Проверка функциональных свойств смарт-контрактов методом символьной верификации модели

Е.С.Шишкин <evgeniy.shishkin@gmail.com>
ИнфоТеКС, Научный отдел
127287, Россия, Москва, Старый Петровско-Разумовский проезд, 1/23, стр.1, этаж 2

Аннотация. В статье рассматривается подход к проверке функциональных свойств смарт-контрактов платформы Ethereum методом символьной верификации модели. Описанный подход позволяет верифицировать выполнение 3х видов свойств на трассах ограниченной длины, а также осуществлять проверку выполнимости инварианта. Описана математическая модель среды исполнения смарт-контрактов, проведена формализация выделенных видов свойств в рамках этой модели, описана процедура трансляции всего перечисленного в язык ограничений SMT-решателя. Жизнеспособность предлагаемого подхода иллюстрируется на примере верификации макетного смарт-контракта MiniDAO, упрощённой версии известного TheDAO. За несколько секунд макет находит контр пример одному нетривиальному функциональному требованию, указывая на ошибку в бизнес-логике смарт-контракта. Насколько нам известно, эта работа - одна из первых попыток описать инструмент, помогающий осуществлять формальную проверку функциональных свойств смарт-контрактов в автоматическом режиме.

Ключевые слова: символьная верификация модели; смарт-контракты; блокчейн; формальная спецификация

DOI: 10.15514/ISPRAS-2018-30(5)-16

Для цитирования: Шишкин Е.С. Проверка функциональных свойств смарт-контрактов методом символьной верификации модели. Труды ИСП РАН, том 30, вып. 5, 2018 г., стр. 265-288. DOI: 10.15514/ISPRAS-2018-30(5)-16

1. Введение

В конце 2008 г. Сатоши Накамото[1] опубликовал работу, в которой описал принципиальное устройство первой полностью децентрализованной платёжной системы под названием BitCoin [10].

Представляя собой распределенный реестр операций со счетами пользователей, система обладает уникальным сочетанием полезных свойств отказоустойчивости, неподменяемости вводимой информации, практической невозможностью цензурирования доступа, прозрачностью проводимых операций. До публикации Накамото, в мире и раньше использовали распределенные базы данных с повышенными гарантиями отказоустойчивости и аутентичности операций через использование ЭЦП, но все они опираются на наличие некоторой единой точки доверия — например администратора, который безраздельно обладает контролем над системой и сохраняемыми данными. В системе BitCoin, напротив, отсутствует доверенная сторона - пользователи доверяют исключительно описанному протоколу.

Семейство протоколов, подобных протоколу BitCoin, получило совокупное название протоколов типа блокчейн из-за их использования технологии зацепления блоков информации через вставку хэша предыдущего блок в следующий. Вскоре стало понятно, что технология блокчейн может быть использован не только как средство обмена ценностью через перевод криптовалюты, но и как распределенная вычислительная платформа по созданию надёжных отказоустойчивых нецензурируемых сервисов.

В 2015 г. состоялся первый релиз вычислительной платформы на базе блокчейн под названием Ethereum [18]. Платформа позволяет пользователям загружать в распределенную систему некую программу, описывающую желаемый бизнес-процесс. После публикации программы в блокчейне Ethereum, заинтересованные лица могут совершать вызовы в эту программу, меняя её внутреннее состояние, совершать криптовалютные переводы и, таким образом, взаимодействовать с остальными участниками процесса и программой. Программа в данном контексте называется смарт-контрактом, а пользовательские вызовы — транзакциями.

Неизменяемость смарт-контракта после публикации в блокчейне позволяет участникам бизнес-процесса «верить» в его «честность» - все возможные действия, а также уже совершенные транзакции программы видны любому желающему, поэтому не остаётся места различного рода манипуляциям с данными или бизнес-логикой. К сожалению, это же качество представляет и угрозу: если в логику смарт-контракта прокралась ошибка (случайная или специально привнесённая на этапе разработки), после записи контракта в блокчейн её невозможно исправить, а злоумышленник может воспользоваться уязвимостью в любой момент для извлечения выгоды.

[1] Это псевдоним. Настоящее имя автора до сих пор неизвестно.

Так, в 2016 г. из смарт-контракта TheDAO вывели объем криптовалюты, оцениваемый по биржевому курсу на момент инцидента в 60 млн. долларов [11]. Уязвимость была связана со спецификой поведения одной из программных конструкций языка программирования смарт-контрактов Solidity, на котором был записан TheDAO. С того момента имело место ещё несколько громких инцидентов [12] [13].

Необходимость строгой проверки смарт-контрактов на корректность на ранних стадиях разработки стала очевидна для разработчиков и заказчиков. Со временем стали появляться аудиторские компании, предоставляющие услугу ручного исследования смарт-контракта на соответствие заявленным требованиям. Зачастую аудит проводится неформально, «глазами», без строго обоснования сделанных заключений. Составляется отчёт о проведённом исследовании, где выдаются рекомендации по исправлению найденных ошибок, либо выносится заключение о надёжности смарт-контракта.

В этой связи стоит вспомнить, что смарт-контракт TheDAO также подвергался аудиту со стороны экспертов, включая самих создателей языка Solidity *и* специалиста по формальной верификации из Ethereum Foundation, однако это не помогло предотвратить печальных последствий! [14] Данный факт может служить аргументом к тезису о том, что, каков бы ни был уровень подготовки проверяющего эксперта, в его работе желательно присутствие инструмента *формальной* проверки программного артефакта. Люди склонны совершать ошибки.

