Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра информационной безопаснсти
Лаборатория безопасности информационных систем

Николайчук Артём Константинович

Обзор методов тестирования zk circuits для проверки полноты их ограничений

Курсовая работа

Научный руководитель:
м.н.с
М.С.Воронов

Москва, 2024

Аннотация

В настоящее время всё большую популярность получают решения второго уровня(L2) на блокчейне. L2 уровень на блокчейне обозначает второй уровень масштабируемости и решает проблемы масштабирования и скорости транзакций, которые могут возникать на первом уровне блокчейна (L1). L2 может быть представлен различными протоколами, такими как Lightning Network для биткоина или Polygon для Ethereum. Эти протоколы позволяют совершать децентрализованные транзакции и выполнять смарт-контракты вне цепи блоков, что уменьшает нагрузку на основную сеть и повышает ее масштабируемость. L2 уровень также обеспечивает более быстрые и дешевые транзакции, что делает использование блокчейна более удобным и доступным для всех участников сети. В таких протоколах часто используют zk circuit - схемы, которые позволяют доказывать, что какие-то действия были выполнены пользователем, например что пользователь совершил транзакцию токена со своего аккаунта, либо то, что пользователь знает какой-то определённый секрет. Цена ошибки или бага в таких схемах может быть невероятно высокой - потенциальные уязвимости могут быть использованы для кражи всех токенов протокола. Допустить ошибку при разработке легко - иногда достаточно забыть добавить одно условие. Для поиска недостаточно ограниченных схем разумно применять методы автоматического тестирования. Такие методы рассматриваются в этой работе.

Содержание

1 Введение

1.1 zk-rollups

Описание на примере блокчейна Эфириум. ZK-rollup - это протокол вне основной сети, который работает поверх блокчейна Ethereum и управляется встроенными смарт-контрактами Ethereum. ZK-rollup выполняет транзакции за пределами основной сети, но периодически передает пакеты транзакций вне сети в сетевой накопительный контракт. Эта запись транзакции является неизменяемой, как и в блокчейне Ethereum, и формирует цепочку ZK-rollup.

Базовая архитектура ZK-rollup представлена на рисунке 1 и состоит из следующих компонентов:

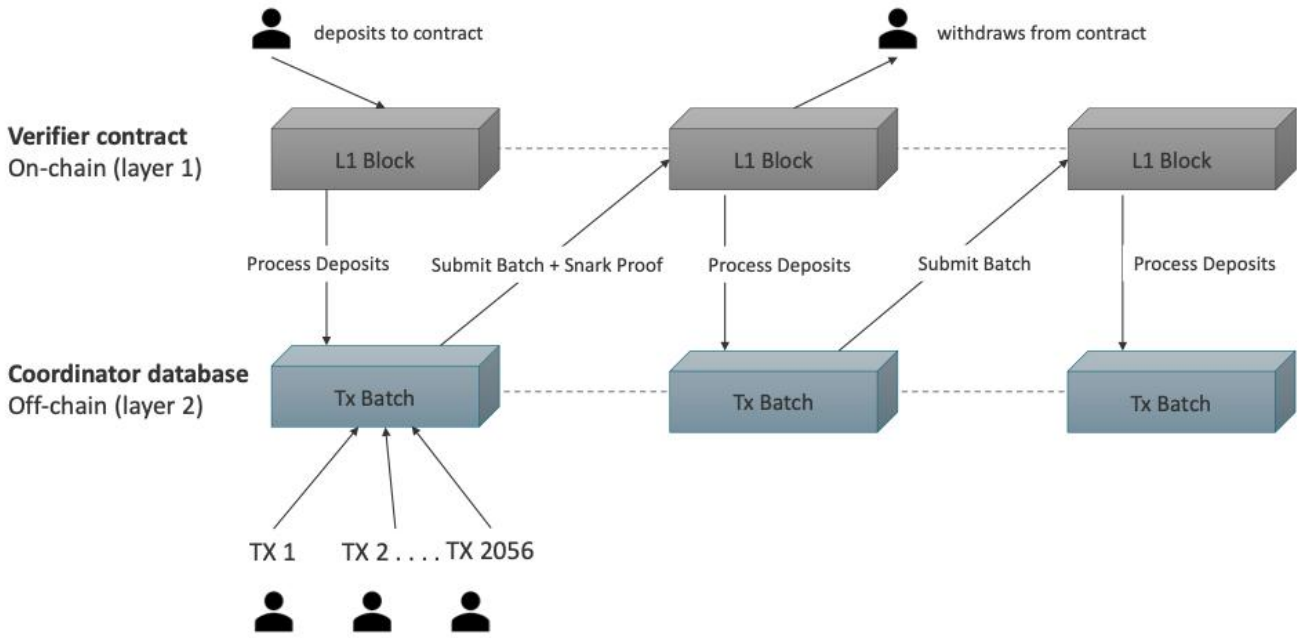

Рисунок 1: Архитектура ZK-rollup

- Сетевые контракты. Протокол ZK-rollup управляется смарт-контрактами, работающими на Ethereum. Это включает в себя основной контракт, который хранит накопительные блоки, отслеживает депозиты и отслеживает обновления состояния. Другой сетевой контракт (verifier contract) проверяет доказательства с нулевым разглашением, представленные производителями блоков. Таким образом, Ethereum служит базовым уровнем или "L1"для ZK-rollup.

