ИНСТИТУТ ИНТЕЛЛЕКТУАЛЬНЫХ КИБЕРНЕТИЧЕСКИХ СИСТЕМ

Кафедра
«Криптология и кибербезопасность»

ОТЧЕТ

о научно-исследовательской работе

«Верификация примитивов в цепях доказательств с нулевым разглашением при помощи символического исполнения»

Исполнитель:
студент гр. Б20-505 Соколов А.Д.
подпись, дата

Научный руководитель: Сенновский И.И.
подпись, дата

Зам. зав. кафедры №42 Когос К.Г.
подпись, дата

Москва – 2022

РЕФЕРАТ

Отчет 40 с., 38 рис., 18 источников.

ZERO-KNOWLEDGE PROOFS, ДОКАЗАТЕЛЬСТВА С НУЛЕВЫМ РАЗГЛАШЕНИЕМ, CIRCOM, БЕЗОПАСНОСТЬ АРИФМЕТИЧЕСКИХ СХЕМ, СИМВОЛЬНОЕ ИСПОЛНЕНИЕ

Предмет исследования в работе — доказательства с нулевым разглашением и арифметические схемы, написанные на DSL Circom.

Цель работы: оценка целесообразности использования символьного исполнения для проверки примитивов в цепях доказательств с нулевым разглашением.

Актуальность работы обусловлена тем, что доказательства с нулевым разглашением все больше входят в эксплуатацию, а написание арифметических схем является нетривиальной задачей.

В ходе работы рассматривались основные понятия в сфере доказательств с нулевым разглашением, реальные протоколы, основанные на ZKP, а также проанализирован язык Circom и предложен подход к анализу примитивов, реализованных на этом языке.

Область применения — верификация ПО.

В результате работы был проанализирован один из примитивов, реализованный на языке Circom.

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИИ, СОКРАЩЕНИЯ

В настоящей работе применяются следующие термины с соответствующими определениями, обозначениями и сокращениями.

ZK	—	Zero-Knowledge, нулевое разглашение
ZKP	—	Zero-Knowledge Proofs, Доказательства с Нулевым Разглашением
DSL	—	Domain-Specific Language, Предметно-Ориентированный язык
zk-SNARK	—	Zero-Knowledge Succinct Non-Interactive Argument of Knowledge, Краткий неинтерактивный аргумент знания с нулевым разглашением
R1CS	—	Rank-1Constraint System, Система ограничений ранга 1
ПО	—	Программное Обеспечение
КАП	—	Квадратичная арифметическая программа
PLONK	—	Permutations over Lagrange bases for Oecumenical Noninteractive arguments of Knowledge, Перестановки над базисами Лагранжа для универсальных неинтерактивных аргументов знания
Протокол	—	Стандарт, описывающий правила взаимодействия функциональных блоков при передаче данных
Узел	—	Устройство, соединённое с другими устройствами, как часть компьютерной сети
Аутентификация	—	Процедура проверки подлинности предъявленного пользователем идентификатора

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ 4
ВВЕДЕНИЕ 5
1 Доказательства с нулевым разглашением и их применение … 6
1.1 Доказательства с нулевым разглашением 6
1.1.1 Интерактивные ZKP 7
1.2 zk-SNARKs 9
1.2.1 Арифметические схемы 10
1.2.2 R1CS и Квадратичные арифметические программы 11
1.2.3 Trusted Setup 12
1.3 PLONK 13
2 Circom… 15
2.1 Circom Language 15
2.2 Circom Compiler 17
3 Безопасность арифметических схем… 18
3.1 Зачем нужна верификация … 18
3.2 Тестирование с помощью символьного исполнения… 19
3.2.1 Z3… 19
3.3 В каких случаях полезно символьное исполнение …..... 20
4 Верификация примитива SmallSigma 21
ЗАКЛЮЧЕНИЕ 25
ПРИЛОЖЕНИЕ 26
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 39

ВВЕДЕНИЕ

С каждым годом количество сервисов, работающих в интернете, растет с неимоверной скоростью. Все больше различных услуг доступно для обычных пользователей. Денежные переводы, обмен сообщениями и письмами, аутентификация — все это желательно делать быстро и конфиденциально. С помощью криптографии стало возможно покрыть второй пункт, а с грамотной реализацией и скорость перестает быть большой проблемой. Однако протоколы имеют свойство устаревать и поэтому появляются новые, более эффективные и надежные способы передачи информации.

Одним из важнейших достижений криптографии за последние годы стали доказательства с нулевым разглашением (Zero-Knolwedge Proofs, далее ZKP). Это метод, благодаря которому одна сторона может доказать другой стороне, что некоторое утверждение правдиво, не раскрывая никакой дополнительной информации об этом утверждении. Значительную популярность он получил в том числе из-за своей значимости для блокчейна и интернета вещей. Более привычными же примерами использования ZKP являются: анонимное голосование и выборы, аутентификация, доказательства правдивости источников новостей.

Главными проблемами в реализации протоколов на основе ZKP долгое время были стандартизация формирования таких доказательств, а также большое время их проверки. Когда эти проблемы были частично решены, стало важным упростить создание доказательств для обычных разработчиков без опыта в криптографии. В последние годы эта отрасль развилась значительно. Было представлено множество исследований на тему ZKP, предложены алгоритмы реализации формирования доказательств и их проверки, а также созданы новые предметно-ориентированные языки (далее DSL), позволяющие реализовать ZK протоколы. Однако логика написания таких программ значительно отличается от привычных языков, следовательно, необходимо разработать методы поиска уязвимостей в них.

1 Доказательства с нулевым разглашением и их применение

В данной главе будет рассказано про принципы работы ZKP, их подмножество zk-SNARKs и конкретный zk-SNARK протокол PLONK.

1.1 Доказательства с нулевым разглашением

ZKP (Zero-Knowledge proofs, Доказательства с нулевым разглашением) — это доказательства, которые позволяют убедить проверяющую сторону в их достоверности и в то же время не раскрывают ничего, кроме факта верности доказываемого утверждения [1].

Задачей ZKP является сокрытие любой важной информации, необходимой для формирования доказательства. Обычно стороны обозначают как Доказывающий (Prover) и Проверяющий (Verifier), далее *P* и *V*.

Три главных свойства, определяющие ZKP:

- Completeness (Полнота) — если утверждение *P* верно, то он сможет убедить в этом *V* (по крайней мере с большой вероятностью);
- Soundness (Корректность) — *P* может убедить *V* в верности своего утверждения только в случае, если оно действительно верно;
- Zero-knowledge (Нулевое разглашение) — *V* не узнает ничего важного об утверждении *P*, кроме факта его верности.

Пусть *V'* — некоторый проверяющий, никак не связанный с изначальным *V*, но имеющий те же полномочия, что и *V*. Симулятор — любой вероятностный алгоритм, работающий за полиномиальное время, который способен симулировать результат взаимодействия *V'* с *P* и эта симуляция будет статистически неотличима от результата взаимодействия оригинального *V* с *P*.

Существует несколько типов нулевого разглашения:

- Идеальное нулевое разглашение (perfect zero-knowledge, PZK) — даже сторона с неограниченными ресурсами для вычислений не способна отличить доказательство от симуляции доказательства;

- Статистическое нулевое разглашение (statistical zero-knowledge, SZK) — сторона с неограниченными ресурсами для вычислений способна отличить настоящее доказательство от симуляции доказательства с незначительной вероятностью;
- Вычислительное нулевое разглашение (computational zero-knowledge, CZK) — не существует эффективного алгоритма, который способен различить настоящее доказательство и симулированное.

Возможность отличать настоящее доказательство от симулированного является ключевой для злоумышленника.

Так как CZK проще всего реализуется, оно является самым распространенным типом нулевого разглашения.

Аргументы знания (arguments of knowledge) — ZKP, которые входят в класс CZK. Они очень полезны для доказательства множества утверждений, которые мы используем в реальной жизни [2].

Примеры использования в реальной жизни:

- Пусть *F* — программа, которая занимает две недели, чтобы завершиться на вашем ноутбуке и два дня в дата-центре. С помощью ZKP дата-центр может прислать результат работы *F*, а также доказательство того, что вычисления были верными;
- Доказательство того, что последний блок в блокчейне, а также все предыдущие блоки являются верными. Это полезно, когда размер цепи достигает нескольких гигабайт и не нужно каждый раз проверять каждый предыдущий блок по отдельности.