Индустрии, на наш взгляд, требуется инструмент, который бы помогал проводить проверку соответствия смарт-контракта заявленной функциональной спецификации в автоматическом режиме.

В данной работе рассматривается возможность построения инструмента формальной верификации некоторых видов функциональных свойств смарт-контрактов. В качестве эталонной блокчейн платформы была выбрана платформа Ethereum, с языком программирования контрактов Solidity. Наличие эталонной платформы позволяет уже сейчас экспериментировать с реальными смарт-контрактами, при этом оставляя возможность перенести результаты работы на другую платформу, близкую по архитектуре.

Смарт-контракты в нашем случае записываются на языке программирования Sol - подмножестве языка Solidity, специально выделенном для упрощения процедуры проверки смарт-контракта, но достаточном для реализации нетривиальной бизнес-логики, в то время как функциональная спецификация задаётся либо в виде предикатов над состоянием контракта, либо в виде допустимых цепочек событий, испускаемых контрактом и взаимосвязях между этими событиями. Проверка осуществляется методом символьной верификации модели (symbolic model-checking). Модель извлекается из программы на языке Sol автоматически.

Вклад работы:

- Сформулировано несколько классов функциональных свойств, которые помогают описывать желаемое поведение смарт-контракта.
- Описан язык программирования смарт-контрактов Sol; язык выделен для целей формального анализа.
- Описана процедура кодирования программы Sol и спецификации в вид, пригодный для проверки SMT-решателем.
- Сделан макет, реализующий описанный метод на пример смарт-контракта MiniDAO - «младшего брата» смарт-контракта TheDAO. Проводится формальная проверка выполнимости нескольких функциональных требований. За несколько секунд макет находит контрпример, указывая на ошибку в логике MiniDAO.

Несмотря на то, что метод символьной верификации моделей программ — это хорошо известный способ проверки темпоральных свойств реагирующих систем, данная работа, насколько нам известно, представляет первую попытку применить этот метод к верификации функциональных свойств смарт-контрактов. Кроме этого, в литературе нам не попадалось работ, предлагающих способы специфицирования желаемого поведения смарт-контракта на формальном языке. Наша работа отчасти восполняет указанные пробелы. Результаты макетирования убеждают нас в том, что описанный подход является перспективным и заслуживает дальнейшей детальной проработки.

2. Смарт-контракты на платформе Ethereum

Объектом верификации в нашей работе являются смарт-контракты платформы Ethereum, записанные на подмножестве языка Solidity. Для краткости, мы опускаем описание работы платформы и языка программирования, ознакомиться с основами можно по материалам [18].

Язык Sol. Чтобы сделать процедуру символьной верификации программы смарт-контракта осуществимой на практике, мы выделили из языка Solidity некоторое подмножество, достаточно выразительное для осуществления интересных бизнес-сценариев, но не приводящее к астрономическому росту порождаемых состояний.

Для формирования представления о том, какие программные конструкции пользуются «спросом», в июле 2018 года было проведено небольшое исследование базы данных смарт-контрактов платформы Ethereum, снабжённых текстами программ на языке Solidity[2]. Из 27341 доступных программных текстов смарт-контрактов, только 23% используют хотя бы одну из форм циклов, и 26% используют динамические массивы. Между тем, эти программные конструкции сложнее всего подвергаются анализу, поэтому на данном этапе нашей работы было решено исключить их из рассмотрения.

[2]По базе https://etherscan.io/contractsVerified

В Solidity присутствует тип mapping который позволяет находить элемент без необходимости итерирования по коллекции. Этот тип частично снимает зависимость от циклов как средства организации вычисления. Следует вспомним также, что один из самых известных контрактов блокчейна Ethereum - TheDAO - не содержит циклов и рекурсии.

Отличия языка Sol от Solidity. 1) отсутствуют циклы for, do/while, рекурсия, взаимные вызовы функций; 2) отсутствуют механизмы динамического создания и удаления объектов через new и delete; 3) поддержка только статических массив; 4) ключевое слово var запрещено, все типы указываются явно; 5) в программе может быть единственный объект contract; 6) события не должны содержать более 4х аргументов; 7) тип address задаётся конечным множеством уникальных идентификаторов. В тексте программы запрещается напрямую указывать значение адреса; 8) нельзя вызывать методы других контрактов, а также динамически создавать экземпляры контрактов; 9) временно не поддерживаются операции со строками и битовые операции с целыми числами.

3. Модель системы

В нашей работе мы ограничимся системами, в которых существует единственный смарт-контракт, и в него производятся пользовательские вызовы. Такая аппроксимация подходит для большинства практических сценариев.

Введём такие обозначения: $\mathrm{N}_{256} \stackrel{\text{def}}{=} \{0 \dots 2^{256}-1\}$, $Addr \stackrel{\text{def}}{=} \{a_0 \dots a_n\}$, $\mathbb{B} = \{true, false\}$. Условимся, что каждой переменной состояния контракта присвоен уникальный постоянный номер. Обозначим множество всевозможных принимаемых значений переменных контракта как Val.

Пусть Ф означает множество публичных функций смарт-контракта, Е множество событий смарт-контракта.