- Автономная виртуальная машина (ВМ). В то время как протокол ZK-rollup работает в Ethereum, выполнение транзакций и хранение состояния происходит на отдельной виртуальной машине. Эта автономная виртуальная машина является средой выполнения транзакций в ZK-rollup и служит в качестве дополнительного уровня или "L2"для протокола ZK-rollup. Доказательства достоверности, проверенные в сети Ethereum, гарантируют корректность перехода состояний в автономной виртуальной машине.

Состояние ZK-rollup, включающее счета и балансы L2, представлено в виде дерева Меркла. Криптографический хэш корня дерева Меркла хранится в контракте основного блокчейна, позволяя протоколу rollup отслеживать изменения в состоянии ZK-rollup.

Оператор(L2), инициировавший переход в другое состояние, должен вычислить новый корневой код состояния и выполнить отправку по цепочке контрактов. Помимо самого состояния операторов генерирует доказательство, которое подтверждает, что все переходы были легитимными и новый корень дерева Меркла действителен. Если подтверждение достоверности, связанное с пакетом, аутентифицируется контрактом верификации, новый корень дерева становится корнем канонического состояния ZK-rollup. Именно тут открываются большие возможности для злоумышленников в случае ошибки в схеме доказательста.

Помимо вычисления корней состояний, оператор ZK-rollup также создает пакетный корень — корень дерева Меркла, содержащий все транзакции в пакете. При отправке нового батча в сводном контракте сохраняется корень пакета, что позволяет пользователям подтвердить, что транзакция (например, запрос на вывод средств) была включена в пакет. Пользователи должны будут предоставить информацию о транзакции, корень пакета и подтверждение Merkle, показывающее путь включения.

1.2 Деревья Меркла

Деревья Меркла - это особый вариант древовидной структуры данных, в котором новые данные вставляются в виде листа в нижней части дерева и для каждого уровня пары из двух листьев хэшируются вместе. Соответственно, количество узлов уменьшается вдвое на каждом новом уровне, пока не останется один с единственным хэшем, корневым хэшем дерева Меркле.

Иллюстрация базового дерева Меркле с восемью элементами данных представлена на рисунке 2

Эта структура данных особенна важна для ZK-rollup, потому что именно она используется для хранения состояния в основной сети. И для доказательства корректности перехода в новое состояние используются различные реализации этих деревьев. Отличительной особенностью реализаций является использование специфичных хеш-функций, таких как Poseidon-hash.[1]

1.3 Схемы арифметизации

Как отмечено ранее, очень важный шаг перехода в новое состояние на L1 уровне это верификация того, что новый корень дерева Меркла создан корректно. Для создания и проверки этого доказательства используются различные протоколы ZK-SNARKs. Отличительной особенностью таких протоколов являтется то, что они создают короткие доказательства(несколько сотен байт),

