Министерство науки и высшего образования Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Институт прикладной математики и компьютерных наук
Кафедра компьютерной безопасности

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Руководитель ООП
канд. техн. наук, доцент
_____ С. А. Останин
«26» января 2021 г.

ДИПЛОМНАЯ РАБОТА

Анализ уязвимостей смарт-контрактов платформы Ethereum на базе формальной верификации

на основной образовательной программе подготовки специалистов «Анализ безопасности компьютерных систем» специальности подготовки 10.05.01 Компьютерная безопасность

Аламов Владимир Александрович

Руководитель ВКР
канд. техн. наук, доцент
_____ Тренькаев В. Н.
«25» января 2021 г.

Автор работы
студент группы № 1155
_____ Аламов В. А.

Томск – 2021

Министерство науки и высшего образования Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Институт прикладной математики и компьютерных наук
Кафедра компьютерной безопасности

УТВЕРЖДАЮ
Руководитель ООП
канд. техн. наук, доцент
_____ С. А. Останин
«9» сентября 2020 г.

ЗАДАНИЕ
по подготовке ВКР специалиста студенту Аламова Владимира Александровича группы 1155.

1. Тема ВКР: «Анализ уязвимостей смарт-контрактов платформы Ethereum на базе формальной верификации».
2. Срок сдачи студентом выполненной дипломной работы:

а) на кафедре: 19.01.21

б) в ГЭК: 1.02.21

3. Исходные данные к работе: Цель работы: исследовать возможность проведения анализа уязвимостей смарт-контрактов с использованием метода формальной верификации. Объекты исследования: блокчейн Ethereum, смарт-контракты, уязвимости смарт-контрактов, методы обнаружения уязвимостей смарт-контрактов, формальная верификация. Методы исследования: аналитический обзор современной научно-технической литературы, затрагивающей проблематику ВКР, апробация на практике предложенного метода анализа уязвимостей смарт-контрактов на базе формальной верификации.
4. Краткое содержание работы: изучение технологии смарт-контрактов в блокчейн Ethereum, существующих уязвимостей смарт-контрактов, существующих решений по выявлению уязвимостей смарт-контрактов на языке Solidity (к 01.11.2020), анализ подходов по формальной верификации смарт-контрактов (к 01.12.2020), разработка метода анализа уязвимостей смарт-контрактов на базе формальной верификации (к 16.12.2020), анализ полученных результатов и написание текста дипломной работы (к 17.01.2021).
5. Работа выполняется по заданию кафедры информационной безопасности Томского государственного университета.
6. Дата выдачи задания 5 сентября 2020 г.

Руководитель ВКР
канд. техн. наук, доцент _____ / Тренькаев В. Н./
Задание принял к исполнению
студент группы № 1155 _____ / Аламов В. А./

РЕФЕРАТ

В работе рассматриваются методы формальной верификации смарт-контрактов и предлагается способ анализа уязвимостей смарт-контрактов на базе формальной верификации, а именно проверки на модели. Реализован инструмент для проведения верификации на базе графого представления смарт-контрактов. Произведено тестирование предложеного метода верификации смарт-контрактов на реальных примерах смарт-контрактов.

Ключевые слова: блокчейн Ethereum, смарт-контракт, уязвимость смарт-контракта, формальная верификация.

Работа содержит 36 страницы, 14 литературных источников, 19 листингов.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Смарт-контракт - компьютерный алгоритм, предназначенный для формирования, контроля и представления информации о владении чем-либо в сети Ethereum. При создании смарт-контрактов долгое время безопасности не уделялось должного внимания. Стимулом, для развития безопасности смарт-контрактов стал инцидент с DAO[1]. В настоящее время, смарт-контракты еще не достигли пика популярности, однако многие компании на рынке ценных бумаг стараются оцифровать свои нотариальные документы и перевести их в смарт-контракты [2]. В 2018 году произошло резкое увеличение интереса к блокчейн технологиям из-за роста ценности Bitcoin. В данный момент происходит новый рост цен криптовалют и интереса к ним. Это сопровождается увеличением количества смарт-контрактов в сети Ethereum.

Для аудита смарт-контрактов используются автоматические средства анализа безопасности [3] [4]. Они основаны на статическом анализе, который имеет высокий процент ошибок первого рода. Также возможен анализ смарт-контрактов на базе формальной верификации. Существует два основных подхода к построению формальных доказательств того, что модель программной системы удовлетворяет своей спецификации:

- логический вывод, т.е. автоматизированное построение доказательства теоремы о том, что анализируемая модель удовлетворяет проверяемой спецификации, и
- проверка на модели, когда ищется вариант функционирования анализируемой модели, при котором нарушается свойство, выражаемое проверяемой спецификацией. [5].

В данной работе используется метод формальной верификации - проверка на модели. При верификации необходимо построить формальную модель системы, после чего представить модель на языке, который понимает средство верификации. Как правило при построении модели необходимо абстрагироваться от деталей для упрощения модели.

В работе рассматривается задача трансляция кода программы на языке Solidity в код на языке Promela. *Promela* - абстрактный язык спецификации алгоритмов [6], который используется в верификаторе *SPIN*. Предлагается способ построения графовой модели смарт-контракта по его коду на языке Solidity. Далее по графовой модели строится код на языке Promela, что затем позволяет использовать популярное средство верификации *SPIN* для анализа уязвимостей смарт-контрактов.

1 Технология Блокчейн

Ethereum - криптовалюта на основе технологии блокчейн.

1.1 Технология блокчейн

Блокчейн - совместно используемый, неизменный реестр, предназначенный для записи транзакций, учета активов и построения доверительных отношений.

Особенности технологии блокчейн:

- «Распределенный реестр», т.е. вся информация распределяется между всеми участниками сети. Таким образом, у каждого участника сети есть вся цепочка блоков за весь период существования блокчейна. Несмотря на то, что каждый участник сети видит все блоки, он обладает доступом только к своим блокам за счет шифрования доступа к каждому из них.

- Для защиты от несанкционированного изменения каждый блок содержит собственную хеш-сумму и хеш-сумму предыдущего блока. Т.о. для редактирования информации в блоке, придется редактировать и все следующие блоки. А так как копии блоков хранятся и на других машинах, то вам нужно редактировать блоки на 51% всех машин, чтобы эти изменения были приняты сетью.

- Передача доступа к блокам происходит в автоматическом режиме по принципу цифровой подписи – ввел код, подтвердил передачу права, и процесс завершен. Первым применением технологии блокчейн стала криптовалюта биткойн. Однако область применения технологии блокчейн гораздо шире, это финансовые операции, идентификация пользователей, создание технологий кибербезопасности.

- Любые изменения без подтверждения криптографическими ключами отклоняются. Эта функция особенно привлекательна для компаний,т.к. благодаря ей легко проверить подлинность личности без посещения офиса.

- Передача закрытого ключа предоставляет полный доступ к блоку (деньгам и иным активам). Это облегчает регистрацию сделок, которые проходят через онлайн-ресурсы.

1.2 Виртуальная машина Ethereum

Ethereum Virtual Machine представляет собой среду выполнения смарт-контрактов в системе Ethereum. Ее основной особенностью является изолированность от внешнего мира, то есть код, работающий внутри EVM, не имеет доступа к сети, файловой системе или другим подобным элементам. Смарт-контракты также могут иметь ограниченный доступ к другим смарт-контрактам.

