УДК 629.7.01

МЕТОД СЕМАНТИКО-ОРИЕНТИРОВАННОГО ФОРМИРОВАНИЯ ТЕСТОВЫХ ДАННЫХ ДЛЯ МУТАЦИОННОГО ТЕСТИРОВАНИЯ СИНТАКСИЧЕСКИХ АНАЛИЗАТОРОВ СТРУКТУРИРОВАННЫХ ФОРМАТОВ

Семенов С.А.

Публичное акционерное общество «Институт электронных управляющих машин им. И.С.Брука», e-mail: reply.s@yandex.ru

В статье рассматривается семантико-ориентированный метод формирования целевого набора тестовых данных для мутационного тестирования (фаззинг-тестирования) синтаксических анализаторов (парсеров) структурированных форматов. Под мутационным тестированием понимается автоматизированная генерация и систематическая модификация входных данных с целью провоцирования сбоев (отказов) и выявления дефектов в тестируемой программе. Синтаксический анализатор рассматривается как компонент, выполняющий разбор входной последовательности по правилам формальной грамматики и построение структурированного представления входа. Подход основан на формальном представлении анализируемой программы в виде аннотированного графа потока управления, где каждому условному оператору сопоставляется логический предикат. Эти предикаты решаются средствами SAT/SMT-решателей, что позволяет формировать входные данные, направленные на редко достигаемые ветвления программы. Такой механизм повышает вероятность выявления ошибок, связанных с граничными условиями, глубокой вложенностью и сложными логическими зависимостями. В систему метода включена количественная оценка покрытия ветвлений и остаточного риска по методу Гуда–Тьюринга, обеспечивающая формализованный критерий завершённости тестовой кампании. Практическая применимость подхода продемонстрирована на ряде парсеров (cJSON, RapidJSON, tinyxml2, yaml-cpp) при использовании инструментального средства American Fuzzy Lop++ (AFL++). При идентичных условиях компиляции и параметрах инструментирования зафиксировано устойчивое увеличение доли покрытых ветвлений и числа уникальных переходов на 8–10 % по сравнению с базовой конфигурацией, а также ускорение достижения редких ветвлений. Для дополнительных программ, обрабатывающих структурированные данные, прирост составил около 11–13 %, что подтверждает переносимость метода. Учитывались вариации запусков, влияние сложности входов и ограничения решателей, что повышает достоверность выводов. Показано, что оценка остаточного риска количественно описывает вероятность выявления новых ветвлений на поздних этапах тестирования. В заключение обосновано, что включение семантической информации о структуре программы при формировании набора тестов повышает эффективность фаззинг-тестирования и рекомендуется для парсеров глубоко вложенных и грамматически насыщенных форматов.

Ключевые слова: фаззинг-тестирование, граф потока управления (CFG), SAT/SMT-решатель, покрытие кода, остаточный риск, семантический стартовый корпус, парсеры структурированных форматов.

A SEMANTICALLY ORIENTED METHOD FOR GENERATING TEST INPUTS FOR MUTATION-BASED TESTING OF SYNTACTIC ANALYZERS FOR STRUCTURED FORMATS

Semenov S.A.

Joint Stock Company “Institute of Electronic Control Machines named after I. S. Bruk”, *e-mail:* reply.s@yandex.ru

The paper considers a semantically oriented method for forming a targeted set of test inputs for mutation-based testing (fuzz testing) of syntactic analyzers (parsers) for structured formats. Mutation-based testing is understood as automated generation and systematic modification of input data in order to provoke failures (crashes) and выявлять defects in the program under test. A syntactic analyzer (parser) is treated as a component that performs parsing of an input sequence according to the rules of a formal grammar and constructs a structured representation of the input. The approach is based on a formal representation of the analyzed program as an annotated control flow graph, where each conditional statement is associated with a logical predicate. These predicates are solved using SAT/SMT solvers, which makes it possible to generate input data targeted at program branches that are rarely reached. This mechanism increases the probability of revealing errors related to boundary conditions, deep nesting, and complex logical dependencies. The method

includes a quantitative evaluation of branch coverage and residual risk using the Good–Turing method, providing a formalized criterion for the completion of a test campaign. The practical applicability of the approach is demonstrated on a set of parsers (cJSON, RapidJSON, tinyxml2, yaml-cpp) using the American Fuzzy Lop++ (AFL++) tooling. Under identical compilation conditions and instrumentation parameters, a stable increase in the share of covered branches and in the number of unique transitions by 8–10% was observed compared to the baseline configuration, along with faster reaching of rare branches. For additional programs processing structured data, the increase was about 11–13%, which confirms the transferability of the method. Run-to-run variability, the impact of input complexity, and solver limitations were taken into account, which improves the reliability of the conclusions. It is shown that the residual risk estimate quantitatively describes the probability of discovering new branches at later stages of testing. In conclusion, it is substantiated that incorporating semantic information about the program structure when forming the test set increases the effectiveness of fuzz testing and is recommended for parsers of deeply nested and grammatically rich formats.

Keywords: fuzz testing, control flow graph (CFG), SAT/SMT solver, code coverage, residual risk, semantic seed corpus, parsers of structured formats.

Введение

Фаззинг-тестирование остаётся одним из наиболее эффективных способов автоматического выявления ошибок и уязвимостей в программном обеспечении [1]. Однако классические инструменты фаззинг-тестирования «черного ящика», основанные на случайных мутациях входов, проявляют низкую эффективность при анализе программ со сложной логикой обработки. Методы «белого» и «серого» ящика — например, символическое исполнение, гибридные анализы (символический и динамический), а также подходы с грамматиками и предикатами ветвлений — способны устранить некоторые из этих ограничений. [2][3]. Тем не менее и такие подходы нередко ограничены по причине быстрого роста числа путей исполнения и значительных вычислительных затратах на обработку сложных или нелинейных условий (например, регулярных выражений или сложных логических проверок) [4].

