Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Верификация контрактов в криптовалюте Ethereum

Магистерская диссертация

Выполнил: Соковикова Светлана
Алексеевна, 17.М04-мм
Научный руководитель:
д.ф.-м.н., проф. Терехов А.Н.
Рецензент: Пантюхин И.С.

Санкт-Петербург

2019

Введение

Блокчейн – технология для поддержания функционирования распределенных полностью реплицированных баз данных. Впервые эта технология была применена в сети криптовалюты Bitcoin. Сеть Bitcoin состоит из аккаунтов, совершающих друг с другом финансовые сделки, называемые транзакциями. Данные обо всех транзакциях специальным образом собираются в блоки, блоки последовательно связаны друг с другом (отсюда термин блокчейн – цепочка блоков). Процесс сбора транзакций в блоки называется майнингом. Цепочка блоков представляет собой базу данных, содержащую информацию обо всех совершенных финансовых сделках. Так как постоянно совершаются новые транзакции, цепочка блоков постоянно обновляется: в конец последовательности добавляются новые блоки, старые при этом не изменяются (что чрезвычайно важно). Каждый аккаунт хранит собственный экземпляр цепочки блоков. Блокчейн-сеть принимает все необходимые меры, чтобы цепочки на всех аккаунтах были одинаковы. Единая централизованная база данных отсутствует, а значит, невозможно фальсифицировать данные в единственном месте так, чтобы ввести этим в заблуждение все аккаунты сети. Кроме того, аккаунты обмениваются между собой данными без участия посредников (peer-to-peer).

В роли аккаунтов в блокчейн-сети могут выступать онлайн-сервисы. Цепочка блоков может хранить данные не только о балансах и переводах денег, но и любые представляющие интерес данные. Эти возможности блокчейн-технологии успешно используются сетью Ethereum. Также как сеть Bitcoin, сеть Ethereum представляет собой множество аккаунтов, обменивающихся между собой информацией в процессе транзакций. Криптовалюта Ethereum предназначена для совершения аккаунтами финансовых сделок. Существуют аккаунты двух видов: внешние и внутренние. Внешние аккаунты не имеют исходного кода, и все решения об их действиях в сети принимаются человеком извне. Внутренние аккаунты иначе называются контрактами или смарт-контрактами. Каждый внутренний аккаунт представляет собой программу, полностью определяющую действия этого аккаунта в сети.

Как любая программа, контракт может быть подвержен уязвимостям. Уязвимости в кодах контрактов привели к нескольким известным атакам злоумышленников в сети Ethereum с крупными финансовыми потерями. Так, ущерб от атаки TheDAO в июне 2016 года составил $74 млн [14], а за 2017 год было украдено $300 млн [15].

История потерь говорит о необходимости тщательной проверки кода контрактов на уязвимости. Проверка исходного кода экспертом может применяться лишь в единичных исключительных случаях, для широкого использования нужны инструменты, выполняющие автоматическую проверку кода.

Формальная верификация кода – строгое математическое доказательство соответствия кода спецификациям (формальным требованиям). В частности,

спецификации могут быть сформулированы как формализованные условия отсутствия в коде определенных уязвимостей. В этом случае формальная верификация становится не чем иным, как строгим доказательством отсутствия в коде уязвимостей.

Один из инструментов для формальной верификации – система Spin, работающая на основе проверки моделей. Система принимает на вход модель, представленную в виде кода на встроенном языке программирования Promela, а также спецификации, на соответствие которым нужно проверить модель. Спецификации должны быть описаны также на языке Promela. Если в роли модели подать системе исходный код контракта, транслированный на язык Promela, а в роли спецификаций – формальные требования, означающие отсутствие уязвимостей, Spin выполнит необходимую проверку контракта на уязвимости.

Цель и постановка задачи

Целью данной работы является создание инструмента для автоматического поиска уязвимостей в кодах контрактов.

Для достижения этой цели были сформулированы следующие задачи.

1. Разработать архитектуру инструмента.
2. Реализовать трансляцию контрактов с внутреннего представления блокчейна Ethereum в модели на языке Promela для последующей верификации системой Spin.
3. Проверить инструмент на тестовых примерах контрактов.

Обзор

Обзор блокчейна Ethereum

Ethereum – платформа для создания онлайн-сервисов на базе технологии блокчейн. Сеть Ethereum была запущена 30 июля 2015 года. В сети используется специальная валюта – ether, обеспечивающая функционирующим онлайн-сервисам возможность совершать финансовые сделки.

Участники сети Ethereum называются аккаунтами. Аккаунт – единица, имеющая возможность совершать операции, изменяющие состояние сети. Существует два вида аккаунтов: внешние и внутренние. Внешний аккаунт менее формально справедливо можно назвать «аккаунт-человек», так как все действия этого аккаунта инициируются человеком извне. Внутренний аккаунт можно воспринимать как «аккаунт-контракт» – аккаунт, действия которого полностью определяются его исходным кодом. Так, контрактами (а также смарт-контрактами или умными контрактами) называются аккаунты, действия которых определяются исходным кодом. Для наглядности виды аккаунтов представлены на рис. 1.

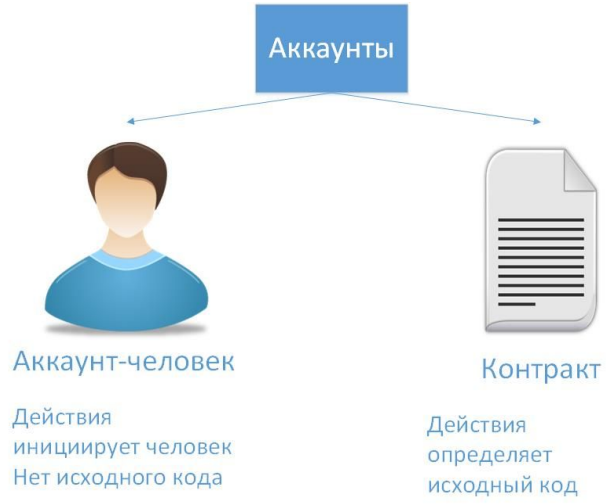

Рис. 1. Виды аккаунтов

Действия любых аккаунтов могут изменять состояние сети. Состояние сети – это совокупность состояний всех аккаунтов. Состояние аккаунта-человека полностью определяется его балансом (в валюте ether). Состояние контракта определяется его балансом и данными в его хранилище. Хранилище – это база данных типа “ключ-значение”, в которой каждый ключ и каждое значение представлены 32-байтовым числом, и таким образом каждый контракт имеет возможность хранить 2^{256} значений по 32 байта. Хранилище перманентно, т.е. данные, записанные в него, не теряются после завершения транзакции.

Состояние сети изменяется посредством транзакций (иначе говоря, транзакции – это действия аккаунтов). Транзакция инициируется некоторым аккаунтом и адресуется другому аккаунту. Если транзакция адресована аккаунту-человеку, единственное, что она может осуществить – перевод денег (ether) от инициатора к получателю. Если же транзакция направлена контракту, то она обязательно вызывает исполнение кода контракта. Что касается перевода денег в этом случае, он может быть направлен от инициатора к адресату, а может вовсе не осуществляться.

Так как состояния сети и переходы из одного состояния в другое были только что определены, сеть Ethereum можно рассматривать как конечный автомат, переходы в котором – это транзакции.

О жизненном цикле транзакции стоит сказать более подробно. После того, как транзакция была отправлена в сеть инициатором, она включается в блок (процесс сбора транзакций в блоки называется mining). После того, как блок собран, транзакции из него исполняются. За исполнение транзакции взимается комиссионная плата, называемая gas в сети Ethereum. Оплачивает комиссионную плату инициатор транзакции.

Действия контракта в сети определяются его исходным кодом. Исходный код контрактов пишется на специальном javascript-подобном языке высокого уровня – Solidity. На листинге 1 представлен код одного небольшого контракта на Solidity.

```
1  contract HoneyPot {
2    mapping (address => uint) public balances;
3    constructor() payable public {
4      put();
5    }
6    function put() payable public {
7      balances[msg.sender] = msg.value;
8    }
9    function get() public {
10     (succeed, ) = msg.sender.call.value(balances[msg.sender])("")
11     if (!succeed) {
12       revert();
13     }
14       balances[msg.sender] = 0;
15   }
16   function() public {
17     revert();
18   }
19 }
```

Листинг 1. Пример кода контракта на Solidity

Как видно, в коде определено несколько функций. Адресованная контракту транзакция – это вызов одной высокоуровневой функции.

Код на языке Solidity переводится в код инструкций Ethereum Virtual Machine (EVM). EVM представляет собой стековую машину, которая оперирует со стеком, памятью и хранилищем. Стек представляет собой обычный стек, размер слова

которого равен 32 байтам. Память устроена в виде обычного массива байтов. О хранилище было сказано ранее. Содержимое стека и памяти теряется после завершения транзакции, тогда как хранилище перманентно. EVM включает в себя много инструкций, подробное обсуждение которых здесь не имеет смысла. Полная документация EVM представлена в Ethereum Yellow Paper [1]. На рис. 2 представлен отрывок кода на уровне инструкций EVM.

