. doi: 10.1007/978-3-031-25538-0_38

[143]. Z. Zhou, and C. Wang. Practical Anti-fuzzing Techniques with Performance Optimization. IEEE Open Journal of the Computer Society, vol. 4, pp. 206-217, 2023. doi: 10.1109/OJCS.2023.3301883

[144]. J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim. FUZZIFICATION: Anti-fuzzing Techniques. Proc. of 28-th USENIX Conference on Security Symposium (SEC'19), pp. 1913–1930, 2019. doi: 10.5555/3361338.3361471

[145]. E. Güler, C. Aschermann, A. Abbasi, and T. Holz. ANTIFUZZ: Impeding Fuzzing Audits of Binary Executables. Proc. of 28-th USENIX Conference on Security Symposium (SEC'19), pp. 1931-1947, 2019. doi: 10.5555/3361338.3361472

[146]. ANTIFUZZ. URL: https://github.com/RUB-SysSec/antifuzz (доступ 05.12.2023)

[147]. Y. Li, G. Meng, J. Xu, C. Zhang, H. Chen, X. Xie, H. Wang, and Y. Liu. Vall-nut: Principled Anti-grey Box – Fuzzing. Proc. of IEEE 32-nd International Symposium on Software Reliability Engineering, pp. 288-299, 2021. doi: 10.1109/ISSRE52982.2021.00039

[148]. Z. Hu, Y. Hu, and B. Dolan-Gavitt. Chaff Bugs: Deterring Attackers by Making Software Buggier, 2018, arXiv:1808.0065. URL: https://arxiv.org/abs/1808.00659 (доступ 05.12.2023)

[149]. D. R. Kaprekar. On Kaprekar Numbers. Journal of Recreational Mathematics, 13(2):81-82, 1980.

[150]. E. Bartocci, Y. Falcone (eds). Lectures on Runtime Verification. Introductory and Advanced Topics. LNCS 10457, Springer, 2018. ISBN: 9783319756318

[151]. D. Drusinsky. The Temporal Rover and the ATG Rover. In: K. Havelund, J. Penix, W. Visser. (eds). SPIN Model Checking and Software Verification (SPIN 2000). LNCS 1885:323-330, 2000, Springer. doi: 10.1007/10722468_19

[152]. K. Havelund and G. Roşu. Java PathExplorer – A Runtime Verification Tool. Proc. of 6-th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS'01), 2001.

[153]. M. Leucker and C. Schallhart. A Brief Account of Runtime Verification. Journal of Logic and Algebraic Programming, 78(5):293-303, 2009. doi: 10.1016/j.jlap.2008.08.004

[154]. Y. Falcone, S. Krstić, G. Reger, and D. Traytel. A Taxonomy for Classifying Runtime verification Tools. International Journal on Software Tools for Technology Transfer, 23:255-284, 2021. doi: 10.1007/s10009-021-00609-z

[155]. C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstić, J. M. Lourenço, D. Nickovic, G. J. Pace, J. Rufino, J. Signoles, D. Traytel, and A. Weiss. A Survey of Challenges for Runtime Verification from Advanced Application Domains (beyond Software). Formal Methods in System Design, 54:279-335, 2019. doi: 10.1007/s10703-019-00337-w

[156]. A. R. Cavalli, T. Higashino, and M. Núñez. A Survey on Formal Active and Passive Testing with Applications to the Cloud. Annals of Telecommunications, 70:85-93, 2015. doi: 10.1007/s12243-015-0457-8

[157]. I. Itkin, R. Yavorskiy. Overview of Applications of Passive Testing Techniques. Modeling and Analysis of Complex Systems and Processes, 2019. URL: https://ceur-ws.org/Vol-2478/paper9.pdf (доступ 20.06.2023)

[158]. A. Edwards, T. Jaeger, and X. Zhang. Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. Proc. of 9-th ACM Conference on Computer and Communications Security, pp. 225-234, 2002. doi: 10.1145/586110.586141

[159]. M. K. Sarrab. Policy-Based Runtime Verification of Information Flow. PhD Thesis, Software Technology Research Laboratory, De Monfort University, UK, 2011.