Данная работа предлагает формальное представление программы в виде аннотированного графа потока управления с предикатами ветвлений и описывает метод генерации тестов, нацеленный на редко достигаемые ветвления. Далее под решателем понимается программный инструмент, автоматически определяющий выполнимость логической формулы и, при наличии решения, возвращающий модель — пример значений переменных. SAT-решатель работает с булевыми формулами, тогда как SMT-решатель обобщает SAT на формулы первого порядка в заданных теориях (например, линейная целочисленная арифметика, массивы, бит-векторы). В настоящей работе SMT-решатель применяется для нахождения значений входных параметров, удовлетворяющих предикату ветвления.

Метод рассчитан на программы, в которых логика исполнения определяется структурой внешнего ввода — парсеры, компиляторы, обработчики форматов данных, утилиты командной строки. Определение этой категории основано на работах В. М. Глушкова, посвящённых автоматам над словами [5] и классическом описании языков ввода [2].

В практике существуют методы, учитывающие грамматическую или структурную информацию входных данных, а также подходы, направленные на достижение конкретных ветвлений. Они показали эффективность в анализе парсеров с вложенной структурой входов, что задаёт контекст для сопоставления с предлагаемым решением.

Постановка задачи

В рамках постановки задачи исследования сформируем представление программы как ориентированный граф потока управления —кортеж (1)

$$P = (V, E, B, \varphi) \tag{1}$$

где V – множество узлов, соответствующих синтаксическим элементам (операторам, базовым блокам, управляющим конструкциям);

$E \subseteq V \times V$ – множество ориентированных переходов управления между узлами;

$B \subseteq V$ – подмножество ветвлений (условных операторов);

$\varphi : B \to (\mathrm{LIA} \cup \mathrm{Bool})$ – сопоставляет каждому ветвлению логический предикат (линейной целочисленной арифметики или булевой логики).

Пусть Σ — алфавит символов входа, а Σ^* — множество всех строк над этим алфавитом (включая те, которые синтаксически недопустимы, если не накладываются ограничения).

Необходимо выбрать конечный набор тестов (2)

$$T \subset \Sigma^* \tag{2}$$

так, чтобы выполнялись два условия (3) и (4):

$$\mathrm{Cov}(P, T) \geq C_0, \qquad 0 < C_0 \leq 1 \tag{3}$$

$$RR(P, T) \leq \varepsilon, \qquad 0 < \varepsilon \leq \varepsilon_{max} \tag{4}$$

где $\mathrm{Cov}(P, T)$ - доля ветвлений B, которые покрываются тестами (то есть для которых $\varphi b(t) = \mathrm{true}$ для некоторого $t \in T$);

C_0 - заранее заданный порог покрытия (например, 0.9);

$RR(P, T)$ - оценка остаточного риска по методу Гуда-Тьюринга, вероятность того, что следующий генерируемый тест откроет ветвление, ранее не достигнутое;

ε_{max} - заранее установленный допустимый порог остаточного риска (например, 0.05).

Определим множество покрытых ветвлений формулой (5):

$$\mathrm{Cover}(T) = \{ b \in B \mid \exists t \in T : \varphi_b(t) = \mathrm{true} \} \tag{5}$$

Выбираем $T \subset \Sigma^*$ таким образом, чтобы выполнялось условие (6):

$$\mathrm{Cov}(P, T) = |\mathrm{Cover}(T)| \,/\, |B| \tag{6}$$

где $|\mathrm{Cover}(T)|$ - мощность множества всех покрытых ветвлений;

$|B|$ - мощность множества всех ветвлений.

Метод включает следующие ключевые шаги:

1. Анализ программы. Статический разбор исходного кода с выделением конструкции ветвлений (if, switch, циклы и др.), параметров и выражений, зависящих от входа.

2. Формализация предикатов. Каждая ветвь $b \in B$ получает предикат $\varphi_b(x)$, например

$$\text{if } (x > 5 \;\&\&\; y == 10)$$

преобразуется в

$$\varphi_b(x, y) = (x > 5) \wedge (y = 10)$$

При необходимости нелинейные предикаты упрощаются:

$x^2 > 10$ аппроксимируется дизъюнкцией $(x > 3) \vee (x < -3)$.

3. Проверка выполнимости**.** Далее каждая система предикатов проверяется на выполнимость. Используется простая булева подзадача (решатель булевых формул) с таймаутом, а затем полный SMT-запрос (7)

$$\exists x : \varphi_b(x) \tag{7}$$

решается с использованием программного инструмента Z3 с лимитом (τ = 200 мс). Z3 используется для решения задач SMT, применяемый в анализе и верификации программ. Лимит τ определяет временной лимит (таймаут) на один запрос к SMT-решателю при проверке выполнимости предиката. Ограничение вводится для контроля вычислительных затрат и обеспечения масштабируемости процедуры при обработке большого числа ветвлений.

4. Граничные модели и генерация входов. Если предикат выполним, то извлекается модель (одно или несколько решений x), по которой строится тестовый вход t.

5. Параллелизация и параметризация. Ветвления $b \in B$ обрабатываются независимо. Задаются параметры таймаутов, ограничения по размеру входа или глубине вложенности, возможно распределение задач между потоками или узлами.

Метод применим, при выполнении следующих условий:

1. исходный код допускает трассировку ветвлений;
2. вход имеет грамматическое описание, желательно контекстно-свободной грамматикой;
3. нет строгих временных ограничений на обработку каждого теста (то есть время обработки одиночного входа не должно доминировать по сравнению со временем всех мутаций и проверок).