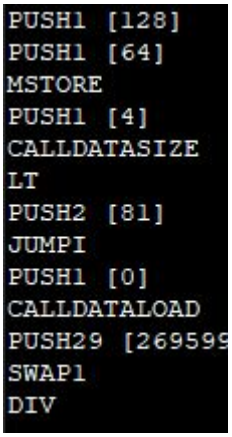

Рис. 2. Отрывок кода на уровне инструкций EVM

Стоит отметить лишь те инструкции EVM, которые служат разметкой низкоуровневого кода. Низкоуровневый код представляет собой единую последовательность инструкций. Инструкции, предназначенные для передачи управления – это JUMP для безусловного перехода и JUMPI для условного перехода. EVM не имеет инструкций цикла – все циклы реализуются с использованием только JUMP и JUMPI. Каждая инструкция имеет свой адрес, по которому происходит переход при JUMP и JUMPI. Переход разрешен только на те адреса, в которых стоит инструкция JUMPDEST, единственное назначение которой – пометка своего адреса как возможного адреса для JUMP или JUMPI (название JUMPDEST означает “jump destination”). Инструкции STOP и RETURN служат для завершения транзакции и сохранения сделанных ею изменений (баланс и хранилище), REVERT – для завершения транзакции с отменой всех сделанных ею изменений. Отдельное место занимает инструкция SUICIDE, после исполнения которой контракт перестает существовать.

Инструкция CALL инициирует новую транзакцию, т.е. вызывает исполнение одной высокоуровневой функции какого-либо стороннего контракта. Чрезвычайно важен тот факт, что в сети Ethereum все вызовы CALL синхронны. Более того, контракты не поддерживают параллельное исполнение нескольких своих функций.

В связи с тем, что транзакция – это вызов одной высокоуровневой функции, а низкоуровневый код не разделен на функции, возникает вопрос: как происходит выбор функции при исполнении низкоуровневого кода? Ответ прост: в начале низкоуровневого кода (до первой, реже – до второй инструкции JUMPDEST) происходит диспетчеризация. Запрос на вызов функции содержит в себе хэш нужной

функции. Этот хэш по очереди сравнивается с хэшем каждой из высокоуровневых функций контракта, и в случае равенства происходит передача управления на адрес в низкоуровневом коде, соответствующий началу вызванной функции.

Типы уязвимостей контрактов

Созданный инструмент позволяет находить уязвимости трех типов: реентерабельность (reentrancy), недостаточная защищенность деструктора и недостаточная защищенность переводящих средства функций. Однако, помимо этих трех типов уязвимостей известны несколько других, поиск которых не реализован в созданном инструменте, но реализован в некоторых из аналогов. Те типы уязвимостей, поиск которых реализован, будут рассмотрены подробно, остальные лишь вкратце. Подробное описание десяти наиболее часто встречающихся типов уязвимостей представлено в [2].

Реентерабельность

Название этой уязвимости может быть переведено как «перевызываемость». Для того чтобы объяснить, в чем она заключается, рассмотрим коды двух контрактов – Victim и Attacker – на листинге 2 [3].

Начало атаки – вызов функции attack() контракта Attacker. В теле этой функции в строке 25 будет вызвана анонимная функция контракта Victim, в результате чего userbalances[msg.sender] получит некоторое ненулевое значение. В строке 26 будет вызвана функция withdraw() контракта Victim.

В теле функции withdraw() на 5 строке будет проверено наличие средств на счету. Условие окажется истиной, и выполнение функции продолжится. Далее в 6 строке будет вызвана анонимная функция контракта Attacker. При этом userbalances[msg.sender] еще не приравнивалась к 0, ее значение по-прежнему положительно, и ровно такая сумма средств будет отправлена контракту Attacker. В свою очередь анонимная функция контракта Attacker снова вызовет функцию withdraw(). Баланс Attacker, записанный в userbalances, так и не изменился, и снова будет отправлен контракту Attacker. Из функции withdraw() снова будет вызвана анонимная функция, и так процесс зациклится, пока у контракта Victim не закончатся средства. Все средства Victim в результате этого цикла будут переведены контракту Attacker.

```
1. contract Victim {
2.     mapping(address => uint) userbalances;
3.
4.     function withdraw() {
5.         if (userbalances[msg.sender] > 0) {
6.             if (msg.sender.call.value(userbalances[msg.sender])()) {
7.                 userbalances[msg.sender] = 0;
8.             }
```

```
9.          }
10.     }
11.
12.     function() {
13.         userbalances[msg.sender] += msg.value;
14.     }
15. }
16.
17. contract Attacker {
18.     Victim v;
19.
20.     function Attacker(address dest) {
21.         v = Victim(dest);
22.     }
23.
24.     function attack() {
25.         v.call.value(msg.value)();
26.         v.withdraw();
27.     }
28.
29.     function() {
30.         if (msg.gas > 100000) {
31.             v.withdraw();
32.         }
33.     }
34. }
```

Листинг 2. Реентерабельность.

Именно такая уязвимость была использована при взломе TheDAO в июне 2016 года. Из-за возможности подобных атак инструменты поиска уязвимостей выдают сообщения о вызове call на адрес, отправивший транзакцию: возможно, действия анонимной функции расположенного по этому адресу контракта будут иметь для нашего контракта катастрофические последствия.

Стоит отметить, что если бы функция withdraw() имела вид, приведенный на листинге 3, эта атака не несла бы опасности для Victim, так как условие наличия средств на счету (строка 2 на рис. 5) оказалось бы ложно уже при втором вызове функции withdraw().

```
1. function withdraw() {
2.     if (userbalances[msg.sender] > 0) {
3.         uint x = userbalances[msg.sender];
4.         userbalances[msg.sender] = 0;
5.         if (!msg.sender.call.value(x)()) throw;
6.     }
7. }
```

Листинг 3. Исправление реентерабельности

Недостаточная защищенность переводящих средства функций

Эта уязвимость не имеет общепринятого названия. Речь здесь идет о ситуации, когда общедоступная (public) функция переводит средства на адрес msg.sender или на адрес, принятый ею в качестве одного из аргументов. Также возможен случай, когда адрес, на который будут переведены средства, проверяется на равенство некоторой переменной, значение которой, в свою очередь, может быть изменено извне.

```
1. contract Crowdfunding {
2.
3.   mapping(address => uint) public balances;
4.   address public owner;
5.   uint256 INVEST_MIN = 1 ether;
6.   uint256 INVEST_MAX = 10 ether;
7.
8.   modifier onlyOwner() {
9.     require(msg.sender == owner);
10.    _;
11.  }
12.
13.  function crowdfunding() {
14.    owner = msg.sender;
15.  }
16.
17.  function withdrawfunds() onlyOwner {
18.    msg.sender.transfer(this.balance);
19.  }
20.
21.  function invest() public payable {
22.    require(msg.value > INVEST_MIN && msg.value < INVEST_MAX);
23.
24.    balances[msg.sender] += msg.value;
25.  }
26.
27.  function getBalance() public constant returns (uint) {
28.    return balances[msg.sender];
29.  }
30.
31.  function() public payable {
32.    invest();
33.  }
34. }
```

Листинг 4. Контракт Crowdfunding с незащищенным переводом средств

Такую уязвимость можно увидеть в контракте Crowdfunding, код которого приведен на листинге 4. Любой посторонний контракт может вывести средства с этого

контракта по следующей схеме. Сначала вызвать общедоступную функцию crowdfunding() (строка 6), тем самым установив свой адрес в переменную owner (иначе говоря, назначить себя хозяином Crowdfunding). Затем вызвать функцию withdrawfunds() (строка 17). Модификатор onlyOwner примет значение истина благодаря предыдущему действию, и средства будут успешно выведены.

Недостаточная защищенность деструктора

Эта уязвимость очень похожа на предыдущую. В некоторых контрактах деструктор (функция, вызывающая исполнение инструкции SUICIDE) общедоступен. Например, контракт может иметь функцию, код которой представлен на листинге 5.

```
1. function kill() public {
2.     selfdestruct(msg.sender);
3. }
```

Листинг 5. Общедоступный деструктор

Вызов деструктора может быть чуть сложнее, тем не менее, доступным извне. Представим, что в контракте с рис. 6 вместо функции withdrawfunds() (строка 17) определена функция, код которой приведен на листинге 6.

```
1. function kill() onlyOwner {
2.     selfdestruct(msg.sender);
3. }
```

Листинг 6. Защищенный деструктор, общедоступный по вине других функций контракта

Сценарий атаки совпадает с предыдущей атакой на Crowdfunding с той лишь разницей, что вместо withdrawfunds() нужно вызывать kill().

Другие типы уязвимостей

Кратко обозначим известные типы уязвимостей, поиск которых не осуществляется созданным инструментом, но осуществляется какими-либо из аналогов.

1. Integer Underflow / Overflow.
2. Использование delegatecall().
   Использование delegatecall может представлять опасность, если его аргументы не проверяются.
3. Использование tx.origin.
   Tx.origin выдает адрес инициатора всей цепочки транзакций (это всегда внешний аккаунт, так как любой контракт начинает исполнение функции только по поступившему сообщению). Tx.origin далеко не всегда совпадает с

msg.sender, и этот факт может привести к серьезным просчетам, которые приведут к уязвимости.