1.1.1 Интерактивные ZKP

Изначально ZKP задумывались как интерактивный протокол. В общем он включал в себя некоторое количество коммуникаций между *P* и *V*, в которых *V* посылает *P* определённое вызов. *P* посылает ответ, после чего *V* на его основе либо не принимает доказательство, либо отправляет еще один вызов,

пока вероятность того, что все вызовы были пройдены наугад, не станет незначительной.

На рисунке 1 показана диаграмма последовательности интерактивного ZKP.

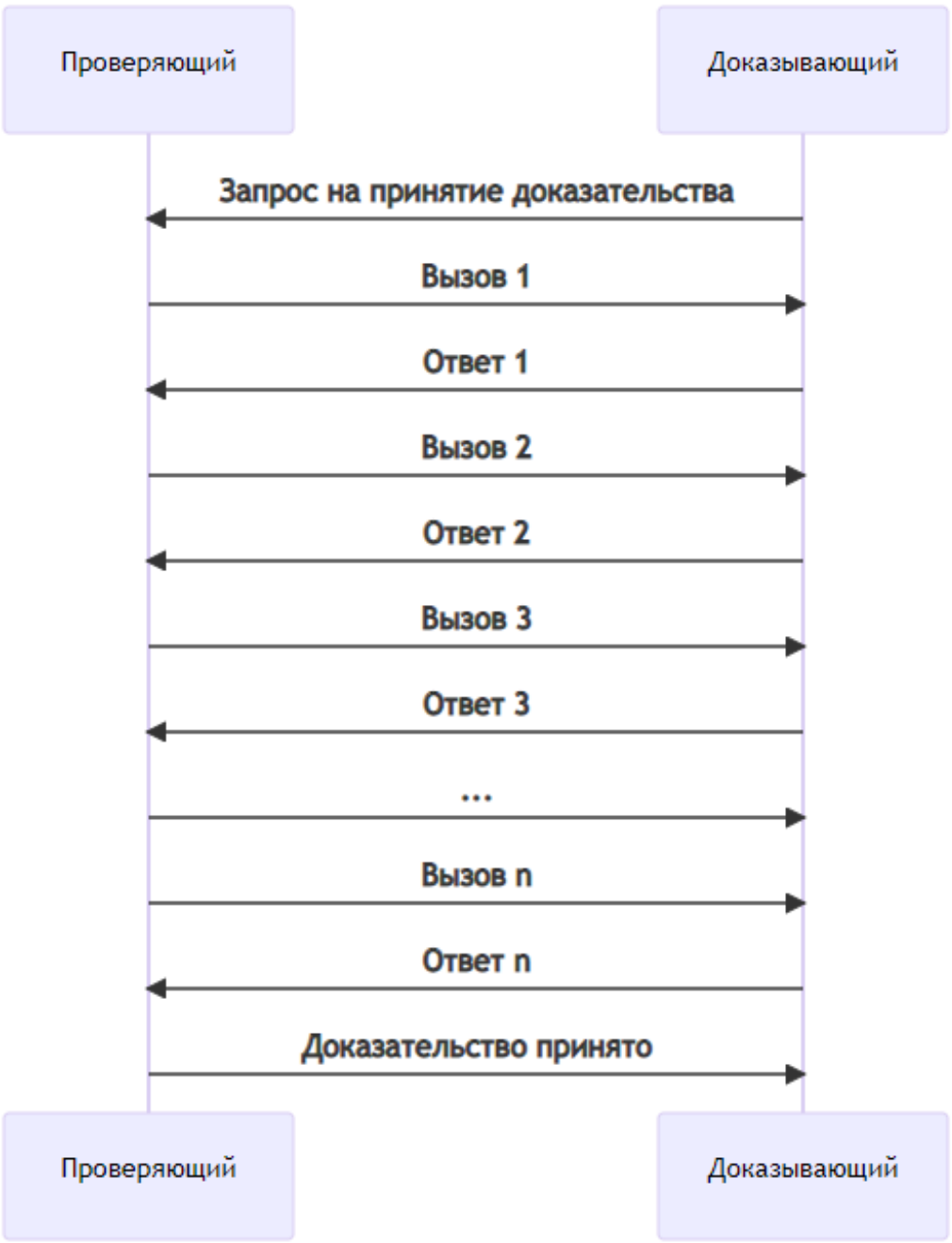

Рисунок 1 — диаграмма интерактивного ZKP

Преобразование Фиата-Шамира — методика трансформации интерактивного ZKP в неинтерактивное. Последнее означает, что доказательство возможно сгенерировать на стороне *P* и далее его может проверить кто угодно, тогда как в случае интерактивного ZKP уверенным в утверждении может быть только один *V*. Также это позволяет не тратить время на продолжительную коммуникацию сторон.

Будем назвать *V* честным, если все испытания, которые он посылает *P*, основываются ни на чём кроме работы генератора случайных чисел. Это значит, что *V* будет следовать протоколу и не будет пытаться узнать секретную информацию посредством отправки специально подготовленных испытаний.

Если определенный генератор случайных чисел может быть построен с помощью данных, известных обеим сторонам, тогда любой интерактивный протокол может быть трансформирован в неинтерактивный [3]. Еще одним важным условием является наличие честного *V*.

1.2 zk-SNARKs

zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge, Краткие неинтерактивные аргументы знания с нулевым разглашением) — это семейство протоколов, основанных на ZKP. В них одна сторона может доказать другой стороне, что она владеет информацией, без раскрытия этой информации. Также в данном протоколе отсутствует взаимодействие между сторонами.

Основным преимуществом данного семейства протоколов является очень малое время проверки доказательства, даже если предмет доказательства требует большого количества вычислений, а также малый размер доказательства [4].

zk-SNARKs можно использовать в том числе для доказательства правильности вычислений компьютерной программы.

В основном, в таких доказательствах необходимо проверить каждую часть выполнения программы, фактически не запуская эту программу. Это становится возможно, если представить программу в виде арифметической схемы.

1.2.1 Арифметические схемы

В теории компьютерных вычислений, арифметическая схема - стандартная модель для вычисления многочленов. В данной модели можно складывать или умножать выражения, которые уже были вычислены, на основе входных данных. Формально это ориентированный ациклический граф. Каждая вершина с нулевой степенью входа - входной вентиль (input gate). Каждая грань - провод(wire). Каждый вентиль имеет два входных и несколько выходных проводов [5].

На рисунке 2 представлен пример арифметической схемы.

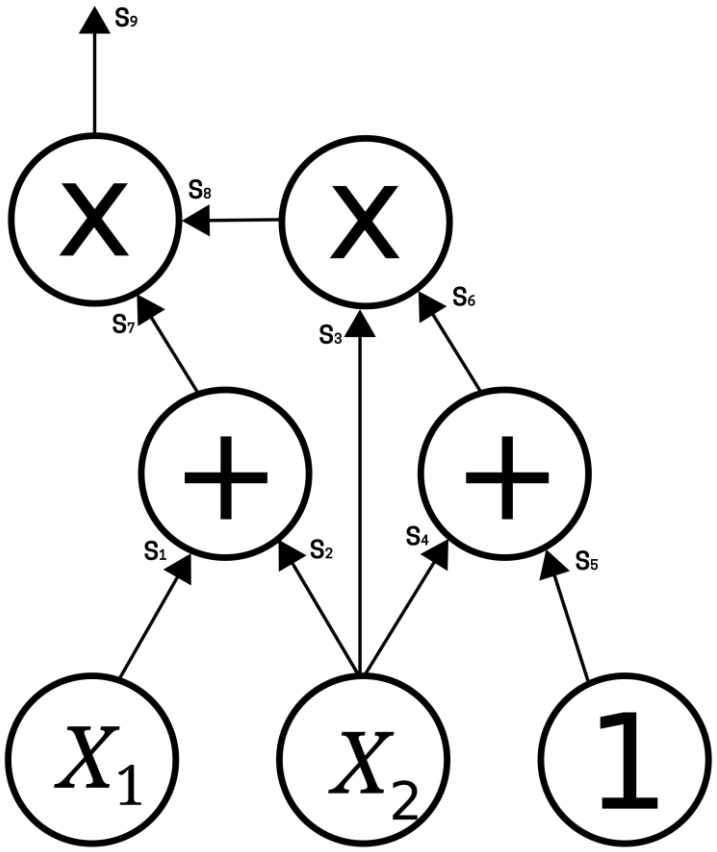

Рисунок 2 — арифметическая схема для многочлена (x1 + x2)x2(x2 + 1)

Выполнение арифметической схемы означает последовательное выполнение всех операций на всех вентилях. В случае использования арифметических схем в ZKP, считается свидетель (witness) — значения на всех проводах такие, чтобы входы и выходы каждого вентиля удовлетворяли ограничению, определяемому операцией вентиля. На рисунке 2 свидетелем является w = (s1, s2, s3, s4, s5, s6, s7, s8, s9). И все они удовлетворяют ограничениям:

- s1 + s2 = s6 (Вентиль 1);
- s4 + s5 = s7 (Вентиль 2;
- s3s6 = s8 (Вентиль 3);
- s7s8 = s9 (Вентиль 4).