Состояние всей системы зададим тройкой: $\sigma \stackrel{\text{def}}{=} \langle \sigma_c, b, t \rangle$ где σ_c - это состояние смарт-контракта, $b: Addr \to \mathrm{N}_{256}$ - балансы адресов системы, $t: \mathrm{N}_{256}$ - время последнего блока, относящегося к смарт-контракту. Мы обозначаем множество всевозможных состояний системы (не обязательно достижимых) через Σ, т.е. $\sigma \in \Sigma$.

Состояние смарт-контракта определим как $\sigma_c \stackrel{\text{def}}{=} \langle \sigma_{cs}, alive, eventlog \rangle$, где $\sigma_{cs}: \mathbb{N} \to Val$ задаёт текущее состояние переменных контракта (отображение уникального номера переменной в значение), $alive : \mathbb{B}$ - индикатор, указывающий является ли контракт активным, или он был удалён, $eventlog: \{\emptyset\} \cup E$-событие, сгенерированное смарт-контрактом в процессы выполнения транзакции, либо его отсутствие. Заметим, что в Solidity возможно генерировать сразу множество событий в процессе выполнения функции, и все они попадут в лог событий данной транзакции, но мы намеренно ограничиваемся только такими контрактами, в которых функция генерирует не более одного события. Это, с одной стороны, упрощает модель и процедуру проверки, с другой стороны *каждую конечную цепочку исходных событий можно при желании закодиро-*

вать одним событием. Поэтому налагаемое ограничение не является принципиальным.

Поясним природу переменной t. Транзакции в *Ethereum* выполняются узлами не по принципу FIFO, а путём группировки их в наборы, и последующей обработки сразу всего набора транзакций (последовательность выполнения транзакций из набора не определена). Эти наборы называются блоками. Каждому блоку в момент создания присваивается значение *blocktime* - время блока – момент времени, в который блок был создан. Это значение строго монотонно возрастает от блока к блоку и зачастую используется в бизнес-логике как источник относительного времени. Из программы смарт-контракта это значение можно прочитать вызовом функции *now*.

Поясним устройство множества Ф. В программе смарт-контракта определение функции состоит из названия функции, набора формальных аргументов, модификаторов, типа возвращаемого значения, и тела функции. Модификаторы могут задавать видимость (public, private), а также накладывать ограничение на возможность отправлять криптовалюту в смарт-контракт вместе с вызовом функции (payable). Принимая всё это во внимание, мы для каждой публичной функции смарт-контракта

$$functionf(arg_{0,}arg_{1,\dots},arg_n)public[payable][returns(T)]$$

ставим в соответствие функцию $f'(\sigma_i, v, s, t, p)$, где $\sigma_i \in \Sigma$ это состояние всей системы на момент выполнения функции, $v \in N_{256}$- количество криптовалюты, посылаемое вместе с вызовом, $s \in Addr$ - адрес отправителя транзакции,

$t \in N_{256}$ - время блока, в котором выполняется транзакция,

$p = (arg_0, arg_1, \dots, arg_n) \in \Pi$ - кортеж, состоящий из набора формальных параметров функции. Множество Ф задано набором таких функций.

Тело функции f' получается из тела функции f серией подстановок: вызов функции *now* заменяется на значение t, *msg.sender* заменяется на s, *msg.value* заменяется на v и т. д.

Функция f' возвращает изменённое состояние системы, которые мы обозначим за σ_{i+1}; само возвращаемое значение функции f на данный момент игнорируется - оно редко используется внешними пользователями для проверки результата выполнения функции, т.к. его неудобно отслеживать. Вместо этого чаще пользуются механизмом событий.

Введём несколько понятий, помогающих моделировать систему смарт-контракта во времени.

Определение. (Множество начальных состояний). Множество начальных состояний системы определяется как

$$I \stackrel{\text{def}}{=} \{\{\sigma_c^0, b_{0,} t_0\} \mid b_0 : Addr \to N_{256,} t_0 \in N_{256}\},\ \sigma_c^0 = \{\sigma_{cs}^0, true, \emptyset\},$$

здесь b_0 - функция, задающая балансы пользователей системы и смарт-контракта, t_0 время блока, в котором смарт-контракт был записан, σ_{cs}^0 состо-

яние переменных смарт-контракта сразу *после успешного вызова функции конструктора*.

В языке Sol мы запрещаем в конструкторе использовать любые выражения, способные привести к возникновению исключения, а также запрещается вызывать функции *transfer* и *selfdestruct*. Поэтому мы считаем, что конструктор всегда выполняется успешно и весь его побочный эффект заключается в присвоении значений переменным контракта.

Определение. (Трасса). Любая конечная последовательность состояний системы $\sigma = \sigma_0\sigma_1 \dots \sigma_{k-1}, \sigma_i \in \Sigma$ называется *трассой,* если справедливо $trace(\sigma) \stackrel{\text{def}}{=} \forall i \in \mathbb{N}, 0 \leq i < len(\sigma).\delta(\sigma_i, \sigma_{i+1})$, где $\delta(\sigma_i, \sigma_j)$ называется отношением шага (определено ниже), $\sigma_0 \in I$, и $len(\sigma)$ задаёт длину последовательности.

Множество всех возможных последовательностей состояний (не обязательно трасс) обозначим как 2^{Σ}.