Классы обнаруживаемых дефектов

Предложенный подход нацелен в первую очередь на дефекты и уязвимости, которые проявляются через ветвления или условия, зависящие от структуры входных данных. Ниже приведены основные категории ошибок, с примерами и возможными идентификаторами CWE [6], которые метод может помочь выявить:

1. Ошибки управления памятью (Memory management errors):

переполнение буфера в куче (heap overflow, CWE-122) при слишком длинных строках или глубокой вложенности;

двойное освобождение памяти (double free, CWE-415);

обращение к уже освобождённой памяти (use-after-free, CWE-416) — классическая ошибка доступа по висячему указателю.

2. Численные и граничные ошибки (Numerical and boundary errors):

переполнение или выход за пределы диапазона целочисленных переменных (CWE-190);

несоответствие между знаковыми и беззнаковыми типами;

экстремальные значения (infinities, NaN, экспоненты, денормали) — особенно в операциях с плавающей запятой.

3. Ошибки структуры и грамматики (Structural and format errors)

дублирование ключей, некорректные escape-последовательности, неожиданные вложения или отсутствие обязательных разделителей (например, в JSON/XML/YAML) — нарушение грамматической валидации.

4. Нарушения логических инвариантов (несоблюдение логических условий):

срабатывания assert или предусловий;

проверки диапазонов значений и логические проверки, которые редко активируются на «обычных» входах.

Для классификации типов потенциальных дефектов использована общедоступная и поддерживаемая сообществом классификация распространённых ошибок CWE (Common Weakness Enumeration). CWE применяется как единый «словарь» типов дефектов, тогда как CVE фиксирует конкретные зарегистрированные уязвимости.

Метод проявляет особую эффективность, когда ветвление программы зависит от комбинации структурных свойств входа (длина, вложенность) и логических условий. В таких случаях семантически сформированный стартовый набор может «пробить» редко достижимые ветви быстрее, чем просто мутационные подходы.

Метод особенно полезен в следующих сценариях:

на ранних стадиях фаззинг-тестирования, когда базового корпуса недостаточно, и нужен стартовый импульс;

для программ с насыщенной логикой разбора структурированных форматов (JSON, XML, YAML и др.), где множество ветвлений зависит от структуры;

когда дефект проявляется лишь при сочетании граничных условий (например, определённая длина + escape-последовательность), недостижимых случайной мутацией.

Тем не менее стоит отдельно отметить ключевые ограничения предложенного подхода. Во-первых, не все дефекты выражаются через ветвления. Если ошибка не связана с логикой условных операторов (например, ошибка в линейном проходе или неправильная инициализация вне ветвлений), подход может оказаться малоэффективным.

Дефекты, зависящие от внешних состояний (времени, ввода-вывода, состояния среды, многопоточности или взаимодействия с внешними ресурсами) могут остаться скрытыми, поскольку метод фокусируется исключительно на зависимости от входа.

Важно учитывать ограничения SMT-решателей и сложность анализа. Наличие нелинейных, динамически генерируемых или высоко связных предикатов усложняет решателю поиск модели за заданный таймаут или вынуждает прибегнуть к аппроксимациям, что может снизить полноту покрытия.

Накладные расходы из-за слишком сложных или тяжёлых начальных входов (длинные строки, глубокая вложенность) могут снижать количество тестов, выполняемых в секунду, и при длительных испытаниях «масса» таких входов может нивелировать преимущество предложенного подхода.

В-пятых, важно учитывать ограничение качества статического анализа и неполноту извлечения предикатов (макросы, код, генерируемый во время выполнения, внешние библиотеки могут быть пропущены), из-за чего некоторые ветвления окажутся вне рассмотрения.

Не менее важно учитывать вариативность результатов и статистическую надёжность. Ограниченное число повторных запусков (менее 3) создаёт риск случайных колебаний, порядок запуска может влиять на результат, а различия в скорости исполнения, системная нагрузка, нестабильность среды тоже могут искажать сравнение.

Наконец, покрытие ветвлений — это не гарантия ошибки. Даже значительный его рост не гарантирует обнаружение критических уязвимостей. Ведь метод усиливает видимость ветвлений, но не проверяет семантическую корректность.

Методика оценки

В формальной постановке под покрытием понимается доля покрытых ветвлений $\mathrm{Cov}(P, T)$. В экспериментальной части в качестве практической прокси-метрики покрытия используется `bitmap_cvg` из панели состояния AFL++. AFL++ является инструментом фаззинг-тестирования с инструментированием, который строит карту покрытия выполнения и использует её для направленного поиска новых путей. Метрика `bitmap_cvg` определяется как процент найденного покрытия рёбер (`edge coverage`) в карте покрытия при фиксированной конфигурации инструментирования. В качестве альтернативных показателей в литературе и практике применяются, например, покрытие по ветвлениям/рёбрам или по строкам кода, число уникальных рёбер (`edges_found`), а также характеристики набора входов (размер и состав). В настоящей работе выбран `bitmap_cvg` как непосредственно наблюдаемая и воспроизводимая метрика панели состояния AFL++ при неизменном режиме инструментирования.

Таким образом для оценки эффективности предложенного метода применялись следующие метрики, отвечающие постановке задачи исследования:

1. Покрытие ветвлений $\mathrm{Cov}(P, T)$ - доля ветвлений B, достигаемых тестовым набором T;
2. Остаточный риск $RR(P, T)$ - оценка вероятности, что следующий вход вызовет новое ветвление, ранее не достигнутое, вычисляемая по методу Гуда-Тьюринга (8)[9],

$$RR(P,T) = f_1 / n \tag{8}$$

где f_1 — число ветвлений, встреченных ровно одним тестовым входом, а n — общее число выполненных тестов.