Размер схемы – количество вентилей в ней.

1.2.2 R1CS и Квадратичные арифметические программы

R1CS — rank-1 constraint system (система ограничений ранга 1). Это последовательность строк, хранящая в себе значения, которым должны соответствовать переменные во время работы программы. Она же связывает отношения между всеми ними во время вычисления. Эти отношения называются ограничениями (constraints) или вентилями (gates) [7] [8].

Допустим, w = (s1, s2, s3, s4, …, sn). Тогда для каждого вентиля будут существовать такие линейные комбинации вектора w — Ai, Bi и Ci, что Ai * Bi – Ci = 0. Ai = (w, ai), Bi = (w, bi), Ci = (w, ci). Где ai, bi, ci – система ограничений, (w, x) – скалярное произведение векторов.

Наборы таких ai, bi, ci для всех ограничений в схеме и будут составлять R1CS.

На рисунке 3 показан пример системы ограничений для вентиля 4 из рисунка 2. Как видно из рисунка, A4 * B4 – C4 = (w, a4) * (w, b4) - (w, c4) = (s8 * 1) * (s7 * 1) - (s9 * 1) = 0.

	a4	b4	c4
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	1	0
8	1	0	0
9	0	0	1

Рисунок 3 — система ограничений для вентиля 4

КАП (Квадратичная арифметическая программа, QAP) — специальная форма программы, которая получена из R1CS, преобразованием её в арифметическое выражение, с использованием многочленов. Это делается с помощью R1CS представления программы, все уравнения вида (w, ai) * (w, bi) - (w, ci) = 0 могут быть записаны с помощью трех многочленов, которые принимают значения (w, ai), (w, bi), (w, ci) при определенном аргументе ri. Многие zk-SNARKs используют это представление.

Выполнимость арифметических схем — NP-полный язык. Значит для любого из NP-вычислений можно построить арифметическую схему для этого вычисления таким образом, что свидетель, удовлетворяющий схеме — свидетель оригинального вычисления. Таким образом убеждаясь в правильности свидетеля для КАП, V одновременно убеждается в правильности свидетеля из оригинального вычисления [6].

1.2.3 Trusted Setup

Доверенная установка — вид многостороннего вычисления, который требует несколько участников, которые генерируют случайные значения, и хотя бы одну честную сторону. Данная установка необходима для генерации

стандартных параметров, использующихся в системе доказательств. Во время установки генерируются случайные значения (секреты), шифруются, используются для создания параметров, а затем удаляются навсегда. Если хотя бы одна сторона выполнит последний пункт, система считается безопасной. Такой процесс называется Церемонией доверенной установки [15].

1.3 PLONK

PLONK (Permutations over Lagrange bases for Oecumenical Non-interactive arguments of Knowledge, Перестановки над базисами Лагранжа для универсальных неинтерактивных аргументов знания). PLONK предоставляет универсальную и обновляемую доверенную установку.

- Универсальная — в отличие от церемоний доверенных установок, которые нужно проводить каждый раз для разных доказательств, PLONK предоставляет установку, которую можно использовать повторно;
- Обновляемая — генерация параметров выполняется последовательно, значит возможно дополнять параметры через некоторое время после первоначальной церемонии.

PLONK также добавляет безопасности системы доказательств, используя в качестве одной из компонентов спаривание точек на эллиптической кривой. Их безопасность основывается на сложности вычисления дискретного логарифма в группе сложения точек на эллиптической кривой, а также на решительном предположении Диффи-Хеллмана. Спаривание точек является билинейным отображением. Это значит, что оно линейно по обоим своим аргументам [17].

В отличие от R1CS, в PLONK ограничения, связанные с вентилем i, представлены в виде уравнения на рисунке 4.

$$(QL_i)a_i + (QR_i)b_i + (QO_i)c_i + (QM_i)a_ib_i + QC_i = 0$$

Рисунок 4 — уравнение, описывающее ограничение для вентиля с номером i

- QLi — переключатель, отвечающий за левый входной провод;

- QRi — переключатель, отвечающий за правый входной провод;
- QOi — переключатель, отвечающий за выходящий провод;
- QMi — переключатель, отвечающий за операцию произведения;
- ai — значение на левом входном проводе;
- bi — значение на правом входном проводе;
- ci — значение на выходящем проводе;

Переключатель — значение равное 0 или 1 в зависимости от i, которое включает данное значение в уравнение [16].

2 Circom

В данном разделе будут рассмотрены предметно-ориентированный язык Circom и компилятор для этого языка — circom compiler.

Это один из самых первых и самых используемых DSL для написания схем на данный момент. Удобным для использования его также делает стандартная библиотека базовых примитивов, таких как операции с двоичным представлением числа, операции на эллиптических кривых, sha256 и т.д.

К примеру, Tornado Cash — децентрализованный протокол, который позволял анонимизировать транзакции в сети Etherium и ряде других блокчейнов, с использованием ZKP. В нем использовался circom для формирования и проверки доказательств. Так же Tornado Cash называют монетным миксером — сервисом, который позволяет пользователям скрывать происхождение и назначение транзакций [18].

Circom и другие DSL используются для описания вычислений вместе с рядом ограничений на входные и выходные значения, называемые сигналами (signals). Существует два типа сигналов: открытые и закрытые. Открытые известны как P, так и V, закрытые же являются секретом P.

2.1 Circom Language

Circom — предметно-ориентированный язык, который был разработан для написания арифметических схем используемых в ZKP. В частности, он был разработан для работы с javascript библиотекой snarkjs [13].

В snarkjs реализованы базовые инструменты для работы с арифметическими схемами. С её помощью можно генерировать свидетеля, проводить церемонии доверенной установки, формировать доказательства и проверять их.

Circom позволяет создавать массивные схемы, используя много маленьких компонентов.

Этот язык с одной стороны подтверждает верность работы программы, а с другой стороны описывает все вычисления. Вычисление и Проверка - разные операции, описанные одной схемой.

Circom значительно отличается от привычных нам языков программирования. Однако синтаксис языка заимствуется у языков javascript и C с добавлением нескольких операторов:

- <==, ==> — используются для передачи значений сигналов и наложения ограничений на сигналы;
- <--, --> — используются для передачи значений сигналов;
- === — используется для наложения ограничений на сигналы.

```
1   pragma circom 2.0.0;
2
3   template NAND() {
4       signal input a;
5       signal input b;
6       signal output out;
7
8       out <== 1 - a*b;
9       a*(a-1) === 0;
10      b*(b-1) === 0;
11  }
12
13  component main = NAND();
```

Рисунок 5 — пример схемы, реализующей битовую операцию “не и”

Как мы видим, есть два входных сигнала (a, b) и выходной сигнал (out). Сигналу out присвоено значение 1 - a * b и наложено ограничение, что out должен действительно быть равен 1 - a * b при проверке. Далее проверяется что a и b находятся в множестве {0, 1}.

Все ограничения должны быть в квадратичном виде.

Данная схема работает с элементами конечного поля по модулю простого числа p = 21888242871839275222246405745257275088548364400416034343698204186575808495617.

2.2 Circom Compiler

Circom Compiler — компилятор языка Circom, написанный на языке программирования Rust. Он используется для генерации R1CS файла с ограничениями, наложенными схемой и программой, которая будет эффективно считать свидетеля. Свидетель — набор значений, удовлетворяющий всем ограничениям, наложенным схемой. Также он создает программы Prover и Verifier. Prover может быть использован для вычисления схемы, используя открытые и закрытые входные сигналы, вместе с доказательством того, что вычисление было выполнено корректно. Verifier с помощью открытых входных сигналов и выходного сигнала вычисления может быть использован для проверки верности доказательства, созданного о prover.