4. Зависимость от порядка транзакций.
   Зависимость от порядка транзакций трудно считать уязвимостью. Если это и уязвимость, то ею трудно воспользоваться. Однако, если контракт подвержен зависимости от порядка транзакций и предполагает частые “клиентские” обращения к нему, это может быть неудобно для клиентов, поэтому многие из аналогичных инструментов выдают об этом предупреждения.
5. Предсказуемая генерация случайных чисел.
   Этой уязвимостью также трудно воспользоваться. Предсказуемой рандомизации стоит особенно избегать в контрактах-лотереях.
6. Необработанные исключения.
   Суть этой уязвимости вполне отражена ее названием. Необработанные исключения часто не препятствуют “успешному” (без отката) завершению транзакции, что ведет скорее к неправильной логике работы, чем к хакерским атакам.
7. Вечная блокировка средств.
   Эту ошибку трудно считать уязвимостью. Она больше похожа на ошибку разработки. Контракт считается подверженным вечной блокировке средств, если в нем нет переводящих средства функций, а также нет деструктора. Очевидно, средства, оказавшиеся на счету такого контракта, никогда не могут быть использованы.

Существующие инструменты поиска уязвимостей

Все существующие инструменты в результате анализа составляют список рекомендаций для разработчика, на что в коде стоит обратить особое внимание и что, возможно, стоит исправить. Созданный в этой работе инструмент отличается тем, что выдает не рекомендации, а возможные сценарии атаки. Еще одна отличительная черта созданного инструмента – осуществление верификации на основе проверки моделей, тогда как большинство аналогов использует верификацию на основе символьного исполнения.

Напомним, что созданный инструмент анализирует контракты на три типа уязвимостей:

- реентерабельность;
- недостаточная защищенность переводящих средства функций;
- недостаточная защищенность деструктора.

Эти типы уязвимостей выбраны неслучайно: их проще всего использовать злоумышленникам, а значит, они представляют наибольшую опасность.

Для сравнения существующие инструменты поиска уязвимостей будут перечислены с указанием используемого подхода к верификации и обнаруживаемых типов уязвимостей, если удалось установить эти характеристики.

1. MAIAN.

   Этот инструмент проверяет контракты на три типа уязвимостей [4]:

- вечная блокировка средств;
- возможность вывода средств любым пользователем (недостаточная защищенность переводящих средства функций);
- возможность вызова деструктора любым пользователем (недостаточная защищенность деструктора).

   Анализ производится путем символьного исполнения.

2. Mythril.

Поиск уязвимостей – лишь одна из многих встроенных в Mythril функций, анализирующая код на уровне инструкций EVM. Для анализа используется символьное исполнение. Также Mythril имеет такие возможности, как дизассемблирование (преобразование шестнадцатиричного кода в листинг инструкций EVM), построение графа потока управления контракта, поиск в сети контрактов, содержащих определенный фрагмент кода, поиск контрактов по их ссылкам на другие контракты, вычисление хеша функции по ее названию и сигнатуре [5].

Mythril проверяет контракты на следующие типы уязвимостей [6]:

- Integer Underflow/Overflow;
- реентерабельность;
- использование delegatecall;
- недостаточная защищенность деструктора;
- использование tx.origin;
- предсказуемая генерация случайных чисел;
- необработанные исключения;
- недостаточная защищенность переводящих средства функций.

3. Porosity.

Главная функция этого инструмента – попытка восстановления кода на Solidity по шестнадцатиричному коду контракта [7]. Шестнадцатиричный код преобразуется в инструкции EVM (дизассемблирование), а затем по инструкциям восстанавливается высокоуровневый код на Solidity (декомпиляция). Поиск уязвимостей осуществляется одновременно с декомпиляцией на основе анализа инструкций.

4. Manticore.

Manticore – инструмент для символьного исполнения как Ethereum-контрактов, так и любых исполняемых файлов. Включает большие возможности для анализа контрактов [8]: кроме упомянутого выше символьного исполнения, также встроены

ассемблер и дизассемблер, компилятор и анализатор кода Solidity, Python API для работы с инструкциями EVM.

5. Securify.

Этот инструмент с закрытым исходным кодом реализован в виде веб-сайта с очень приятным пользовательским интерфейсом [9]. Securify предназначен специально для поиска уязвимостей в Ethereum-контрактах. На вход принимает либо код на Solidity, либо шестнадцатиричный код контракта. Securify проверяет контракты на уязвимости следующих типов:

- зависимость от порядка исполнения транзакций;
- реентерабельность;
- необработанные исключения;
- использование tx.origin;
- Integer Overflow/Underflow;
- вечная блокировка средств;
- предсказуемая генерация случайных чисел.

6. Oyente.

Oyente – инструмент для поиска уязвимостей в смарт-контрактах. Oyente работает с инструкциями EVM, проводя анализ путем символьного исполнения [10]. Контракты анализируются на следующие типы уязвимостей:

- Integer Overflow/Underflow;
- зависимость от порядка выполнения транзакций;
- предсказуемая генерация случайных чисел;
- необработанные исключения;
- реентерабельность.

Система Spin и язык Promela

Система Spin предназначена для верификации темпоральных моделей. Понятие темпоральности означает не только то, что состояние модели меняется со временем, но и то, что спецификации, на соответствие которым проверяется модель, сформулированы с учетом временного аспекта, т.е. оперируют понятиями “никогда”, “всегда”, “когда-либо”, “в конце концов”.

Подробное рассмотрение теории математической логики и алгоритмов, на основе которых работает система Spin, можно найти в [11]. Здесь же дадим лишь краткое описание принципа работы системы Spin. По полученному для анализа коду на языке Promela строится автомат Бюхи, каждое состояние которого соответствует одному набору значений всех глобальных переменных, а операции над переменными соответствуют переходам в автомате. Кроме того, строится автомат Бюхи по отрицанию формальных требований к коду. Затем строится пересечение языков,

порождаемых первым и вторым автоматами. Если пересечение оказывается пусто, значит, код требованиям соответствует, иначе – не соответствует.

Полное описание возможностей встроенного языка программирования Promela можно найти в документации [12]. Скажем здесь о нескольких из них, представляющих наибольший интерес для данной работы. Promela позволяет определить множество типов процессов (proctype), исходный код каждого из которых задается отдельно. Важен тот факт, что одновременно может быть запущено несколько процессов одного типа (исходные коды этих процессов будут одинаковы, но это будут разные процессы). Процесс может принимать на вход аргументы, но не может вернуть значение. Есть процесс, отличающийся от всех остальных – это процесс с именем init, аналог функции main, не принимающий на вход аргументов. Все остальные процессы запускаются асинхронно по вызову run <имя процесса> из кода на Promela.

Promela не поддерживает функции и процедуры в их классическом понимании. Однако, поддерживает макросы, задаваемые ключевым словом inline [13]. Пусть макрос имеет вид inline block_one() { <некоторый код> }. Тогда если далее (но не ранее!) в коде встретится “слово” block_one(), это будет эквивалентно следующему: { <некоторый код> }.

Promela поддерживает метки и goto – безусловный переход на метку.

Promela позволяет задать несколько действий, из которых будет недетерминированно выбрано и исполнено одно. Эта возможность очень удобна для осуществления полного перебора. Также есть возможность задать цикл, который может прекращаться при наступлении некоторого условия либо быть вечным.

Отдельное выражение, значение которого имеет логический тип (bool), воспринимается как await. То есть, например, конструкция вида: …; a == 0; … сработает как …; await <a == 0>; …

Есть возможность задать атомарную последовательность действий. Для этого достаточно заключить последовательность в блок atomic { }. Атомарность, безусловно, может влиять на корректность верификации. Но даже если не сказывается на корректности, то может сказываться на времени верификации, изменяя его в десятки, сотни, даже тысячи раз.

Spin предназначен именно для проверки моделей. Средств для символьного исполнения в нем не предусмотрено, несмотря на то что это было бы чрезвычайно полезно. Самостоятельная реализация символьного исполнения в моделях для Spin была бы очень трудоемкой и едва ли имела бы смысл, так как изначально Spin для этого не предназначен.

Архитектура инструмента

Верификация исходных кодов контрактов осуществляется по следующей схеме. Код на уровне инструкций EVM транслируется в модель на языке Promela. Условия отсутствия в контракте уязвимостей (реентерабельности, недостаточной защищенности деструктора и недостаточной защищенности переводящих средства функций) формулируются также на языке Promela в рамках построенной модели. Таким образом, условия отсутствия в коде контракта уязвимостей служат спецификациями, на которые будет проверяться модель. Непосредственно проверка модели выполняется системой Spin. Если модель соответствует спецификациям (что означает, что код не подвержен уязвимостям), Spin просто сообщает этот факт. В случае же несоответствия модели спецификациям (т.е. подверженности кода уязвимостям) Spin сообщает не только сам факт несоответствия, но и выдает контрпример, по которому легко составить сценарий атаки.

Для того чтобы автоматически построить модель работы контракта, необходимы реализации инструкций EVM на языке Promela. Реализации были выполнены в рамках данной работы в виде макросов inline. Все реализации собраны в едином файле-шаблоне (файл .pml, что означает, что файл содержит код на Promela), который используется при анализе каждого контракта. Далее этот файл-шаблон будет упоминаться как pml-шаблон.

Кроме того, Promela предоставляет возможность делать вставки кода на C, которая используется для реализации нескольких инструкций. C-код, реализующий эти инструкции (файлы .c, далее – C-шаблоны), также используется при анализе каждого контракта и хранится вместе с pml-шаблоном. pml-шаблон обращается к коду на C посредством специальной директивы #include, используемой в языке Promela аналогично языку C. Иначе говоря, pml-шаблон использует C-шаблоны как библиотеки. UML-диаграмма компонентов на рис. 3 отражает рассмотренные зависимости модели и шаблонов (пунктирная стрелка означает импортирование и направлена к импортируемому файлу, пунктирная линия только лишь соединяет компонент с примечанием к нему).

