Мост между SAST и фаззингом: как из сработки SAST получить подтверждённую уязвимость

Источник. Блог компании INFERA Security на HABR

https://habr.com/ru/companies/infera_security/articles/1061880/

Инструменты статического анализа (SAST) лишь подсвечивают вероятные уязвимости, генерируя гипотезы. Динамическое тестирование (DAST) и фаззинг, напротив, выявляют реальные сбои на работающем приложении и фиксируют вектор атаки, но не указывают на конкретную строку в исходниках. Традиционно эти два подхода существуют в изоляции, образуя методологический разрыв, преодолеть который способен только AppSec-эксперт путем кропотливого ручного триажа.

В этой статье мы расскажем про то, как в INFERA AI.SafeCode уменьшаем поток ложных срабатываний в целом, зачем для этого формируем единый реестр знаний о проекте, и как строим мост от SAST к автоматически подтверждаемой уязвимости.

1. Настоящая проблема – не сканеры, а ложные срабатывания

Сканеров сегодня много, и по отдельности они работают неплохо. Но если начать их использовать для реального повышения защищённости, то оказывается, что на большой кодовой базе инструменты статического анализа выдают тысячи предупреждений в неделю, и большая часть из них – **ложные срабатывания** (false positive). Сканер честно говорит «здесь подозрительная конструкция», но между «подозрительно» и «реально эксплуатируется» чаще всего возникает пропасть.

Эту пропасть закрывает человек. AppSec-инженер вынужден вручную анализировать код: мысленно выстраивать цепочку прохождения данных от точки входа (entry point) до уязвимого вызова (sink), проверять наличие механизмов фильтрации или санитизации, и лишь затем выносить вердикт реальная это угроза или ложное срабатывание (false positive). Разбор одного такого алерта занимает от пары минут до десятков, если требуется глубоко погрузиться в контекст. При потоке в тысячу срабатываний в неделю компании требуется отдельная команда, которая не исправляет код, а лишь фильтрует шум сканеров. Иначе это превращается в бесконечный бэклог безопасности, разбор которого растягивается на месяцы или даже годы.

Поэтому задача, которую мы решаем, звучит не как «найти побольше», а как **резко сократить долю ложных срабатываний, не потеряв настоящие уязвимости**. Это принципиально разные задачи. Найти больше обычно несложно: достаточно понизить пороги у любого сканера. Найти больше и при этом не утонуть в шуме – вот это сложнее.

Ключевая мысль всей статьи: высокая точность не достигается за счёт одного секретного приёма, а состоит из **нескольких шагов (слоёв)**, каждый из которых отрезает часть шума. А чтобы эти слои работали, нужно то, чего у отдельно взятого сканера просто отсутствует из-за особенностей его архитектуры – **единая память о проекте**.

Но для начала, начнем с того, почему без человека здесь до сих пор не обходились.

2. Почему связать статику и динамику обычно может только эксперт

Статический и динамический анализ отвечают на разные вопросы, и отвечают на разных языках.

- **Статический анализ (SAST)** позволяет «читать» исходный код, не запуская его. Инженер видит структуру: вот запрос к базе, вот в него подставляется переменная. Но он не знает, дойдёт ли до этой строки реальный запрос пользователя, и не может проверить свою же догадку.
- **Динамический анализ (DAST) и фаззинг** подразумевает запуск программы и направление в неё входных данных. Они дают убедительное доказательство: конкретный запрос или конкретный набор байтов, на котором приложение отвечает как уязвимое или падает. Но это не позволяет понять, *какое* место в коде стоит проверять. Если мы говорим конкретно о фаззинге, то чтобы прицелиться, кто-то должен заранее написать нагрузку для фаззера (небольшую обёртку (harness), которая знает, какую функцию и как дёргать).

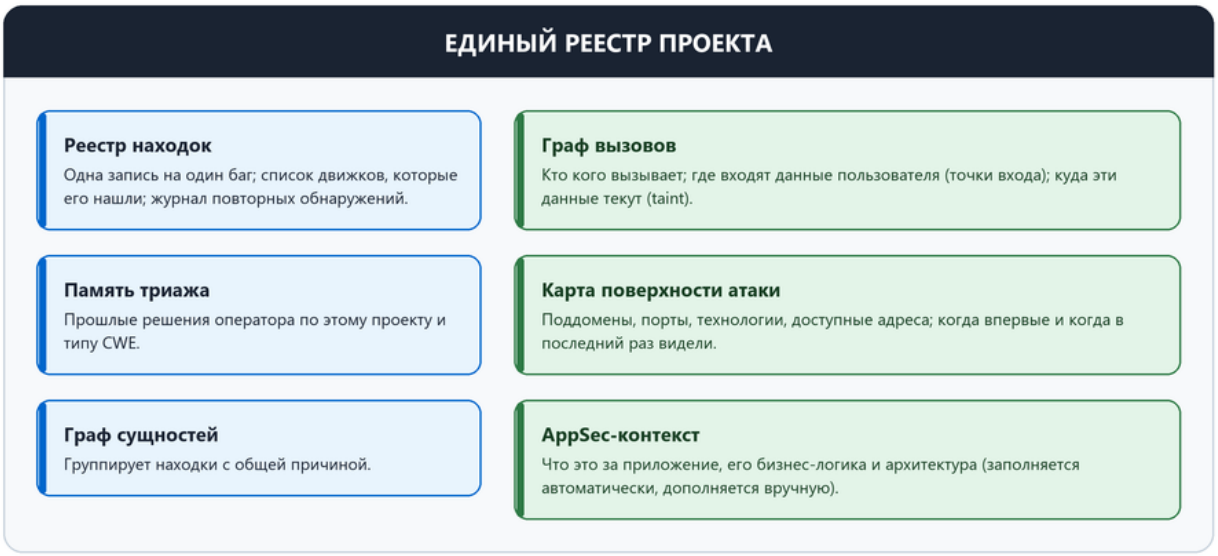
Между этими двумя мирами и сидит эксперт. Именно он делает работу, которую ни один из инструментов сам не делает:

1. смотрит на статическую находку и решает, **достижима** ли опасная строка от реального пользовательского ввода;
2. проверяет, нет ли на пути **очистки данных** (санитайзера, параметризованного запроса), которая делает находку безопасной;
3. если сомневается, то придумывает, **как это воспроизвести**: пишет запрос к стенду или готовит вход для фаззера;
4. связывает статическую догадку с динамическим доказательством и выносит финальный вердикт.

Это дорого и сложно масштабируется. Наша цель: переложить эту цепочку рассуждений на платформу. Не «заменить эксперта одной нейросетью», а **разложить его работу на воспроизводимые шаги и автоматизировать каждый**. Самый наглядный из этих шагов – сделать автоматический мост от статической находки к сбою в фаззинге.

3. Единый реестр: что платформа знает о проекте

Отдельный сканер видит только то, что нашёл прямо сейчас, и забывает это после прогона. Эксперт так не работает: у него в голове держится цельная картина приложения. Мы воспроизводим эту картину в виде **единого реестра проекта** – это общее хранилище, которое хранится в INFERA AI.SafeCode, куда стекается вся информация о том, как проект устроен и как он функционирует. Реестр накапливается от скана к скану и не привязан к одному инструменту.



Что в него входит:

- **Реестр находок.** Одна и та же уязвимость, найденная тремя движками на одной строке, и это не три записи, а одна, с пометкой, кто её видел. При каждом новом скане в журнал добавляется факт «эту находку снова обнаружили тогда-то». Так исчезают дубли и появляется история.
- **Граф вызовов.** Модель того, как код вызывает сам себя: какие функции кого дёргают, где в приложение входят данные пользователя (точки входа) и куда эти данные дальше текут. Именно граф отвечает на вопрос «а достижима ли вообще эта опасная строка».
- **Карта поверхности атаки.** Всё, что «торчит наружу»: поддомены, открытые порты, обнаруженные технологии, доступные адреса. Данные копятся во времени и видно, что появилось недавно, а что живёт давно.
- **Граф сущностей.** Связывает находки, у которых общая первопричина, чтобы чинить корень, а не десять симптомов по отдельности.
- **AppSec-контекст.** Человеческое описание проекта: что это за сервис, какая у него бизнес-логика и архитектура. Часть заполняется автоматически, часть руками владельца.
- **Память триажа.** Прошлые вердикты оператора: «в этом проекте по такому классу уязвимостей мы уже пять раз решили, что это ложное срабатывание,

вот по какой причине». Без этой памяти любой автоматический судья повторяет одни и те же ошибки на типовых для проекта паттернах.

Смысл реестра простой: **находка оценивается не в вакууме, а на фоне всего, что мы знаем о проекте.** Это первое, что делает эксперт, и первое, чего лишён одиночный сканер.

4. Кто какой информацией пользуется

Реестр не пассивный склад. Каждый инструмент и каждый шаг проверки читает из него ровно тот срез, который ему сейчас нужен, и кладёт результат обратно, обогащая картину для следующих шагов.

Шаг / инструмент	Что читает из реестра	Что кладёт обратно
Слой SAST-сканеров	исходный код репозитория	находки с меткой каким движком найдено
Кросс-дедупликация	все находки скана и их «отпечатки»	одна запись вместо N; список движков, согласных по этому месту
Построение графа вызовов	исходный код	граф вызовов: узлы, рёбра, точки входа, источники недоверенных данных
Классификатор достижимости	граф вызовов и путь находки	метка: достижимо / вероятно / недостижимо / неизвестно
Консенсус-детектор	список нашедших движков и соседние находки	сколько независимых движков согласны
AST-фильтр	фрагмент кода из находки	есть ли рядом безопасная конструкция (параметризованный запрос, санитайзер)
Проверка на стенде (DAST)	адрес стенда + статус проверки эксплойта	сработал ли эксплойт на живом приложении
Fuzz-подтверждение	находка (файл, строка, CWE) + «свои» пространства имён проекта	воспроизводимый сбой (crash-proof) либо «не подтверждено»
LLM-триаж (судья)	фрагмент кода или доказательства эксплуатации, память	вердикт «настоящая / ложная» и уверенность