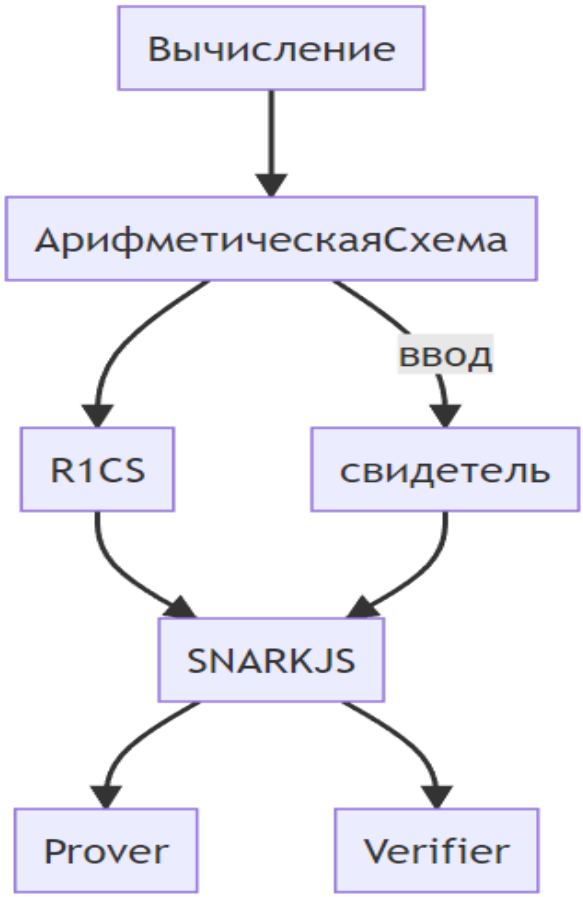

Рисунок 6 — схема создания доказательства, используя circom и snarkjs

В приложении 1 представлен пример проведения церемонии доверенной установки, создания доказательства и его проверки, с помощью утилит circom и snarkjs.

3 Безопасность арифметических схем

Все больше и больше людей заинтересовано во внедрении ZKP в свои системы. Есть множество развивающихся отраслей, например, таких как блокчейн. ZKP позволяет сделать их более надежными, быстрыми и дешёвыми. В большинстве своём сегодня ZKP используют арифметические схемы. Их безопасность не так хорошо изучена, как у стандартных языков программирования. Это всё говорит о том, что необходимо выработать подходы к тестированию такого рода ПО, оценить целесообразность этих подходов и выяснить границы их применимости.

3.1 Зачем нужна верификация

Поиск ошибок в программах необходим во всех видах разработки. Написание арифметических схем ничем от этого не отличается. Однако в случае с уже известными языками программирования способы отладки и поиска неисправностей налажены систематически. В случае же языков для написания арифметических схем могут встречаться неисправности, которые сложно найти, а отладить не получится в силу специфики языков.

Арифметические схемы всегда являются составной частью какого-то большего приложения. Ошибка в схеме может привести к формированию неправильных доказательств или к принятию подделанных доказательств. Оба этих поведения неприемлемы как для разработчика, так и для пользователя. В основном наличие такого рода ошибок приведет к необходимости внесения радикальных изменений в ПО со стороны разработчика и переустановки приложения со стороны пользователя. Наличие различного рода ошибок может привести к потере денежных средств и утечке конфиденциальных данных.

Очень важно выявлять такие ошибки еще во время написания ПО. Однако, так как отрасль довольно новая и представлено немного классов уязвимостей таких приложений, требуется выявление систематических подходов к тестированию такого ПО. Одним из них является верификация частей схемы с использованием символьного исполнения.

Данный метод очень хорошо себя проявляет в связке с другими способами поиска ошибок в арифметических схемах. Самые распространённые:

- Статический анализ кода — сканирует код на наличие распространенных ошибок и уязвимостей;
- Фаззинг — использует динамическое исполнение для тестирования частей программ, с помощью случайных данных.

Все эти методы затрагивают разные части работы схемы, а также имеют разную вовлеченность человека.

3.2 Тестирование с помощью символьного исполнения

Тестирование с помощью символьного исполнения — это средство анализа программы для определения того, какие входные данные вызывают выполнение каждой части программы. Оно полезно для генерации тестовых данных и подтверждения качества программы. Исполнение требует выбора путей, которые осуществляются набором значений данных. Программа, которая выполняется с использованием фактических данных, приводит к выводу ряда значений. В символьном исполнении данные заменяются символьными значениями с набором выражений, по одному выражению на выходную переменную [9]. Символьное исполнение позволяет проверить свойства программы, и то, что результат работы нескольких программ всегда приводит к одинаковым значениям.

Символьное исполнение обладает одним из больших преимуществ — оно нивелирует определенный класс ошибок, которые бывает сложно выявить человеку во время аудита. С другой стороны затраты человеческих ресурсов в этом случае тоже довольно высоки.

3.2.1 Z3

z3 — SMT (Satisfiability Modulo Theories) — утилита для доказательства определенного набора теорем [10]. Она базируется на символьном исполнении и в основном работает с математическими выражениями. Данная утилита прекрасно подходит для нахождения ошибок в арифметических схемах, ведь их структура сама по себе является набором уравнений.

3.3 В каких случаях полезно символьное исполнение

Выделим несколько случаев, кода нам может понадобиться символьное исполнение [11]:

- Слабая верификация (уникальный ввод/вывод) — если для данного ввода, вывод КАП должен иметь однозначно определенные значения;
- Уникальность свидетеля — все значения свидетеля, которые появляются во всех уравнениях, определены однозначно;
- Строгая уникальность — если КАП должна быть строго эквивалентна некоторой математической спецификации.

4 Верификация примитива SmallSigma

В данном разделе я приведу результаты анализа реализации примитива sha256 SmallSigma, разработанного на языке Circom [12].

В ходе работы были разработаны вспомогательные инструменты, для удобства обработки данных. Все реализованные инструменты, будут в приложении 2 [14].

Первым шагом в тестировании данного примитива, была его реализация на языке python.

```python
def SmallSigma(inp, ra, rb, rc):
    rota = RotR(32, ra, inp)
    rotb = RotR(32, rb, inp)
    shrc = ShR(32, rc, inp)

    s = xor3(rota, rotb, shrc, 32)
    return s


def xor3(a, b, c, n):
    mid = []
    for k in range(n):
        mid.append(b[k] ^ c[k] ^ a[k])
    return mid


def ShR(n, r, inp):
    out = []
    for i in range(n):
        if i + r >= n:
            out.append(0)
        else:
            out.append(inp[i + r])
    return out


def RotR(n, r, inp):
    out = []
    for i in range(n):
        out.append(inp[(i + r) % n])
    return out
```

Рисунок 7 — реализация SmallSigma на языке python

Большим преимуществом z3 является наличие типа данных BitVec. С помощью него можно очень эффективно обращаться с числами и применять к ним бинарные операции.

```
constrs = json.load(open("constraints/constr.json"))
witness = json.load(open("witness/witness.json"))
nvars = constrs["nVars"]
nout = constrs["nOutputs"]
out0 = [int(x) for x in witness[1 : 1 + nout]]


inp = [BitVec(f"f_{i}", 1) for i in range(32)]
ar, br, cr = 1, 2, 3
out = SmallSigma(inp, ar, br, cr)

s = Solver()
for i in range(len(out)):
    s.add(out[i] == out0[i])

w = []
i = 0
print("_____________________________________________________")
print("Найденные Входные сигналы:")
if argv[1] == "1":
    if s.check() == sat:
        m = s.model()
        for x in inp:
            w.append(str(m[x]))
    else:
        print("unsat")
else:
    while s.check() == sat:
        m = s.model()
        for x in inp:
            w.append(str(m[x]))

        print("".join(w))

        new = []
        for x in inp:
            new.append(x != m[x])
        s.add(Or(new))
        w = []
        i += 1
        if i > 20:
            print("too much")
            exit()
```

Рисунок 8 — использование z3 для верификации

На рисунке 8 реализовано нахождение всевозможных входных данных для функции SmallSigma, дающие известный нам результат.