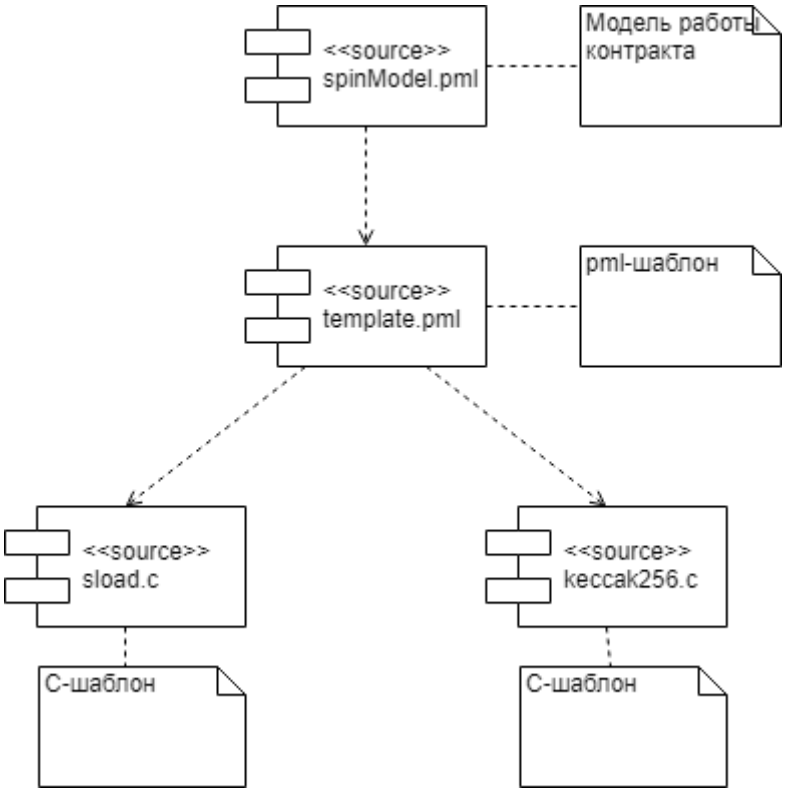

Рис. 3. Схема компонентов, сопутствующих Promela-модели контракта

Собственно трансляцию кодов контрактов из инструкций EVM в язык Promela осуществляет Python-скрипт. На рис. 4 представлена диаграмма потоков данных, отображающая связь всех стадий работы инструмента (трансляция, верификация) в единое целое.

На диаграмме можно видеть такой внешний элемент, как Ethereum-сеть. Для того, чтобы контракт мог быть верифицирован, необходимо, чтобы он был загружен в некоторую Ethereum-сеть (это может быть не только основная сеть Ethereum, но и любая из тестовых Ethereum-сетей или даже локальная Ethereum-сеть, поднятая только на одном компьютере). В процессе верификации требуются обращения по API в эту Ethereum-сеть для чтения хранилища контракта, о чем будет сказано далее в этой главе. Обращения реализованы на языке C (C-шаблоны).

Работа инструмента начинается с запуска пользователем Python-скрипта, в качестве аргумента которому подается адрес в сети верифицируемого контракта. Python-скрипт обращается по API к Ethereum-сети и получает исходный код контракта (на уровне инструкций) по заданному адресу. Затем полученный исходный код транслируется в язык Promela, в результате чего получается Promela-модель. Promela-модель подается на верификацию системе Spin (подача должна быть произведена пользователем специальной командой в консоли). Модель построена таким образом, что в процессе верификации требуются обращения в Ethereum-сеть. Когда верификация выполнена, Spin выдает результаты (по умолчанию на консоль, но можно направить вывод в текстовый файл).

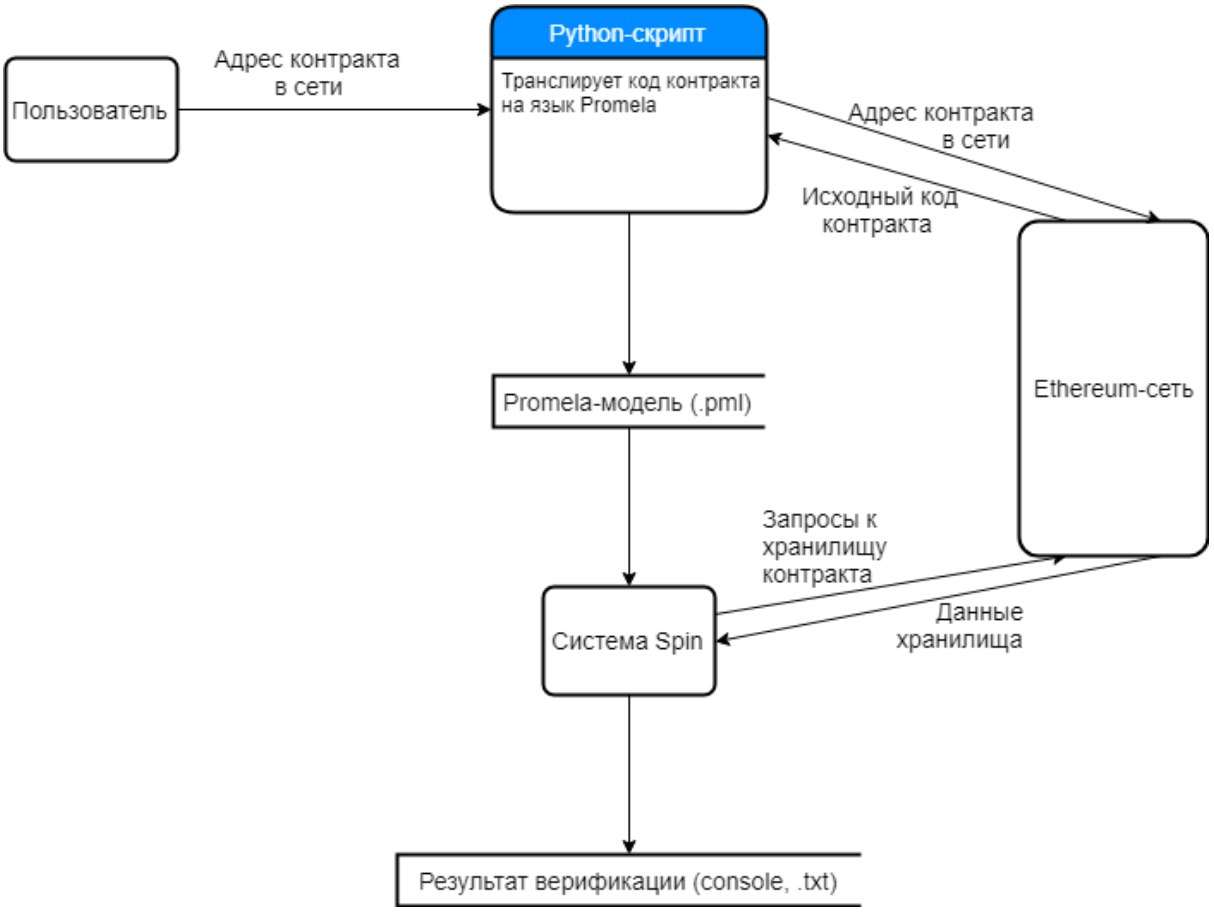

Рис. 4. Диаграмма потоков данных созданного инструмента

Алгоритм трансляции, реализуемый Python-скриптом, прост: каждая инструкция заменяется на свою Promela-реализацию. Однако, необходим определенный предварительный анализ кода контракта, выполняемый этим же Python-скриптом. Обстоятельно предварительный анализ будет рассмотрен далее в этой и следующей главах.

Принцип трансляции кодов контрактов в язык Promela представляет собой не что иное, как реализацию на Promela следующей схемы. Будем называть жертвой контракт, который проверяется на уязвимости (чисто условный термин, вовсе не обозначающий, что контракт уязвим). Предполагаемый злоумышленник – это тоже контракт, который отправляет жертве несколько транзакций таким образом, чтобы нанести ущерб (в нашем случае либо вывести все деньги, либо вовсе разрушить жертву). Транзакция – это вызов одной высокоуровневой функции контракта (в нашем случае жертвы). Так, если злоумышленник поставил цель нанести ущерб жертве, ему нужно построить такую последовательность вызовов функций жертвы, которая нанесет ей вред.

Трансляция кода контракта проецирует приведенную схему на понятия системы Spin. Код каждой функции контракта выделяется в отдельный тип процесса. Роль злоумышленника играет процесс init. Чрезвычайно важен тот факт, что тело каждой функции целиком атомарно. Это отражается как на корректности верификации, так и

на ее времени: экспериментально было установлено, что благодаря атомарности время сокращается в тысячи раз.

В начале работы процесс init назначает жертве и злоумышленнику одинаковые не нулевые балансы. В процессе init (злоумышленник) задан цикл, прекращающийся либо при нулевом балансе на счету жертвы, либо при разрушении жертвы, либо после вызова функций жертвы определенное число раз. В теле цикла происходит недетерминированный выбор и вызов одной из функций контракта. Очевидно, в такую модель вписывается любая последовательность вызовов функций.