[1] https://www.poseidon-hash.info/

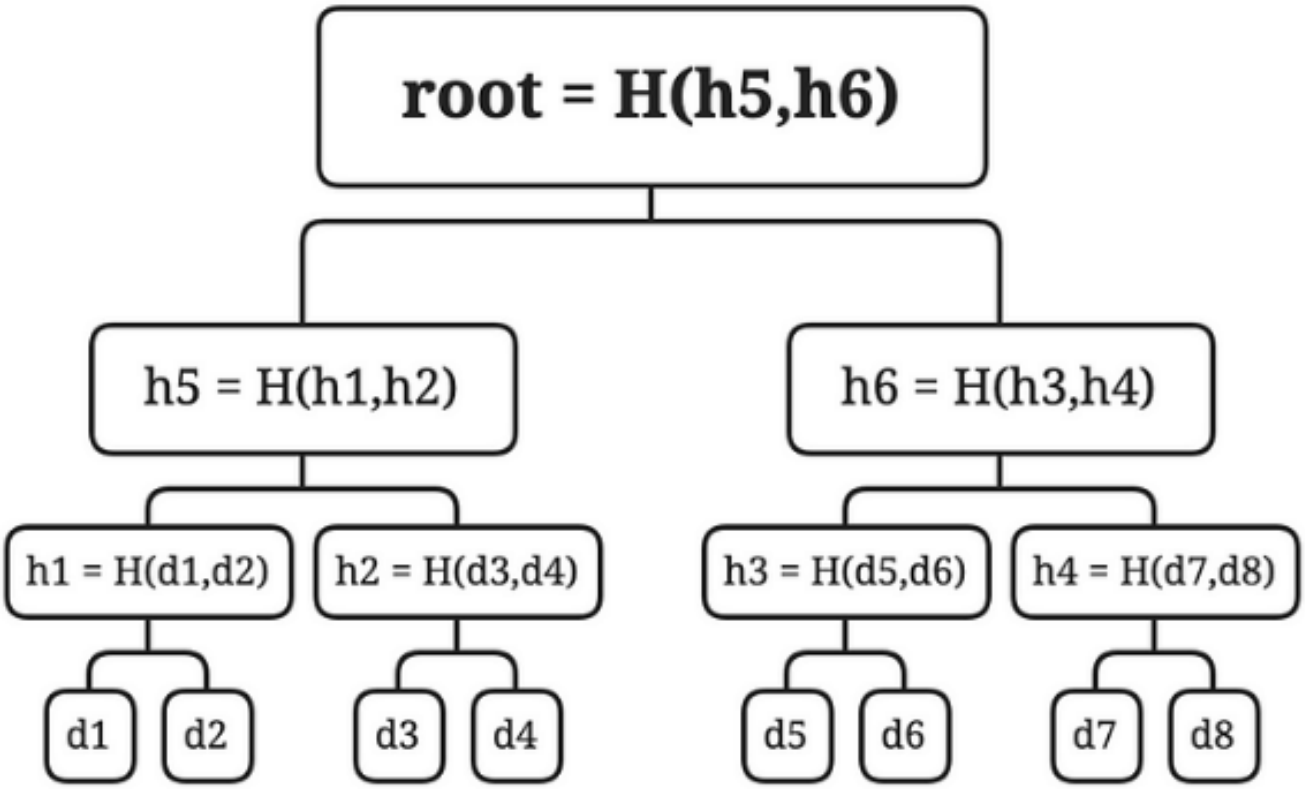

Рисунок 2: Пример дерева Меркла

которые быстро проверяются контрактом в основной сети. Общая схема работы таких протоколов представлена на рисунке **??**. Ключевым свойством протоколов zk-snark является то, что подходящее средство доказательства и верификации может быть автоматически сгенерировано из арифметической схемы, представляющей некоторые вычисления. Арифметическая схема принимает некоторые входные сигналы, которые являются значениями в диапазоне $[0, p)$, и выполняет сложение и умножение по модулю простого числа p. В результате каждого сложения и умножения генерируется сигнал: промежуточный сигнал от нескольких промежуточных операций и выходной сигнал от конечной операции. В частности, для конкретной арифметической схемы, компиляторы SNARK создают средство доказательства и верификации, преобразуя схему в набор полиномиальных уравнений. Арифметизация - процесс создания генерации схемы по программе. Чтобы упростить этот процесс, криптографическое сообщество разработало такие языки, как Circom[1], Zokrates [2], Halo 2 [3] и другие, которые позволяют пользователям выражать свои предполагаемые вычисления естественным образом.

[1]https://docs.circom.io/

[2]https://zokrates.github.io/

[3]https://zcash.github.io/halo2/index.html

```
template Decoder(w) {
    signal input inp;
    signal output out[w];
    signal output success;
    var lc=0;

    for (var i=0; i<w; i++) {
        out[i] <-- (inp == i) ? 1 : 0;
        out[i] * (inp-i) === 0;
        lc = lc + out[i];
    }

    lc ==> success;
    success * (success -1) === 0;
}
```

```
inp · out[0] = 0
(inp − 1) · out[1] = 0
(inp − 2) · out[2] = 0
out[0] + out[1] + out[2] − success = 0
(success − 1) · success = 0
```

Рисунок 3: Пример программы на языке circom и соответствующая ей схема R1CS

1.3.1 R1CS

R1CS(Rank 1 Constraint System) - схема арифметизации, которая представляет из себя набор полиномов над полем F_p, где p - большое простое число. Система содержит n переменных - это все сигналы, используемые в программе. При этом каждое ограничение(каждый полином) должны быть квадратичными, линейными или постоянными уравнениями

$$\begin{cases} (a_{11} * s_1 + ... + a_{1n} * s_n) * (b_{11} * s_1 + ... + b_{1n} * s_n) + (c_{11} * s_1 + ... + c_{1n} * s_n) = 0 \\ (a_{21} * s_1 + ... + a_{2n} * s_n) * (b_{21} * s_1 + ... + b_{2n} * s_n) + (c_{21} * s_1 + ... + c_{2n} * s_n) = 0 \\ (a_{31} * s_1 + ... + a_{3n} * s_n) * (b_{31} * s_1 + ... + b_{3n} * s_n) + (c_{31} * s_1 + ... + c_{3n} * s_n) = 0 \\ ... \\ (a_{m1} * s_1 + ... + a_{mn} * s_n) * (b_{m1} * s_1 + ... + b_{mn} * s_n) + (c_{m1} * s_1 + ... + c_{mn} * s_n) = 0 \end{cases}$$

Пример арифметизации для программы на языке circom представлен на рисунке 3

1.3.2 Plonkish

Plonk(Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge) арифметизация представляет из себя таблицу со значениями из конечного поля F_p. Элементы таблицы называются ячейками. Схемы состоят из элементов управления, которые являются ограничениями, применяемыми к вложенным таблицам (набору ячеек) в таблице. Столбцы таблицы делятся на три основных типа: instance, advice и fixed. Столбцы Instance хранят общедоступные входные данные, столбцы advice хранят частные секретные данные(witness в общей терминологии) (частные входные данные и промежуточные значения, используемые проверяющим в схеме), а столбцы fixed хранят фиксированные значения, которые являются частью описания схемы. Для этого типа арифметизации используются ограничения на значения в таблице в виде полиномов(гейты) $(q_L)a + (q_R)b + (q_O)c + (q_M)ab + (q_C) = 0$. Гейты - это общие поли-