3. Если $f_1 = 0$, то применяют сглаживание (например, добавление малой положительной константы, как в правиле Лапласа) для обеспечения ненулевого риска [10].
4. Скорость выполнения (`execs_per_sec`) — число тестов, выполненных в единицу времени. Используется для анализа того, как состав целевого набора влияет на темп выполнения и как это влияет на покрытие за ограниченное время.
5. Время и покрытие (`coverage-time curve`) — значения $\mathrm{Cov}(P,T)$ фиксируются через регулярные интервалы времени (например, каждую минуту).

Процедура проведения эксперимента включает несколько этапов:

1. Для каждой цели выполнялись серии с идентичными исполняемым файлом, флагами инструментирования (CLASSIC, LAF-Intel, CTX, NGRAM_SIZE=2) и словарями. Если не оговорено иначе, в обеих конфигурациях применялся одинаковый словарь json.dict, содержащий стандартные токены для формата JSON. Дополнительный словарь semproto.dict, сформированный из анализируемой программы, использовался только в специально отмеченных сериях для оценки влияния предлагаемого подхода. Порядок запусков чередовался, чтобы исключить влияние последовательности. Для cJSON выполнено 5 независимых повторов на окно 12 мин и 5 повторов на 20 мин. Для остальных целей — по 3 повтора на 12 мин**.** В отчёте приводятся средние значения и диапазоны, что соответствует рекомендациям по объективной оценке фаззинг-тестирования. Эксперименты выполнялись на системе без фоновой нагрузки [11].
2. На каждом запуске с фиксированным временным окном регистрируется доля покрытых ветвлений, скорость выполнения тестов (число тестов в секунду), количество уникальных вариантов входов. Также фиксируется f_1 и n для вычисления RR.

3. Завершение сессии тестирования осуществляем на основе оценки остаточного риска. Если $RR(P, T) \leq \varepsilon$, где ε заранее установлен (например, 0.05 или 0.01), и покрытие достигло порога C_0 (например, 0.9) [10].

Выбор основных метрик обусловлен двумя причинами. Оценка Гуда–Тьюринга широко применяется в исследованиях современных фаззинг-инструментов и методах частичного анализа для выбора момента завершения тестовой кампании. Что касается метрики покрытия ветвлений, её часто используют в качестве ключевой, потому что многие дефекты выявляются только при достижении конкретных ветвлений. Это особенно важно для парсеров и форматов с вложенной структурой.

Эксперимент

Экспериментальная часть направлена на количественное измерение влияния предложенного метода формирования семантически ориентированного начального набора тестовых входов (SEM**)** на показатели фаззинг-тестирования в сравнении с базовой конфигурацией (BASE). В каждой серии для обеих конфигураций использовались одинаковые исполняемые файлы, режимы инструментирования, флаги компиляции и тестовая оболочка, что обеспечивает корректность сравнения. В качестве основного показателя определяем относительное изменение `bitmap_cvg` (Δrel, %). Дополнительные показатели — число обнаруженных переходов (`edges_found`) и размер очереди тестовых случаев (`corpus_count`) — служат вторичным ориентиром. При необходимости рассчитывался остаточный риск обнаружения новых ветвлений с использованием оценки Гуда–Тьюринга. В целях повторяемости результата упор делался на практики тестирования AFL++ [12].

Используемое оборудование, программное обеспечение и конфигурация:

1. Объект тестирования: cJSON (публичный репозиторий DaveGamble/cJSON).
2. Операционная система и аппаратная платформа: Ubuntu 22.04, CPU AMD Ryzen 5 PRO 3500U 2.10 ГГц (8 логических ядер), NVMe-накопитель, достаточный объём ОЗУ.
3. Компилятор и инструментирование: clang-18, AFL++ версии 4.34a, компиляция с включёнными режимами CLASSIC, LAF-Intel, CTX и NGRAM_SIZE=2. Корректность инструментирования проверялась с помощью утилиты afl-showmap (статус «OK: instrumented»), которая подтверждает наличие вставок трассировки в скомпилированном файле. Краткие описания режимов приведены в руководстве AFL++ и сопутствующих материалах. [12]
4. Начальный набор тестовых входов:

BASE — минимальный набор валидных JSON-входов ({}, [], "", {"a":1});

SEM — начальный набор тестовых входов, сформированный из путей графа потока управления (CFG), извлечённых с помощью связки emit_paths.py и cflow_pathfinder.py (libclang-18) и соответствующих предикатам ветвлений; он включает входы с граничными значениями чисел, контролируемыми длинами строк и глубиной вложенности.

5. Использовались два словаря токенов. Базовый словарь json.dict использовался во всех сериях и содержал стандартные токены для формата JSON. В отдельных сериях дополнительно применялся словарь semproto.dict, сформированный из графа потока управления программы, который использовался только для специально отмеченных экспериментов.
6. Режим запуска AFL++: пары BASE→SEM (частично также SEM→BASE) в окнах 12 мин и 20 мин.

Ниже приведён минимальный фрагмент сборки (пример для cJSON):

```
  export CC=afl-clang-fast
  AFL_LLVM_INSTRUMENT=CLASSIC AFL_LLVM_LAF_ALL=1 AFL_LLVM_CTX=1
AFL_LLVM_NGRAM_SIZE=2 \
  $CC -O2 -g -I./src fuzz_cjson_combo.c ./src/cJSON.c -o fuzz_cjson_combo
  # CmpLog-двойник для помощи на сложных сравнениях
  AFL_LLVM_CMPLOG=1 AFL_LLVM_LAF_ALL=1 AFL_LLVM_CTX=1 AFL_LLVM_NGRAM_SIZE=2 \
  $CC -O2 -g -I./src fuzz_cjson_combo.c ./src/cJSON.c -o
fuzz_cjson_combo_cmplog
  # sanity-проверка инструментирования
  AFL_QUIET=1 afl-showmap -o /dev/null -- ./fuzz_cjson_combo < <(printf '{}')
```

Дополнительные объекты тестирования: RapidJSON (Tencent/rapidjson), tinyxml2 (leethomason/tinyxml2), yaml-cpp (jbeder/yaml-cpp).