Рассмотрим описанную схему на небольшом примере (листинг 7). Кроме init определены два типа процессов, что соответствует контракту с двумя высокоуровневыми функциями. Процесс init работает так, как и было описано. Стадия ожидания ответа выделена потому, что в Ethereum вызовы функций всегда синхронны, в отличие от Spin. Если проверяемый контракт разрушен или ограблен, цикл прекращается, так как это означает, что уязвимость найдена. Чтобы проверка неуязвимых контрактов не была вечной, цикл в init прекращается еще при некоторых других условиях, о которых пойдет речь в следующей главе.

```
proctype function_1 {
    ...
}

proctype function_2 {
    ...
}


proctype init {
начало:
    недетерминированно: либо run function_1
                        либо run function_2
    ожидание ответа
    проверка, не разрушен ли и не ограблен ли проверяемый контракт
    goto начало
}
```

Листинг 7. Общая схема работы созданного инструмента

Первая решаемая при предварительном анализе подзадача – сбор адресов начала всех высокоуровневых функций. Из того, что было сказано на этот счет в обзоре, элементарно следует, что эта подзадача легко решается путем просмотра кода до первой (реже – до второй) инструкции JUMPDEST.

В процессе исполнения функции используются стек и память. Содержимое стека и памяти хранится до завершения исполнения функции, вследствие этого им соответствуют локальные переменные типа процесса. Стек и память реализованы в

виде массивов, логика заполнения которых типична для стека и обычного массива байтов (память).

Содержимое хранилища сохраняется после завершения исполнения функций, поэтому хранилищу соответствует глобальная переменная модели. Хранилище реализовано в виде map-структуры (точнее говоря, в виде ассоциативного массива, так как Promela не имеет встроенного типа map). Сохранение значения в хранилище по некоторому ключу key реализовано обычным образом, тогда как с получением значения дело обстоит сложнее. При первом получении значения хранилища по ключу key происходит обращение в Ethereum-сеть по специальному API, и в соответствующую хранилищу переменную записывается полученное значение. При последующих запросах к хранилищу по ключу key значение берется из соответствующей хранилищу переменной.

Балансы хранятся в ассоциативном массиве, реализующем map-структуру: адрес контракта – баланс. Работа со сторонними контрактами (все контракты кроме верификатора и проверяемого) не реализована, и map-структура для балансов не обязательна, она служит лишь заделом для возможной поддержки обращений к сторонним контрактам в дальнейшем.

Чрезвычайно важны логические глобальные переменные was_destructed и robbed. was_destructed истинна в том и только том случае, если проверяемый контракт разрушен (исполнена инструкция SUICIDE), robbed – если баланс проверяемого контракта равен нулю. was_destructed ложна изначально и принимает истинное значение при исполнении проверяемым инструкции SUICIDE. Значение переменной robbed обновляется в определенные моменты, о которых пойдет речь в следующей главе. В те же моменты происходит проверка значений was_destructed и robbed. Имея эти две переменные, нетрудно сформулировать спецификации, эквивалентные отсутствию в контракте трех обнаруживаемых типов уязвимостей: достаточно потребовать, чтобы всякий раз при проверке значений was_destructed и robbed были ложны.

Для простоты составления сценария атаки уязвимого контракта в модель введена глобальная переменная history, представляющая собой массив из записей специального вида. Всякий раз, когда процесс init запускает какой-либо другой процесс, в переменную history добавляется запись, содержащая информацию о типе вызванного процесса (proctype, однозначно соответствующий функции проверяемого контракта) и еще некоторые данные, которые сейчас не важны. Если модель не соответствует спецификациям (т.е. контракт уязвим), Spin сообщит этот факт и выведет в том числе значения всех глобальных переменных в тот момент работы модели, когда спецификации нарушились. Таким образом будет выведено в том числе значение history, представляющее собой не что иное, как последовательность вызовов функций жертвы, приводящая к ее разрушению или ограблению.

Трансляция контрактов в язык Promela

Результатом трансляции является Promela-модель работы контракта, подаваемая на верификацию системе Spin. В процессе реализации архитектуры, рассмотренной в предыдущей главе, пришлось столкнуться с несколькими нетривиальными вопросами, решению которых посвящена эта глава. Именно, речь идет о следующих вопросах.

1. Реализация “прыжков” по коду для инструкций JUMP и JUMPI.
2. Синхронизация процессов. В Ethereum вызов функций всегда синхронен, тогда как в Spin запуск процессов асинхронен.
3. Реализация возможности полного отката транзакции для инструкции REVERT.
4. Ограничения на вызов верификатором функций жертвы. Ограничения накладываются для того, чтобы верификация, во-первых, не длилась бесконечно, во-вторых, укладывалась в приемлемое время.
5. Работа с аргументами функций. Для работы с заранее неизвестными аргументами изобретена идея символьного исполнения. Однако, Spin не содержит средств для символьного исполнения, из-за чего работа с аргументами должна быть проведена дополнительно.
6. Выбор верификатором value вызова. Value вызова – сумма денег, которая будет переведена верификатором жертве при вызове функции.

Реализация “прыжков” по коду для инструкций JUMP и JUMPI

Инструкция JUMP – это безусловный переход по коду на адрес, лежащий на вершине стека. Инструкция JUMPI – это условный переход, при котором адрес перехода лежит на вершине стека, условие – сразу под адресом в стеке; если условие не равно нулю, происходит переход на адрес, если же условие равно нулю, выполняется следующая инструкция. Инструкция JUMPDEST помечает адрес, на который допускается прыжок.

Назовем блоком часть кода от одной инструкции JUMPDEST до следующей (пусть для определенности первая JUMPDEST входит в блок, вторая – нет). Каждый блок кроме первого (реже – первого и второго) оформляется в виде макроса inline с именем block_<address>, где <address> – адрес инструкции JUMPDEST. Первый (реже еще и второй) блоки отвечают за диспетчеризацию и обрабатываются иным образом (об этом было сказано в предыдущей главе). Определения макросов для блоков следуют в том же порядке, в котором блоки следуют в исходном коде контракта.

После определений макросов для всех блоков следует макрос jump(). О нем подробнее будет сказано совсем скоро.

Далее идет макрос contract_code, представляющий собой полный код контракта в следующем виде. Снова в том же порядке, что блоки в исходном коде контракта, следуют имена соответствующих этим блокам макросов. Причем перед каждым

именем макроса стоит метка вида b_<address> (тот же address, что в имени макроса). Если адрес начала блока является также адресом начала какой-либо из высокоуровневых функций контракта, перед именем соответствующего макроса стоит также метка вида func_<address>.

Что касается макроса jump(), он реализован следующим образом. На уровне Python-скрипта выбираются адреса всех инструкций JUMPDEST. Реализация jump() снимает со стека верхнее значение (адрес перехода), сравнивает его с каждым из JUMPDEST-адресов и в случае равенства выполняет переход goto b_<JUMPDEST-адрес>.

После макроса contract_code() следуют типы процессов, соответствующие функциям контракта. Типы процессов имеют вид, представленный на листинге 8 (предположим, 687 – адрес начала функции).

```
proctype function_687(){
   объявление переменной-стека;
   объявление переменной-памяти;
   goto func_678;
   contract_code()
}
```

Листинг 8. Схема реализации каждого типа процесса

И, наконец, в конце кода, получившегося в результате трансляции – несколько вспомогательных макросов для init и сам процесс init.

Синхронизация процессов

Синхронизация обеспечивается двумя глобальными переменными: run_verifier и callstack. Run_verifier – логическая переменная, принимающая значение истина, когда верификатор может продолжать работу, и ложь, когда должен ожидать ответа от проверяемого. Callstack – стек вызовов: всякий раз, когда вызвана какая-либо функция (как верификатором, так и проверяемым), на этот стек кладется одна запись, содержащая информацию о вызвавшем, вызванном, value вызова (сумму денег, переведенную адресату) и определенные данные, о которых пойдет речь в следующем пункте этой главы.

Принцип действия переменной run_verifier иллюстрирует UML-диаграмма последовательности на рис. 5. В начале работы модели run_verifier получает значение true (не отмечено на рисунке). Каждый раз перед тем, как вызвать функцию проверяемого, верификатор сбрасывает значение этой переменной в ложь (run_verifier = false), а затем ожидает, пока оно снова станет истиной (await (run_verifier == true)). Когда же проверяемый в процессе работы встречает одну из инструкций, завершающих транзакцию (STOP, RETURN или REVERT), он убеждается, что сам был

вызван именно верификатором (т.е. убеждается, что в верхней записи стека вызовов вызвавший – это верификатор, что на рисунке не отмечено для краткости), и если в этом убедился, то присваивает run_verifier = true. Сейчас в созданном инструменте не реализована возможность обращений к сторонним контрактам (контрактам кроме верификатора и проверяемого), поэтому верхнюю запись стека вызовов можно не проверять; проверка выполняется только как задел для реализации этой возможности.