Если говорить проще, то EVM – это своего рода виртуальный компьютер, на котором все узлы сети Ethereum соглашаются работать. Когда есть код (или данные) в блокчейне, необходим консенсус, чтобы договориться о том, что этот код будет делать. Все узлы согласны с тем, как должен себя вести EVM (то есть согласны с его правилами), и все имеют одинаковые данные в рамках данного блокчейна, поэтому каждый узел будет обрабатывать одни и те же данные. С этой точки зрения, виртуальная машина EVM похожа на один большой всемирный компьютер.

В сети Ethereum имеется 2 вида учетных записей, имеющих одинаковое адресное пространство: внешние учетные записи, которые управляются парами ключей открытого доступа, и контактные учетные записи, которые контролируются кодом, хранящимся вместе с учетной записью. Адрес внешней учетной записи определяется из открытого ключа, в то время как адрес контракта определяется на момент создания контракта (он получается из адреса создателя и количества транзакций, отправленных с этого адреса). Независимо от того, хранит ли код учетная запись или нет, два типа одинаково обрабатываются EVM. Каждая учетная запись имеет постоянное хранилище значений ключа, сопоставляющее 256-битные слова с 256-битными словами, называемыми хранилищем.

Транзакция представляет собой сообщение, которое отправляется из одной учетной записи в другую учетную запись. Оно может включать двоичные данные (полезную информацию) и эфир (Ether). Если целевая учетная запись содержит код, этот код выполняется и полезная информация предоставляется в качестве входных данных. Если целевой учетной записью является нулевая учетная запись (с адресом 0), транзакция создает новый контракт. Полезная информация такой транзакции создания контракта принимается как байт-код EVM и выполняется. Результат этого исполнения постоянно хранится как код контракта.

После создания каждой транзакции взимается определенное количество газа, целью которого является ограничение объема работы, необходимой для прохождения транзакции, и для оплаты этого исполнения. Хотя EVM выполняет транзакцию, газ постепенно истощается в соответствии с конкретными правилами. В системе учитывается цена на газ, то есть стоимость, установленная создателем транзакции, которая должна оплачивать газ. Если после исполнения остаётся некоторый газ, он возвращается таким же образом. Если газ расходуется (т. е. его количество становится отрицательным), запускается исключение типа «отсутствие газа», которое возвращает все изменения, сделанные в текущем кадре вызова. Как уже упоминалось ранее, каждая учетная запись имеет постоянную область памяти, которая называется хранилищем (storage). Это хранилище ключей, которое ставит в соответствие одни 256-битные слова другим 256-битным словам. Невозможно «перебрать» хранилище в рамках контракта, и его сравнительно дорого читать и тем более изменять это хранилище. Контракт не может ни читать, ни писать в любое другое хранилище, кроме его собственного. Вторая область памяти называется собственно памятью (memory), из которой контракт получает только «очищенный» экземпляр объекта для каждого вы-

зова сообщения. Память является линейной и может адресоваться на уровне байтов, но чтение ограничено шириной в 256 бит, тогда как запись может быть либо 8-битного, либо 256-битного формата.

EVM не является регистровой машиной, она использует стековую модель, поэтому все вычисления выполняются в области, называемой стеком. Он имеет максимальный размер 1024 элемента и содержит слова из 256 бит. Доступ к стеку ограничивается верхней планкой следующим образом: возможно скопировать один из самых верхних 16 элементов в верхнюю часть стека или заменить верхний элемент одним из 16 элементов под ним. Все остальные операции берут верхние два (или один или более, в зависимости от операции) элемента из стека и помещают результат в стек. Конечно, можно перемещать элементы стека в хранилище или память, но невозможно просто получить доступ к произвольным элементам глубоко в стеке, не удалив сначала верхнюю часть стека.

Набор команд EVM поддерживается минимальным, чтобы избежать неправильных реализаций, которые могут вызвать проблемы с консенсусом. Все инструкции работают с базовым типом данных, то есть 256-битными словами. В наборе команд EVM присутствуют обычные арифметические, битовые, логические операции и операции сравнения. Возможны условные и безусловные переходы. Кроме того, контракты могут получить доступ к соответствующим свойствам текущего блока, например, узнать его номер и временную метку.

1.3 Технология смарт-контрактов

«Смарт-контракт» - это программа, выполняющаяся в блокчейне Ethereum. Это набор кода (его функции) и данных (его состояние), который находится по определенному адресу в цепочке блоков Ethereum. Смарт-контракты - это разновидность учетной записи Ethereum. Это означает, что у них есть баланс, и они могут отправлять транзакции в сеть. Однако они не контролируются пользователем, вместо этого они развертываются в сети и работают в соответствии с алгоритмом. Учетные записи пользователей могут затем взаимодействовать со смарт-контрактом, отправляя транзакции, которые выполняют функцию, определенную в смарт-контракте. Смарт-контракты могут определять правила, как обычный контракт, и автоматически обеспечивать их соблюдение с помощью кода. Возможно, лучшая метафора смарт-контракта - это торговый автомат, описанный Ником Сабо. При правильных входах гарантирован определенный выход. Чтобы купить закуску в торговом автомате:

```
деньги + выбор закусок = снек
```

Эта логика запрограммирована в торговом автомате. В смарт-контракт, как в торговый автомат, заложена логика. Вот простой пример того, как этот торговый автомат может выглядеть как смарт-контракт:

```
1 pragma solidity 0.6.11;
```

```
2
3  contract VendingMachine {
4
5      address public owner;
6      mapping (address => uint) public cupcakeBalances;
7
8      constructor() public {
9          owner = msg.sender;
10         cupcakeBalances[address(this)] = 100;
11     }
12
13     function refill(uint amount) public {
14         require(msg.sender == owner, "Only the owner can
       refill.");
15         cupcakeBalances[address(this)] += amount;
16     }
17
18     function purchase(uint amount) public payable {
19         require(msg.value >= amount * 1 ether, "You must pay
       at least 1 ETH per cupcake");
20         require(cupcakeBalances[address(this)] >= amount,
       "Not enough cupcakes in stock to complete this purchase");
21         cupcakeBalances[address(this)] -= amount;
22         cupcakeBalances[msg.sender] += amount;
23     }
24 }
```

Листинг 1 — Пример торгового автомата

Подобно тому, как торговый автомат устраняет необходимость в сотруднике поставщика, смарт-контракты могут заменить посредников во многих отраслях. Такие контракты записываются в виде кода, существующего в распределенном реестре — блокчейне, который поддерживается и управляется сетью компьютеров. Другими словами, смарт контракты позволяют обмениваться активами, не прибегая к услугам посредников. Смарт-контракты общедоступны в Ethereum и могут рассматриваться как открытые API. Это означает, что вы можете вызывать другие смарт-контракты в своем собственном смарт-контракте, чтобы значительно расширить возможности. Контракты могут даже использовать другие контракты. Сами по себе смарт-контракты не могут получить информацию о «реальных» событиях, потому что они не могут отправлять HTTP-запросы. Это сделано намеренно, поскольку использование внешней информации может поставить под угрозу консенсус, который важен для безопасности и децентрализации. Смарт-контракты позволяют совершать надежные транзакции без участия третьих лиц. Кроме того, такие тран-

закции являются прослеживаемыми, прозрачными и необратимыми. Смарт-контракты не только содержат информацию об обязательствах сторон и санкциях за их нарушение, но и сами автоматически обеспечивают выполнение всех условий договора. Преимущества смарт-контрактов:

- Автономность - соглашаетесь только вы; нет необходимости полагаться на брокера, юриста или других посредников для подтверждения. Между прочим, это также исключает опасность манипуляции со стороны третьей стороны, поскольку выполнение управляется автоматически сетью, а не одним или несколькими, возможно, предвзятыми людьми, которые могут ошибаться.

- Доверие - ваши документы зашифрованы в общей бухгалтерской книге. Никто не может сказать, что потерял его.