[160]. D. Efremov and I. Shchepetkov. Runtime Verification of Linux Kernel Security Module. Proc. of International Workshop on Formal Methods, LNCS 12233:185-199, Springer, 2020. doi: 10.1007/978-3-030-54997-8_12. URL: https://arxiv.org/pdf/2001.01442.pdf

[161]. Д. В. Ефремов, В. В. Копач, Е. В. Корныхин, В. В. Кулямин, А. К. Петренко, А. В. Хорошилов, И. В. Щепетков. Мониторинг и тестирование модулей операционных систем на основе абстрактных моделей поведения системы. Труды Института системного программирования РАН, 33(6):15-26б 2021. doi: 10.15514/ISPRAS-2021-33(6)-2

[162]. E. Bartocci, B. Bonakdarpour, and Y. Falcone. First International Competition on Runtime Verification. In: B. Bonakdarpour, S. A. Smolka (eds.). Runtime Verification 2014. LNCS 8734:1-9, Springer, 2014. doi: 10.1007/978-3-319-11164-3_1

[163]. Y. Falcone, D. Ničković, G. Reger, and D. Thoma. Second International Competition on Runtime Verification. In: E. Bartocci, R. Majumdar (eds). Runtime Verification 2015. LNCS 9333:405-422, Springer, 2015. doi: 10.1007/978-3-319-23820-3_27

[164]. G. Reger, S. Hallé, and Y. Falcone. Third International Competition on Runtime Verification. In: Y. Falcone, C. Sánchez (eds). Runtime Verification 2016. LNCS 10012:21-37, Springer, 2016. doi: 10.1007/978-3-319-46982-9_3

[165]. M. Delahaye, N. Kosmatov, and J. Signoles, Common Specification Language for Static and Dynamic Analysis of C Programs. Proc. of 28-th Annual ACM Symposium on Applied Computing, pp. 1230-1235, 2013. doi: 10.1145/2480362.2480593

[166]. E-ACSL. URL: https://frama-c.com/fc-plugins/e-acsl.html (доступ 21.06.2023)

[167]. Код E-ACSL. URL: https://github.com/evdenis/e-acsl (доступ 21.06.2023)

[168]. ANSI/ISO C Specification Language. https://frama-c.com/html/acsl.html (доступ 21.06.2023)

[169]. S. Navabpour, Y. Joshi, C. W. W. Wu, S. Berkovich, R. Medhat, B. Bonakdarpour, S. Fischmeister. RiTHM: a Tool for Enabling Time-Triggered Runtime Verification for C Programs. Proc. of 9-th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013), pp. 603-606, 2013. doi: 10.1145/2491411.2494596

[170]. R. Medhat, Y. Joshi, B. Bonakdarpour, and S. Fischmeister. Accelerated Runtime Verification of LTL Specifications with Counting Semantics. In Y. Falcone, C. Sánchez (eds). Runtime Verification 2016, LNCS 10012:251-267, Springer, 2016. doi: 10.1007/978-3-319-46982-9_16. URL: https://arxiv.org/abs/1411.2239

[171]. C. Colombo, G. J. Pace, and G. Schneider. LARVA — Safer Monitoring of Real-Time Java Programs. Proc. of 7-th IEEE International Conference on Software Engineering and Formal Methods, pp. 33-37, 2009. doi: 10.1109/SEFM.2009.13

[172]. LARVA. URL: http://www.cs.um.edu.mt/~svrg/Tools/LARVA/ (доступ 21.06.2023)

[173]. Код LARVA. URL: https://github.com/ccol002/larva-rv-tool (доступ 21.06.2023)

[174]. C. Colombo, G. J. Pace, and G. Schneider. Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. Proc. of Formal Methods for Industrial Critical Systems (FMICS 2008), LNCS 5596:135-149, Springer, 2008. doi: 10.1007/978-3-642-03240-0_13

[175]. Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O'Neil Meredith, T.-F. Serbanuta, and G. Roşu. RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties. In: B. Bonakdarpour and A. Smolka (eds). Runtime Verification 2014, LNCS 8734:285-300, Springer, 2014. doi: 10.1007/978-3-319-11164-3_24

[176]. Код RV-Monitor. URL: https://github.com/runtimeverification/rv-monitor (доступ 21.06.2023)

[177]. Y. Falcone, P. Meredith, T. F. Şerbănuţă, S. Shiriashi, A. Iwai, and G. Roşu. RV-Android: Efficient Parametric Android Runtime Verification, a Brief Tutorial. In: E. Bartocci, R. Majumdar (eds). Runtime Verification 2015. LNCS 9333:342-357, Springer, 2015. doi: 10.1007/978-3-319-23820-3_24

[178]. G. Reger, H. C. Cruz, and D. E. Rydeheard. MarQ: Monitoring at Runtime with QEA. Proc. of 21-st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015), LNCS 9035:596-610, Sringer, 2015. doi: 10.1007/978-3-662-46681-0_55

[179]. N. Decker, J. Harder, T. Scheffel, M. Schmitz, and D. Thoma. Runtime Monitoring with Union-Find Structures. Proc. of 22-nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016), LNCS 9636:868-884, Springer, 2016. doi: 10.1007/978-3-662-49674-9_54