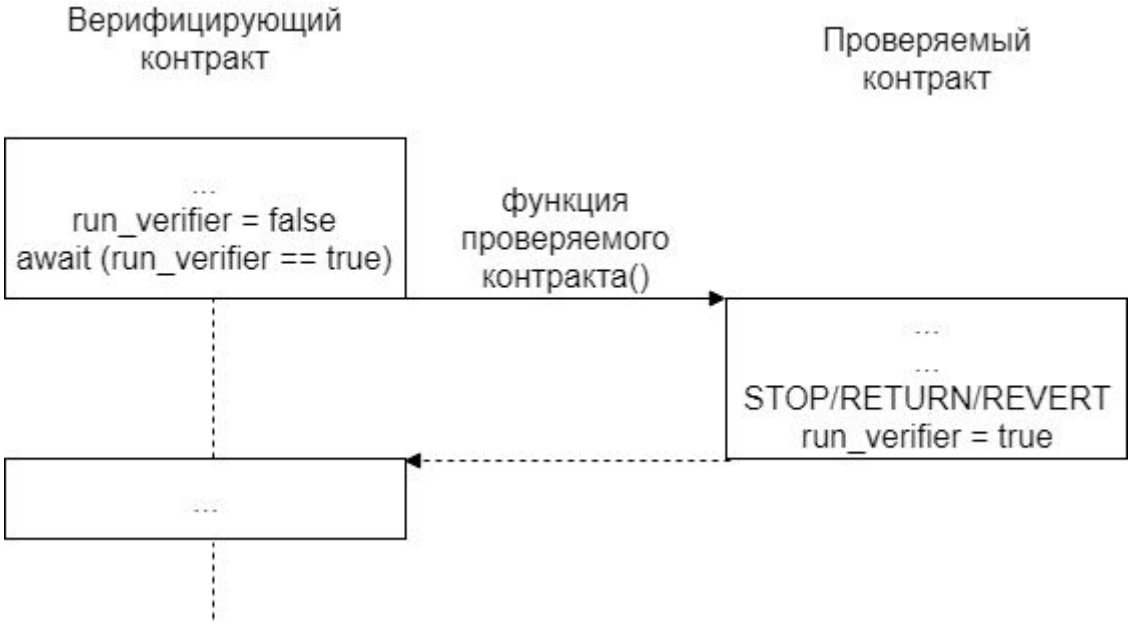

Рис. 5. Изменения состояния переменной run_verifier

Инструкция CALL, встреченная проверяемым контрактом, усложняет схему работы. На данный момент реализован только случай, когда проверяемый вызывает транзакцию верификатора, но не какого-либо другого контракта. Такой случай проиллюстрирован на рис. 6. Обратимся сразу к моменту, когда проверяемый встретил CALL и отправил вызов верификатору (в результате верификатор оказался в состоянии II). Верификатор, в свою очередь, может просто дать проверяемому разрешение на продолжение process_1, а может снова вызвать какую-либо функцию проверяемого (именно этот случай показан на рис. 6). Предположим, была вызвана какая-либо функция проверяемого, встретившая инструкцию завершения транзакции (STOP, RETURN или REVERT). Тогда, как и в предыдущем примере, верификатор узнает об этом благодаря переменной run_verifier и снова перейдет в состояние II. Заметим различие между состояниями верификатора I и II: в состоянии I нет процессов проверяемого, ожидающих ответа от верификатора, тогда как в состоянии II такие процессы есть.

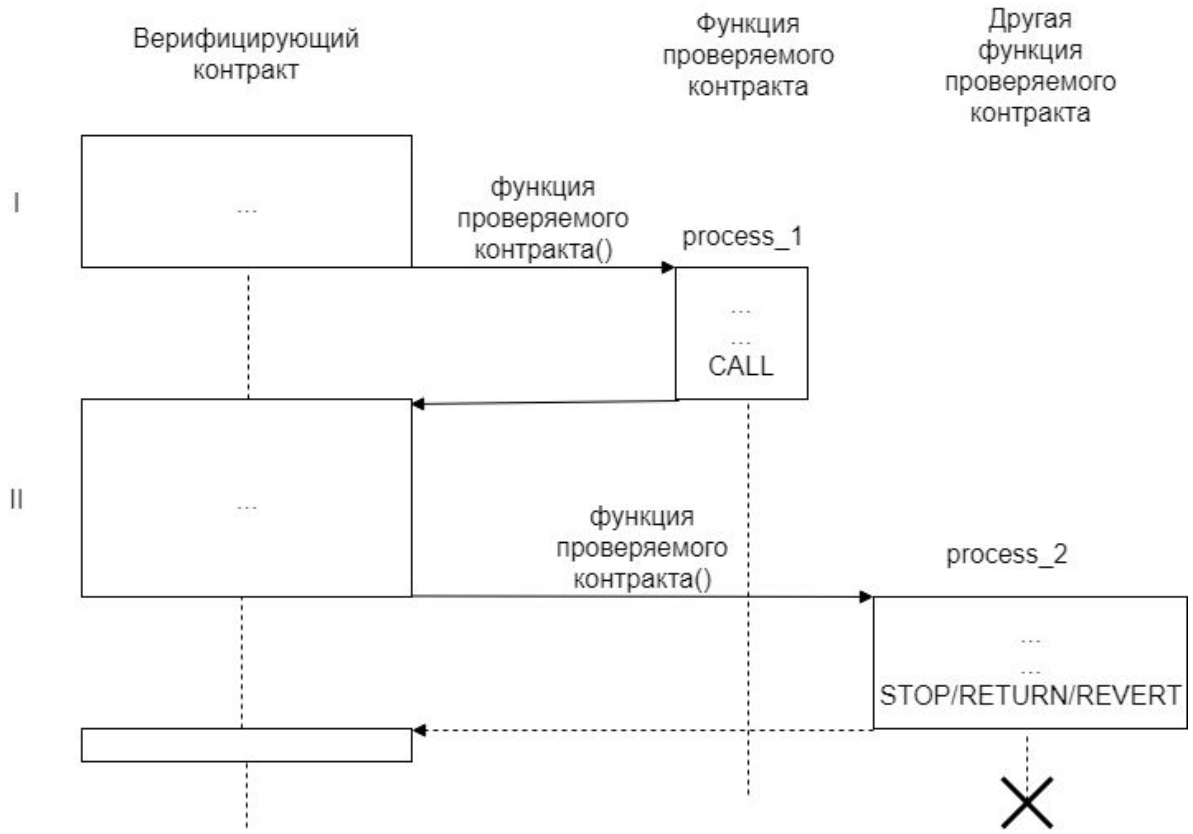

Рис. 6. Краткая схема обработки инструкции CALL

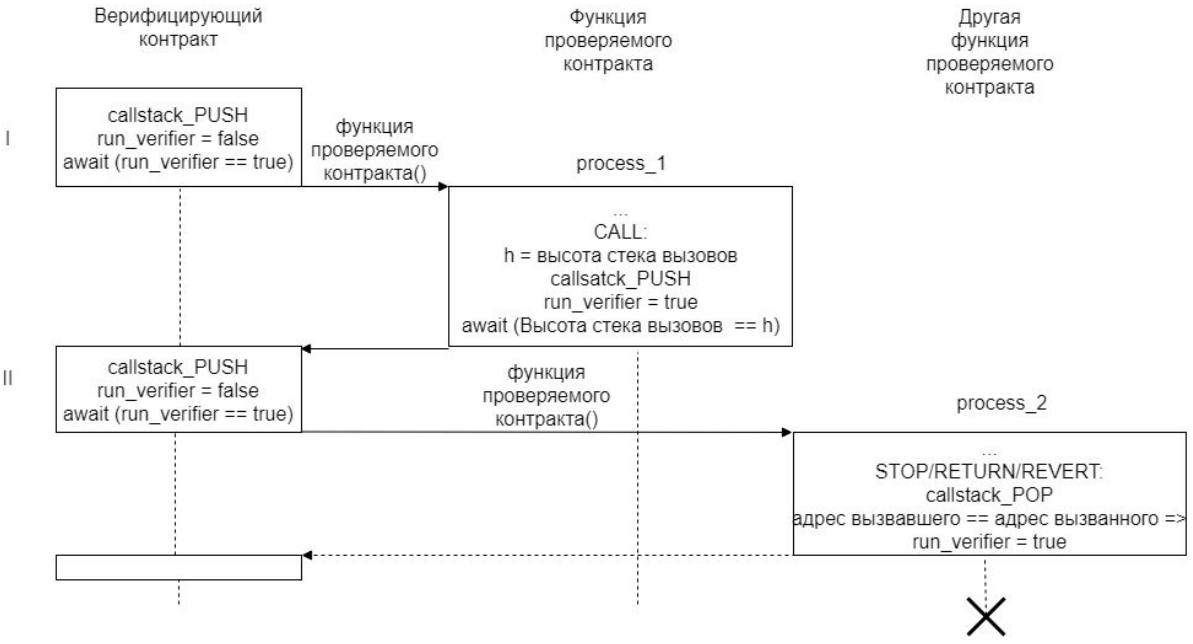

Рис. 7. Схема влияния переменной callstack на последовательность вызовов

Для того, чтобы ожидающий разрешения на продолжение process_1 успешно получил это разрешение и затем продолжил работу, служит переменная callstack.

Принцип работы с этой переменной (и заодно run_verifier при работе с CALL) представлен на рис. 7. callstack_PUSH кладет одну запись на стек вызовов, callstack_POP снимает одну запись со стека вызовов. Встретив инструкцию CALL, проверяемый запоминает высоту стека вызовов, кладет на стек вызовов одну запись, а затем ожидает, пока высота стека вызовов снова станет равна запомненому значению. Встретив STOP/RETURN/REVERT, проверяемый снимает со стека вызовов одну запись. Теперь рассмотрим момент, когда проверяемый выполнил одну из инструкций STOP/RETURN/REVERT, и верификатор вернулся после этого в состояние II (иначе говоря, после обозначенной пунктиром стрелки). process_2 завершен, остался один ожидающий процесс – process_1. Теперь для того, чтобы дать процессу process_1 разрешение на продолжение, верификатору достаточно сбросить значение run_verifier в ложь, снять одну запись со стека вызовов и поставить await(run_verifier == true).