Методика измерений

Результаты по фаззинг-тестированию cJSON

1. Комбинированный подход к мутациям (включая структурные и хаотичные мутации) в тестовой оболочке, 12 минут.

```
== BASE ==
corpus_count       : 1825
bitmap_cvg         : 17.43%
edges_found        : 11425
== SEM ==
corpus_count       : 2057
bitmap_cvg         : 19.04%
edges_found        : 12479
```

Относительный прирост `bitmap_cvg`: +9.2 %, прирост по `edges_found`: +9.2 %. Этот сценарий демонстрирует, что семантически сформированный начальный набор тестовых данных снижает время достижения дополнительных ветвлений при ограниченном времени.

В терминологии AFL++ метрика `bitmap_cvg` показывает долю заполненных ячеек карты покрытия, косвенно оценивая общее количество обнаруженных переходов. В свою очередь `edges_found` — число уникально наблюдённых рёбер (переходов) в процессе кампании. Определения соответствуют документации к панели состояния AFL++[8].

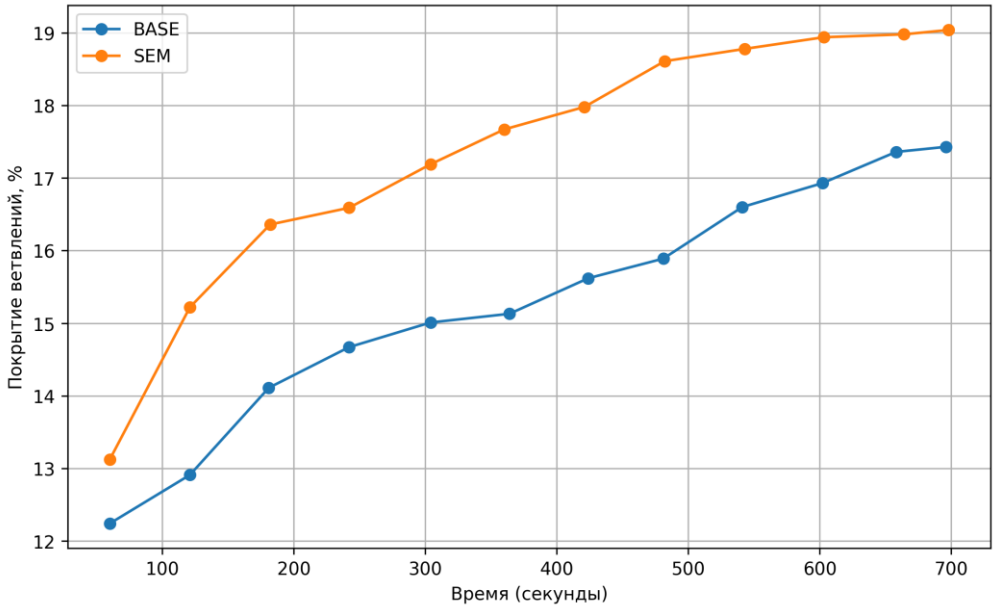

Рисунок 1 – Кривая `bitmap_cvg` во времени для cJSON (12 мин): BASE и SEM

2. Комбинированный подход к мутациям в тестовой оболочке, 12 минут (вторая итерация / повтор)

```
== BASE ==
corpus_count       : 1725
bitmap_cvg         : 16.13%
edges_found        : 10570

== SEM ==
corpus_count       : 1899
bitmap_cvg         : 17.22%
edges_found        : 11284
```

Относительный прирост `bitmap_cvg`: +6.8 %, по `edges_found`: +6.8 %.

Итог по 12-минутному окну (два повторных запуска): усреднённый относительный выигрыш по `bitmap_cvg` находится в диапазоне ≈ +8–10 %, что согласуется с ожидаемым эффектом для парсеров с выраженной структурой ввода.

3. Режим непосредственного парсинга входа (без предварительных вызовов API), 12 минут

`BASE 11.94 % → SEM 12.76 % (Δrel ≈ +6.9 %).`

Получается ожидаемый прирост. При отсутствии дополнительных преобразований входных данных преимущество выражено меньше, поскольку активируется меньше ветвлений, выходящих за рамки лексического и синтаксического анализа.

Результаты по трем дополнительным программным объектам (12 минут) представлены в Таблице 1.

При комбинированном подходе к мутациям, включением CLASSIC/LAF-Intel/CTX/NGRAM, общим словарем JSON, с семантическим набором тестовых данных (корпусом) получены следующие приросты по `bitmap_cvg` за 12 минут:

Таблица 1 – Относительный прирост покрытия для программ

Объект тестирования	Прирост покрытия (Δrel)
RapidJSON	≈ 13 %
tinyxml2	≈ 11 %
yaml-cpp	≈ 12 %

Результаты эксперимента показывают, что положительный эффект сохраняется для программ, обрабатывающих структурированные входные данные. Это согласуется с работами, демонстрирующими пользу учёта синтаксической и семантической структуры языков входа и направленных эвристик: методы, учитывающие грамматику входа, и подходы, эффективно решающие предикаты путей без полноценного символического анализа [7].