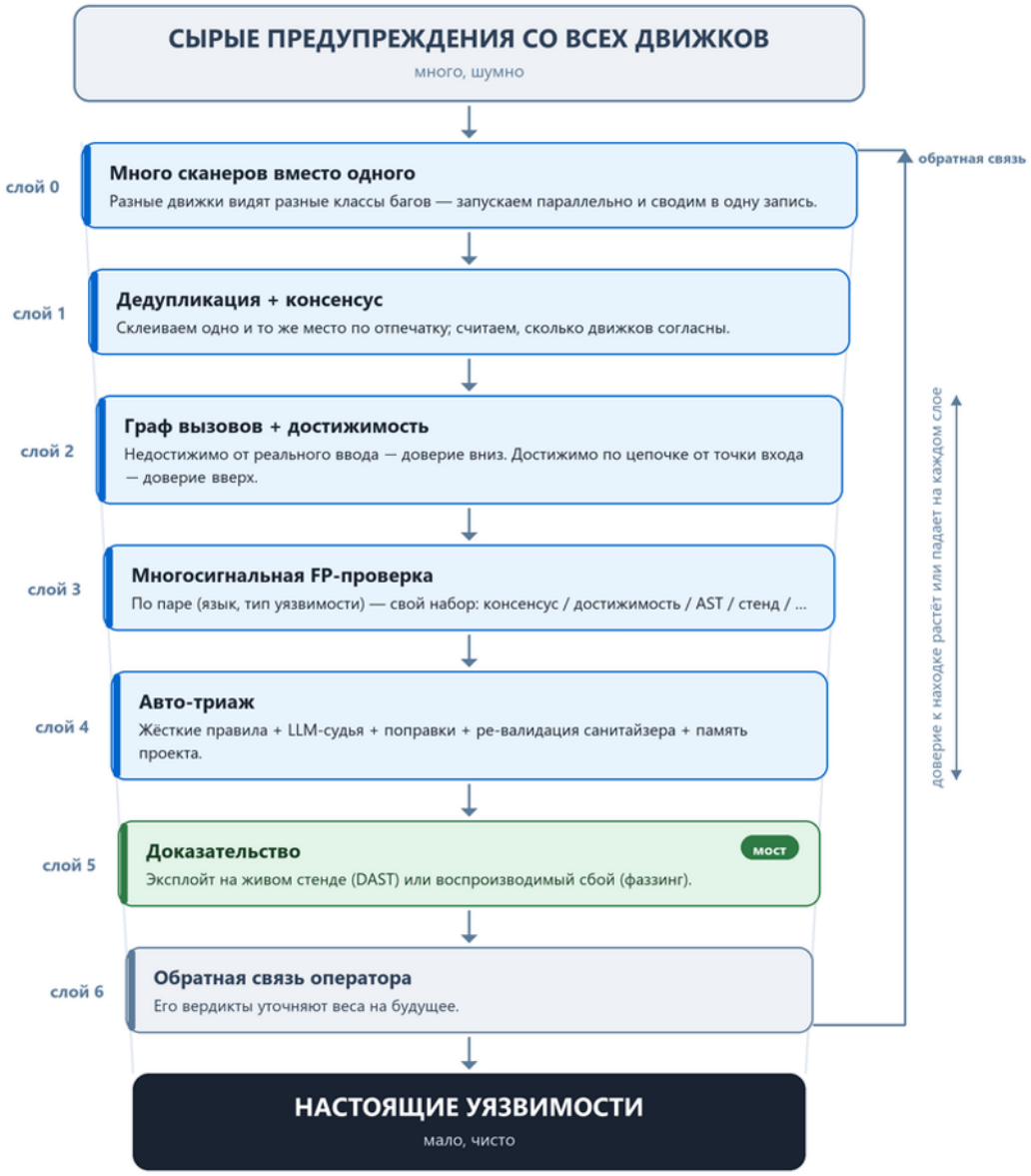
Шаг / инструмент	Что читает из реестра	Что кладёт обратно
	триажа, сигналы детекторов	
Наведение DAST (pre-seed)	контракты API, точки входа из графа, история сканов	приоритетные маршруты, куда бить в первую очередь для проверки

Обратите внимание на связи между строками. Граф вызовов строит один инструмент, а его результатом – метками достижимости – пользуются и классификатор достижимости, и авто-триаж, и наведение DAST.

Метку «этот движок тоже нашёл это место» ставит дедупликация, а читает её консенсус-детектор. Именно так разрозненные сканеры перестают быть разрозненными: они общаются не напрямую, а через общий реестр памяти. Это и есть автоматизация той работы, которую раньше делал AppSec-инженер.

5. Как набирается точность: слои фильтрации

Ни один отдельный сигнал не даёт высокой точности. Параметризованный запрос бывает опасным, а согласие двух движков ещё не гарантия уязвимости. Поэтому точность мы набираем также слоями: сырой поток предупреждений проходит через воронку, где каждый слой либо отсекает шум, либо повышает или понижает доверие к находке.



Пройдём по слоям.

Слой 0: несколько сканеров вместо одного. Часто разные движки сильны в разном: один хорош на инъекциях, другой на небезопасной десериализации, третий на утечках секретов. Под разными движками мы понимаем не только два движка одного семейства (например, SAST на правилах и ML модель для поиска уязвимостей), но и разные типы инструментов. Далее сводим результаты в одну запись. Это не про «больше шума», а про полноту: то, что пропустит один, вероятнее всего поймает другой.

Слой 1: дедупликация и консенсус. Одинаковые находки от разных движков склеиваются по устойчивому «отпечатку» (файл, тип уязвимости, нормализованное место). Дальше работает простое, но сильное наблюдение: если три независимых движка показали на одно и то же место с одним классом CWE, вероятность ложного

срабатывания резко падает. Одиночная находка – это не приговор что это «ложь», но и не повод для высокого доверия.

Слой 2: граф вызовов и достижимость. Здесь мы отвечаем на главный вопрос эксперта: «а дойдёт ли до этой строки реальный ввод пользователя?». По графу вызовов ищем путь от точки входа (обработчик HTTP-маршрута, чтение аргументов командной строки) до опасного места. Нашли путь – ставим метку «достижимо», доверие поднимается вверх. Строка лежит в тестах, в примерах или в мёртвом коде, куда ввод не попадает, – «недостижимо», доверие вниз. Граф строится один раз и обслуживает все находки скана.