Такую последовательность вызовов функций, которая начинается с состояния I и заканчивается также в состоянии I, далее будем называть цепочкой вызовов. Цепочка включает в себя вызовы, направленные в обе стороны: как от верификатора к проверяемому, так и наоборот. Назовем полной цепочку, которая возвращается в состояние I (рис. 5). Неполной назовем цепочку, которая не возвращается в состояние I (рис. 6).

Реализация возможности полного отката транзакции для инструкции REVERT

Реализация этой возможности осложняется тем, что при откате транзакции должен быть выполнен откат всех ее подтранзакций. Если tx – транзакция, то ее подтранзакции – это все транзакции из той же цепочки, что и tx, которые были вызваны позже, чем сама tx. Например, на рис. 6 транзакция, соответствующая process_2, является подтранзакцией по отношению к транзакции, соответствующей process_1.

Возможность отката поддержана в созданном инструменте стандартным образом – посредством ведения журнала изменений. Журнал изменений реализован в виде стека изменений: чем новее изменение, тем выше в журнале запись о нем. Из-за сходства принципа заполнения журнала со стеком длину журнала изменений будем называть высотой журнала. При изменении балансов посредством вызовов с переводом ненулевой суммы денег или при выполнении проверяемым контрактом инструкции SSTORE (сохранение значения в свое хранилище по ключу) наверх журнала изменений кладется запись о проделанном действии.

Откат изменений возможен благодаря следующей схеме. Запись, которая кладется на стек вызовов перед началом транзакции, содержит текущую высоту журнала изменений, которую обозначим за changesBefore. Повторимся, что в каждый момент времени есть ровно один процесс, находящийся в состоянии работы (остальные – в ожидании), и заметим, что вызову этого процесса соответствует верхняя запись стека вызовов. Таким образом, все изменения, сделанные исполняющейся

транзакцией и ее подтранзакциями, зафиксированы в журнале изменений выше позиции с номером changesBefore из верхней записи стека вызовов. Зная это, откатить сделанные транзакцией изменения элементарно. Именно так реализован откат изменений в инструкции REVERT. Кроме того, список изменений позволяет легко определить, изменилось ли в результате транзакции состояние проверяемого контракта.

Записи из журнала изменений удаляются по следующим правилам.

1. При выполнении проверяемым контрактом инструкции REVERT все изменения, сделанные исполняющейся транзакцией и ее подтранзакциями, сначала откатываются, а затем удаляются из журнала.
2. Журнал полностью очищается при выполнении проверяемым контрактом инструкций STOP или RETURN в случае, если нет ожидающих процессов. Условие отсутствия ожидающих процессов обеспечивает каждой транзакции в любой момент вплоть до ее завершения доступность журнала изменений, сделанных всеми ее подтранзакциями.

Ограничения на вызов верификатором функции жертвы

Назовем сценарием последовательность вызовов верификатором функций жертвы (возможно, один из них является сценарием атаки). Перечислим ограничения, предъявляемые созданным инструментом к рассматриваемым сценариям. Сценарии, не удовлетворяющие этим ограничениям, не рассматриваются.

1. Если последняя цепочка вызовов не изменила состояние проверяемого контракта (баланс и данные в хранилище), сценарий прерывается (т.е. прекращается цикл в init).
2. Длина цепочки вызовов ограничена: если высота стека вызовов равна 10, верификатор больше не вызывает функции проверяемого, а только дает ожидающим разрешение на продолжение.
3. Сценарий может содержать не более 4 вызовов функций проверяемого, сделанных верификатором из состояния I (т.е. из состояния, когда нет ожидающих процессов). Экспериментальным путем было установлено, что выбранное для этого ограничения число заметно отражается на времени верификации.
4. Если со стека вызовов была снята одна запись, то верификатор не может вызывать функции проверяемого (т.е. запускать новые процессы), пока высота стека вызовов не станет равной нулю. С того момента, как высота стека вызовов уменьшилась, и до момента, пока она станет равной нулю, верификатор может только давать ожидающим процессам разрешение на продолжение. Логической глобальной переменной justPushed присваивается значение true всякий раз, когда на стек вызовов кладется запись, и значение false, когда запись со стека

вызовов снимается, и по значению justPushed верификатор всегда может определить, начала ли убывать высота стека вызовов.

5. Заметим, что если верификатор находится в состоянии II, то он сам был вызван проверяемым. В результате этого вызова он мог получить ненулевую сумму денег, но мог ее не получить. Только в том случае, если получил ненулевую сумму, он может снова вызвать функцию проверяемого (т.е. запустить новый процесс), причем только ту же самую высокоуровневую функцию, которую вызывал в предыдущий раз.

Работа с аргументами функций

В EVM есть инструкция CALLDATALOAD. Она кладет на стек 32-байтный отрезок данных, полученных контрактом от инициатора транзакции. В нашем случае инициатор транзакции – это сам верификатор (предполагаемый злоумышленник). Значит, в конечном счете верификатор решает, что положит на стек CALLDATALOAD.

Также есть инструкция CALLVALUE, которая кладет на стек сумму денег, полученную от инициатора. Эту сумму денег также определяет верификатор.

Далее, то, что положено на стек инструкциями CALLVALUE и CALLDATALOAD, может повлиять на условие в условном переходе.

На стадии предварительного анализа кода контракта после выбора адресов начала высокоуровневых функций происходит симуляция исполнения каждой из них. Чтобы симуляция не уходила в бесконечный цикл, нить исполнения обрывается при входе в блок кода, в котором она уже успела побывать. Нитью исполнения здесь называется последовательность адресов инструкций, получающаяся при выборе одной конкретной ветви при каждом встреченном условном переходе. В процессе симуляции отслеживается содержимое стека и памяти в строковых представлениях (например, “storage[1]”, “calldata[4] + 7”, иногда встречаются представления простого вида, такие как “32”). Кроме строкового представления, для каждого элемента стека и памяти хранится логический признак, принимающий истинное значение тогда и только тогда, когда элемент зависит от значения, положенного в стек инструкцией CALLDATALOAD либо CALLVALUE. Этот логический признак эквивалентен влиянию на элемент верификатора (злоумышленника). Этот признак поддерживается в корректном состоянии следующим образом. У каждого элемента, который кладется на стек инструкцией CALLDATALOAD либо CALLVALUE, признак назначается истиной. Далее для каждой инструкции выполняется проверка: если хоть у одного операнда этот признак истинен, то он истинен также у результата инструкции, иначе у результата инструкции признак ложен.

Задача симуляции – выбрать все адреса условных переходов JUMPI, на условие которых влияет верификатор. Выбранные JUMPI транслируются не в обычные условные переходы, а в недетерменированные “развилки”. Однако, среди выбранных

переходов JUMPI есть исключения, которые все-таки транслируются в обычные условные переходы. Во-первых, это те переходы, условие которых имеет вид CALLVALUE == 0, так как value влияет не только на условный переход, но и на балансы. Во-вторых, если в пределах одной функции верификатор влияет на элемент стека, но элемент с точно таким же строковым представлением задает сумму перевода для инструкции CALL (в проверяемом контракте), то все переходы JUMPI, условие которых совпадает со строковым представлением этого элемента, транслируются в обычные условные переходы. Это сделано для того, чтобы созданный инструмент не игнорировал выполняемые жертвой проверки запрошенной злоумышленником суммы. От того факта, что в условные переходы во втором случае транслируются только те инструкции JUMPI, строковое представление которых в точности совпадает с представлением суммы перевода для CALL, корректность работы инструмента падает: если сумма перевода связана с тем значением, которое было проверено, но не совпадает с ним в точности, и на проверенное значение может повлиять верификатор, то такая проверка будет проигнорирована инструментом и транслирована как недетерминированная “развилка”.

Те инструкции JUMPI, которые при трансляции становятся недетерменированными “развилками”, транслируются таким образом, что в Promela-модели в историю вызовов функций (глобальная переменная history) записывается, по которой из ветвей пошло исполнение и адрес соответствующей инструкции JUMPI в коде контракта. Python-скрипт (тот самый, который выполняет первичный анализ и трансляцию) выводит на консоль адреса всех JUMPI, транслированных в недетерминированные “развилки”, и соответствующие им условия. Таким образом, видя переменную history Promela-модели и вывод Python-скрипта, можно подобрать подходящие для атаки аргументы функций и value вызовов.

Что же касается реализации CALLDATALOAD в pml-шаблоне, эта инструкция кладет на стек значение, равное изначальным балансам верификатора и проверяемого. Такая реализация сделана потому, что удобна в случае, если аргумент функции играет роль запрашиваемой суммы. Это очень упрощенное предположение, сделанное потому, что реализация символьного исполнения на данном этапе крайне бедна и сильно осложняется тем, что система Spin для этого не предназначена.

Проследить за тем, что значение испытывает на себе влияние верификатора, созданный инструмент может только в рамках одной функции. Например, если в результате функции fun_1(int x) значение x было записано в хранилище проверяемого контракта по ключу 0, то в процессе исполнения следующих функций значение хранилища по ключу 0 уже не будет считаться подверженным влиянию верификатора. В этом один из недостатков, присущих созданному инструменту.

Выбор верификатором value вызова

Симуляция, выполняемая при первичном анализе исходного кода, различает функции, принимающие деньги и функции, не принимающие денег. При попытке вызвать не принимающую денег функцию с ненулевым value, произойдет откат транзакции.

Пусть функция не принимает денег. Одна из нитей исполнения такой функции – это проверка CALLVALUE на равенство нулю и откат транзакции, если CALLVALUE != 0. В этой нити исполнения есть всего одно условие, на которое влияет верификатор: CALLVALUE == 0. Если у функции есть такая нить исполнения, в которой верификатор влияет на единственное условие, имеющее вид CALLVALUE == 0, то симулятор считает, что эта функция не принимает денег.