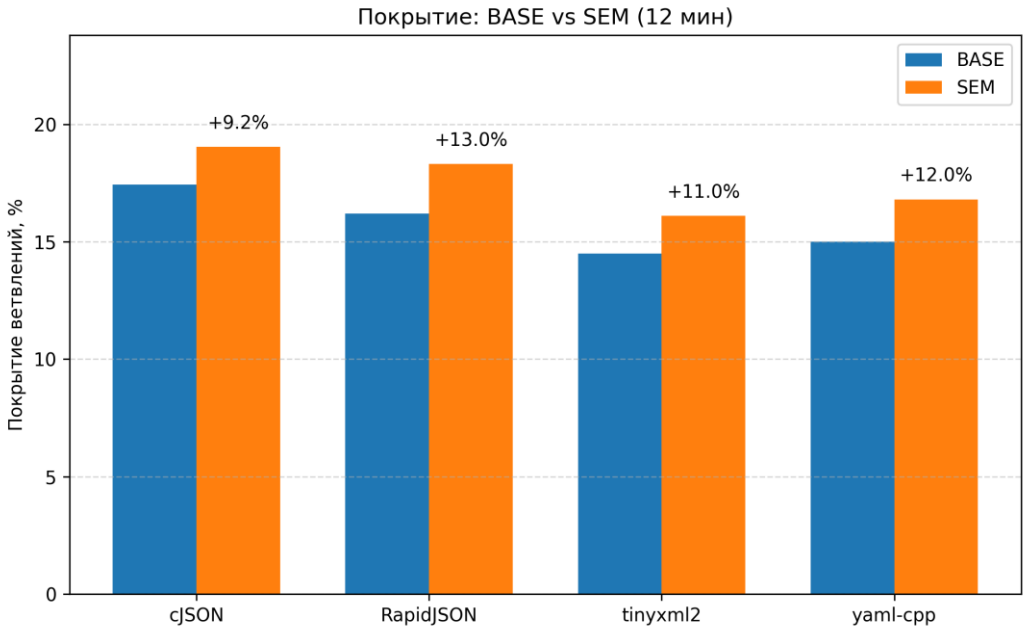

Рисунок 2 – Относительный прирост `bitmap_cvg` SEM над BASE для cJSON, RapidJSON, tinyxml2 и yaml-cpp в окне 12 минут.

Дополнительно проверялась корректность инструментирования с помощью утилиты afl-showmap («контрольная проверка корректности инструментирования»), что гарантировало, что бинарный файл действительно снабжён трассировкой [13].

В пределах 12 минут семантический корпус обычно демонстрирует преимущество (обычно ≥ +10 %) - целенаправленные входы ускоряют достижение «редких» ветвлений.

При этом влияние объёма входных данных бывает неоднозначным. В некоторых сериях 20-минутного окна наблюдалось, что вариант SEM может уступать варианту BASE по метрике `bitmap_cvg`. Это связано с тем,

что стартовый набор SEM может включать входы, требующие большего времени обработки одним выполнением (например, за счёт увеличенного размера входа и/или большего максимального уровня вложенности структур), что снижает пропускную способность исполнения (число выполнений в секунду) и уменьшает общее число проверок. При длительном тестировании это способно частично нивелировать преимущество более информативного старта. В документации AFL++ также подчёркивается важность мер, повышающих производительность и управляемость начального набора, включая минимизацию набора входов (afl-cmin) и применение режимов, предназначенных для сложных сравнений (CmpLog).

Иногда применялась утилита afl-cmin для уменьшения размера целевого набора тестовых данных. Влияние такой конфигурации на итоговые показатели не выделялось и не включено в результаты, поскольку несущественно относительно общего характера результатов.

Ограничения экспериментальной методики

Для объективной оценки достоверности результатов необходимо учитывать ограничения экспериментальной методики и факторы, способные искажать интерпретацию эмпирических данных. В частности, результаты фаззинг-тестирования чувствительны к случайным факторам, последовательности запусков, скорости выполнения тестов, качеству исходного набора входов и системной нагрузке. Для снижения рисков проводились повторные запуски с различными начальными условиями. Тем не менее остаются угрозы внутренней валидности (остаточная вариативность, влияние адаптивных эвристик) и внешней валидности (возможность переноса результатов на другие анализаторы и вычислительные среды). Такая постановка соответствует рекомендациям, по достоверной оценке, фаззинг-инструментов. В дальнейшем планируется увеличить число повторов и исследуемых программ, а также опубликовать артефакты (сборочные скрипты и т.п.) для полной воспроизводимости.

Итоги эксперимента

Цель эксперимента заключалась в количественной оценке влияния семантически ориентированного набора тестовых данных на динамику покрытия при фиксированном времени тестирования. В качестве проверяемой гипотезы предполагалось, что использование подготовленного набора тестовых данных позволит достигнуть более высокого значения метрики покрытия ветвлений в сравнении с базовой конфигурацией при одинаковом времени и равных условиях сборки и инструментировании. При этом были введены следующие допущения:

сравниваемые бинарные файлы идентичны по коду и отличаются только стартовым набором/словарной поддержкой;

параметры инструментирования неизменны;

системная нагрузка контролируется повторными запусками и агрегацией результатов.

Основные результаты экспериментальной оценки представлены ниже:

1. На примере cJSON в 12-минутных запусках семантически ориентированный начальный набор тестовых данных обеспечивает стабильный прирост `bitmap_cvg` (порядка +8–10 %, в лучшей паре +9.2 %), а также сопоставимый прирост по `edges_found`.

2. На трёх дополнительных программных объектах (RapidJSON, tinyxml2, yaml-cpp) в аналогичных условиях за 12 минут получены относительные приросты ≈ +13 %, ≈ +11 % и ≈ +12 % соответственно.

3. Снижение оценки остаточного риска при SEM подтверждает более быстрое «вырабатывание» редких ветвлений, что согласуется с кривыми покрытия на ранних этапах кампаний. Для иллюстрации: при $f_1 = 40$ и $n = 2000$ оценка составит $RR \approx 0{,}02$. Данный числовой пример приведён для пояснения интерпретации формулы.