- Резервное копирование. Представьте, что ваш банк потерял сберегательный счет. В блокчейне каждый из ваших друзей поддерживает вас. Ваши документы многократно дублируются.

- Безопасность - криптография , шифрование веб-сайтов, обеспечивает безопасность ваших документов. Взлома нет. На самом деле, чтобы взломать код и проникнуть в него, потребуется невероятно умный хакер.

- Скорость - обычно вам приходится тратить много времени и документов на ручную обработку документов. Смарт-контракты используют программный код для автоматизации задач, тем самым сокращая часы работы ряда бизнес-процессов.

- Экономия - Смарт-контракты экономят ваши деньги, поскольку они исключают присутствие посредника. Например, вам придется заплатить нотариусу, чтобы засвидетельствовать вашу сделку.

- Точность. Автоматизированные контракты не только быстрее и дешевле, но и позволяют избежать ошибок, возникающих при заполнении кучи форм вручную.

Обязательные атрибуты смарт-контракта:

1. использование методов электронной подписи на основе публичных и приватных ключей, имеющихся у двух или более сторон соглашения;

2. наличие приватной децентрализованной среды (например, Ethereum), в которую записываются смарт-контракты и которая поддерживает входы и выходы для оракулов, обеспечивающих связь реального и цифрового мира;

3. сам предмет договора и наличие необходимых для его исполнения инструментов (криптовалютных расчетных счетов, программ-оракулов и т. д.);

4. точно описанные условия его исполнения, которые участники договора подтверждают подписью, а также достоверность источника цифровых данных.

При исполнении смарт-контракты заверяются цифровой подписью каждой из сторон.

1.4 Язык программирования Solidity

Solidity — JavaScript-подобный объектно-ориентированный язык для разработки смарт-контрактов. Является кроссплатформенным, но на практике используется преимущественно на Ethereum[7]. Был создан в 2014 году командой программистов под руководством Кристиана Райтвизнера на основе идеи Гевина Вуда. Похожесть синтаксиса на JavaScript была рассчитана на быструю адаптацию разработчиков, которые до этого на нем разрабатывали подобные протоколы. Solidity поддерживает наследование, в том числе множественное. В сравнении с JS есть пара существенных отличий: Solidity — статически типизированный язык, динамическими являются только возвращаемые типы. Т.о. он больше напоминает TypeScript - строго типизированный JavaScript. Полноценной версии языка не вышло (текущая — 0.7.8), поэтому многие функциональность пока в урезанном виде, а количество багов достаточно велико. Однако с поставленными задачами Solidity справляется, а визуально это все тот же ECMAScript:

```
1  contract mortal {
2          address owner;
3
4          function mortal() { owner = msg.sender; }
5
6          function kill() { if (msg.sender == owner)
       selfdestruct(owner); }
7      }
8
9      contract greeter is mortal {
10         string greeting;
11
12         function greeter(string _greeting) public {
13             greeting = _greeting;
14         }
15
16         function greet() constant returns (string) {
17             return greeting;
18         }
19     }
```

Листинг 2 — Пример контракта на языке Solidity

Solidity не имеет классов, вместо них реализованы контракты.

2 Уязвимости смарт-контрактов

DASP (Decentralized Application Security Project) Top 10 спецификация уязвимостей децентрализованных приложений[8][9] .

2.1 Рекурсивный вызов

Рекурсивный вызов (Reentrancy), вероятно, самая известная уязвимость Ethereum, удивила всех, когда была обнаружена впервые. Впервые она была обнаружена во время многомиллионного ограбления, которое привело к хард-форку Ethereum. Рекурсивный вызов происходит, когда вызовам внешнего контракта разрешено делать новые вызовы вызывающего контракта до завершения первоначального выполнения. Для функции это означает, что состояние контракта может измениться в середине его выполнения в результате вызова ненадежного контракта или использования функции низкого уровня с внешним адресом.

Убыток : оценивается в 3,5 миллиона ETH (на тот момент 50 миллионов долларов США). Уязвимый контракт совершает вызов к другому контракту, при этом внешний контракт может сделать ответный вызов функций уязвимого контракта внутри начального вызова.

Рассмотрим пример уязвимого контракта:

```
1  contract Purse {
2          mapping (address => uint) private balances;
3
4          function depositFunds() public payable {
5              balances[msg.sender] += msg.value;
6          }
7
8          function withdraw(uint _amount) {
9              require(balances[msg.sender] >= _amount);
10             msg.sender.call.value(_amount)();
11             balances[msg.sender] -= _amount;
12         }
13     }
```

Листинг 3 — Пример уязвимого контракта Reentrency

Пример атакующего контракта:

```
1  import "purse.sol"
2      contract Attacker {
3          Purse public vuln;
4
5          function withdrawFromVuln() {
6              vuln.withdraw(100);
```

```
7           }
8           function () payable {
9               vuln.withdraw(100);
10          }
11      }
```

Листинг 4 — Пример атакующего контракта Reentrency

1. Смарт-контракт отслеживает баланс ряда внешних адресов и позволяет пользователям получать средства с его общественной withdraw()-функцией.

2. Смарт-контракт злоумышленника использует withdraw()-функцию, чтобы получить весь баланс.

3. Контракт жертвы выполняет call.value(amount)() функцию низкого уровня, чтобы отправить эфир к вредоносному договору перед обновлением баланса договора злоумышленника.

4. Договор злоумышленника имеет payable fallback()-функцию, которая принимает денежные средства, а затем вызывается снова в withdraw()-функции контракта жертвы.

5. Это второе выполнение вызывает перевод средств: помните, что баланс вредоносного контракта все еще не обновлялся после первого вывода. В результате контракт злоумышленника успешно снимает весь баланс во второй раз.

Контракт Attacker внутри функции withdrawFromVuln() вызывает функцию withdrawSomeMoney() контракта Vuln. Но при отправке токенов функцией msg.sender.call.value() вызывается fallback-функция контракта Attacker. Это функция без названия, которая в данном случае используется для получения контрактом ether, поэтому она отмечена модификатором payable. Внутри нее контракт Attacker снова вызывает функцию withdrawSomeMoney(), причем важно заметить, что с баланса аккаунта в кошельке ether еще не списался, значит, проверка достаточности баланса успешна, и мы снова попадаем в fallback-функцию. Так происходит, пока на контракте Vuln совсем не останется средств.

2.2 Управление доступом

Проблемы контроля доступа характерны для всех программ, а не только для смарт-контрактов. Фактически, это номер 5 в топ-10 OWASP. Обычно доступ к функциям контракта осуществляется через его публичные или внешние функции. В то время как небезопасные настройки видимости предоставляют злоумышленникам простые способы доступа к личным значениям или логике контракта, обходы контроля доступа иногда более тонкие. Эти уязвимости могут возникнуть, когда контракты используют устаревшие tx.origin, содержащие сложную логику авторизации с длинным require, для проверки

тех, кто вызывает смарт-контракт и необдуманное использование delegatecall в прокси - библиотеках или прокси - контрактах .

Убыток : оценивается в 150000 ETH (30 миллионов долларов США на тот момент). Пример уязвимого контракта:

```
contract Wallet {
        address public owner;

        constructor (address _owner) {
            owner = _owner;
        }

        function () public payable {}

        function withdrawAll(address _luckyAddress) public {
            require(tx.origin == owner); <-- Vulnerability
            _luckyAddress.transfer(this.balance);
        }
    }
```

Листинг 5 — Пример уязвимого контракта "Управление доступом"

Пример атакующего контракта:

```
import "wallet.sol";
    contract Attacker {
        Wallet poorWallet;
        address attacker;

        constructor (Wallet _poorWallet, address
    _attackerAddress) {
            poorWallet = _poorWallet;
            attacker = _attackerAddress;
        }

        function () payable {
            poorWallet.withdrawAll(attacker);
        }
    }
```

Листинг 6 — Пример атакующего контракта "Управление доступом"

1. Смарт контракт задает адрес, который инициализирует его в качестве владельца контракта. Это обычная схема предоставления повышенных привилегий, таких как возможность вывода средств по контракту.