При выполнении команды “python veryfye.py 2”.

```
sarkoxedaf@orgasmotron ~/Working/verification_repo/shaparts/sigma [master]
$$ % python veryfye.py full
_____________________________________________________________________

Найденные Входные сигналы:
10000101010111100001001001001100
01011110001100111010010010010111
```

Рисунок 9 — нахождение входных сигналов

```
sarkoxedaf@orgasmotron ~/Working/verification_repo/shaparts/sigma [master]
$$ % cat data/input.json                                                !10055
{"in": ["1", "0", "0", "0", "0", "1", "0", "1", "0", "1", "0", "1", "1", "1", "1", "0", "0",
 "0", "0", "1", "0", "0", "1", "0", "0", "1", "0", "0", "1", "1", "0", "0"]}
```

Рисунок 10 — тестовый входной сигнал

На рисунке 9 видно, что существует два входных сигнала, удовлетворяющих данному нам выходному сигналу. Это вполне логично, что существует несколько входных данных, приводящих к одинаковому выводу, ведь в SmallSigma присутствует операция shr, не являющаяся обратимой. Как мы можем заметить, первый из них полностью совпадает с тестовым.

При выполнении команды “make check” запустится процесс в котором будут посчитаны все ограничения на лежащей в директории src схеме main.circom, посчитан свидетель на основе входных данных из data/input.json, обработан файл constraints/main.r1cs с помощью скрипта tools/map.js и проведена верификация работы арифметической схемы с помощью скрипта tools/verify.py.

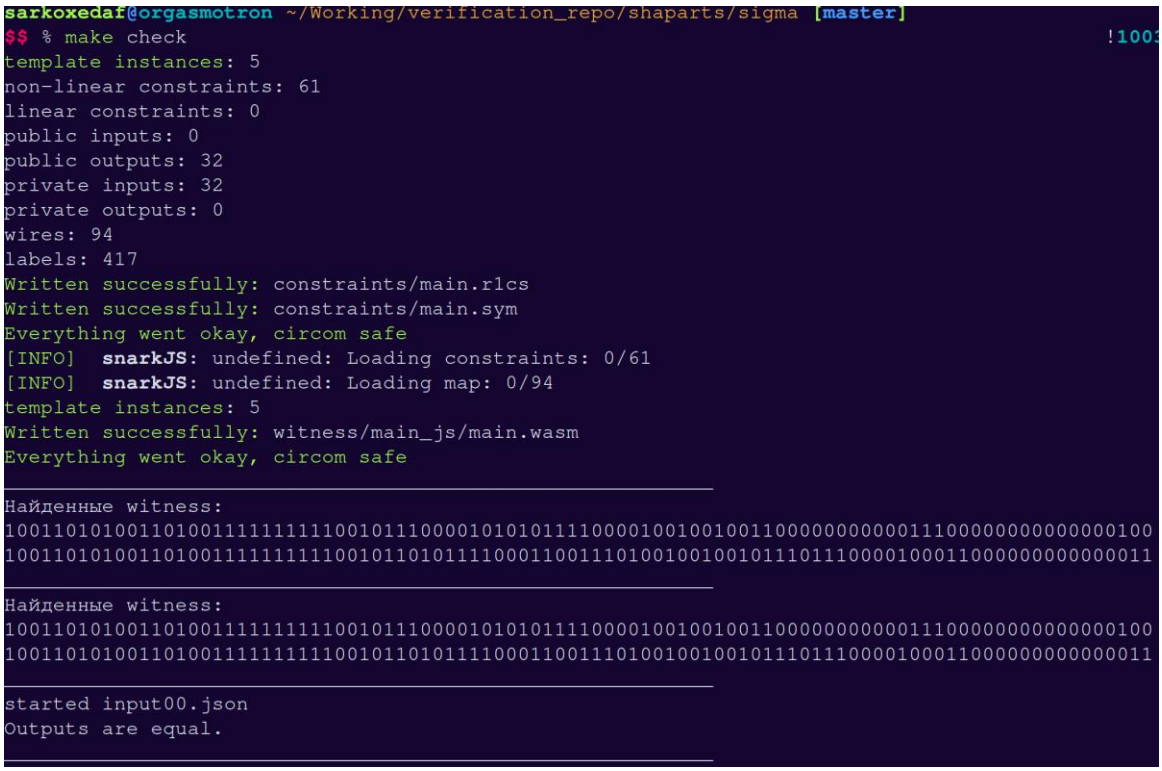

Рисунок 11 — вывод команды “make check”

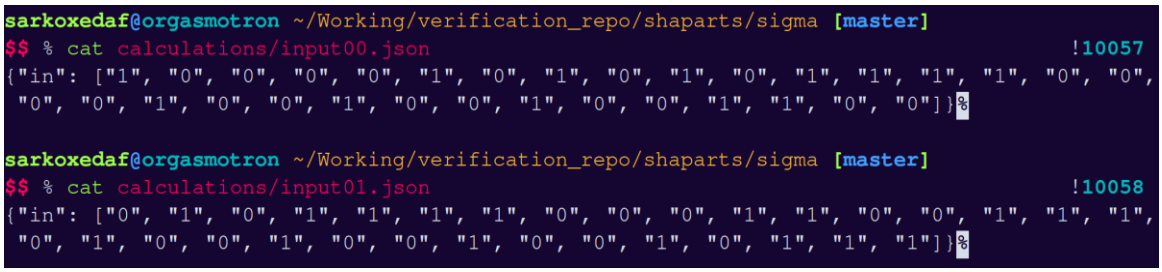

Рисунок 12 — посчитанные входные сигналы

На рисунке 12 видно, что все посчитанные входные сигналы идентичны тем, что были посчитаны с помощью прямой реализации функции SmallSigma.

Далее с помощью команды “python very_equal.py” будет проверено, что при любых входных данных результаты работы схемы и функции идентичны.

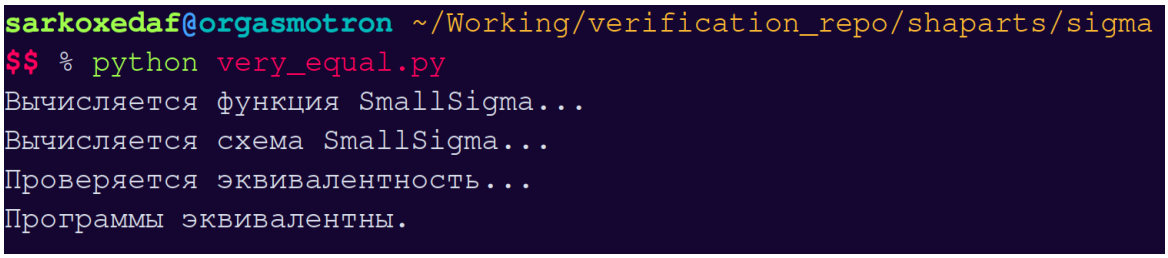

Рисунок 13 — результат работы команды “python very_equal.py”

ЗАКЛЮЧЕНИЕ

В научно-исследовательской работе приведено описание доказательств с нулевым разглашением. Также были затронуты способы их практической реализации и внедрения в повседневную жизнь, на примере PLONK. Рассмотрен язык для написания арифметических схем Circom.

Был проведён анализ безопасности этих схем. Подробно был рассмотрен метод тестирования схем при помощи символьного исполнения.

Результаты проделанной работы показали целесообразность использования символьного исполнения при тестировании. Также были представлены случаи, когда символьное исполнение может быть применено.

В будущих исследованиях планируется максимально автоматизировать процесс тестирования и закончить анализ sha256. Дополнительно планируется протестировать больше примитивов из библиотеки circomlib и схем, написанных на других DSL, например, на новом языке Noir. Кроме того, планируется переход с z3 на cvc5(другой SMT-solver).

В первом разделе были рассмотрены основные понятия в области ZKP, семейство протоколов zk-SNARKs и их представитель PLONK.

Во втором разделе был рассмотрен DSL Circom.

В третьем разделе была проанализированы безопасность схем и подходы к их верификации. Подробно было рассмотрено символьное исполнение.

В четвертом разделе был проанализирован примитив SmallSigma из библиотеки circomlib и была проведена его верификация.