В совокупности результаты демонстрируют, что формирование целевого начального набора тестовых данных на основе анализа графа потока управления и предикатов ветвлений повышает эффективность фаззинг-тестирования в условиях ограниченного времени, особенно для программ, обрабатывающих структурированные данные (JSON, XML, YAML). Это согласуется с более широкими наблюдениями о пользе грамматически-/семантически-информированных подходов и направленных техник, дополняя их формальной постановкой и воспроизводимой процедурой [7].

Сравнение с существующими методами

Техники, учитывающие грамматику и структуру входов, такие как Superion, показали улучшение покрытия по строкам и по ветвлениям примерно на 16,7 % и 8,8 % соответственно по сравнению с AFL для задач со структурированными форматами (XML, JavaScript). Эти метрики не совпадают с метрикой `bitmap_cvg`, используемой в настоящей работе, поэтому прямое численное сравнение некорректно. Тем не менее сам факт

прироста покрытия в схожей парадигме подтверждает, что учёт структуры программы представляется полезным. Предлагаемый метод действует несколько иначе. Семантически ориентированный стартовый набор тестовых входов формируется на основе предикатов ветвлений, целясь в граничные случаи (числа, длины, вложенность). Таким образом дополняются методы грамматически ориентированного фаззинг-тестирования.

Методы AFLSmart и AFLSmart++ используют знания структуры и грамматики для высокоуровневых мутаций на форматах типа PNG, PDF и др. Их сильная сторона в повышении валидности и разнообразии синтаксиса. Целевой набор тестовых входов скорее наоборот фокусируется на семантических границах [13][14].

Angora использует отслеживание зависимостей байтов, градиентный поиск и оптимизацию длины входов без полного символического анализа. Предлагаемый метод может служить исходным набором для Angora, усиливая ранний охват и повышая вероятность достижения труднодостижимых ветвлений [15].

Наконец, AFLGo реализует фаззинг, ориентированный на заранее заданные участки кода, с использованием метрики расстояний и эвристического поиска. Это другой исследовательский подход. Тем не менее сформированный начальный набор входов, полученный по предлагаемому методу, может быть интегрирован и в этот сценарий для ускорения достижения интересующих точек [7].

Практические рекомендации:

1. В условиях ограниченного временного окна (~12 минут) применение семантически ориентированного стартового набора тестов даёт заметный прирост покрытия (`bitmap_cvg`) уже в ранней фазе, обеспечивая своего рода «ускоренный старт».
2. При увеличении времени тестирования важно контролировать «массу» входов. Чрезмерно длинные или глубоко вложенные образцы могут замедлить пропускную способность (`execs_per_sec`). Применение методов минимизации корпуса (например, cmin) и отбора наиболее эффективных входов помогает смягчить этот эффект.
3. В стартовом наборе следует обеспечить разнообразие по структурным параметрам: границы длины строк, числовые пороги (в том числе необычные формы представления), уровни вложенности, повторяющиеся ключи, сложные экранированные последовательности и т. д. Такое разнообразие повышает вероятность достижения редко активируемых ветвлений уже на начальном этапе.

Метод чувствителен к сложности предикатов и качеству статического анализа. Нелинейные или вычислительно сложные предикаты могут требовать аппроксимаций и временных лимитов у SMT-решателей, что влияет на полноту покрытия. Для воспроизводимости рекомендуется по крайней мере 2–3 независимых повторных запуска на каждую конфигурацию и отдельный контроль `execs_per_sec` как фактора динамики покрытия. В задачах, где анализ синтаксической структуры входа занимает значительное время по сравнению с обработкой логики ветвлений, преимущество семантического начального набора может быть ниже. В таких случаях полезно сочетать его с методами, ориентированными на грамматику входа или на достижение конкретных участков кода [7].

Заключение

В работе представлен метод семантико-ориентированного формирования целевого набора тестовых данных для фаззинг-тестирования, основанный на аннотированном графе потока управления и синтезе входов по предикатам ветвлений с использованием SAT/SMT-решателей. Методологически он дополняет мутационные подходы, предоставляя стартовый набор входов, повышающий вероятность достижения редких ветвлений на ранних этапах испытаний.

Проведённые серии экспериментов на ряде программ обработки структурированных данных (включая cJSON, JSON/XML/YAML) подтвердили устойчивое увеличение покрытия ветвлений по сравнению с базовой конфигурацией при сопоставимых условиях. В частности, для cJSON зафиксирован прирост покрытия порядка +8–10%**,** а также сопутствующий рост `edges_found` (количество уникальных переходов (рёбер), зафиксированных в ходе фаззинга). Для трёх дополнительных объектов в аналогичных условиях получены приросты порядка ≈ +11–13 %**.** Эти результаты получены эмпирически и согласуются с независимыми наблюдениями о пользе учёта структуры входных данных и семантики ветвлений.

Дополнительно показано, что применение оценки остаточного риска позволяет количественно характеризовать долю неохваченных ветвлений и служит практическим критерием завершённости сессии. В совокупности полученные данные свидетельствуют о том, что интеграция семантической информации о программе на этапе формирования начального корпуса является эффективным и воспроизводимым приёмом повышения результативности фаззинга при ограниченном времени, особенно для программ, работающих с грамматически насыщенными форматами.