В общем случае, из своего текущего состояния смарт-контракт может переходить в различные состояния, т.к. заранее неизвестно какую функцию и с какими параметрами пожелает вызвать тот или иной пользователь. Это закладывает недетерминизм в формировании любого последующего состояния трассы. Поэтому, когда мы рассуждаем про контракт, вместо одной трассы мы рассматриваем сразу множество всех возможных трасс. Только так мы можем гарантировать, что не упустим ошибочных состояний из внимания.

Определение. (Поведение). Пусть $\Sigma^{\star} \stackrel{\text{def}}{=} \{\sigma \mid \sigma \in 2^{\Sigma} \wedge trace(\sigma)\}$ - множество всех возможных трасс системы. Назовём это множество поведением системы.

Чтобы состояние σ_{i+1} «имело право» следовать за состоянием σ_iв последовательности трассы, пара (σ_i, σ_{i+1}) должна находится в определённом отношении, которое мы называем *отношением шага*. Для описания этого отношения, необходимо разобраться в природе транзакции.

Частично заданные функции. Функции $f' \in \Phi$ способны порождать исключения, т.е. прерывать выполнение функции с откатом всех ранее внесённых системных изменений, включая переводы криптовалюты, изменение значений переменных состояния контракта и т.д. Про такие функции мы говорим, что они заданы *частично*, т.е. определены не на всех значениях входных аргументов. Этот феномен, вероятно, можно было бы промоделировать, сделав отношение шага $\delta(\sigma_i, \sigma_j)$ рефлексивным. Но в этом случае, у нас бы появилось большое количество «мусорных» переходов, т.е. таких переходов, которые не ведут в новые состояния, и значит не представляют интереса в смысле проверки корректности поведения смарт-контракта. Чтобы отбросить такие переходы, мы вводим понятие предусловия для функций из Ф.

Определение. (Предусловие функции смарт-контракта). Назовём предусловием функции $f'(\sigma_i, v, s, t, p)$предикат $f_{pre}(\sigma, v, s, t, p)$ такой, что если он выполняется для заданных аргументов

$\sigma \in \Sigma,\ v \in N_{256},\ s \in Addr,\ t \in N_{256},\ p \in \Pi,$ то функция $f' \in \Phi$ определена на этих параметрах. Зададим множество $\Phi^{pre} \stackrel{\text{def}}{=} \{(f', f_{pre})\}$, т.е. каждую функцию $f' \in \Phi$ снабдим её предусловием f_{pre}.

Определение. (Отношение шага). Из состояния σ_iвозможно сделать шаг в состояние σ_{i+1}если существует набор v, s, t, p такой, что хотя бы одна функция $f' \in \Phi$ определена на значениях σ_i, v, s, t, p.

Множество всех таких пар состояний мы называем отношением шага:

$$\Delta \stackrel{\text{def}}{=} \{(\sigma_i, \sigma_j) : \exists (f', f_{pre}) \in \Phi^{pre}, v, s, t, p.\, \sigma_j = f'(\sigma_i, v, s, t, p) \wedge f_{pre}(\sigma_i, v, s, t, p)\}$$

Принадлежность к этому множеству определяется предикатом

$$\delta(\sigma_i, \sigma_j) \stackrel{\text{def}}{=} ((\sigma_i, \sigma_j) \in \Delta)$$

4. Задача верификации

Ранее была описана модель системы взаимодействия пользователей со смарт-контрактом. Сформулируем решаемую задачу проверки функциональных свойств смарт-контракта, заданных на языке спецификации.

Определение. (Задача верификации). Пусть P - предикат над трассой. Нужно установить, что $\forall \sigma \in \Sigma^{\star}.\, \sigma \vDash P$, т.е. что поведение смарт-контракта удовлетворяет P. Предикат P в этом контексте называем (формальной) спецификацией на смарт-контракт.

В зависимости от вида свойства (свойство конкретных состояний или событийное свойство), предикат P принимает либо одно состояние $P : \Sigma \to \mathbb{B}$ либо цепочку длины не более k $P : \Sigma_k^{\star} \to \mathbb{B}$.

Здесь и далее $\Sigma_k^{\star} = \{\sigma \in \Sigma^{\star} \mid len(\sigma) \leq k\}$, где $len(\sigma)$ задаёт длину последовательности.

Сформулируем виды функциональных свойств, которые мы хотим уметь проверять, в терминах изложенной ранее модели.

Определение. (Инвариант). Предикат $P : \Sigma \to \mathbb{B}$ называется инвариантом системы, если

$$\forall \sigma_0 \in I.\, P\ \sigma_0 \wedge \forall \sigma_i, \sigma_j \in \Sigma. \left(\delta(\sigma_i, \sigma_j) \wedge P\ \sigma_i\right) \to\ P\ \sigma_j$$

Определение. (Свойство на трассах длины k).