2. К сожалению, функцию инициализации может быть вызвана кто угодно - даже после того, как она уже была вызвана. Это позволяет любому стать владельцем контракта и забрать ег средства.

В кошельке Parity с несколькими подписями эта функция инициализации была отделена от самих кошельков и определена в «библиотечном» контракте. Ожидалось, что пользователи инициализируют свой собственный кошелек, вызывая функцию библиотеки через файл delegateCall. К сожалению, как и в нашем примере, функция не проверяла, был ли кошелек уже инициализирован. Хуже того, поскольку библиотека была смарт-контрактом, любой мог инициализировать саму библиотеку и потребовать ее уничтожения.

2.3 Арифметические особенности

Целочисленные переполнения и потери значимости не являются новым классом уязвимостей, как и в других языках, возникают из-за ограниченного размера памяти, выделенного на переменную, но они особенно опасны в смарт-контрактах, где преобладают беззнаковые целые числа, а большинство разработчиков привыкли к простым int типам (которые часто являются просто целыми числами со знаком). В случае переполнения многие кажущиеся безобидными кодовые пути становятся векторами кражи или отказа в обслуживании. Максимальное значение для переменной типа uint (uint256) равно $2^{256} - 1$, минимальное — 0. Выйти за эти границы не получится: значение либо обнулится (при переполнении вверх), либо станет максимальным (при переполнении вниз). Пример :

1. Смарт-контракт с withdraw()-функцией позволяет получить эфир пожертвованный контракту, до тех пор, пока ваш баланс остается положительным после операции.

2. Злоумышленник пытается вывести больше, чем его или ее текущий баланс.

3. Результат проверки withdraw()-функции всегда положительный, что позволяет злоумышленнику снять больше, чем разрешено. В результате баланс истощается и становится на порядок больше, чем должен быть.

Примеры кода:

1. Самый простой пример - это функция, которая не проверяет целочисленное переполнение, что позволяет снимать бесконечное количество токенов:

```
1 function withdraw(uint _amount) {
2         require(balances[msg.sender] - _amount > 0);
3         msg.sender.transfer(_amount);
4         balances[msg.sender] -= _amount;
5 }
```

Листинг 7 — Пример уязвимого контракта "Арифметические особенности"

2. Второй пример (замеченный во время конкурса кодирования Underhanded Solidity) - это ошибка с точностью до единицы, чему способствует тот факт, что длина массива представлена целым числом без знака:

```
function popArrayOfThings() {
        require(arrayOfThings.length >= 0);
        arrayOfThings.length--;
    }
```

Листинг 8 — Пример уязвимого контракта "Арифметические особенности"

3. Третий пример представляет собой вариант первого примера, где результатом арифметической операции двух целых чисел без знака является целое число без знака:

```
function votes(uint postId, uint upvote, uint downvotes) {
        if (upvote - downvote < 0) {
            deletePost(postId)
        }
    }
```

Листинг 9 — Пример уязвимого контракта "Арифметические особенности"

4. В четвертом примере используется var как ключевое слово, поддержка которого скоро будет прекращена. Поскольку var изменится на наименьший тип, необходимый для хранения присвоенного значения, он станет uint8 для хранения значения 0. Если цикл предназначен для итерации более 255 раз, он никогда не достигнет этого числа и остановится, когда выполнение выйдет за пределы выделенного газа:

```
for (var i = 0; i < somethingLarge; i ++) {
        // ...
    }
```

Листинг 10 — Пример уязвимого контракта "Арифметические особенности"

2.4 Отсутствие проверки возвращаемых значений низкоуровневых вызовов

В Solidity существуют функции низкого уровня call(), callcode(), delegatecall(), transfer() и send(). Их поведение при учете ошибок сильно отличается от других функций Solidity, поскольку они не будут распространяться (или всплывать) и не приводят к полному откату текущего выполнения. Вместо этого они вернут установленное логическое значение false, и код продолжит выполнение. Это может удивить разработчиков и, если возвращаемое значение таких низкоуровневых вызовов не проверяется, может привести к нежелательным результатам.

1. send()

Отправляет заданное количество монет на адрес, при возникновении ошибки возвращается false.

2. call()

   Возвращает булево значение, показывающее, завершился ли вызов функции успешно (true) или брошено исключение EVM (false).

3. transfer()

   Отправляет заданное количество монет на адрес, откатит транзакцию.

Пример: Следующий код является примером того, что может пойти не так, если забыть проверить возвращаемое значение send(). Если вызов используется для отправки эфира в смарт-контракт, который их не принимает (например, потому что он не имеет оплачиваемой резервной функции), EVM заменит свое возвращаемое значение на false. Поскольку в нашем примере возвращаемое значение не проверяется, изменения функции в состоянии контракта не будут отменены, и etherLeft переменная в конечном итоге будет отслеживать неверное значение:

```
1 function withdraw(uint256 _amount) public {
2     require(balances[msg.sender] >= _amount);
3     balances[msg.sender] -= _amount;
4     etherLeft -= _amount;
5     msg.sender.send(_amount);
6 }
```

Листинг 11 — Пример уязвимого контракта "Отсутствие проверки возвращаемых значений низкоуровневых вызовов"

2.5 Отказ в обслуживании

Отказ в обслуживании смертоносен в мире Ethereum: в то время как другие типы приложений могут в конечном итоге восстановиться, смарт-контракты могут быть отключены навсегда в результате одной из этих атак. Многие способы приводят к отказу в обслуживании, в том числе злонамеренное поведение при получении транзакции, искусственное увеличение количества газа, необходимого для вычисления функции, злоупотребление средствами контроля для доступа к частным компонентам смарт-контрактов, использование путаницы и халатности и т.д. Класс атак включает в себя множество различных вариантов и, вероятно, получит значительное развитие в ближайшие годы.

Убыток : оценивается в 514 874 ETH (около 300 миллионов долларов США на тот момент). Пример:

1. Договор аукцион позволяет пользователям делать ставки на различные активы.

2. Чтобы сделать ставку, пользователь должен вызвать bid(uint object) функцию с желаемым количеством эфира. Аукционный контракт будет хранить эфир на условном депонировании до тех пор, пока владелец объекта не примет ставку или первоначальный участник торгов не отменит ее. Это означает, что в контракте об аукционе должна содержаться полная стоимость любой не разрешенной заявки на балансе.

3. Договор аукциона также содержит withdraw(uint amount) функцию, которая позволяет администраторам получать средства из договора. Поскольку функция отправляет на amount на жестко заданный адрес, разработчики решили сделать функцию общедоступной.

4. Злоумышленник видит потенциальную атаку и вызывает функцию, направляя все договора средства к его администраторам. Это разрушает обещание условного депонирования и блокирует все ожидающие предложения.

5. Хотя администраторы могут вернуть депонированные деньги в контракт, злоумышленник может продолжить атаку, просто снова сняв средства.