ПРИЛОЖЕНИЕ

1 Использование утилит circom и snarkjs

```
1   pragma circom 2.0.0;
2
3   template NAND() {
4       signal input a;
5       signal input b;
6       signal output out;
7
8       out <== 1 - a*b;
9       a*(a-1) === 0;
10      b*(b-1) === 0;
11  }
12
13  component main = NAND();
```

Рисунок 14 — схема main.circom

Для компиляции схемы необходимо воспользоваться утилитой “circom”

```
sarkoxedaf@sarkoxedaf ~/test
$$ % circom main.circom --wasm --sym --r1cs --output out/

template instances: 1
non-linear constraints: 3
linear constraints: 0
public inputs: 0
public outputs: 1
private inputs: 2
private outputs: 0
wires: 4
labels: 4
Written successfully: out/main.r1cs
Written successfully: out/main.sym
Written successfully: out/main_js/main.wasm
Everything went okay, circom safe
```

Рисунок 15 — компиляция схемы main.circom

На рисунке 15 происходит компиляция схем с созданием R1CS файла, sym файла (свидетель в симметричном формате) и программы, считающей свидетеля на языке программирования WebAssembly (низкоуровневый язык программирования для написания скриптов для браузера).

Для подсчета свидетеля необходимо использовать утилиту “snarkjs”.

```
sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs wc out/main_js/main.wasm input.json witness.wtns

sarkoxedaf@sarkoxedaf ~/test
$$ % ls
out   input.json  main.circom  witness.wtns
```

Рисунок 16 — подсчёт свидетеля

1.1 Проведение церемонии доверенной установки

В случае PLONK церемонию не проводят сами, а используют уже проведенную. Однако, для примера вот как она происходит.

```
sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs powersoftau new bn128 12 pot12_0000.ptau -v
[DEBUG] snarkJS: Calculating First Challenge Hash
[DEBUG] snarkJS: Calculate Initial Hash: tauG1
[DEBUG] snarkJS: Calculate Initial Hash: tauG2
[DEBUG] snarkJS: Calculate Initial Hash: alphaTauG1
[DEBUG] snarkJS: Calculate Initial Hash: betaTauG1
[DEBUG] snarkJS: Blank Contribution Hash:
                 786a02f7 42015903 c6c6fd85 2552d272
                 912f4740 e1584761 8a86e217 f71f5419
                 d25e1031 afee5853 13896444 934eb04b
                 903a685b 1448b755 d56f701a fe9be2ce
[INFO]  snarkJS: First Contribution Hash:
                 9e63a5f6 2b96538d aaed2372 481920d1
                 a40b9195 9ea38ef9 f5f6a303 3b886516
                 0710d067 c09d0961 5f928ea5 17bcdf49
                 ad75abd2 c8340b40 0e3b18e9 68b4ffef
```

Рисунок 17 — начало церемонии

```
sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs powersoftau contribute pot12_0000.ptau pot12_0001.ptau --name="First contribution" -v
Enter a random text. (Entropy): NIR2022
[DEBUG] snarkJS: Calculating First Challenge Hash
[DEBUG] snarkJS: Calculate Initial Hash: tauG1
[DEBUG] snarkJS: Calculate Initial Hash: tauG2
[DEBUG] snarkJS: Calculate Initial Hash: alphaTauG1
[DEBUG] snarkJS: Calculate Initial Hash: betaTauG1
[DEBUG] snarkJS: processing: tauG1: 0/8191
[DEBUG] snarkJS: processing: tauG2: 0/4096
[DEBUG] snarkJS: processing: alphaTauG1: 0/4096
[DEBUG] snarkJS: processing: betaTauG1: 0/4096
[DEBUG] snarkJS: processing: betaTauG2: 0/1
[INFO]  snarkJS: Contribution Response Hash imported:
                 97a65627 900c1db0 7b48dd5c 66017b69
                 47cb3f07 901aedb1 cdf1d17a 57a2dfbb
                 3e68dceb 9021acd6 8e30d9ea d84ae0e5
                 5b08622b 48c3c709 d39d6e70 75b6d51e
[INFO]  snarkJS: Next Challenge Hash:
                 c887dffd db5d2b4a 5080477c 977587c5
                 2e84c3ba 34921ed0 efed23f3 f73c4c66
                 0b9eb10d a5c4cbe9 37609909 85fc740c
                 bf52e521 d6bf5a34 8baf3f00 5ece9066
```

Рисунок 18 — Первый вклад в церемонию

```
sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs powersoftau contribute pot12_0001.ptau pot12_0002.ptau --name="Second contribution" -v -e="NIR2022\!"
[DEBUG] snarkJS: processing: tauG1: 0/8191
[DEBUG] snarkJS: processing: tauG2: 0/4096
[DEBUG] snarkJS: processing: alphaTauG1: 0/4096
[DEBUG] snarkJS: processing: betaTauG1: 0/4096
[DEBUG] snarkJS: processing: betaTauG2: 0/1
[INFO]  snarkJS: Contribution Response Hash imported:
                 1c826c6c 0a3f73ab e31ba2fb 5e80c9dc
                 5f8e66bb 7cef83d8 9369b656 6b7fa942
                 2cf506fc b4581668 5bbf2b5d 6d879670
                 af8bbc10 5519fb09 24c79d87 9c750df8
[INFO]  snarkJS: Next Challenge Hash:
                 51adb0fb 61546110 d1aac2c1 bba13613
                 fceca251 7f855fa6 cf9b1ee9 ad9340d6
                 196f699c 7b841222 24d59c45 349f13aa
                 b4a707e3 6958e3ed f6891700 25ae27ec
```

Рисунок 19 — Второй вклад в церемонию

```
sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs powersoftau challenge contribute bn128 challenge_0003 response_0003 -e="NIR2022\!"
[INFO]  snarkJS: Claimed Previous Response Hash:
                 1c826c6c 0a3f73ab e31ba2fb 5e80c9dc
                 5f8e66bb 7cef83d8 9369b656 6b7fa942
                 2cf506fc b4581668 5bbf2b5d 6d879670
                 af8bbc10 5519fb09 24c79d87 9c750df8
[INFO]  snarkJS: Current Challenge Hash:
                 51adb0fb 61546110 d1aac2c1 bba13613
                 fceca251 7f855fa6 cf9b1ee9 ad9340d6
                 196f699c 7b841222 24d59c45 349f13aa
                 b4a707e3 6958e3ed f6891700 25ae27ec
[INFO]  snarkJS: Contribution Response Hash:
                 e016c738 a4a92549 93b3ed9b da686fda
                 ee34cf9b 5e71541d 1f7f9369 ee04766b
                 d8c5fb09 242a02d8 40506951 5b587d64
                 b9c5f8ac 61498e27 4f2c4871 8f97cefe
```

Рисунок 20 — Третий вклад, используя стороннее ПО (1)

```
sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs powersoftau import response pot12_0002.ptau response_0003 pot12_0003.ptau -n="Third contribution"
[INFO]  snarkJS: Contribution Response Hash imported:
                 e016c738 a4a92549 93b3ed9b da686fda
                 ee34cf9b 5e71541d 1f7f9369 ee04766b
                 d8c5fb09 242a02d8 40506951 5b587d64
                 b9c5f8ac 61498e27 4f2c4871 8f97cefe
[INFO]  snarkJS: Next Challenge Hash:
                 f1452db3 77f1618a fc9d6b51 dd53d015
                 d5c0e705 c97bc45f 6f9f24d5 d07e721c
                 d60cb2a4 7d74f2a5 d4e23e76 d0ab452f
                 065aa85f 209321ac d30746f5 dc146071
```

Рисунок 21 — Третий вклад, используя стороннее ПО (2)

```
sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs powersoftau verify pot12_0003.ptau
[INFO]  snarkJS: Powers Of tau file OK!
[INFO]  snarkJS: Next challenge hash:
                 f1452db3 77f1618a fc9d6b51 dd53d015
                 d5c0e705 c97bc45f 6f9f24d5 d07e721c
                 d60cb2a4 7d74f2a5 d4e23e76 d0ab452f
                 065aa85f 209321ac d30746f5 dc146071
[INFO]  snarkJS: ------------------------------------------------------
[INFO]  snarkJS: Contribution #3: Third contribution
[INFO]  snarkJS: Next Challenge:
                 f1452db3 77f1618a fc9d6b51 dd53d015
                 d5c0e705 c97bc45f 6f9f24d5 d07e721c
                 d60cb2a4 7d74f2a5 d4e23e76 d0ab452f
                 065aa85f 209321ac d30746f5 dc146071
```

Рисунок 22 — Проверка предыдущих шагов (1)

```
[INFO]  snarkJS: ----------------------------------------------------
[WARN]  snarkJS: this file does not contain phase2 precalculated values. Please run:
   snarkjs "powersoftau preparephase2" to prepare this file to be used in the phase2 ceremony.
[INFO]  snarkJS: Powers of Tau Ok!
```