Предикат $P : \Sigma \to \mathbb{B}$ называется свойством трассы длины k, если

$$\forall \sigma^{\star} \in \Sigma_k^{\star}, i \in \mathbb{N}.\, i \leq len(\sigma^{\star}) \to P(\sigma_i^{\star})$$

Определение. (Паттерн возникновения событий). Свойства этого класса задаются предикатами над трассами длины не более k , т.е. $P : \Sigma_k^{\star} \to \mathbb{B}$. Рассмотрим одно из них, остальные определяются по аналогии. Если произошло событие $E_1(p_0, \ldots, p_n)$, после которого в какой-то момент происходит событие $E_2(m_0, \ldots, m_k)$, то между ними не должно возникнуть событие $E_3(n_0, \ldots, n_m)$, т.е.

$$P(\sigma^{\star}) = \exists i,j \in N.\, \sigma_i^{\star}[eventlog] = E_1 \wedge \sigma_j^{\star}[eventlog] = E_2 \wedge i < j \leq len(\sigma_i) \rightarrow$$
$$\forall k \in N, i < k < j.\, \sigma_k^{\star}[eventlog] \neq E_3$$

Если есть дополнительные зависимости между параметрами событий $p_0,\ldots,p_n,m_0,\ldots,m_k,n_0,\ldots,n_m$, то они добавляются к указанному предикату.

Определение. (Возможность выполнения транзакции). Свойства этого класса задаются предикатом над трассами длины не более k, т.е. $P{:}\,\Sigma_k^{\star} \rightarrow \mathbb{B}$. Рассмотрим одно из них, остальные определяются по аналогии. Если произошло событие $E_1(p_0,\ldots,p_n)$, после которого в какой-то момент происходит событие $E_2(m_0,\ldots,m_q)$, то между ними всегда возможно успешно выполнить $f'(\sigma,v,s,t,p)$.

$$P(\sigma^{\star}) = \forall i,j \in N.\ \sigma_i^{\star}[eventlog] = E_1(p_{0,}\ldots,p_n) \wedge$$
$$\sigma_j^{\star}[eventlog] = E_2(m_{0,}\ldots,m_q) \wedge$$
$$i < j \leq len(\sigma^{\star}) \rightarrow \forall k \in N, i < k < j.\, f_{pre}(\sigma_k^{\star},v,s,t,p)$$

Если есть дополнительные зависимости между параметрами событий $p_0,\ldots,p_n,m_0,\ldots,m_q$и параметрами вызываемой функции σ,v,s,t,p , то они добавляются к указанному предикату.

5. Конструирование модели смарт-контракта

Построить модель смарт-контракта и спецификации означает задать такой набор объектов: $(\Phi^{pre},E,Addr,I,k,\Sigma_k^{\star},P)$, где k означает максимальную длину анализируемой трассы. Обсудим процедуру построения каждого из этих объектов для какого-то заданного смарт-контракта.

Множество $Addr$. В системе Ethereum множество *Addr* совпадает с множеством $\{0\ldots 2^{160}-1\}$, но для целей символьной верификации такое множество оказывается слишком большим. Дело в том, что его размер влияет на количество достижимых состояний: параметр s в функции $f'(\sigma_i,v,s,t,p)$ выбирается из множества *Addr*, а значит чем оно крупнее, тем больше возможностей выбора. Поэтому, в верифицируемых моделях множество адресов $\{a_0\ldots a_n\}$ мы стараемся выбирать как можно меньшего размера, но такого, чтобы было возможно пройти по всем принципиальным сценариям выполнения. Оптимальный размер этого множества можно определить только исходя из понимания бизнес логики конкретного контракта и на данный момент никак не автоматизировано. По умолчанию, полагаем $Addr = \{noAddr, addr_0, addr_1, addr_2, contractAddr\}$.

Элемент *noAddr* соответствует отсутствию адреса - то, что в коде обычно обозначается как address(0). Элемент *contractAddr* задаёт адрес анализируемого контракта.

Множество Φ^{pre}. Строится автоматически: для каждой *публичной* функции f' смарт-контракта (кроме конструктора) строится предикат f_{pre} над набором

переменных σ, v, s, t, p. Это делается методом символьного исполнения кода функции. В местах кода, где происходит вызов другой функции, мы производим встраивание тела вызываемой функции.

Потенциально опасные конструкции. Конструкции, способные привести к возникновению исключения, на данный момент: операция деления целых чисел, взятие остатка от деления; функции mulmod, addmod; отправка криптовалюты через transfer; функции assert, require**,** revert; оператор throw; вызов нерayable функции с параметром $v > 0$; совпадение адреса s с адресом смарт-контракта *contractAddr;* попытка вызова функции смарт-контракта, который был удалён, т.е. $\sigma_{cs}[alive] = false$

Этот список будет расширяться впоследствии. Так как в языке Sol нет циклов и рекурсии (инструмент верификации проводит синтаксическую проверку программы перед построением модели), то символьное исполнение кода функции гарантировано завершается, и набор предикатов f_{pre} будет получен за конечное время.

Множество E. Строится автоматически из списка определённых в смарт-контракте событий. События, которые не используются ни в одной из функций, отбрасываются.

Множество I. Множество начальных состояний смарт-контракта.

Значение k. Максимальная длина анализируемой трассы. Задаётся в явном виде пользователем. Если проверяется инвариант, этот параметр игнорируется.

Множество $\Sigma_k^\star$. Множество задаётся неявно, через построение системы ограничений на наборе состояний $\sigma_{[0..k-1]}$и наборе параметров (v, s, t, p), где i-й набор соответствует параметрам, передаваемым в i-й по счёту вызов одной из функций смарт-контракта.