В следующем примере (вдохновленном King of the Ether) функция игрового контракта позволяет вам стать президентом, если вы публично подкупите предыдущего. К сожалению, если предыдущий президент является смарт-контрактом и вызывает возврат платежа, передача власти не удастся, и вредоносный смарт-контракт останется президентом навсегда. Своего рода как диктатура:

```
1 function becomePresident() payable {
2         require(msg.value >= price);
3         president.transfer(price);
4         president = msg.sender;
5         price = price * 2;
6 }
```

Листинг 12 — Пример уязвимого контракта "Отказ в обслуживании"

Во втором примере вызывающий может решить, кого вознаградит следующий вызов функции. Из-за дорогостоящих инструкций в цикле for злоумышленник может ввести число, слишком большое для повторения (из-за ограничений газового блока в Ethereum), что эффективно блокирует работу функции.

```
1 function selectNextWinners(uint256 _largestWinner) {
2     for(uint256 i = 0; i < largestWinner, i++) {
3         // heavy code
4     }
5     largestWinner = _largestWinner;
6 }
```

Листинг 13 — Пример уязвимого контракта "Отказ в обслуживании"

2.6 Проблема псевдослучайных чисел

Уязвимые алгоритмы генерации случайных чисел. Блокчейн Ethereum основан на детерминированном алгоритме, поэтому получение случайных чисел — задача трудная. В Ethereum сложно исправить случайность. Хотя Solidity предлагает функции и переменные, которые могут получить доступ к явно трудно предсказуемым значениям, они, как правило, либо более общедоступны, чем кажется, либо подвержены влиянию майнеров. Поскольку эти источники случайности в некоторой степени предсказуемы, злоумышленники обычно могут воспроизвести их и атаковать функцию, полагаясь на ее предсказуемость.

Убыток : более 400 ETH Пример :

1. Умный контракт использует номер блока в качестве источника случайности для игры.

2. Злоумышленник создает вредоносный контракт, который проверяет, является ли текущий номер блока победителем. Если это так, он вызывает первый смарт-контракт , чтобы выиграть; поскольку вызов будет частью одной и той же транзакции, номер блока останется одинаковым в обоих контрактах.

3. Злоумышленнику достаточно лишь вызвать ее злонамеренный контракт, пока он не выиграет.

Пример кода :

1. В первом примере private seed используется в сочетании с iteration числом и keccak256 хэш-функцией, чтобы определить, выиграет ли вызывающий. Несмотря на то, что seed является private, он должен быть установлен через транзакцию в какой-то момент времени и, таким образом, виден в цепочке блоков.

```
1  uint256 private seed;
2
3  function play() public payable {
4      require(msg.value >= 1 ether);
5      iteration++;
6      uint randomNumber = uint(keccak256(seed + iteration));
7      if (randomNumber % 2 == 0) {
8          msg.sender.transfer(this.balance);
9      }
10 }
```

Листинг 14 — Пример уязвимого контракта "Плохая псевдослучайность"

2. В этом примере block.blockhash используется для генерации случайного числа. Этот хэш неизвестен, если blockNumbe установлено текущее значение block.number(по очевидным причинам) и, следовательно, установлено значение 0. В случае, если ранее

было blockNumber установлено более 256 блоков, он всегда будет равен нулю. Наконец, если он установлен на предыдущий номер блока, который не слишком старый, другой смарт-контракт может получить доступ к тому же номеру и вызвать игровой контракт как часть той же транзакции.

```
1 function play() public payable {
2     require(msg.value >= 1 ether);
3     if (block.blockhash(blockNumber) % 2 == 0) {
4         msg.sender.transfer(this.balance);
5     }
6 }
```

Листинг 15 — Пример уязвимого контракта "Плохая псевдослучайность"

2.7 Опережение

В Ethereum все транзакции сначала добавляются в пул, который виден всем участникам сети, а далее майнеры добавляют их в блок в произвольном порядке. Поэтому результат выполнения транзакции не должен зависеть от ее положения в блоке. Поскольку майнеры всегда получают вознаграждение в виде платы за газ за запуск кода от имени внешних адресов (EOA), пользователи могут указать более высокую плату, чтобы их транзакции выполнялись быстрее. Поскольку блокчейн Ethereum является общедоступным, каждый может видеть содержимое ожидающих транзакций других. Это означает, что если данный пользователь раскрывает решение головоломки или другой ценный секрет, злоумышленник может украсть решение и скопировать свою транзакцию с более высокой комиссией, чтобы упредить исходное решение. Если разработчики смарт-контрактов не будут осторожны, эта ситуация может привести к практическим и разрушительным атакам с опережением. Пример :

1. Умный контракт публикует ряд RSA (N = prime1 x prime2).
2. Вызов его submitSolution() публичной функции с правом prime1 и prime2 вознаграждает вызывающего.
3. Алиса успешно факторизует номер RSA и отправляет решение.
4. Кто-то в сети видит транзакцию Алисы (содержащую решение), ожидающую майнинга, и отправляет ее с более высокой ценой на газ.
5. Вторую транзакцию сначала забирают майнеры из-за более высокой уплаченной комиссии.
6. Злоумышленник получает приз.

2.8 Зависимость от времени

От блокировки продажи токенов до разблокировки средств в определенное время для игры, контрактам иногда необходимо полагаться на текущее время. В Solidity переменная now соответствует глобальной переменной block.timestamp, которую задает майнер, поэтому ее нельзя использовать в качестве источника энтропии. Поскольку у майнера транзакции есть свобода действий в сообщении времени, когда произошел майнинг, хорошие умные контракты не будут сильно полагаться на объявленное время. Обратите внимание, что block.timestamp это также иногда (неправильно) используется при генерации случайных чисел. Пример :

1. Сегодня в полночь игра производит выплаты самому первому игроку.
2. Майнер злоумышленник включает свою попытку выиграть игру и устанавливает отметку времени на полночь.
3. Незадолго до полуночи майнер добывает блок. Реальное текущее время «достаточно близко» к полуночи (текущая установленная метка времени для блока), другие узлы в сети решают принять блок.

Пример кода :

Следующая функция принимает только вызовы, поступившие после определенной даты. Поскольку майнеры могут влиять на временную метку своего блока (в определенной степени), они могут попытаться добыть блок, содержащий их транзакцию, с временной меткой блока, установленной в будущем. Если он достаточно близок, он будет принят в сети, и транзакция предоставит майнеру эфир до того, как любой другой игрок сможет попытаться выиграть игру:

```
1 function play() public {
2     require(now > 1521763200 && neverPlayed == true);
3     neverPlayed = false;
4     msg.sender.transfer(1500 ether);
5 }
```

Листинг 16 — Пример уязвимого контракта "Зависимость от времени"

2.9 Манипулирование длиной адреса

Адрес в Ethereum состоит из 20 байт. Для осуществления атаки злоумышленнику нужно сгенерировать адрес, который заканчивается на нулевой байт. Если указать сгенерированный адрес без последнего байта, то недостающий байт адреса будет взят из следующего аргумента - *amount*, а в конец EVM допишет нулевой байт. Для вызова функций контрактов используется ABI (Application Binary Interafce) — бинарный интерфейс приложения, согласно которому вызов функции представляется в виде последовательности байтов. Первые четыре байта — это начало хеша keccak256 от подписи функции (ее

название и типы входных данных), далее идут значения входных параметров. Но если указать адрес без последнего байта, то недостающий байт адреса будет взят из следующего аргумента (amount), а в конец EVM допишет нулевой байт. Получается, вместо 1 ether жертва отправит 0x100=256 ether. Хотя эту уязвимость еще предстоит использовать в реальных условиях, она является хорошей демонстрацией проблем, возникающих при взаимодействии между клиентами и блокчейн Ethereum. Существуют и другие проблемы вне сети: важной из них является глубокое доверие экосистемы Ethereum к определенным интерфейсам Javascript, плагинам браузера и общедоступным узлам. Печально известный эксплойт вне сети был использован взлом ICO Coindash, который изменил адрес Ethereum компании на своей веб-странице, чтобы обманом заставить участников отправлять эфиры на адрес злоумышленника.