Рисунок 23 — Проверка прошла

```
sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs powersoftau prepare phase2 pot12_beacon.ptau pot12_final.ptau -v
[DEBUG] snarkJS: Starting section: tauG1
[DEBUG] snarkJS: tauG1: fft 0 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 0 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 1 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 1 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 2 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 2 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 3 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 3 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 4 mix start: 0/2
[DEBUG] snarkJS: tauG1: fft 4 mix start: 1/2
[DEBUG] snarkJS: tauG1: fft 4 mix end: 0/2
[DEBUG] snarkJS: tauG1: fft 4 mix end: 1/2
[DEBUG] snarkJS: tauG1: fft  4  join: 4/4
[DEBUG] snarkJS: tauG1: fft 4 join  4/4  1/1 0/1
[DEBUG] snarkJS: tauG1: fft 5 mix start: 0/4
[DEBUG] snarkJS: tauG1: fft 5 mix start: 1/4
[DEBUG] snarkJS: tauG1: fft 5 mix start: 2/4
[DEBUG] snarkJS: tauG1: fft 5 mix start: 3/4
```

Рисунок 24 — Подготовка к генерации доказательства

```
sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs powersoftau verify pot12_final.ptau
[INFO]  snarkJS: Powers Of tau file OK!
[INFO]  snarkJS: Next challenge hash:
                 2a260b7e 03801b7a eac40dc7 c53e7b13
                 71b34e77 9801c9bc 680a8670 d8d5f2e9
                 e44b6740 0acb9170 be0532b0 91b89814
                 34bdedea 5d116749 480b65a3 42662e88
```

Рисунок 25 — Проверка публичных параметров

```
[INFO]  snarkJS: ----------------------------------------------------
[INFO]  snarkJS: Powers of Tau Ok!
```

Рисунок 26 — Проверка прошла

```
sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs plonk setup out/main.r1cs pot12_final.ptau circuit_final.zkey
[INFO]  snarkJS: Reading r1cs
[INFO]  snarkJS: Plonk constraints: 4
[INFO]  snarkJS: Setup Finished

sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs zkey export verificationkey circuit_final.zkey verification_key.json

sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs plonk prove circuit_final.zkey witness.wtns proof.json public.json

sarkoxedaf@sarkoxedaf ~/test
$$ % snarkjs plonk verify verification_key.json public.json proof.json
[INFO]  snarkJS: OK!

sarkoxedaf@sarkoxedaf ~/test
$$ % ls
out                input.json       pot12_0001.ptau  pot12_beacon.ptau  public.json           witness.wtns
challenge_0003     main.circom      pot12_0002.ptau  pot12_final.ptau   response_0003
circuit_final.zkey pot12_0000.ptau  pot12_0003.ptau  proof.json         verification_key.json
```

Рисунок 27 — доказательство и проверка

2 Вспомогательные утилиты для верификации

```
1  pragma circom 2.0.0;
2
3  include "sigma.circom";
4
5  component main = SmallSigma(1, 2, 3);
```

Рисунок 28 — main.circom

```
19  pragma circom 2.0.0;
20
21  template ShR(n, r) {
22      signal input in[n];
23      signal output out[n];
24
25      for (var i=0; i<n; i++) {
26          if (i+r >= n) {
27              out[i] <== 0;
28          } else {
29              out[i] <== in[ i+r ];
30          }
31      }
32  }
33
```

Рисунок 29 — shift.circom

```
21  template RotR(n, r) {
22      signal input in[n];
23      signal output out[n];
24
25      for (var i=0; i<n; i++) {
26          out[i] <== in[ (i+r)%n ];
27      }
28  }
```

Рисунок 30 — sigma.circom

```
34  template Xor3(n) {
35      signal input a[n];
36      signal input b[n];
37      signal input c[n];
38      signal output out[n];
39      signal mid[n];
40
41      for (var k=0; k<n; k++) {
42          mid[k] <== b[k]*c[k];
43          out[k] <== a[k] * (1 -2*b[k]  -2*c[k] +4*mid[k]) + b[k] + c[k] -2*mid[k];
44      }
45  }
```

Рисунок 31 — xor3.circom

```
19  pragma circom 2.0.0;
20
21  include "xor3.circom";
22  include "rotate.circom";
23  include "shift.circom";
24
25  template SmallSigma(ra, rb, rc) {
26      signal input in[32];
27      signal output out[32];
28      var k;
29
30      component rota = RotR(32, ra);
31      component rotb = RotR(32, rb);
32      component shrc = ShR(32, rc);
33
34      for (k=0; k<32; k++) {
35          rota.in[k] <== in[k];
36          rotb.in[k] <== in[k];
37          shrc.in[k] <== in[k];
38      }
39
40      component xor3 = Xor3(32);
41      for (k=0; k<32; k++) {
42          xor3.a[k] <== rota.out[k];
43          xor3.b[k] <== rotb.out[k];
44          xor3.c[k] <== shrc.out[k];
45      }
46
47      for (k=0; k<32; k++) {
48          out[k] <== xor3.out[k];
49      }
50  }
```

Рисунок 32 — sigma.circom

```
var fastFile = require('fastfile');

const r1csfile = require("r1csfile");
var r1csName = "constraints/main.r1cs";
var symName =  "constraints/main.sym";

//const cir = r1csfile.readR1cs(r1csName, true, true, false);

async function map(symFileName){
    const fd = await fastFile.readExisting(symFileName);
    const buff = await fd.read(fd.totalSize);
    const symsStr = new TextDecoder("utf-8").decode(buff);
    const lines = symsStr.split("\n");

    var varIdx2Name = [ "one" ];

    for (let i=0; i<lines.length; i++) {
        const arr = lines[i].split(",");
        if (arr.length!=4) continue;
        if (varIdx2Name[arr[1]]) {
            varIdx2Name[arr[1]] += "|" + arr[3];
        } else {
            varIdx2Name[arr[1]] = arr[3];
        }
    }
    await fd.close();

    var data = {
        map: varIdx2Name
    };

    var jsonData = JSON.stringify(data);


    var fs = require('fs');
    fs.writeFile("data/map.json", jsonData, function(err) {
        if (err) {
            console.log(err);
        }
    });
}

map(symName);
```

Рисунок 33 — map.js (программа для обработки R1CS файла)

```python
from z3 import BitVec, Solver, sat, Or
import json
from sys import argv


constrs = json.load(open("constraints/constr.json"))
witness = json.load(open("witness/witness.json"))
map = json.load(open("data/map.json"))["map"]

nvars = constrs["nVars"]
nout = constrs["nOutputs"]
ninp = constrs["nPrvInputs"]

p = int(constrs["prime"])

out = [int(x) for x in witness[1 : 1 + nout]]
# print(out)

vars = [BitVec(f"var_{i}", 1) for i in range(nvars)]

s = Solver()
s.add(vars[0] == 1)

for constr in constrs["constraints"]:
    abc = []
    for i in range(3):
        t = 0
        for varn, var in constr[i].items():
            n = int(varn)
            v = int(var)
            if v > p // 2:
                v = v - p
            t += vars[n] * v
        abc.append(t)
    a, b, c = abc
    s.add(a * b - c == 0)

for i in range(nout):
    s.add(vars[i + 1] == out[i])
```

Рисунок 34 — использование R1CS файла для верификации(1)

```
w = []
i = 0
print("_____________________________________________________________")
print("Найденные witness:")
if argv[1] == "1":
    if s.check() == sat:
        m = s.model()
        for x in vars:
            w.append(str(m[x]))
        with open("twitness/witness.json", "wt") as f:
            json.dump(w, f)
        with open("data/calculated_input.json", "wt") as f:
            json.dump([w[x] for x in range(len(w)) if "main.in" in map[x]], f)
    else:
        print("unsat")
else:
    while s.check() == sat:
        m = s.model()
        for x in vars:
            w.append(str(m[x]))

        print("".join(w))#len(w))
        with open(f"calculations/input{str(i).zfill(2)}.json", "wt") as f:
            json.dump({"in": [w[x] for x in range(len(w)) if "main.in" in map[x]]}, f)

        new = []
        for x in vars:
            new.append(x != m[x])
        s.add(Or(new))
        w = []
        i += 1
        if i > 20:
            print("too much")
            exit()
```