номиальные ограничения, определенные разработчиком схемы. Каждое такое ограничение применяется к ряду соседних строк логического элемента. Следовательно, схема, табличное представление которой содержит несколько строк, может иметь несколько полиномиальных ограничений, все из которых должны выполняться одновременно. Другой тип ограничений - ограничения на равенство (также известные как ограничения на перестановку или ограничения на копирование) могут заставить определенную группу ячеек иметь одинаковые значения в строках и столбцах. Таблица и соответствующая её система ограничений - это и есть арифметизация Plonkish. В Halo2 элементы управления активируются с помощью переменных селекторов. Селектор - это двоичная переменная, которая умножается на полиномиальное ограничение для каждой строки: оно равно 0, если элемент управления выключен, или 1, если он включен. Ограничений в виде многочлена выполнено, если его значение равно 0 во всех строках. Все полиномиальные ограничения в системе проверки должны исчезать во всех строках для получения действительной пары секретных и общедоступных входных данных. Это делается либо путем установки несущественных переменных селектора в 0, либо путем предоставления (возможно, секретных) назначений ячейкам таблицы, которые приводят к вычислению полинома равным 0.

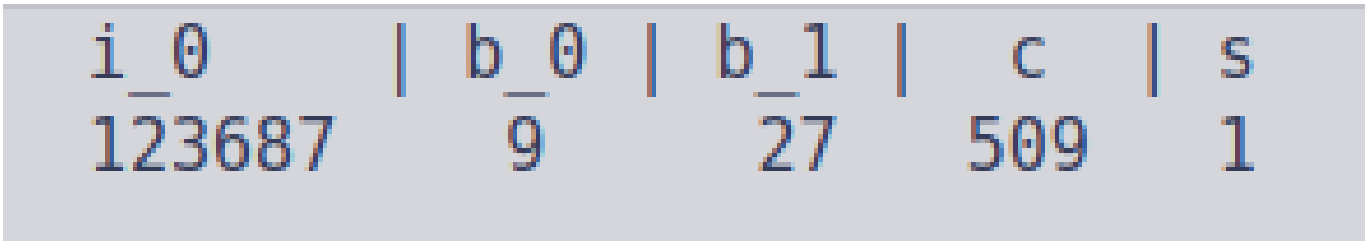

Рисунок 4: Пример арифметизации plonkish

Пример таблицы продемонстрирован на рисунке 4. Здесь i_0 - публичная входная переменная, b_0, b_1 - секретные переменные, c - константа, s - селектор. Соответствующее ограничение в виде полинома - $s \cdot (c \cdot b_0 \cdot b_1 - i_0) = 0$. То есть схема будет верна только, если доказывающий знает какое-то разложение числа на множители.

2 Цель работы

Цель данной работы - сделать обзор методов и средств, используемых при автоматическом тестировании схем арифметизации. Для достижения этой цели требуется:

- Сформировать критерии для сравнения методов.
- Проанализировать существующие решения по выделенным критериям.
- Дать оценку каждому критерию.

3 Анализ предметной области

3.1 Получение начального множества тестов

Все существующиме анализаторы Базовый алогоритм работы состоит из шагов:

1. Предобработка тестируемого программного обеспечения(далле SUT - Soft Under Test). Например, она может заключаться в компиляции со специальными флагами или подготовке окружения для тестирования.

2. Получить начальное множество входных данных и создать из него пул тестов.

3. Выбрать из пула один или несколько примеров и получить результат их тестирования в SUT.

4. Решить будет ли полезен этот тест в будущем.

5. Мутировать тест и добавить его в пул.

6. Перейти к шагу 3.

3.2 Получение начального множества тестов

Первый способ получения входных данных - использовать тесты, которые написали разработчики для SUT. Этот способ позволяет сразу получить хорошее покрытие кода. При обнаружении бага разработчики исправляют его и часто добавляют тест, который проверяет работоспособность программы в этом месте. В этом смысле тесты, как начальное множество, позволяют сразу добираться до "слабых"мест в SUT. Минус этого способа - тесты разработчиков не всегда доступны.

Второй вариант - собрать пул тестов из примеров в интернете. Например, в случае fuzz-тестировании интерпретатора javascript можно в начальное множество добавлять примеры javasrcipt кода с гитхаба. Таким способом можно получить широкий пул тестов.

Третий - можно сгенерировать множество самостоятельно. В этом случае появляется возможность подтолкнуть фаззер в определённом направлении. Детали процесса генерации описаны ниже.

3.3 Генерация тестов

3.3.1 Способы генерации

Алгоритм создания тестовых данных опирается на знание их структуры. Как было отмечено ранее, эту структуру удобно задавать контекстно-свободной грамматикой. В этом случае процесс генерации нового теста заключается в построении его AST. Шаги алгоритма[1]:

1. Положить в корень дерева стартовую вершину и добавить её в очередь вершин.
2. Взять текущую вершину из очереди.
3. Если текущая вершина - терминальная, то перейти к следующей вершине.
4. Каким-то способом выбрать продукцию из правила вывода для текущего нетерминала и добавить все символы из неё в очередь.
5. Перейти к шагу два.