Пример :

1. API обмена имеет торговую функцию, которая принимает адрес получателя и сумму.
2. Затем API взаимодействует с transfer(address _to, uint256 _amount) функцией смарт-контракта с дополненными аргументами: он добавляет к адресу (ожидаемой длины 20 байтов) 12 нулевых байтов, чтобы сделать его длиной 32 байта.
3. Боб (0x3bdde1e9fbaef2579dd63e2abbf0be445ab93f00) просит Алису передать ему 20 токенов. Он злонамеренно дает ей свой адрес, усеченный, чтобы удалить завершающие нули.
4. Алиса использует API обмена с более коротким 19-байтовым адресом Боб (0x3bdde1e9fbaef
5. API заполняет адрес 12 нулевыми байтами, делая его 31 байтом вместо 32 байтов. Фактически похищение одного байта из следующего _amount аргумента.
6. В конце концов, EVM, выполняющая код смарт-контракта, заметит, что данные не заполнены должным образом, и добавит недостающий байт в конец _amount аргумента. Эффективная передача в 256 раз больше токенов, чем предполагалось.

2.10 Unknown unknowns

Уязвимости не входящие в этот список. Ethereum все еще находится в зачаточном состоянии. Основной язык, используемый для разработки смарт-контрактов, Solidity, еще не достиг стабильной версии, а инструменты экосистемы все еще являются экспериментальными. Некоторые из наиболее разрушительных уязвимостей смарт-контрактов удивили всех, и нет оснований полагать, что не будет других, столь же неожиданных или столь же разрушительных. Пока инвесторы решают вкладывать большие суммы денег в сложный, но плохо проверяемый код, мы будем продолжать видеть новые уязвимости, ведущие к ужасным последствиям. Методы формальной проверки смарт-контрактов еще не разработаны, но они кажутся многообещающими как способы преодоления нынешнего

шаткого статус-кво. Поскольку новые классы уязвимостей продолжают обнаруживаться, разработчикам необходимо оставаться настороже, и нужно будет разработать новые инструменты, чтобы найти их до того, как это сделают злоумышленники.

3 Формальная верификация программ

Верификацией программ, или, шире, компьютерных систем, называется деятельность, направленная на выяснение их правильности или, наоборот, ошибочности. Все методы верификации можно поделить на 3 группы [10]:

- Формальные методы - используют математический строгий анализ модели программы и модели требований;
- Методы тестирования - осуществляют проверку реального поведения программы на некотором наборе сценариев:
- Экспертиза - выполняется людьми на основе их опыта и знаний;

В рамках данной работы будет рассмотрена только формальная верификация.

Формальная верификация в своей основе имеет математическое моделирование программ и их требований. Идея следующая:

1. создается модель - некоторое абстрагированное описание объекта исследования;
2. модель исследуется при помощи математических методов;
3. результаты переносятся на оригинальный объект исследования.

Существует несколько подходов к формальной верификации:

- Проверка моделей
- Дедуктивный анализ
- Символьное выполнение
- Проверка эквивалентности и др.

Наиболее часто встречающимися подходами являются символьное исполнение и проверка моделей.

3.1 Символьное исполнение

Основная идея символического исполнения заключается в том, что в любой момент выполнения можно выразить значения всех переменных как функции начальных значений. В символьном исполнении данные заменяются символическими значениями с набором выражений, по одному выражению на выходную переменную. Общий подход к символьному выполнению состоит в том, чтобы выполнить анализ программы, что приводит к созданию потокового графа. Блок-схема идентифицирует точки принятия решений и назначения, связанные с каждым потоком. При обходе потокового графа из точки входа создается список операторов присваивания и предикатов ветвления. Однако для программ с бесконечными деревьями выполнения символическое тестирование не может быть исчерпывающим и не может быть установлено абсолютное доказательство правильности[11].

3.2 Проверка на модели

При проверке на модели требования к исследуемому объекту представляются в виде логических формул определенного вида, а сами программы - структурами, интерпретирующими эти формулы. При представлении требований часто используют темпоральные логики - логики, позволяющие задавать связь между событиями во времени. Для представления структур часто используют структуры Крипке или родственные формализмы. Проверка моделей не так далека от разработки, как может показаться. Для программного описания модели чаще всего используется язык Promela, а для верификации корректности модели используется SPIN (Simple Promela Interpreter).

3.3 Формальная верификация смарт-контрактов

При исследовании темы формальной верификации смарт-контрактов было найдено ограниченное количество статей и ни одного рабочего инструментального решения. Наиболее часто используемым подходом является трансляция подмножества языков Solidity в другой язык и дальнейшая верификация программы уже на другом языке. Данный подход является точным, однако довольно сложным в реализации. Например, в статье [12], рассматривается трансляция языка Sol, который является упрощенной версией Solidity (не содержит массивов, циклов, некоторых типов) в язык F*, который является функциональным языком программирования. Поверхностно описывается алгоритм трансляции и так и не приводится практический пример реализации. Другим примером можно считать работу разработчиков компании Microsoft, они также используют трансляцию языка Sol, но в промежуточный верифицируемый язык Boogie. Однако этот проект так и не вышел, а команда прекратила над ним работать больше года назад [13].

4 Анализ уязвимостей смарт-контрактов на базе проверки на модели

В результате анализа существующих решений по анализу уязвимостей смарт-контрактов за основу предлагаемого решения выбран метод проверка на модели. Для построения модели предлагается следующая схема:

- По Solidity коду строится его представление в виде графа;
- На основе графового представления строится код на языке Promela;
- Код на языке Promela верифицируется при помощи программы SPIN;

При трансляции графа в код на языке Promela сложность составляет отсутствие некоторых типов данных, которые есть в языке Solidity, например: address и string.

4.1 Построение графовой модели смарт-контракта

Построение графа по коду программы не является сложной задачей. Однако для построения графа по коду Solidity стоит сделать некоторые ремарки. В качестве основы будет использован граф потока управления. Граф потока управления используется для обнаружения ошибок в программе, что прекрасно подходит для цели данной работы. Ключевое слово Contract является аналогом Class в других ООП языках, например C++. В одном контракте может быть множество структур типа Contract. Каждая из них является частью общего контракта и они связаны между собой. Будем называть эти структуры субконтрактами.

Предикатом будем называть условное выражение, либо модификаторы (кроме payable).

При построении графа, в качестве входа выступает весь контракт. Далее, при нахождении субконтракта строится новый граф, где входом уже является субконтракт. Если в субконтракте встречается функция, то строится еще один граф, где входом является найденная функция. Исключением являются некоторые конструкции языка Solidity.