Слой 3: многосигнальная проверка на ложность. Универсального детектора ложных срабатываний не существует: инъекция в SQL проверяется иначе, чем DOM-XSS, а утечка секрета – иначе, чем небезопасная десериализация в Java. Поэтому каждую находку мы направляем к своему набору детекторов, подобранному по паре **(язык, тип уязвимости)**:

- **консенсус** – сколько движков согласны;
- **достижимость** – метка из графа вызовов;
- **AST-фильтр** – смотрит на сам фрагмент кода: есть ли параметризованный запрос, известный санитайзер, нормализация пути;
- **проверка на стенде** – сработал ли эксплойт на живом приложении;
- **проверка секрета** – действителен ли найденный ключ на самом деле;
- **консенсус по цепочкам десериализации** – специально для «гаджет-цепочек» в Java.

Каждый детектор оставляет сигнал с уверенностью: «похоже на настоящую», «похоже на ложную» или «нейтрально». Сигналы не перетирают друг друга, а собираются вместе.

Слой 4: авто-триаж. Здесь сигналы сводятся в решение. Сначала работают **жёсткие правила**:

- есть воспроизведённый эксплойт или падение, это подтверждённая уязвимость;
- по графу вызовов место недостижимо, значит ложное срабатывание.

Если ни одно жёсткое правило не сработало, подключается **LLM-судья**: ему мы передаём фрагмент кода (для статики) или доказательства эксплуатации (для динамики) и подсказку из памяти проекта: «раньше по этому классу здесь мы решали, что это …..».

Вердикт судьи затем корректируется детерминированными **поправками**: достижимо – прибавить уверенности, согласны три движка – ещё прибавить, найден санитайзер – убрать, одиночная находка – ещё убрать. Сумма сравнивается с порогами и попадает в одну из трёх корзин: «настоящая», «ложная», «на ручной разбор».

И последний штрих – **ре-валидация санитайзера**. Если судья сказал «ложное срабатывание, потому что тут есть очистка данных», мы не верим на слово:

проверяем, действительно ли названный санитайзер присутствует во фрагменте. Если не нашли, вердикт «FP» оспаривается, и находка не подавляется.

Слой 5: доказательство. Всё, что дошло до этого слоя, можно попробовать доказать по-настоящему: воспроизвести эксплойт на стенде или получить сбой в фаззинге. Это самый дорогой и самый убедительный слой, т.к. именно здесь по-настоящему оживает мост между SAST и фаззингом.

Слой 6: обратная связь. Финальный вердикт всегда за человеком. Но каждое его решение мы запоминаем и складываем в память триажа и в общий журнал. Со временем это позволит перевести веса детекторов от равных к пропорциональным их реальной точности, так система учится на решениях оператора и текущем проекте.

Насколько слои режут шум, лучше показать на цифрах. Наивный подход «просто запустим фаззер по сотне находок» даёт порядка 50 сбоев, из которых настоящих уязвимостей получается около 5: аналитик всё равно разбирает 50 отчётов. Те же находки, пропущенные через слои, дают примерно 5–7 подтверждённых и около 45 «не подтверждено». В результате и человек смотрит на 7, а не на 50.

6. Мост между SAST и фаззингом крупным планом

Идея моста простая. **Если у нас уже есть статическая находка с конкретным файлом, строкой и категорией CWE, то этого достаточно, чтобы автоматически собрать нагрузку и нацелить фаззинг именно на эту точку.** А значит, можно автоматизировать «проверку гипотезы», которую обычно делают руками.

Реализован он как отдельный шаг конвейера: сканер находит подозрительное место, платформа по CWE подбирает категорию опасных точек (sink), запускает короткий прогон фаззера с остановкой на первом сбое (таймаут 15 минут по умолчанию), и результат пишет обратно в находку. Самое интересное – что именно считать «доказательством».

6.1 Конвейер целиком

Сначала расскажем как всё устроено сверху, без деталей классификации:

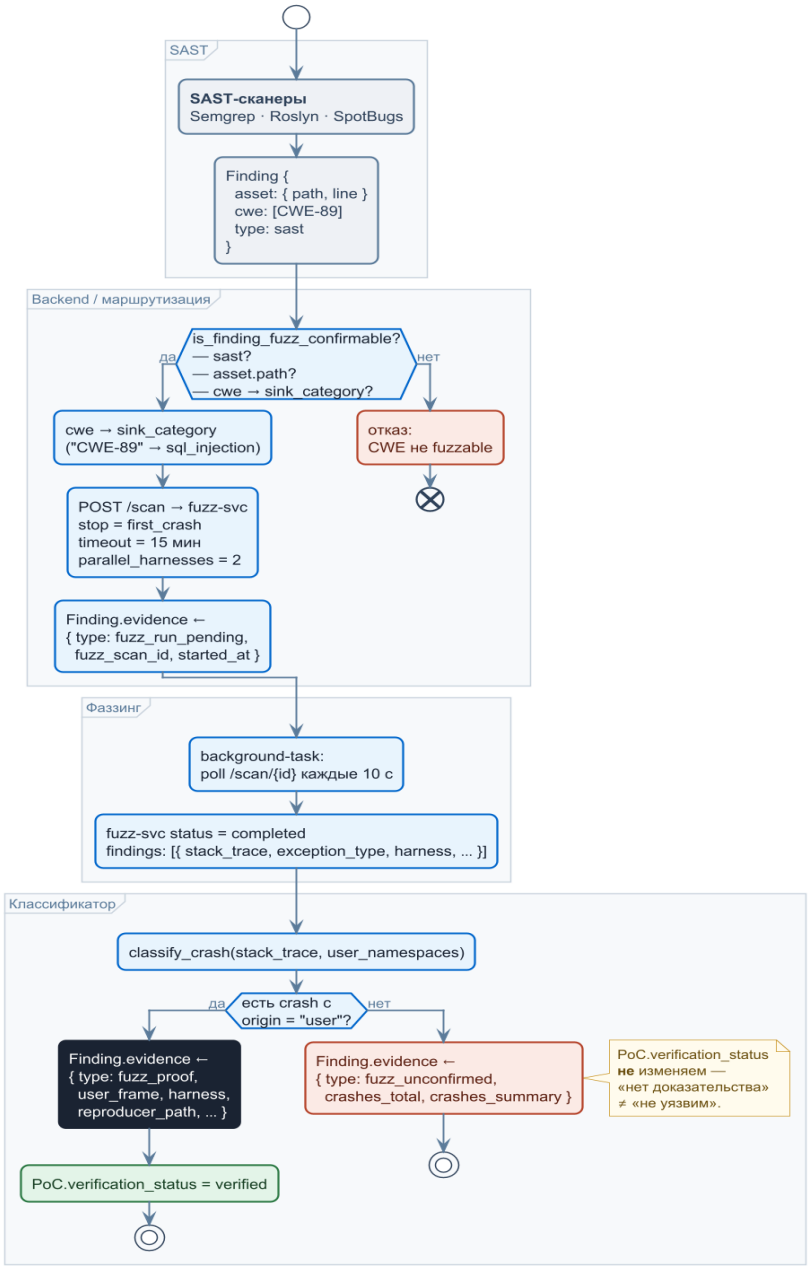


Схема 1. Конвейер от SAST к подтверждению. От находки сканером до записи доказательства

На схеме две неочевидные точки. Первая – проверка «можно ли вообще подтвердить фаззингом»: не каждую статическую находку имеет смысл фаззить (про это в 6.2). Вторая – классификация сбоя: именно она держит низкий уровень шума (про это в 6.3).

6.2 Какие уязвимости так проверяются, а какие нет

Фаззинг работает там, где есть локальное опасное место, принимающее внешние данные, и заметный побочный эффект – падение, исключение, зависание, – который виден в стеке вызовов. Если же уязвимость про разрешения, про логику между

запросами или про мультиролевую авторизацию, фаззер на стек не отзовётся: такое ловят динамическим тестом или анализом графа вызовов.

Поддерживаемое отображение «от CWE к тому, что ищет фаззер»:

CWE	Категория опасных точек	Что фаззер ищет	Опасный эффект
CWE-22	обход пути	чтение файла по собранному пути	выход за пределы базовой директории
CWE-78	инъекция команды	запуск процесса с собранной командной строкой	«лишний» процесс или падение
CWE-89	SQL-инъекция	запрос с конкатенацией	обработчик падает или зависает
CWE-91	XML/XPath	инъекция через парсер XML	XPathException, XmlException
CWE-94	инъекция кода	динамическое вычисление выражений	ScriptException, ArgumentException
CWE-502	небезопасная десериализация	BinaryFormatter и аналоги	SerializationExceptio, нехватка памяти
CWE-611	внешние сущности XML (XXE)	расширение внешних сущностей	XmlException, чтение внешних файлов
CWE-643	XPath-инъекция	инъекция через SelectNodes и аналоги	
CWE-918	подделка запроса (SSRF)	сетевой вызов по собранному URL	таймаут на подконтрольный канал
CWE-1333	«злая» регулярка (ReDoS)	регулярное выражение без таймаута	зависание проверки

Что **не** покрыто и почему:

CWE	Почему не фаззингом
CWE-862 (нет проверки доступа)	Нет «плохого входа», есть отсутствующий вызов проверки. Фаззер этого не увидит; нужен анализ графа вызовов.

CWE	Почему не фаззингом
CWE-639 (обход по идентификатору, IDOR)	Проблема в логике авторизации между запросами разных пользователей – нужно обрабатывать бизнес логику.
CWE-915 (переназначение полей)	Уязвимость в сопоставлении полей объекта. Падения нет; нужна семантика «это поле не должно было прийти от пользователя».
CWE-79 (отражённая XSS)	Эффект в браузере, а не в серверном процессе. Это работа DAST с проверкой через рендеринг.

Проверка «можно ли подтвердить фаззингом» для всех неподдерживаемых CWE честно возвращает «нет» с причиной, а интерфейс прямо показывает: «для этой находки фаззинг-подтверждение неприменимо, нужен такой-то метод».

6.3 Главная тонкость: настоящее падение против шума

Самое неинтуитивное наблюдение во всей этой работе: **большинство падений, которые находит фаззер, к коду пользователя отношения не имеют.**

Если кормить случайные байты в стандартный десериализатор JSON, рано или поздно он упадёт где-то глубоко внутри себя. Это не уязвимость, а нормальная реакция парсера на мусор. Записать такое падение как «найдено», значит вернуть тот самый поток ложных срабатываний, ради борьбы с которым всё и затевалось.

Решение: **классификация падения по стеку вызовов**. Каждый кадр стека (frame) относим к одной из категорий:

- **свой код** – кадр в пространстве имён проекта (имя класса начинается с известного префикса проекта);
- **стандартная библиотека** – System.*, java.*, builtins.* и подобные;
- **сторонняя зависимость** – всё, что не первое и не второе;
- **неизвестно** – не удалось извлечь имя.

Падение считается доказательством **только** если хотя бы один кадр попал в свой код. Если весь стек в стандартной библиотеке, то это шум парсера: находка получает пометку «не подтверждено», но **не** закрывается как «не уязвима» (отсутствие доказательства – это не доказательство отсутствия).