Если функция не принимает денег, она всегда вызывается с нулевым value. Если функция принимает деньги, она вызывается либо с нулевым value, либо с value, равным изначальным балансам верификатора и проверяемого.

Проверка инструмента на тестовых примерах контрактов

План тестирования говорит о том, что созданный инструмент является работающим прототипом, но не финальной готовой к использованию версией. Созданный инструмент рассчитан на поиск трех типов уязвимостей: реентерабельности, недостаточной защищенности деструктора и недостаточной защищенности переводящих средства функций. Сначала убедимся, что на каждый из этих типов уязвимостей будет построен корректный сценарий атаки. Затем проверим, что не подверженный уязвимостям контракт будет признан соответствующим спецификациям.

Тестирование инструмента на реентерабельном контракте

Начнем тестирование с реентерабельного контракта, код которого приведен на листинге 9.

```
contract HoneyPot {
  mapping (address => uint) public balances;

  constructor() payable public {
    put();
  }

  function put() payable public {
    balances[msg.sender] = msg.value;
  }

  function get() public {
    bool succeed;
    (succeed, ) = msg.sender.call.value(balances[msg.sender])("");
    if (!succeed) {
      revert();
    }
      balances[msg.sender] = 0;
  }

  function() external {
    revert();
  }
}
```

Листинг 9. Код тестового реентерабельного контракта

Все объяснения были даны в обзоре, сейчас лишь заметим, что этот контракт может быть ограблен по следующему сценарию. Сначала злоумышленник вызывает put(), ее выполнение завершается. Затем злоумышленник вызывает get(), в процессе

исполнения которого жертва вызывает злоумышленника. Воспользовавшись вновь появившимся правом вызова, злоумышленник снова вызывает get(). В процессе вновь вызванного get() жертва снова вызывает злоумышленника, в результате чего у жертвы на счету не остается денег. Будучи анализирующим код “экспертом”, нетрудно понять, что снова вызывать get() бессмысленно, дальше стоит только давать ожидающим процессам разрешение на продолжение. Схема построенного “экспертом” сценария может быть представлена в таком виде, как на листинге 10. Если вызов является подтранзакцией для другого вызова, он обозначается на рисунке с отступами.

```
put()
get()
   get()
```

Листинг 10. Сценарий атаки, построенный человеком

Сценарий атаки, построенный с помощью созданного инструмента, выглядит так, как показано на листинге 11.

```
put()
get()
   get()
      get()
```

Листинг 11. Сценарий атаки, построенный созданным инструментом

Построенный сценарий не является кратчайшим, но добивается цели: жертва в результате него оказывается ограблена.

Тестирование инструмента на контракте с незащищенным деструктором

Рассмотрим результат работы инструмента на контракте с приведенным на листинге 12 исходным кодом. Этот контракт очень напоминает предыдущий, однако, он не подвержен реентерабельности (в обзоре этот факт подробно объяснен). Единственная уязвимость этого контракта в том, что функция-деструктор kill(address payable addr) общедоступна. Сценарий атаки в данном случае состоит из единственного вызова kill, причем с любым аргументом (наиболее выгодно подать в качестве аргумента свой адрес, и тогда на него перейдут все деньги жертвы; однако, независимо от поданного адреса, жертва будет разрушена). В результате работы инструмента был построен точно такой сценарий.

```
contract Suicider {
  mapping (address => uint) public balances;

  constructor() payable public {
    put();
  }

  function put() payable public {
    balances[msg.sender] = msg.value;
  }

  function get() public {
    uint x = balances[msg.sender];
    if (x > 0) {
      balances[msg.sender] = 0;
      bool succeed;
      (succeed, ) = msg.sender.call.value(x)("");
      if (!succeed) {
        revert();
      }
    }
  }

  function() external {
    revert();
  }

  function kill(address payable addr) external {
    selfdestruct(addr);
  }
}
```

Листинг 12. Контракт с общедоступным деструктором

Тестирование инструмента на контракте с недостаточно защищенной функцией перевода средств

Рассмотрим контракт, исходный код которого приведен на листинге 14. Сценарий его атаки имеет вид, представленный на листинг 13.

```
crowdfunding()
withdraw(баланс жертвы)
```

Листинг 13. Сценарий атаки, построенный человеком

Созданный инструмент расценил этот контракт как уязвимый и нашел сценарий атаки, который позволяет его ограбить. Однако, построенный сценарий, который можно видеть на листинге 15, далек от оптимального. В начале сценария можно видеть два подряд вызова функции invest(). У верификатора не хватило бы на них денег, но инструмент устроен так, что верификатор может отдавать любую сумму. Также инструмент не следит за тем, чтобы было предоставлено достаточно gas, иначе перевызов withdraw был бы невозможен: transfer предоставляет сумму gas, недостаточную для перевызова.

```
contract Crowd {

  mapping(address => uint) public balances;
  address public owner;
  uint256 INVEST_MIN = 1 ether;
  uint256 INVEST_MAX = 10 ether;

  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  function crowdfunding() public {
    owner = msg.sender;
  }

  function withdraw(uint amount) onlyOwner public {
    msg.sender.transfer(amount);
  }

  function invest() public payable {
    require(msg.value > INVEST_MIN && msg.value < INVEST_MAX);

    balances[msg.sender] += msg.value;
  }

  function getBalance() public view returns (uint) {
    return balances[msg.sender];
  }

  function() external payable {
    invest();
  }
}
```

Листинг 14. Контракт с недостаточно защищенной функцией перевода средств

```
invest()
invest()
crowdfunding()
withdraw(начальный баланс проверяемого и верификатора)
   withdraw(начальный баланс проверяемого и верификатора)
      withdraw(начальный баланс проверяемого и верификатора)
         withdraw(начальный баланс проверяемого и верификатора)
```

Листинг 15. Найденный инструментом сценарий атаки

Тестирование инструмента на контракте, не подверженном уязвимостям

В качестве примера контракта, не подверженного уязвимостям, рассмотрим контракт, представленный на листинге 9, исправленный таким образом, чтобы не быть

подверженным реентерабельности. В обзоре подробно рассмотрено сделанное исправление. На листинге 16 представлен исправленный исходный код.

```
contract Clear {
  mapping (address => uint) public balances;

  constructor() payable public {
    put();
  }

  function put() payable public {
    balances[msg.sender] = msg.value;
  }

  function get() public {
    uint x = balances[msg.sender];
    if (x > 0) {
      balances[msg.sender] = 0;
      bool succeed;
      (succeed, ) = msg.sender.call.value(x)("");
      if (!succeed) {
        revert();
      }
    }
  }

  function() external {
    revert();
  }
}
```

Листинг 16. Исходный код не подверженного уязвимстям контракта

Созданный инструмент в результате анализа признал соответствие кода этого контракта спецификациям, что означает неподверженность контракта уязвимостям.

Заключение

В рамках данной работы были достигнуты следующие результаты.

1. Разработана архитектура инструмента, включающая подход к решению и укрупненную схему его работы.
2. Реализована трансляция кодов контрактов в язык Promela в виде Python-скрипта и общего шаблона на Promela.
3. Инструмент проверен на нескольких тестовых контрактах, подверженных уязвимостям, одном – не подверженном уязвимостям; в результате проверки получены корректные результаты работы инструмента. Проведенная проверка позволяет считать инструмент еще не финальной готовой к использованию версией, но уже работающим прототипом.

Список литературы

[1] Dr. Gavin Wood. (05.05.2019) ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER BYZANTIUM VERSION
[2] DASP – TOP 10. (https://dasp.co/). Просмотрено: 18.05.2019.
[3] Dennis Peterson. (01.07.2016) Writing Secure Solidity. (https://www.blunderingcode.com/writing-secure-solidity/). Просмотрено: 18.05.2019.
[4] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. (2018) Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18). ACM, New York, NY, USA, 653-663. DOI: https://doi.org/10.1145/3274694.3274743
[5] https://github.com/ConsenSys/mythril-classic Просмотрено: 18.05.2019.
[6] https://github.com/ConsenSys/mythril-classic/tree/master/mythril/analysis/modules Просмотрено: 18.05.2019.
[7] https://github.com/comaeio/porosity Просмотрено: 18.05.2019.
[8] https://github.com/trailofbits/manticore/releases/tag/0.1.6 Просмотрено: 18.05.2019.
[9] https://securify.chainsecurity.com/ Просмотрено: 18.05.2019.
[10] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). ACM, New York, NY, USA, 254-269. DOI: https://doi.org/10.1145/2976749.2978309
[11] С.Э.Вельдер, М.А.Лукин, А.А.Шалыто, Б.Р.Яминов. (2011) Верификация автоматных программ. Санкт-Петербург: “Наука”
[12] Basic Spin Manual. (http://spinroot.com/spin/Man/Manual.html). Просмотрено: 18.05.2019.
[13] http://spinroot.com/spin/Man/inline.html Просмотрено: 18.05.2019.
[14] Antonio Madeira. The Dao, the Hack, the Soft Fork and the Hard Fork. (https://www.cryptocompare.com/coins/guides/the-dao-the-hack-the-soft-fork-and-the-hard-fork/). Просмотрено: 18.05.2019.
[15] Alex Hern. (08.11.2017) '$300m in cryptocurrency' accidentally lost forever due to bug. (https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether). Просмотрено: 18.05.2019.