1. Конструкторы - обрабатываются отдельно. Выписываются все переменные из конструктора и становятся узлами;
2. Модификаторы - ключевые слова, которые обозначают определенные свойства. Например, require означает, что условие является необходимым;

Далее представлен пример построения графовой модели по смарт-контракту.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >= 0.7.0;
3
4 contract owned {
```

```
5
6       address public owner;
7
8       constructor() {
9           owner = msg.sender;
10      }
11
12      modifier onlyOwner {
13          require(owner == msg.sender);
14          _;
15      }
16
17      function changeOwner(address _owner) onlyOwner public {
18          owner = _owner;
19      }
20  }
21
22  contract Crowdsale is owned {
23
24      uint sum;
25      uint256 public totalSupply;
26      mapping (address => uint256) public balanceOf;
27
28      event Transfer(address indexed from, address indexed to,
        uint256 value);
29
30      constructor() payable owned() {
31          totalSupply = 21000;
32          sum = 20000;
33          balanceOf[owner] = totalSupply - sum;
34          Transfer(this, owner, balanceOf[owner]);
35      }
36
37      fallback () external payable {
38          uint256 tokens = 5 * msg.value / 10;
39          if (tokens > sum) {
40              tokens = sum;
41              uint valueWei = tokens * 10 / 5;
42              msg.sender.transfer(msg.value - valueWei);
43          }
44          require(tokens > 0);
```

```
45          balanceOf[msg.reciever] += tokens;
46          balanceOf[this] -= tokens;
47          Transfer(this, msg.sender, tokens);
48      }
49  }
```

Листинг 17 — Пример контракта

1. Разделим контракт на субконтракты: owned() и Crowdsale
2. Рассмотрим первый субконтракт - owned():
   (a) Находим переменную: address public owner;
   (b) Находим модификаторы в первом субконтракте: onlyOwner() -> owner = msg.sender
   (c) Находим конструктор: owner = msg.sender;
   (d) Функций с модификатором payable нет, других функций, которые стоит рассматривать тоже
3. Переходим ко второму субконтракту Crowdsale
4. Находим переменные:
   - uint sum;
   - uint256 public totalSupply;
   - mapping (address => uint256) public balanceOf;
5. Модификаторов нет
6. Рассмотрим конструктор:
   - totalSupply = 21000;
   - sum = 20000;
   - balanceOf[owner] = totalSupply - sum;
7. Находим fallback функцию:
   - uint256 tokens = 5 * msg.value / 10; - безусловный переход
   - if (tokens > sum) - условие без модификатора
     - tokens = sum;
     - uint valueWei = tokens * 10 / 5;
   - require(tokens > 0) - условие с модификатором

- balanceOf[msg.reciever] += tokens;
- balanceOf[this] -= tokens;

Для облегчения обработки смарт-контракта, его графовая модель представляется в формате JSON. JSON (JavaScript Object Notation) - простой формат обмена данными, удобный для чтения и написания как человеком, так и компьютером. Рассмотрим представление в формате JSON на предыдущем примере:

```
{
    [
        {
            "state": "value",
            "transitCondition":
            [
                {
                    "toState": "value = value + 100",
                    "condition": ""
                },
                {
                    "toState": "value = value - valueWei",
                    "condition": "tokens > sum"
                }
            ]
        },
        {
            "state": "sum",
            "transitCondition":
            [
                {
                    "toState": "sum = 2000",
                    "condition": ""
                },
                {
                    "toState": "sum = sum + tokens",
                    "condition": "tokens > 0"
                }
            ]
        },
        {
            "state": "balanceOfOwner",
            "transitCondition":
            [
                {
```

```
36                      "toState": "balanceOfOwner =
    balanceOfOwner + 1000",
37                      "condition": ""
38                  }
39              ]
40          },
41          {
42              "state": "balanceOfReciever",
43              "transitCondition":
44              [
45                  {
46                      "toState": "balanceOfOwner =
    balanceOfOwner + 1000",
47                      "condition": ""
48                  }
49              ]
50          },
51          {
52              "state": "totalySupply",
53              "transitCondition":
54              [
55                  {
56                      "toState": "totalySupply = 21000",
57                      "condition": ""
58                  }
59              ]
60          },
61          {
62              "state": "tokens",
63              "transitCondition":
64              [
65                  {
66                      "toState": "tokens = 5 * value / 10",
67                      "condition": "sum > 0"
68                  },
69                  {
70                      "toState": "tokens = sum",
71                      "condition": "tokens > sum"
72                  }
73              ]
74          },
```

```
75          {
76              "state": "valueWei",
77              "transitCondition":
78              [
79                  {
80                      "toState": "valueWei = valueWei * 10 / 5",
81                      "condition": ""
82                  }
83              ]
84          },
85          {
86              "state": "newBalanceOwner",
87              "transitCondition":
88              [
89                  {
90                      "toState": "newBalanceOwner =
    balanceOfOwner - tokens",
91                      "condition": "tokens > 0"
92                  }
93              ]
94          },
95          {
96              "state": "newBalanceOfReciever",
97              "transitCondition":
98              [
99                  {
100                     "toState": "newBalanceOfReciever =
    balanceOfReciever + tokens",
101                     "condition": "tokens > 0"
102                 }
103             ]
104         },
105         {
106             "state": "transfer = false",
107             "transitCondition":
108             [
109                 {
110                     "toState": "transfer = true",
111                     "condition": "sum > 0"
112                 },
113                 {
```

```
114                     "toState": "transfer = true",
115                     "condition": "tokens > 0"
116                 }
117             ]
118         }
119     ]
120 }
```

Листинг 18 — Пример представления графа в формате JSON

- *state*, рассматриваемое состояние
- *transitCondition*, массив объектов, содержащих условие перехода и следующее состояние
- *toState*, следующее состояние
- *condition*, условие перехода в следующее состояние

4.2 Перевод графовой модели смарт-контракта в описание на языке Promela

После генерации представления в формате JSON, мы можем построить код на языке Promela. Рассмотрим на предыдущем примере Promela-код, который получается:

```
1  int totalSupply, balanceOfOwner, balanceOfReciever,
       newBalanceOfOwner, newBalanceOfReciever, tokens, valueWei,
       value, sum;
2  bool transfer=false;
3
4  proctype crowdsale() {
5      atomic {
6          value = value + 100;
7          sum = 20000;
8          balanceOfOwner = totalSupply + 1000;
9          balanceOfReciever = balanceOfReciever + 1000;
10         totalSupply = 21000;
11     }
12     atomic {
13         if
14         ::(sum > 0) -> {tokens = 5 * value / 10;
15             if
16             ::(tokens > sum) -> {
17                 tokens = sum;
18                 valueWei = valueWei * 10 / 5;
```