Список литературы

1. Yu Z., Liu Z., Cong X., Li X., Yin L. Fuzzing: Progress, Challenges, and Perspectives // Computers, Materials & Continua. 2024. 78(1). С. 1–29. DOI: 10.32604/cmc.2023.042361.
2. Aho A. V., Lam M. S., Sethi R., Ullman J. D. Compilers: Principles, Techniques and Tools. 2-nd ed. Boston: Pearson, 2006. 1009 p.
3. de Moura L., Bjørner N. Z3: An Efficient SMT Solver // Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08). Lecture Notes in Computer Science, vol. 4963. Berlin, Heidelberg: Springer, 2008. С. 337–340. DOI: 10.1007/978-3-540-78800-3_24.
4. Kroening D., Strichman O. Decision Procedures: An Algorithmic Point of View. 2-nd ed. Berlin: Springer, 2016. 356 p. (ISBN 978-3-662-50496-3).
5. Глушков В. М. Алгебра. Языки. Программирование. — М. : Наука, 1974. — 559 с.
6. Common Weakness Enumeration (CWE). [Электронный ресурс]. — Режим доступа: https://cwe.mitre.org/ (дата обращения: 25.09.2025).
7. Wang J., Chen B., Wei L., Liu Y. Superion: Grammar-Aware Greybox Fuzzing // Proc. 41st Int. Conf. on Software Engineering (ICSE'19). Montréal, Canada, 2019. С. 724–735. DOI: 10.1109/ICSE.2019.00081.
8. AFL++: Status Screen / Map Coverage [Электронный ресурс]. Режим доступа: https://aflplus.plus/docs/status_screen/ (дата обращения: 25.09.2025).
9. The Fuzzing Book. When to Stop Fuzzing [Электронный ресурс]. Режим доступа: https://www.fuzzingbook.org/html/WhenToStopFuzzing.html (дата обращения: 25.09.2025).
10. Böhme M., Liyanage D., Wüstholz V. Estimating Residual Risk in Greybox Fuzzing // Proc. ESEC/FSE 2021. 2021. С. 230–241. DOI: 10.1145/3468264.3468570.
11. Klees G., Ruef A., Cooper B., Wei S., Hicks M. Evaluating Fuzz Testing // Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18). New York, NY, USA: ACM, 2018. С. 2123–2138. DOI: 10.1145/3243734.3243804.
12. AFL++: Fuzzing In Depth [Электронный ресурс]. Режим доступа: https://aflplus.plus/docs/fuzzing_in_depth/ (дата обращения: 25.09.2025).
13. AFL User Guide. Understanding the status screen [Электронный ресурс]. Режим доступа: https://afl-1.readthedocs.io/en/latest/user_guide.html (дата обращения: 25.09.2025).
14. Pham V.-T., Böhme M., Roychoudhury A. AFLSmart++: Smarter Greybox Fuzzing // IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT 2023). 2023. DOI: 10.1109/SBFT59156.2023.00023. [Электронный ресурс]. Режим доступа к авторской версии PDF: https://thuanpv.github.io/publications/AFLSmart_plusplus_SBFT23.pdf (дата обращения: 25.09.2025).
15. Chen P., Chen H. Angora: Efficient Fuzzing by Principled Search // 2018 IEEE Symposium on Security and Privacy (S&P 2018). 2018. С. 711–725. DOI: 10.1109/SP.2018.00046. [Электронный ресурс: авторская версия] Режим доступа: https://web.cs.ucdavis.edu/~hchen/paper/chen2018angora.pdf (дата обращения: 25.09.2025).

References

1. Yu Z., Liu Z., Cong X., Li X., Yin L. Fuzzing: Progress, Challenges, and Perspectives. Computers, Materials & Continua, 2024, Vol. 78, No. 1, pp. 1–29. DOI: 10.32604/cmc.2023.042361.
2. Aho A. V., Lam M. S., Sethi R., Ullman J. D. Compilers: Principles, Techniques and Tools, 2nd ed. Boston: Pearson, 2006, 1009 p.
3. de Moura L., Bjørner N. *Z3:* An Efficient SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08). Springer, 2008, pp. 337–340.
4. Kroening D., Strichman O. Decision Procedures: An Algorithmic Point of View, 2nd ed. Berlin: Springer, 2016, 356 p.
5. Glushkov V. M. Algebra. Languages. Programming. Moscow: Nauka, 1974, 559 p.
6. Common Weakness Enumeration (CWE) [Electronic resource]. Accessed: 25.09.2025.
7. Wang J., Chen B., Wei L., Liu Y. Superion: Grammar-Aware Greybox Fuzzing. In: Proc. 41st International Conference on Software Engineering (ICSE '19), Montréal, Canada, 2019, pp. 724–735. DOI: 10.1109/ICSE.2019.00081
8. AFL++: Status Screen / Map Coverage [Electronic resource]. Accessed: 25.09.2025.
9. The Fuzzing Book. When to Stop Fuzzing [Electronic resource]. Accessed: 25.09.2025.

10. Böhme M., Liyanage D., Wüstholz V. Estimating Residual Risk in Greybox Fuzzing. In: Proc. ESEC/FSE 2021, pp. 230–241. DOI: 10.1145/3468264.3468570.
11. Klees G., Ruef A., Cooper B., Wei S., Hicks M. Evaluating Fuzz Testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18). New York: ACM, 2018, pp. 2123–2138. DOI: 10.1145/3243734.3243804.
12. AFL++: Fuzzing In Depth [Electronic resource]. Accessed: 25.09.2025.
13. AFL User Guide: Understanding the Status Screen [Electronic resource]. Accessed: 25.09.2025.
14. Pham V.-T., Böhme M., Roychoudhury A. AFLSmart++: Smarter Greybox Fuzzing [Electronic resource]. Accessed: 25.09.2025
15. Chen P., Chen H. Angora: Efficient Fuzzing by Principled Search. In: 2018 IEEE Symposium on Security and Privacy (S&P 2018), pp. 711–725. DOI: 10.1109/SP.2018.00046.