Рисунок 35 — использование R1CS файла для верификации(2)

```
.PHONY: default
default: init

.PHONY: init
init:
	@mkdir src || true
	@mkdir data || true
	@mkdir witness || true
	@mkdir twitness || true
	@mkdir constraints || true
	@mkdir calculations || true
	@cp -r ../tools tools || true
	@touch src/main.circom || true
	@printf "pragma circom 2.0.0;\n\ninclude \"temp.circom\";\n\ncomponent main = temp();" > src/main.circom

.PHONY: verify
verify: constraints witness map veripy

.PHONY: deep_verify
deep_verify: constraints witness map deep_veripy

.PHONY: check
check: deep_verify tools/verify.py constraints/constr.json witness/witness.json tools/map.js
	@rm calculations/*
	@python tools/verify.py 2
	@tools/check.sh

.PHONY: witness
witness: src/*.circom data/input.json
	@circom src/main.circom --wasm -o witness/
	@snarkjs wc witness/main_js/main.wasm data/input.json witness/witness.wtns
	@snarkjs wej witness/witness.wtns witness/witness.json

.PHONY: constraints
constraints: src/*.circom
	@circom src/main.circom --r1cs --sym -o constraints/
	@snarkjs rp constraints/main.r1cs constraints/main.sym > constraints/constr
	@snarkjs rej constraints/main.r1cs constraints/constr.json

```

Рисунок 36 — makefile для удобства работы в директории (1)

```
40  .PHONY: map
41  map: constraints
42          @node tools/map.js
43
44  .PHONY: veripy
45  veripy: tools/verify.py constraints/constr.json witness/witness.json
46          @python tools/verify.py 1
47
48  .PHONY: deep_veripy
49  deep_veripy: tools/verify.py constraints/constr.json witness/witness.json tools/map.js
50          @python tools/verify.py 2
51
52  .PHONY: self_witness
53  self_gen_witness: data/input1.json src/main.circom
54          @circom src/main.circom --wasm -o twitness/
55          @snarkjs wc twitness/main_js/main.wasm data/input1.json twitness/witness.wtns
56          @snarkjs wej twitness/witness.wtns twitness/witness.json
57
58  .PHONY: test
59  test: tools/test.js src/main.circom
60          @mocha tools/test.js
61
62  .PHONY: clean
63  clean:
64          @echo "Cleaning constraints, witness, twitness directories"
65          @rm constraints/* 2>/dev/null
66          @rm -r witness/* 2>/dev/null
67          @rm -r twitness/* 2>/dev/null
```

Рисунок 36 — makefile для удобства работы в директории (2)

```
39  def gen_python(inp, ar=1, br=2, cr=3):
40      out = SmallSigma(inp, ar, br, cr)
41      return out
42
43
44  def gen_r1cs(inp, out, constrs, map, nvars, p):
45      vars = []
46      j = 0
47      for i in range(nvars):
48          if "main.in" not in map[i]:
49              vars.append(BitVec(f"{map[i]}", 1))
50          else:
51              vars.append(inp[j])
52              j += 1
53      s = Solver()
54      s.add(vars[0] == 1)
55
56      for constr in constrs["constraints"]:
57          abc = []
58          for i in range(3):
59              t = 0
60              for varn, var in constr[i].items():
61                  n = int(varn)
62                  v = int(var)
63                  if v > p // 2:
64                      v = v - p
65                  t += vars[n] * v
66              abc.append(t)
67          a, b, c = abc
68          s.add(a * b - c == 0)
69
70      for i in range(nout):  # TODO if "main,out" in ...
71          s.add(vars[i + 1] != out[i])
72      return s
```

Рисунок 37 — very_equal.py (1)

```
constrs = json.load(open("constraints/constr.json"))
map = json.load(open("data/map.json"))["map"]

nvars = constrs["nVars"]
nout = constrs["nOutputs"]
ninp = constrs["nPrvInputs"]
p = int(constrs["prime"])

inp = [BitVec(f"main.in[{i}]", 1) for i in range(ninp)]
print("Вычисляется функция SmallSigma...")
out = gen_python(inp)
print("Вычисляется схема SmallSigma...")
s = gen_r1cs(inp, out, constrs, map, nvars, p)

print("Проверяется эквивалентность...")
res = s.check()
if res == unsat:
    print("Программы эквивалентны.")
else:
    print("Программы не эквивалентны. Возможные входные данные:")
    w = []
    i = 0
    if argv[1] == "1":
        if s.check() == sat:
            m = s.model()
            for x in inp:
                w.append(str(m[x]))
            with open("twitness/witness.json", "wt") as f:
                json.dump(w, f)
            with open("data/calculated_input.json", "wt") as f:
                json.dump([w[x] for x in range(len(w)) if "main.in" in map[x]], f)
        else:
            print("unsat")
    else:
        while s.check() == sat:
            m = s.model()
            for x in inp:
                w.append(str(m[x]))

            print("".join(w))  # len(w))
            with open(f"calculations/input{str(i).zfill(2)}.json", "wt") as f:
                json.dump(
                    {"in": [w[x] for x in range(len(w)) if "main.in" in map[x]]}, f
                )

            new = []
            for x in inp:
                new.append(x != m[x])
```

Рисунок 38 — very_equal.py (1)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Goldreich O. Zero-Knowledge twenty years after its invention [Текст] // Weizmann Institute of Science. — 2002.

2. Zero Knowledge Proofs: An illustrated primer [Электронный ресурс] — Режим доступа к ресурсу: https://blog.cryptographyengineering.com/2014/11/27/zero-knowledge-proofs-illustrated-primer, свободный.

3. Heuristique de Fiat-Shamir [Электронный ресурс] — Режим доступа к ресурсу: https://ru.frwiki.wiki/wiki/Heuristique_de_Fiat-Shamir, свободный.

4. An approximate introduction to how zk-SNARKs are possible [Электронный ресурс] — Режим доступа к ресурсу: https://vitalik.ca/general/2021/01/26/snarks.html, свободный.

5. Arithmetic circuit complexity [Электронный ресурс] — Режим доступа к ресурсу: https://en.wikipedia.org/wiki/Arithmetic_circuit_complexity, свободный.

6. Thomas Chen, Hui Lu, Teeramet Kunpittaya, Alan Luo. A Review of zk-SNARKs // Arxiv. — Доступ к ресурсу URL: https://arxiv.org/pdf/2202.06877.pdf

7. r1cs [Электронный ресурс] — Режим доступа к ресурсу: https://www.zeroknowledgeblog.com/index.php/the-pinocchio-protocol/r1cs, свободный.

8. r1cs bin format [Электронный ресурс] — Режим доступа к ресурсу: https://github.com/iden3/r1csfile/blob/master/doc/r1cs_bin_format.md, свободный.

9. Символическое исполнение [Электронный ресурс] — Режим доступа к ресурсу: https://coderlessons.com/tutorials/kachestvo-programmnogo-obespecheniia/slovar-testirovaniia-programmnogo-obespecheniia/simvolicheskoe-ispolnenie, свободный.

10. Programming z3 [Электронный ресурс] — Режим доступа к ресурсу: https://theory.stanford.edu/~nikolaj/programmingz3.html#sec-intro, свободный.

11. Security of ZK systems [Электронный ресурс] — Режим доступа к ресурсу: https://www.youtube.com/watch?v=SxI8uNBp05k&t=4739s, свободный.

12. Circomlib [Электронный ресурс] — Режим доступа к ресурсу: https://github.com/iden3/circomlib, свободный.

13. Circom Language [Электронный ресурс] — Режим доступа к ресурсу: https://docs.circom.io, свободный.

14. Git репозиторий исследования [Электронный ресурс] — Режим доступа к ресурсу: https://github.com/Sarkoxed/Research.git, свободный.

15. Understanding trusted setups [Электронный ресурс] — Режим доступа к ресурсу: https://blog.pantherprotocol.io/a-guide-to-understanding-trusted-setups, свободный.

16. Understanding PLONK [Электронный ресурс] — Режим доступа к ресурсу: https://vitalik.ca/general/2019/09/22/plonk.html, свободный.

17. Elliptic curve pairings [Электронный ресурс] — Режим доступа к ресурсу: https://medium.com/@VitalikButerin/exploring-elliptic-curve-pairings-c73c1864e627, свободный.

18. Tornado Cash [Электронный ресурс] — Режим доступа к ресурсу: https://github.com/tornadocash/tornado-core, свободный.