```
19                  value = value - valueWei;
20              }
21              ::(tokens > 0) -> {
22                  sum = sum + tokens;
23                  newBalanceOfReciever = balanceOfReciever +
      tokens;
24                  transfer=true;
25              }
26              fi
27          }
28          fi
29      }
30  }
31
32  init {
33      run crowdsale()
34      assert(newBalanceOfReciever > balanceOfReciever)
35  }
36
37  ltl p1 { <> transfer }
```

Листинг 19 — Пример представления модели контракта на языке Promela

В данном примере мы сгруппировали то, что задается в разных структурах внутри atomic. Это значит, что следующий код не выполнится, пока не будут выполнены все условия в atomic, нарушение хотя бы одного из них ведет к нарушению всего блока atomic. Так как язык Promela не содержит сложных типов данных, например строк и больших чисел, мы заменим их на другие.

Таблица 1 – Трансляция типов Solidity в Promela

Транслируемый тип	Результат
int/uint	int
address	int
bool	bool
bytes	byte
string	byte[]
enum	mtype
function	proctype
contract	proctype

4.3 Верификация с использованием SPIN

Для верификации с использованием SPIN, необходимо задать требования, которые проверяются. Для записи требований к модели использует язык темпоральной логи-

ки LTL. В примере используется следующая LTL-формула: *ltl p1 { <> transfer }* Язык Promela состоит из множества атомарных высказываний, логических операций и темпоральных операторов. Верификатор SPIN автоматически строит контрпример как путь в модели, поданной ему на вход. Так как модель на языке Promela эквивалентна исходному графу, обратное преобразование не требуется. Рассмотрим на предыдущем примере верификацию модели с помощью SPIN.

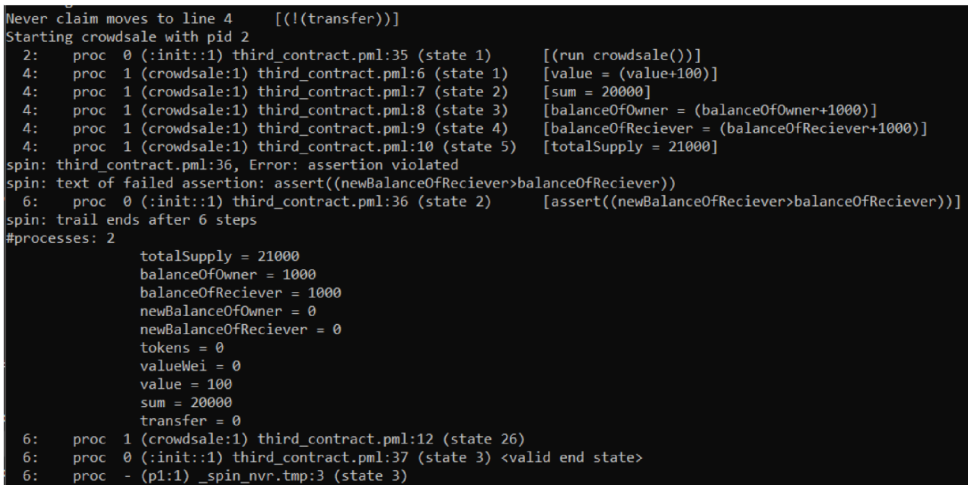


Рисунок 1 — Результат работы верификатора SPIN

Как видно, SPIN показывает, что при определенных условиях, возможна отправка средств. В данном случае была найдена неизвестная уязвимость.

ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломной работы получены следующие результаты:

1. Предложен способ выявления уязвимостей смарт-контрактов на базе верификатора SPIN.

2. Разработан инструмент для перевода смарт-контракта на языке Solidity в описание на языке Promela.

3. Предложенный подход верификации смарт-контрактов протестирован на множестве реальных примеров.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Безопасность смарт-контрактов. — Электрон. дан. — URL: https://xakep.ru/2016/06/17/splitdao-attack/Xaker.

2. Электронные закладные отправляются в блокчейн. — Электрон. дан. — URL: https://www.raiffeisen.ru/about/press/releases/71080/Raiffeisen Bank.

3. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts / Mossberg Mark, Manzano Felipe, Hennenfent Eric, Groce Alex. — 2019.

4. *Loi L.* Making Smart Contracts Smarter / L. Loi, C. Duc-Hiep, O. Hrishi // CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. — №. 5479. — Association for Computing Machinery, 2018. — С. 254–269.

5. *Миронов А.М.* Верификация программ методом Model Checking / Миронов А.М. — 2017.

6. *Шошмина И.В.* Введение в язык Promela и систему комплексной верификации Spin / Шошмина И.В. — 2009.

7. Руководство-по-Solidity. — Электрон. дан. — URL: https://docs.soliditylang.org/en/latest/Solidity Documentation.

8. *Адамович А.И.* Уязвимости смарт-контрактов блокчейн-платформы Ethereum / Адамович А.И. — 2019.

9. *Боровик В.С.* К вопросу о безопасности смарт-контрактов / Боровик В.С., Зенин М.М., Гатчин Ю.А. — 2018.

10. *Камкин А.С.* Введение в формальные методы верификации программ: Учебное пособие / Камкин А.С. — 2018.

11. *James C. King.* Symbolic Execution and Program Testing / James C. King. — 2019.

12. *Vipin Deval A. D.* Formal-Verification of Smart-Contract Languages / A. D. Vipin Deval // PLAS '16: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. — Association for Computing Machinery, 2019. — С. 91–96.

13. Formal Specification and Verification of Smart Contracts for Azure Blockchain. / Lahiri, Shuvendu K., Chen, Shuo, Wang, Yuepeng, Dillig, Isil // *CoRR*. — 2018. — №. abs/1812.08829. http://dblp.uni-trier.de/db/journals/corr/corr1812.html#abs-1812-08829.

АНТИПЛАГИАТ
ТВОРИТЕ СОБСТВЕННЫМ УМОМ

Отчет о проверке на заимствования №1

Автор: Аламов Владимир alamovvladimir@gmail.com / ID: 1619042
Проверяющий: Аламов Владимир (alamovvladimir@gmail.com / ID: 1619042)

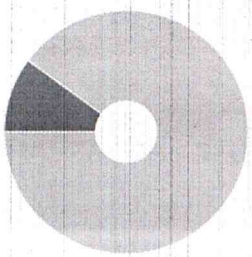
Отчет предоставлен сервисом «Антиплагиат»- http://users.antiplagiat.ru

ИНФОРМАЦИЯ О ДОКУМЕНТЕ

№ документа: 15
Начало загрузки: 25.01.2021 03:37:44
Длительность загрузки: 00:00:01
Имя исходного файла: VKR Alamov.pdf
Название документа: VKR Alamov
Размер текста: 57 кБ
Символов в тексте: 58267
Слов в тексте: 6986
Число предложений: 466

ИНФОРМАЦИЯ ОБ ОТЧЕТЕ

Последний готовый отчет (ред.)
Начало проверки: 25.01.2021 03:37:45
Длительность проверки: 00:00:04
Комментарии: не указано
Модули поиска: Модуль поиска Интернет



ЗАИМСТВОВАНИЯ	САМОЦИТИРОВАНИЯ	ЦИТИРОВАНИЯ	ОРИГИНАЛЬНОСТЬ
9,02%	0%	0%	90,98%

Заимствования — доля всех найденных текстовых пересечений, за исключением тех, которые система отнесла к цитированиям, по отношению к общему объему документа.
Самоцитирования — доля фрагментов текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника, автором или соавтором которого является автор проверяемого документа, по отношению к общему объему документа.
Цитирования — доля текстовых пересечений, которые не являются авторскими, но система посчитала их использование корректным, по отношению к общему объему документа. Сюда относятся оформленные по ГОСТу цитаты; общеупотребительные выражения; фрагменты текста, найденные в источниках из коллекций нормативно-правовой документации.
Текстовое пересечение — фрагмент текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника.
Источник — документ, проиндексированный в системе и содержащийся в модуле поиска, по которому проводится проверка.
Оригинальность — доля фрагментов текста проверяемого документа, не обнаруженных ни в одном источнике, по которым шла проверка, по отношению к общему объему документа.
Заимствования, самоцитирования, цитирования и оригинальность являются отдельными показателями и в сумме дают 100%, что соответствует всему тексту проверяемого документа.
Обращаем Ваше внимание, что система находит текстовые пересечения проверяемого документа с проиндексированными в системе текстовыми источниками. При этом система является вспомогательным инструментом, определение корректности и правомерности заимствований или цитирований, а также авторства текстовых фрагментов проверяемого документа остается в компетенции проверяющего.

№	Доля в отчете	Доля в тексте	Источник	Ссылка	Актуален на	Модуль поиска	Блоков в отчете	Блоков в тексте
[01]	3%	3,04%	Безопасность смарт-контра...	https://xakep.ru	09 Дек 2019	Модуль поиска Интернет	12	12
[02]	0%	3,04%	Безопасность смарт-контра...	https://xakep.ru	12 Авг 2020	Модуль поиска Интернет	0	12
[03]	1,17%	1,83%	Что такое смарт-контракты (...	https://halzen.ru	12 Фев 2020	Модуль поиска Интернет	1	5

Еще источников: 17
Еще заимствований: 4,85%

С результатами ознакомлен.
Руководитель ВКР Трегубов В.И.