Все известные методы опираются при выборе продукции нетерминала в пункте 4 на вероятности, то есть каждой продукции каждого нетерминала задаётся вероятность её выбора. Разные алгоритмы отличаются друг от друга способом задания этой вероятности.

- Выбирать продукцию равновероятно для каждого нетерминала. Плюсы - простая реализация. Минусы - часто будут генерироваться похожие тесты, медленно покрываются всевозможные ветки деревьев. Например: если у стартового символа одна из продукций - один терминальный символ, то большая часть сгенерированных тестов будет состоять из этого символа.
- Алгоритм построения похожих[1]. Если имеется какое-то множество примеров, то каждый из них представляется в виде AST и для каждого нетерминала для каждой продукции подсчитывется частота её встречаемости. По этим частотам можно вычисляется вероятность выбора каждой продукции. Чем чаще встречается переход в тестах, тем чаще он будет использоваться. Плюсы - этот метод позволяет направлять фаззер, путём изменения множества. Выбор символа становится более осмысленным с точки зрения программирования. Минусы - требуется начальное множество тестов.
- Алгоритм построения отличных[1]. Отличие этого способа от предыдущего - вероятность выбора становится обратно пропорциональна частоте встречаемости, то есть чем чаще встречается переход в тестах, тем реже он будет генерироваться. Это позволяет двигаться фаззеру в противоположном направлении. Плюсы - чаще будут встречаться неожиданные пути в грамматике. Минусы - чаще будут попадаться неинтересные символы. Например, в javascript часто будет вызываться return.

[1]https://www.fuzzingbook.org/html/Grammars.html#A-Simple-Grammar-Fuzzer

- Ещё один способ - задать вероятности пропорционально количеству возможных поддеревьев в продукции. Если в вершине потенциально много поддеревьев, она будет чаще выбираться. Плюсы - позволяет наиболее полно покрыть грамматику. Минусы - требуется предпосчёт количества деревьев для каждого нетерминала.

3.3.2 Свойства сгенерированных тестов

Сгенерированные тесты должны не только быть синтаксически корректными, но и обладать полезными для fuzz-тестирования свойствами.

- Нужно стремиться создавать короткие тесты. Чем тест длиннее - тем дольше он будет выполняться в SUT, тем сложнее его анализировать, тем дольше будет его обработка.

- Чем тест сложнее и рекурсивнее, тем вероятнее найти на нём ошибку SUT.

Эти свойства не противоречат друг другу. Рассмотрим примеры для грамматики с рисунка **??**. Выражение "5+5+5+5+5+5+5+5+5+5+5+5"является длинным, но не сложным. Скорее всего при парсинге будет вызвана функция обработки знака "+"несколько раз последовательно, что лишь увеличит время работы программы. Другой пример - выражение "(5 + (5) + ((5 + 5) + 5))". Оно короче предыдущего, но при обработке скобок некоторые функции будут вызваны рекурсивно, что увеличивает вероятность найти ошибку.

3.4 Получение обратной связи от запуска теста на SUT

Получение обратной связи от запуска теста - важная часть фаззера. Именно она часто позволяет определять значимость входного набора данных. Метрики, которые полезно оценивать:

- Самая весомая метрика - возникновение в SOT искомых исключений, например double free[1] в языке программирования c++. Зачастую такие ошибки означают, что найден баг и этот тест нужно дополнительно исследовать вручную.

- Покрытие кода(line coverage) и покрытие функций(function coverage) - подсчёт количества строк/функций, которые покрыл тест. В контесте множества тестов можно выявлять те, которые попадают в новые строки/функции. Разным строкам и функциям можно давать разный вес при подсчёте метрики. Это позволяет направлять фаззер в сторону исследования этих функций.

[1] https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory

- Покрытие веток(путей) - более сложный вариант предыдущей метрики. При подсчёте учитывается последовательность выполнения строк кода или вызывов функций. Плюсы - покрытие веток гораздо более информативная метрика, чем предыдущая. Минусы - число веток растёт очень быстро с увеличением количества кода в SUT.

3.5 Оценка полезности теста

После запуска теста требуется узнать полезен ли этот тест или от него можно отказаться. Для этого нужно научиться сравнивать различные входные данные друг с другом. Для этого введём функцию $value : \mathbb{X} \to \mathbb{R}$, которая каждому тесту ставит в соответствие его численную оценку. Конкретных реализаций этой функции может быть много. Приведём основные параметры, от которых она может зависеть:

- При наличии исключения при запуске, значение функции становится бесконечным.
- Чем больше покрытие кода, тем больше значение функции. Если тест покрывает новый участок кода, то функцию можно сделать равной бесконечности.
- Чем короче тест, тем больше значение функции.
- Чем тест разнообразнее, то есть чем больше символов грамматики он покрывает, тем лучше.