[180]. Mufin Project. URL: https://www.isp.uni-luebeck.de/mufin (доступ 21.06.2023)

[181]. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: a Fast Address Sanity Checker. Proc. of USENIX Annual Technical Conference, pp. 309-318, 2012. doi: 10.5555/2342821.2342849

[182]. AddressSanitizer. URL: https://github.com/google/sanitizers/wiki/AddressSanitizer (доступ 22.06.2023)

[183]. QASan (QEMU-AddressSanitizer). URL: https://github.com/andreafioraldi/qasan (доступ 22.06.2023)

[184]. W. Han, B. Joe, B. Lee, C. Song, and I. Shin. Enhancing Memory Error Detection for Large-Scale Applications and Fuzz Testing. Proc. of Network and Distributed System Security Symposium, 2018. doi: 10.14722/ndss.2018.23318.

[185]. S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. ACM SIGPLAN Notices, 44(6):245-258, 2009. doi: 10.1145/1543135.1542504

[186]. S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. ACM SIGPLAN Notices, 45(8):31-40, 2010. doi: 10.1145/1837855.1806657

[187]. B. Lee, C. Song, T. Kim, and W. Lee. Type Casting Verification: Stopping an Emerging Attack Vector. Proc. of 24-th USENIX Security Symposium, pp. 81-96, 2015. doi: 10.5555/2831143.2831149

[188]. I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. TypeSan: Practical Type Confusion Detection. Proc. of ACM SIGSAC Conference on Computer and Communications Security, pp. 517-528, 2016. doi: 10.1145/2976749.2978405

[189]. Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer. HexType: Efficient Detection of Type Confusion Errors for C++. Proc. of ACM SIGSAC Conference on Computer and Communications Security, pp. 2373-2387, 2017. doi: 10.1145/3133956.3134062

[190]. X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. Proc. of 24-th ACM Symposium on Operating System Principles, pp. 260-275, 2013. doi: 10.1145/2517349.2522728

[191]. Valgrind. URL: https://valgrind.org/ (доступ 21.06.2023)

[192]. J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. Proc. of USENIX Annual Technical Conference, pp. 2, 2005. doi: 10.5555/1247360.1247362

[193]. D. Bruening and Q. Zhao. Practical Memory Checking with Dr. Memory. Proc. of International Symposium on Code Generation and Optimization, pp. 213-223, 2011. doi: 10.1109/CGO.2011.5764689

[194]. E. Stepanov and K. Serebryany. MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++. Proc. of IEEE/ACM International Symposium on Code Generation and Optimization, pp. 46-55, 2015. doi: 10.1109/CGO.2015.7054186

[195]. MemorySanitizer in LLVM/Clang. URL: https://clang.llvm.org/docs/MemorySanitizer.html (доступ 22.06.2023)

[196]. W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding Integer Overflow in C/C++. ACM Transactions on Software Engineering and Methodology, 25(1):1-29, 2015. doi: 10.1145/2743019

[197]. UndefinedBehaviorSanitizer in LLVM/Clang. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html (доступ 22.06.2023)

[198]. K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. Proc. of Workshop on Binary Instrumentation and Applications, pp. 62-71, 2009. doi: 10.1145/1791194.1791203

[199]. ThreadSanitizer in LLVM/Clang. URL: https://clang.llvm.org/docs/ThreadSanitizer.html (доступ 22.06.2023)

[200]. R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT — a Formal System for Testing and Debugging Programs by Symbolic Execution. ACM SIGPLAN Notices, 10(6):234-245, 1975. doi: 10.1145/390016.808445