Дерево решений классификатора:

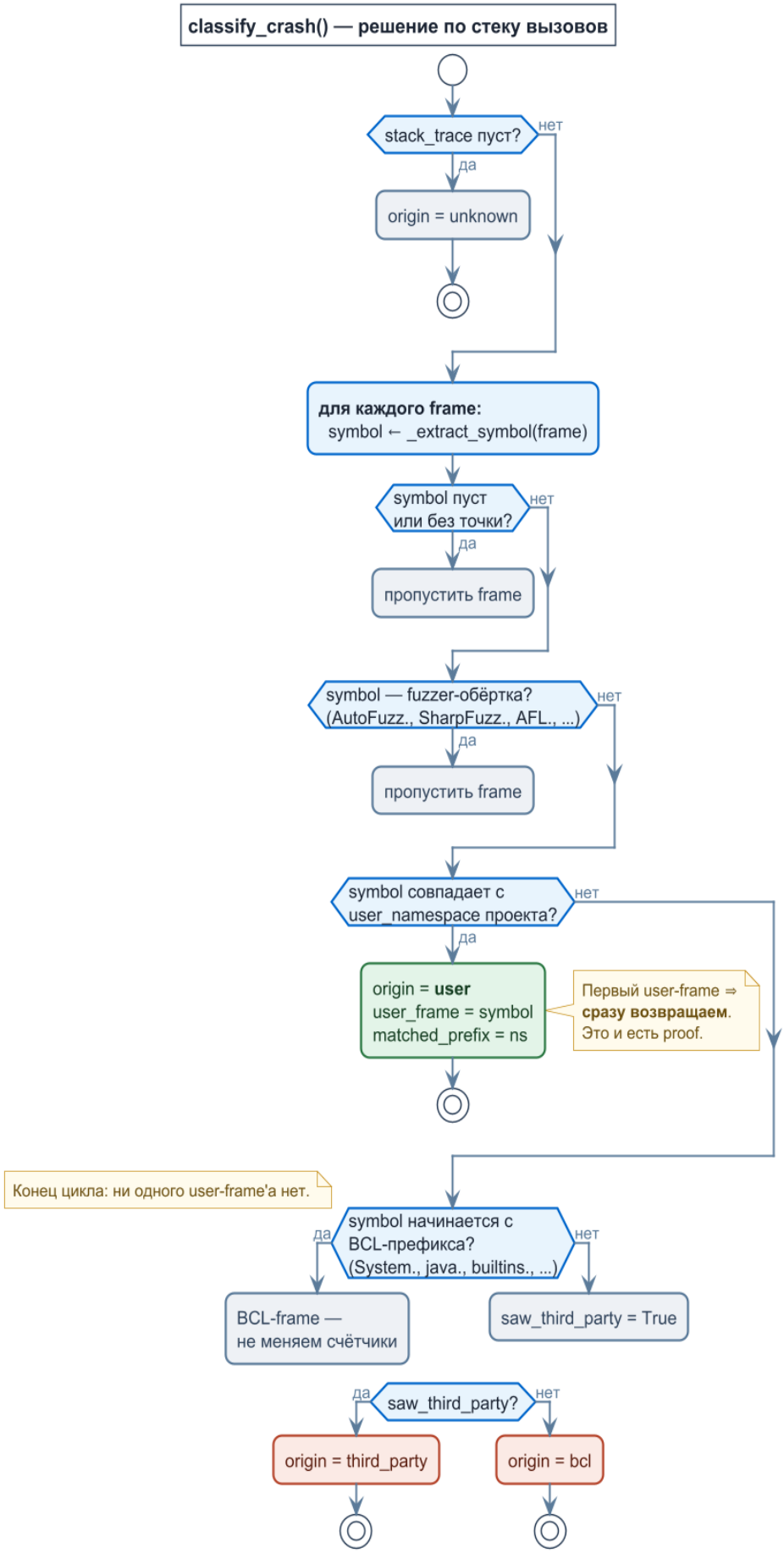

classify_crash() — решение по стеку вызовов
stack_trace пуст?
нет
да
origin = unknown
для каждого frame:
symbol ← _extract_symbol(frame)
symbol пуст
или без точки?
нет
да
пропустить frame
symbol — fuzzer-обёртка?
(AutoFuzz., SharpFuzz., AFL., ...)
нет
да
пропустить frame
symbol совпадает с
user_namespace проекта?
нет
да
origin = user
user_frame = symbol
matched_prefix = ns
Первый user-frame ⇒
сразу возвращаем.
Это и есть proof.
Конец цикла: ни одного user-frame'а нет.
symbol начинается с
BCL-префикса?
(System., java., builtins., ...)
да
нет
BCL-frame —
не меняем счётчики
saw_third_party = True
saw_third_party?
да
нет
origin = third_party
origin = bcl

Схема 2. classify_crash() – дерево решений: каждый frame стека проверяется по очереди, первый user-frame завершает обход с вердиктом proof.

Несколько неочевидных тонкостей.

Обёртки фаззера пропускаются без вердикта. В стеке всегда есть служебные кадры самого фаззера. Они не несут информации ни в какую сторону, поэтому не учитываются, иначе классификатор либо переоценит (примет обёртку за свой код, если префиксы случайно совпадут), либо недооценит.

Первый же «свой» кадр завершает поиск. Идём сверху вниз и на первом совпадении со «своим» кодом останавливаемся. Это важно: глубже по стеку может встретиться кадр стандартной библиотеки, а ещё глубже – снова свой. Если ждать «весь стек свой», доказательство потеряется: в реальности большинство падений упирается в System.*.