3.6 Способы обработки тестового множества

Существует две стратегии обработки тестового множества:

1. Первая - наиболее часто встречающаяся - обрабатывать каждый тест по отдельности. Это позволяет распараллелить процесс fuzz-тестирования, что серьёзно его ускоряет.
2. Вторая стратегия основана на теории эволюции Дарвина. Выбирается множество тестов. Для них всех рассчитывается функция полезности. Затем начинается процесс "выживания". Какой-то процент (например 10) тестов с наибольшей функцией полезности объявляется выжившими. Остальные случайным образом делятся на группы, в которых выживает несколько сильнейших. Потом среди оставшихся несколько случайных тестов объявляются выжившими. Тесты, которые не выжили отбрасываются.

3.7 Методы улучшения тестов

Все успешные тесты необходимо преобразовывать для продолжения процесса fuzz-тестирования. Целями улучшения могут быть:

- Поиск наиболее оптимального теста с точки зрения фаззера, обладающего теми же свойствами, что и улучшаемый. Другими словами - оптимизация процесса fuzz-тестирования.

- Дальнейшее продвижение фаззера, в том числе выход из локальных экстремумов.

3.7.1 Уменьшение размера теста

Как было отмечено ранее, у коротких тестов есть ряд преимуществ по сравнению с длинными. Уменьшенный тест должен сохранить все полезные свойства длинного. Например, если старый тест покрывает какую-то новую функцию, то и новый должен покрывать эту функцию. Стандартные методы минмизации:

- Самый простой способ заключается в построении AST теста и поочерёдном удалении поддеревьев. Если после удаления поддерева, полезность теста не уменьшилась, то старый тест заменяется на новый, и продолжается процесс минимизации. Плюсы - неплохая скокорость работы. С небольшой вероятностью тест существенно укоротится. Минусы - удаление поддерева может сделать тест синтаксически некорректным. Этот способ охватывает только очень локальные изменения теста.

- Уменьшение поддерева. Способ похож на предыдущий, только вместо отбрасывания поддерева, оно заменяется на более короткое. Новый тест всегда будет синтаксически корректным, но перебор всех поддеревьев может занять длительное время.

- Рекурсивное замещение поддерева. По сути этот способ является эвристикой предыдущего. Если в вершине у нетерминала F есть сын F, то производится замена поддерева текущей вершины на поддерево сына. Такое изменение оставит тест синтаксически корректным и приведёт к его упрощению для фаззера. Минус - этот способ может быть редко применим.

- Контролировать размер во время построение теста. Этот способ применим, если используется генерация тестов. Тогда этап уменьшения можно опустить.

3.7.2 Мутации

Мутации являются двигателем фаззера, позволяют ему эволюционировать. Если фаззер "застрял"на каком-то этапе, то мутации могут сделать шаг в сто-

рону и процесс fuzz-тестирования продолжится. Если тесты представимы в виде AST, то удобно описывать мутации изменениями над AST.

- Скрещивание тестов друг с другом. Берустя два успешных теста, представляются в виде AST, и поддерево одного теста заменяется на поддерево другого согласно правилам грамматики. Такое изменение, например, позволяет быстро увеличивать покрытие веток SUT.

- Создаётся пул поддеревьев. Для каждой вершинки в AST теста считается какая-то дополнительная информация. Поддерево этой вершинки может быть заменено только на поддерево из пула с такой же информацией. Например, информацией может быть тип нетерминала, количество идентификаторов. В пул тестов добавляются поддеревья из тестов, которые дают существенное улучшение полезности. Эта мутация позволяет обращать больше внимания на прогресс в фаззере. Например, если тестом покрыта какая-то новая функция, то среди новых тестов будет много тех, которые эту функцию исследуют. Сохранение дополнительной информации позволяет увеличить вероятность попадания в новую функцию.

- Рекурсивное дублирование поддерева. Это обратное действие одному из способов уменьшения. Эта мутация направлена на усложнение структуры выполняемого кода в SUT, например, на создание вложенных циклов или рекурсии.

- Применимы стандартные AFL[1] мутации - бит флип, замена "интересных значений". Они более случайны, их имеет смысл применять, когда остальные не работают. После применения этой мутации тест может стать синтаксически некорректным.

- Замена поддерева на случайно сгенерированное. Эта мутация - адаптированная для fuzz-тестирования с грамматикой версия предыдущего пункта. Основное отличие - тест останется корректным.

- При использовании метода генерации тестов по вероятностной грамматике можно изменять вероятности генерации продукции для конкретного нетерминала. Можно изменять на случайные, можно просто немного прибавить или отнять вероятности у нескольких символов. Новые тесты могут сильно отличаться от предыдущих. Плюс такой мутации - не нужно работать с самими тестами, достаточно просто изменить вероятность при генерации.

[1] https://github.com/google/AFL

4 Анализ инструментов предметной области

Цель анализа - рассмотреть работу инструментов, которые применяются при проверке ограничений в схемах арифметизации. В обзоре рассмотрены самые цитируемые актуальные статьи с google scholar.