[201]. W. E. Howden. Methodology for the Generation of Program Test Data. IEEE Transactions on Computers, C-24(5):554-560, 1975. doi: 10.1109/T-C.1975.224259
[202]. J. C. King. A New Approach to Program Testing. Proc. of International Conference on Reliable Software, pp. 228-233, 1975. doi: 10.1145/800027.808444
[203]. J. C. King. Symbolic Execution and Program Testing. Communications of the ACM, 19(7):385-394, 1976. doi: 10.1145/360248.360252
[204]. C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. Communications of ACM, 56(2):82-90, 2013. doi: 10.1145/2408776.2408795
[205]. R. Baldoni, E. Coppa, D. Cono D'Elia, C. Demetrescu, and I. Finocchi. A Survey of Symbolic Execution Techniques. ACM Computing Surveys. 51:3(1-39), art. 50, 2018. doi: 10.1145/3182657. URL: https://arxiv.org/abs/1610.00502
[206]. T. Avgerinos, S. K. Cha, B.T.H. Lim, and D. Brumley. AEG: Automatic Exploit Generation. Proc. of Network and Distributed System Security Symposium, pp. 283-300, 2011.
[207]. X. Mi, S. Rawat, C. Giuffrida, and H. Bos. LeanSym: Efficient Hybrid Fuzzing Through Conservative Constraint Debloating. Proc. of 24-th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), pp. 62-77, 2012. doi: 10.1145/3471621.3471852
[208]. P. Godefroid. Compositional Dynamic Test Generation. ACM SIGPLAN Notices, 42(1):47-54, 2007. doi: 10.1145/1190215.1190226
[209]. P. Godefroid and D. Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. Proc. of International Symposium on Software Testing and Analysis (ISSTA'11), pp. 23-33, 2011. doi: 10.1145/2001420.2001424
[210]. X. Xie, B. Chen, Y. Liu, W. Le, and X. Li. Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis. Proc. of 24-th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16), pp. 61-72, 2016. doi: 10.1145/2950290.2950340
[211]. K. L. McMillan. Lazy Annotation for Program Testing and Verification. Proc. of 22-nd International Conference on Computer Aided Verification (CAV'10), LNCS 6174:104-118, 2010. doi: 10.1007/978-3-642-14295-6_10
[212]. Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao. Postconditioned Symbolic Execution. Proc. of IEEE 8-th International Conference on Software Testing, Verification and Validation (ICST), pp. 1-10, 2015. doi: 10.1109/ICST.2015.7102601
[213]. V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient State Merging in Symbolic Execution ACM SIGPLAN Notices, 47(6):193-204, 2012. doi: 10.1145/2345156.2254088
[214]. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: a New Approach to Computer Security via Binary Analysis. Proc. of 4-th International Conference on Information Systems Security ((ICISS'08), LNCS 5352:1-25, 2008. doi: 10.1007/978-3-540-89862-7_1
[215]. BitBlaze: Binary Analysis for Computer Security. URL: http://bitblaze.cs.berkeley.edu/ (доступ 27.06.2023)
[216]. D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. Proc. of 23-rd International Conference on Computer Aided Verification (CAV'11), LNCS 6806:463-469, 2011. doi: 10.1007/978-3-642-22110-1_37
[217]. D. Kus. Towards Symbolic Pointers Reasoning in Dynamic Symbolic Execution. arXiv 2109.03698, 2022. URL: https://arxiv.org/abs/2109.03698 (доступ 05.12.2023)
[218]. Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. Proc. of IEEE Symposium on Security and Privacy, pp. 138-157, 2016. doi: 10.1109/SP.2016.17
[219]. S. Poeplau and A. Francillon. Symbolic Execution with SymCC: Don't Interpret, Compile! Proc. of 29-th USENIX Security Symposium, pp. 181-198, 2020. doi: 10.5555/3489212.3489223
[220]. L. Borzacchiello, E. Coppa, C. Demetrescu. FUZZOLIC: Mixing Fuzzing and Concolic Execution. Computers and Security, 108(C), art 102368, 2021. doi: 10.1016/j.cose.2021.102368
[221]. T. Wang, T. Wei, Z. Lin, and W. Zhou. IntScope: Automatically Detecting Integer Overflow Vulnerability in x86 Binary using Symbolic Execution. Proc of Network and Distributed System Security Simposium, 2009.

[222]. Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. SAVIOR: Towards Bug-Driven Hybrid Testing. Proc. of IEEE Symposium on Security and Privacy, pp. 1580-1596, 2020. doi: 10.1109/SP40000.2020.00002. URL: https://arxiv.org/abs/1906.07327

[223]. S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. ParmeSan: Sanitizer-Guided Greybox Fuzzing. Proc. of 29-th USENIX Conference on Security (SEC'20), article 129, pp. 2289-2306. doi: 10.5555/3489212.3489341

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

[225]. I. K. Isaev, D. V. Sidorov. The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs. Programming and Computer Software, 36(40):225-236, 2010. doi: 10.1134/S0361768810040055

[226]. М.К. Ермаков, А.Ю. Герасимов. Avalanche: применение параллельного и распределенного динамического анализа программ для ускорения поиска дефектов и уязвимостей. Труды Института системного программирования РАН, 25:29-38, 2013.

Информация об авторах / Information about authors

Виктор Вячеславович КУЛЯМИН – кандидат физико-математических наук, доцент кафедры Системного программирования ВМК МГУ, ведущий научный сотрудник ИСП РАН. Сфера научных интересов: программная инженерия, тестирование на основе моделей, формальные методы программной инженерии.

Victor Viatcheslavovitch KULIAMIN – PhD, Associate Professor of System Programming Department, Faculty of Computational Mathematics and Cybernetics Moscow State University, Leading Researcher of the Institute for System Programming, Russian Academy of Sciences. Research interests: software engineering, model based testing, formal methods of software engineering.