Сторонняя зависимость не засчитывается за доказательство. Если данные пробросились через чужую библиотеку и упали внутри неё, мы помечаем это как «зависимость» и не считаем доказательством против кода пользователя. Выбор спорный (можно было бы сказать «уязвимость есть, она в зависимости»), но находка изначально привязана к конкретной строке *нашего* кода, а падение в зависимости не доказывает, что уязвима именно эта строка.

6.4 Откуда берутся «свои» пространства имён

Чтобы классификатор понимал, какой кадр считать «своим», ему нужен список префиксов проекта. Просить пользователя вбивать его вручную – плохо, особенно когда проектов десятки. Поэтому список выводим из самой находки – из пути к файлу, отбрасывая «шумовые» сегменты (src, app, lib, main, …).

Эвристика работает на C#, Java, Kotlin, Python, т.е. везде, где имя каталога примерно совпадает с именем модуля. Список «шумовых» сегментов, которые не являются частью имени, зашит явным перечнем:

```
src, source, sources, app, lib, backend, frontend,
main, java, kotlin, scala, test, tests
```

Всё, что за пределами этого перечня, считаем частью пространства имён. Цена решения: иногда мы пропустим настоящий сбой в своём коде, потому что список префиксов не угадал точно. Окупается это тем, что мы **никогда не запишем шумовое падение как доказательство**, и порог доверия к подтверждённой находке остаётся высоким.

6.5 Что попадает в находку и что видит пользователь

Фаззинг-подтверждение – это долгая операция (15+ минут). Поэтому находка проходит через явные состояния, и каждое из них видно в интерфейсе:

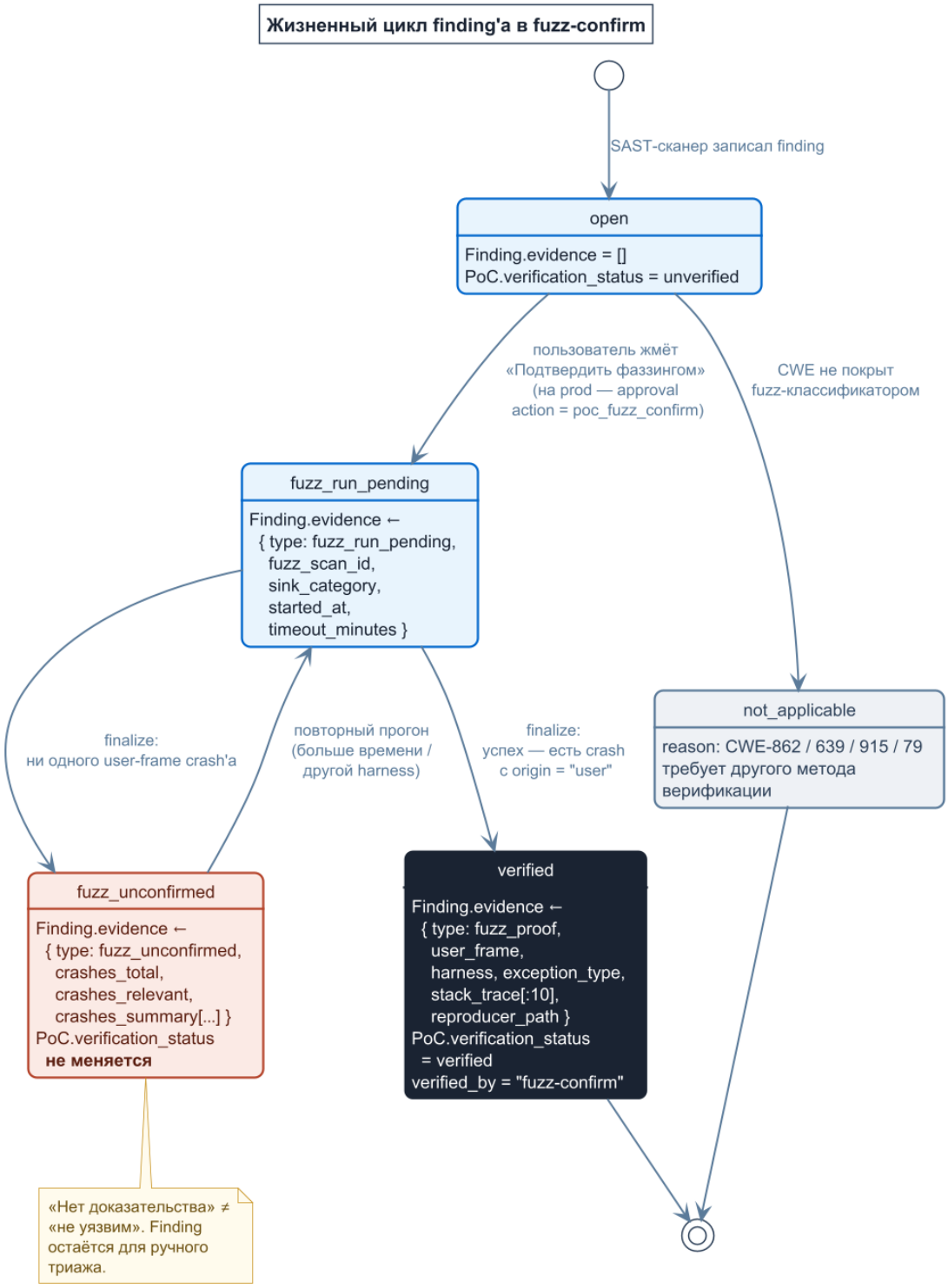


Схема 3. Жизненный цикл finding'а в fuzz-confirm. Конечный автомат жизненного цикла finding'а: open - fuzz_run_pending - verified | fuzz_unconfirmed | not_applicable. Из unconfirmed возможен повторный прогон с другими параметрами.