4.1 QED^2

QED^2[7] - новый метод поиска ошибок в ZKP схемах, вызванных недостаточно ограниченными полиномиальными уравнениями над конечными полями. Метод выполняет семантический анализ уравнений конечного поля, сгенерированных компилятором, чтобы доказать, является ли каждый сигнал уникальным для входных данных. Подход сочетает в себе SMT-решение с упрощенным выводом уникальности, что позволяет эффективно анализировать схемы. В работе рассмотрена конкретная реализация этого метода - инструмент Picus[1].

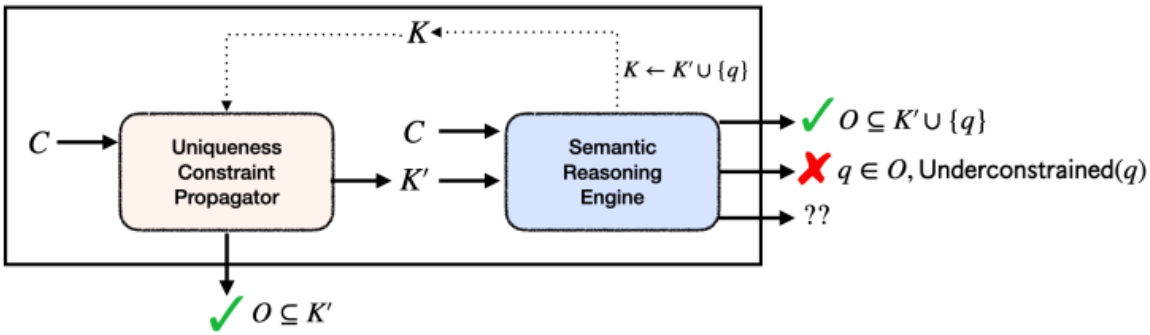

Рисунок 5: Алгоритм работы метода QED^2

Инструмент принимает на вход $R1CS$ схему и пытается определить достаточно ли в ней ограничений. Результат работы инструмента может быть одним из следующих:

- Ограничений достаточно. Это означает, что все скрытые переменные однозначно определяются входными, и инструмент смог это математически подтвердить.
- Ограничений недостаточно. В этом случае инструмент предоставит для конкретных входных значений два набора скрытых переменных, которые удовлетворяют схеме.
- Неопределённость. Инструмент не может дать точный ответ, так как SMT-solver не смог найти решение задачи за выделенное время.

Рассмотрим подробнее алгоритм работы инструмента, представленный на рисунке 5.

[1]https://github.com/chyanju/Picus

На вход алгоритму подаётся схема $C(I, W, O)$, где I - входные переменные, W - промежуточные, O - выходные. K - множество переменных, которые однозначно задаются входными данными, независимо от их значения. Изначально в K лежат все входные переменные. По сути инструмент использует своеобразный метод абстрактной интерпретации - каждая переменная либо однозначно определена, тогда она лежит во множестве K, либо не определена. При этом во время работы метода поддерживается второе состояние абстрактной интерпретации - список возможных значений переменной. Изначально всем переменным присвоены все значения их поля F_p, то есть интервал $0..p-1$. Далее в цикле выполняются следующие два шага, цель каждого из которых - расширить множество K:

$$\frac{\Delta, C, K \Vdash e \qquad \text{Constant}(c) \qquad c \neq 0 \qquad c \times x - e = 0 \in C}{\Delta, C, K \Vdash x}\ (\textsc{Assign}) \qquad \frac{A\vec{x} - \vec{b} = 0 \subseteq C \qquad \Delta, C, K \Vdash \vec{b} \qquad A \in \mathbb{F}_p^{n \times n} \wedge \det(A) \neq 0}{\Delta, C, K \Vdash \vec{x}}\ (\textsc{BigInt-Mul})$$

$$\frac{\sum_{i=0}^{n} c^i \times y_i - x = 0 \in C \qquad \Delta, C, K \Vdash x \qquad y_n < \frac{p}{c^n} - 1 \qquad c > 1 \qquad \forall i \in [0, n].\ \Delta(y_i) \subseteq [0, c-1]}{\Delta, C, K \Vdash y_i \quad \text{for } i \in [0, n]}\ (\textsc{Base-Conv})$$

$$\frac{\sum_{i=0}^{n} y_i = e \qquad \Delta, C, K \Vdash e \qquad \Delta, C, K \Vdash x \qquad \forall i \in [0, n].\ y_i \times (x - i) = 0 \in C}{\Delta, C, K \Vdash y_i \quad \text{for } i \in [0, n]}\ (\textsc{All-But-One-0})$$

Рисунок 6: Правила вывода для абстрактной интерпретации определённости переменной

$$\frac{\Delta \vdash e : \Omega \qquad c \neq 0 \qquad c \times x - e = 0 \in C}{\Delta, C \vdash x : \{v \times c^{-1} \mid v \in \Omega\}}\ (\textsc{Assign})$$

$$\frac{\prod_{i=1}^{n} (x - c_i) = 0 \in C}{\Delta, C \vdash x : \{c_1, \ldots, c_n\}}\ (\textsc{Root}) \qquad \frac{\sum_{i=0}^{n} c^i \times y_i - x = 0 \in C \qquad c > 1 \qquad \forall i \in [0, n].\ \Delta(y_i) \subseteq [0, c-1]}{\Delta, C \vdash x : \{v \in \mathbb{F}_p \mid 0 \leq v < c^{n+1}\}}\ (\textsc{Base-Conv})$$

Рисунок 7: Правила вывода для абстрактной интерпретации значений переменной

1. Шаг распространения определённости на основе правил. На этом шаге используются особенности схемы арифметизации $R1CS$, а именно - 4 правила

вывода, которые эффективно работают на схемах. Правила представлены на рисунке 6. Этот шаг работает до тех пор, пока какое-то из правил работает и в множество K добавляются новые переменные. Если правила неприменимы, то инструмент переходит к следующему шагу. При это параллельно вычисляется абстрактная интерпретация значений переменных по правилам на рисунке 7.