Определим отношение перехода:

$$transition(\sigma_i, \sigma_{i+1}) = \bigvee_{0 \le j < n} (f_j^{pre}(\sigma_i, v_i, s_i, t_i, p_i) \wedge \sigma_{i+1} = f_j'(\sigma_i, v_i, s_i, t_i, p_i)),$$

где $n = |\Phi^{pre}|$

Определим путь между состояниями: $path(\sigma_{[0..k]}) = \bigwedge_{0 \le i < k} transition(\sigma_i, \sigma_{i+1})$

Путь отличен от трассы тем, что первое состояние в последовательности не обязано быть начальным. Путь длины 0 содержит единственное состояние, в нём не совершается ни единого перехода.

Введём дополнительно требование на монотонность времени $T = \bigcup_{i=0..k-1} \{t_i < t_{i+1}\}$, требование на начальные состояния: $I(\sigma_0)$, требование невозможности вызова функций смарт-контракта с адреса самого смарт-контракта: $NoSelfCall = \bigcup_{i \in \{0..k-1\}} \{s_i \neq contractAddr\}$, требование невозможности вызова функций смарт-контракта с адреса *noAddr*:

$$NoAddrCall = \bigcup_{i \in \{0..k-1\}} \{s_i \neq noAddr\}$$

В этом случае, кат $I(\sigma_0) \wedge T \wedge NoSelfCall \wedge NoAddrCall \wedge path(\sigma_{[0..k-1]})$ описывает ограничения, выполнение которых в контексте SMT-решателя задаёт присваивания переменным $\sigma_{[0..k-1]}$ и (v, s, t, p), неявно «генерируя» множество $\Sigma_k^\star$.

Предикат P. Предикат формируется путём трансляции функционального свойства в предикат первого порядка, как это было описано в разделе.

Таким образом, из программы на языке Sol возможно автоматически извлечь модель, пригодную для передачи в SMT-решатель. Единственное, что требуется от пользователя, это указать функциональную спецификацию *P*, длину трассы *k*, количество элементов в множестве *Addr*.

6. Алгоритм проверки спецификаций

В этом разделе мы описываем устройство алгоритмов проверки функциональных свойств. Пусть $P(\sigma)$ задаёт проверяемое свойство. Выражение $SAT(e, Vars)$ означает, что утверждение *e* проверяется SAT/SMT решателем на выполнимость, т.е. ищется такой набор присваиваний для переменных из *Vars* такой, что вся логическая формула *e* становится истиной. Если такое присваивание удаётся найти, то функция возвращает *true*. Иначе возвращает *false*. Результат *unknown* не рассматривается, т.к. мы находимся в рамках полностью разрешимых теорий для которых этот результат означает нехватку выделенного времени на поиск решения. Предикат $path(\sigma_{[0..n]})$ определяется, как было указано ранее. Каждый из алгоритмов возвращает *true*, если свойство $P(\sigma)$ выполняется, иначе *false*. В используемом псевдокоде выражение $Vars = \{p_i : t_i\}$ означает, что для каждого p в набор пропозиционных переменных решателя добавляется переменная типа *t*, либо набор однотипных переменных в случае массива.

Некоторые из этих алгоритмов уже публиковались ранее. Так, алгоритм проверки выполнимости свойства на пути длины k подробно рассмотрен в различных вариациях в [20], а алгоритм проверки инварианта хорошо известен. Тем не менее, чтобы сделать текст самодостаточным, мы приводим псевдокод всех используемых нами верифицирующих алгоритмов в одном месте.

Алгоритм 1. Проверка выполнимости инварианта

```
1: Vars = {σ[0,1]:Σ, v[0,1]:N256, s[0,1]:Addr, t[0,1]:N256, p[0,1]:Π}
2: if (SAT (I(σ0) ∧ ¬P(σ0), Vars)) {
3:     print s0
4:     return false
5: }
6: if (SAT (P(σ0) ∧ δ(σ0,σ1) ∧ ¬P(σ1), Vars)) {
7:     print fpre, σ0, σ1
8:     return false
9: }
10:return true
```

Обоснование алгоритма 1. В строке 2 мы проверяем выполнимость $P(\sigma)$ во всех начальных состояниях. Если проверка в строке 2 прошла успешно, мы переходим к проверке индуктивного шага: предполагая, что $P(\sigma)$ выполняется в каком-либо σ_0 (не обязательно начальном) и из него можно перейти в другое состояние σ_1, то $P(\sigma)$ будет выполняться и в σ_1.

Алгоритм 2. Проверка выполнимости свойства на пути длины k

```
1: Vars = {σ[0..k-1]:Σ, v[0..k-1]:N256, s[0..k-1]:Addr, t[0..k-1]:N256,
           p[0..k-1]:Π}
2: i = 0
3: while (i < k) do {
4:   if (SAT (I(σ0) ∧ path(σ[0..i]) ∧ ¬P(σi), Vars)) {
5:             print σ[0..i]
6:             return false
7:   }
8:   i = i + 1
9: }
10:return true
```

Обоснование алгоритма 2. Мы хотим убедиться, что для любой трассы *длины не более k* свойство $P(\sigma)$ будет выполняться во всех состояниях этой трассы. Для проверки этой гипотезы, мы последовательно, начиная с $i = 0$ (в этом случае мы проверяем отдельные точки – начальные состояния), просим решатель найти хотя бы один пример, в котором бы гипотеза нарушалась, и делаем так вплоть до $i = k - 1$, после чего алгоритм завершается.