Несколько решений, которые стоит выделить.

Сводку по сбоям сохраняем даже при отрицательном вердикте. Если фаззер нашёл падения, но все они в стандартной библиотеке или в зависимостях, то это всё

равно материал для аналитика. Может десериализатор здесь и правда лишний, а может и нет. Решает в данном случае человек, но факты у него под рукой.

Статус проверки PoC при «не подтверждено» не трогаем. PoC мог быть проверен другим способом: ручным запуском или проверкой на стенде. Отсутствие фаззинг-доказательства не должно понижать ранее установленный статус. Только положительный вердикт фаззинга поднимает статус. Сами статусы проверки меняются лишь в сторону большего доверия.

Лимиты по умолчанию – **15 минут, две параллельные нагрузки.** Числа подобраны эмпирически: первое падение на типичной нагрузке (десериализатор, парсер XML, регулярка без таймаута) находится за 3–8 минут. Большие лимиты резко повышают вероятность шума (фаззер начинает падать в стандартной библиотеке по всё более экзотическим путям) и почти не дают новых сбоев в своём коде. Если нужен глубокий прогон, то это уже отдельный режим скана, а не быстрое подтверждение находки.

7. Границы метода

Лучше всего мост работает на парсерах, десериализаторах и операциях с регулярными выражениями. Слабее на цепочках бизнес-логики, где падение возникает не сразу, а через 5–10 шагов. Совсем не работает на семантических уязвимостях: отсутствие проверки, неверная логика авторизации, обход по идентификатору. Именно поэтому мост представляет только один слой из тех, что описаны в разделе 5, а не вся система целиком.

Несколько практических следствий:

- **Не каждая «не подтверждённая» находка – это ложное срабатывание.** Это критично для пользователя. В INFERA AI.SafeCode мы различаем три разных состояния: «фаззинг нашёл сбой в своём коде» (доказано), «фаззинг ничего не нашёл за 15 минут» (не подтверждено) и «для этой CWE фаззинг неприменим» (другой метод). Смешивать их нельзя.
- **Фаззинг здесь короткий и прицельный.** Мы не ищем все возможные баги. За условные 15 минут получаем или не получаем доказательство для конкретной находки. Это не тот режим, в котором фаззеры обычно работают.
- **Опасные прогоны требуют подтверждения.** Любое подтверждение с большим таймаутом или высокой параллельностью проходит через согласование. Не из-за ресурсов, а потому что «фаззинг боевого кода» создаёт нагрузку на сервисы, и человек должен это явно одобрить.

8. Что это даёт командам

Если коротко:

- **Находки делятся на ясные корзины** вместо одной размытой шкалы «возможно уязвимо». Доказано – в приоритет на починку. Не подтверждено –

на ручной разбор. Неприменимо – на другой метод. Неопределённости в потоке становится меньше.

- **Доказательство воспроизводимо.** Мы сохраняем и вход, на котором возникает сбой, и стек, и имя нагрузки – разработчик может прогнать сам и убедиться. Цикл «нашли – починили – закрыли» заметно ускоряется.
- **Шум падает на порядки.** Это не эффект одного приёма, а сумма слоёв: много движков, дедупликация, достижимость по графу вызовов, многосигнальная проверка, авто-триаж и только в конце – дорогое доказательство. Каждый слой отрезает свою часть, и до человека доходит горстка находок, а не сотни.

9. Заключение

Главная идея этой работы: **доказательство уязвимости должно быть встроено в конвейер**, а не оставаться задачей разработчика «поверить инструменту». Иначе разработчик или AppSec захлебнётся в потоке. А чтобы конвейер сам не захлебнулся в ложных срабатываниях, ему нужна память: единый реестр того, как устроен и как работает проект, из которого каждый инструмент берёт нужный ему срез. Тогда разрозненные сканеры перестают быть разрозненными, а работа, которую раньше делал только эксперт (связать статику с динамикой и вынести вердикт), раскладывается на воспроизводимые слои.

Мост между SAST и фаззингом самый наглядный из этих слоёв. Статика даёт точку входа (файл, строка, CWE), фаззинг даёт верификацию (сбой с воспроизводимым входом). Но без классификации стека вызовов этот мост даёт больше шума, чем сигнала, поэтому рабочее определение «подтверждённой находки» мы держим сознательно узким:

> **Находка считается подтверждённой, если динамический тест нашёл воспроизводимый вход, при котором стек вызовов содержит хотя бы один кадр в пространстве имён проекта, и класс сбоя соответствует CWE находки**.

Если сделать шире, будет шум, уже – пропустим настоящее. При текущих метриках (около 85% точности на подтверждённых находках и около 95% полноты на отсеве чисто-библиотечных падений) баланс держится.

Дальше – новые классы уязвимостей (разбираемся, что делать с XSS – там работает только динамика) и более длинные прогоны для критичных находок. Но и базовый рецепт точности – **слои вместо одного приёма**, и его фундамент – **единый реестр знаний о проекте** – остаются неизменными.

В этой статье мы лишь сверху копнули полный набор технологических возможностей. В следующих статьях цикла расскажем про то, как мы учили движок автопентеста искать уязвимости бизнес-логики, как учили плагин IDE самостоятельно фиксить найденные уязвимости внутри IDE, общаясь с моделью, которая для вас генерит код или о том, как строить MLSecOps.