2. Далее каждую из неопределённых переменных алгоритм пытается доопределить при помощи солвера. Для этого формируется вторая схема C' - полная копия первой, кроме названий переменных. Чтобы добавить в формулу для солвера знания о переменных, накопленные на предыдущих шагах алгоритма, определяются две формулы:

   (a) $\phi = \wedge_{u \in K}(u = u')$ - формула сужает перебор - уже определённые переменные должны быть равны.

   (b) - кодируем в формулу все имеющиеся абстрактные интерпретации значений для каждой из переменных.

   Итоговую формула отдаётся решателю CVC5[1]

Среди схем, которые может решить QED^2, QED^2 возвращает результаты, подтвержденные для подавляющего большинства - 89%. Этот результат ожидаем, поскольку многие схемы, входящие в состав circomlib-utils и circomlib-core, написаны криптографами, которые также являются экспертами Circom. Однако существует 13 схем, для которых QED^2 выдает контрпримеры, что означает, что эти схемы доказуемо недостаточно ограничены.

4.2 Выводы

Для fuzz-тестирования шаблонизаторов предлагается использовать наиболее отличившиеся идеи. Среди будущих мутаций обязательно должны присутствовать скрещивание и замена поддерева на случайное, потому что они выделяются среди остальных. Для генерации тестов можно использовать алгоритм построения похожих, вероятности вычислить по реальным программам. Для борьбы с семантическими ошибками можно использовать идеи из Grammarinator. Среди стратегий обработки тестов нет явно выделяющейся, но идеи, описанные в EvoGFuzz, мало изучены, предлагается исследовать именно их.

[1] https://github.com/cvc5/cvc5

5 Результаты

В рамках данной работы были получены следующие результаты:

- Сделан обзор методов fuzz-тестирования программ, принимающих на вход данные, порождаемые КС-грамматикой. Выделены критерии для сравнения таких фаззеров.
- Произведён анализ существующих инструментов. На его основе предложен алгоритм для fuzz-тестирования шаблонизаторов.

Список литературы

[1] PlonK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge [Электронный ресурс]. URL: https://eprint.iacr.org/2019/953.pdf (дата обращения: 26.05.2024)

[2] Automated Analysis of Halo2 Circuit [Электронный ресурс]. URL: https://ceur-ws.org/Vol-3429/paper3.pdf (дата обращения: 26.05.2024)

[3] Certifying Zero-Knowledge Circuits with Refinement Types [Электронный ресурс]. URL: https://eprint.iacr.org/2023/547.pdf (дата обращения: 26.05.2024)

[4] SoK: What don't we know? Understanding Security Vulnerabilities in SNARKs [Электронный ресурс]. URL: https://arxiv.org/pdf/2402.15293 (дата обращения: 26.05.2024)

[5] Halo2 Book [Электронный ресурс]. URL: https://zcash.github.io/halo2/index.html (дата обращения: 26.05.2024)

[6] MoonMath manual [Электронный ресурс]. URL: https://leastauthority.com/community-matters/moonmath-manual/ (дата обращения: 26.05.2024)

[7] Automated Detection of Under-Constrained Circuits in Zero-Knowledge Proofs [Электронный ресурс]. URL: https://eprint.iacr.org/2023/512.pdf (дата обращения: 26.05.2024)

[8] Compositional Formal Verification of Zero-Knowledge Circuits [Электронный ресурс]. URL: https://eprint.iacr.org/2023/1278.pdf (дата обращения: 26.05.2024)

[9] Formal Verification of Zero-Knowledge Circuits [Электронный ресурс]. URL: https://arxiv.org/pdf/2311.08858(дата обращения: 26.05.2024)

[10] Franklyn Wang (2022): Ecne: Automated Verification of ZK Circuits. OxPARC Blog [Электронный ресурс]. URL: https://0xparc.org/blog/ecne(дата обращения: 26.05.2024)

[11] Plookup: A simplified polynomial protocol for lookup tables [Электронный ресурс]. URL: https://eprint.iacr.org/2020/315.pdf(дата обращения: 26.05.2024)

[12] ZERO-KNOWLEDGE ROLLUPS [Электронный ресурс]. URL: https://ethereum.org/en/developers/docs/scaling/zk-rollups/(дата обращения: 26.05.2024)

[13] Circom [Электронный ресурс]. URL: https://docs.circom.io/(дата обращения: 26.05.2024)

[14] Zokrates [Электронный ресурс]. URL: https://zokrates.github.io/(дата обращения: 26.05.2024)