Алгоритм 3. Проверка паттерна возникновения событий

```
1: Vars = {σ[0..k-1]:Σ, v[0..k-1]:N256, s[0..k-1]:Addr,
           t[0..k-1]:N256, p[0..k-1]:Π, m,n,q:N256}
2: i = 3
3: while (i < k) do {
4:     if (SAT (I(σ0) ∧ path(σ[0..i]) ∧  σ^cs_m[eventlog] = E1 ∧
            σ^cs_n[eventlog] = E2 ∧ (m < n) ∧ (q > m) ∧ (q < n) ∧
            σ^cs_q[eventlog] = E3, Vars) {
5:           print σ[0..i]
6:           return false
7:     }
8:     i = i + 1
9:}
10:return true
```

Обоснование алгоритма 3. Мы хотим убедиться, что любая трасса *длины не более k* удовлетворяет условию: если в трассе возникло событие E_1, после которого возникло E_2, то между этими событиями не возникает E_3 (гипотеза). Это условие задаёт паттерн возникновения событий в трассах смарт-контракта, и могло бы быть записано на языке регулярных выражений: $.* E_1(\neg E_3) * E_2.*$, при ограничении длины входных строк параметром k.
Проверка этого свойства делается путём последовательного, начиная с $i = 3$, поиска примера, опровергающего гипотезу. Поиск прекращается после того, как все трассы длины до k были проверены (условие в строке 3).

Алгоритм 4. Проверка возможности осуществления вызова функции

```
1: Vars = {σ[0..k-1]:Σ, v[0..k-1]:N256, s[0..k-1]:Addr, t[0..k-1]:N256,
                    p[0..k-1]:Π, m,n,q:N256}
2: i = 3
3: while (i < k) do {
4:     if (SAT (I(σ0) ∧ path(σ[0..i]) ∧  σ^cs_m[eventlog] = E1 ∧
            σ^cs_n[eventlog] = E2 ∧ (m < n) ∧ (q > m) ∧
            (q < n) ∧ ¬f_pre(σq, vq, sq, tq, pq), Vars))  {
5:                 print σ[0..i], vq, sq, tq, pq
6:                 return false
7:     }
8:     i = i + 1
9: }
10:return true
```

Обоснование алгоритма 4. Мы хотим убедиться, что любая трасса *длины не более k* удовлетворяет условию: если в трассе возникло событие E_1, после которого возникло E_2, то строго между этими событиями во всех состояниях возможно успешно выполнить функцию смарт-контракта $f(\sigma, v, s, t, p)$. Успешность вызова функции с заданными параметрами описывается выполнением предиката $f_{pre}(\sigma, v, s, t, p)$ (гипотеза).

Проверка этого свойства делается путём последовательного, начиная с $i = 3$, поиска примера, опровергающего гипотезу, то есть примера такой трассы, чтобы строго между E_1 и E_2 в каком-то из состояний предикат $f_{pre}(\sigma, v, s, t, p)$ не выполнялся. Взаимное расположение событий друг относительно друга задаётся с помощью отношений номеров состояний трассы, в которых соответствующие события возникли. Поиск прекращается после того, как все трассы длины до k были проверены (условие в строке 3). Трассы длины меньше 4х элементов слишком короткие, чтобы их проверять.

7. Описание макетного образца

В целях апробирования описанной методики, мы запрограммировали смарт-контракт MiniDAO - упрощённый вариант смарт-контракта TheDAO, и попробовали отыскать контр пример, демонстрирующий нарушение описанного в спецификации требования.

Программа смарт-контракта MiniDAO записана на языке Sol, а проверяемые свойства закодированы в виде соответствующих предикатов. Мы оттранслировали вручную оба артефакта в язык ограничений SMT-решателя и провели поиск контр примеров, фиксируя длительность проверки разного типа свойств с различными значениями параметров модели.

Мы кратко опишем логику работы MiniDAO и сформулируем несколько утверждений, которые будут служить частичной спецификацией на этот контракт.

7.1 Смарт-контракт MiniDAO

MiniDAO - это смарт-контракт, реализующий возможность привлечения криптовалютных инвестиций в новый проект. В смарт-контракте предусмотрено два вида участников: инвестор и подрядчик. Инвестор - это тот, кто вносит средства в фонд смарт-контракта, и далее голосует «за» либо «против» поддержки предложенного кем-то проекта. Подрядчик - это сторона, которая предлагает новый проект и занимается его реализацией, возвращая сумму инвестиций и дивиденды инвесторам через смарт-контракт. Интерфейс смарт-

```
interface ERC20Interface { /* Стандартный интерфейс ERC20 */ }
interface MiniDAOInterface {
    function deposit() public payable;
    function vote(uint proposalId, bool supportsProposal)
public;
    function refund() public;
    function propose(address recepient, uint amount,
                     string text) public;
    function execute_proposal() public;
    event Voted(address voter, uint proposalID,
                bool supportsProposal);
    event Refunded(address investor, uint tokens);
    event Deposited(address investor, uint tokens);
    event ProposalAdded(uint amount, uint proposalID);
    event ProposalExecuted(uint proposalID);
    event ProposalRejected(uint proposalID);
}
contract MiniDAO is MiniDAOInterface, ERC20Interface { ... }
```

контракта приведён на рис.1. Полный текст смарт-контракта доступен по ссылке[3].

.Рис. 1. Интерфейс смарт-контракта MiniDAO
Fig. 1. MiniDAO smart-contract interface

Чтобы дать инвестору возможность вывести свои средства из смарт-контракта MiniDAO, реализован метод refund(). Возврат осуществится только в том случае, если инвестор не голосовал за заявку. Смарт-контракт отправляет на адрес инвестора количество эфира пропорционально количеству токенов на внутреннем балансе инвестора, т.е. ровно столько, сколько инвестор вложил в фонд MiniDAO.

Инвесторы могут переводить на счета других инвесторов свои токены miniDAO. Это реализуется через поддержку стандартного интерфейса токенов ERC20.

Предположим, мы заинтересованы привлечь как можно больше инвесторов в наш фонд miniDAO. Чтобы минимизировать страх инвестора потерять свои деньги, мы заявляем, что наш смарт-контракт обладает таким функциональным свойством: «*Если вы не проголосовали ни за одно инвестиционное предложение, то вы всегда сможете забрать свои средства обратно*». В качестве аргумента мы указываем на программный код функции refund, которая отвечает за возврат средств инвестору.

```
  function refund() public {
        address sender = msg.sender;
        uint tokens = balance[sender];
        require (isVoted[0][sender] ==
false);
        require (tokens > 0);
        require (DAO_tokens_emitted >=
tokens);
        DAO_tokens_emitted -= tokens;
        balance[sender] = 0;
        sender.transfer(tokens *
DAO_token_price);
        emit Refunded(sender, tokens);
    }
```

Функция выглядит просто и убедительно. Однако свойство, тем не менее, не выполняется.

Атака большинства. Рассмотрим сценарий возможной атаки. Два инвестора вложили в инвест фонд MiniDAO криптовалюты суммой на X и Y токенов соответственно. Предположим, что появляется третий инвестор, который вкладывает криптовалюты объёмом на 2∗ (X+Y) токенов. У этого инвестора получается большинство голосов (2/3≈66%) при принятии решения о заявке.

[3]https://bitbucket.org/unboxed_type/minidao/src/master/contracts/MiniDAO.sol

Так как инвестору не запрещено быть подрядчиком, то этот инвестор регистрирует собственную заявку с указанием своего адреса и необходимую сумму в размере 3∗ (X+Y) токенов. Далее, из-за того, что его голос - решающий, он голосует за собственную же заявку и после этого вызывает *execute_proposal*. Все средства из фонда MiniDAO перейдут на счёт этого инвестора-злоумышленника, включая те средства, которые внесли первые два инвестора. Получается очевидное нарушение функционального требования: первый и второй инвестор не голосовали, но свои деньги они уже точно не вернут.

Атака большинства (в несколько другой форме) описана в оригинальной работе TheDAO [14]. *Предположим, что мы не знаем про атаку большинства и хотим попросить верифицирующий инструмент проверить, выполняется ли заявленное функциональное свойство.*

7.2 Функциональные свойства MiniDAO

В качестве примера сформулируем три свойства, которые мы хотели бы проверить с помощью инструмента верификации.

Свойство DepositedNotVotedRefund. Если инвестор с адресом inv вносил депозит, но при этом ни разу не голосовал за какую-либо заявку, он всегда сможет вернуть свои средства путём вызова функции refund.

$$DepositNotVotedRefund(\sigma) \stackrel{\text{def}}{=} \exists i, inv, s, 1 \le i < k.$$
$$\sigma_{cs}^{i}[logs] = Deposited(inv, s) \ \wedge \ \exists j, inv_1, s_1, id, 1 \le j < k.$$
$$\sigma_{cs}^{j}[logs] = ProposalAdded(inv_1, s_1, id) \wedge \forall n, id_1, 1 \le n < k.$$
$$\sigma_{cs}^{n}[logs] \ne Voted(inv, id_1, True) \wedge \sigma_{cs}^{n}[logs] \ne Voted(inv, id_1, False) \wedge$$
$$\sigma_{cs}^{n}[logs] \ne Refund(inv) \wedge \sigma_{cs}^{n}[logs] \ne Transfer(inv) \rightarrow$$
$$\forall m, i \le m < k. \exists v, t, p. refund_{pre}(\sigma_m, v, inv, t, p)$$

В данном случае мы используем язык логики первого порядка, так как его легче всего оттранслировать в язык ограничений SMT-решателя.

Свойство InvDaoBalanceEquTokens. В любом достижимом состоянии системы, сумма остатков токенов miniDAO на балансах пользователей всегда должна быть равна количеству эмитированной криптовалюты, т.е. для всех достижимых состояний системы,

$$InvDaoBalanceEquTokens(\sigma_k)$$
$$\stackrel{\text{def}}{=} \sum_{i \in Addr} \sigma_{cs}^{k}[daoBalance[i]] = daoTokensEmitted$$

Свойство RejectedNotExecuted. Инвестиционное предложение, которые было отклонено, не может получить перечисление криптовалюты из смарт-

контракта.

$$RejectedNotExecuted(\sigma) \stackrel{\text{def}}{=} \forall n. \exists i, j \in N.$$
$$\sigma_{cs}^{i}[logs] = ProposalAdded(n, amount) \wedge \sigma_{cs}^{j}[logs] = ProposalRejected(n) \wedge$$
$$i < j \rightarrow \forall k, i < k < j. \sigma_{cs}^{k}[logs] \neq ProposalExecuted(n)$$

7.3 Поиск ошибок в контракте MiniDAO

Мы проверяем смарт-контракт методом символьной верификации модели. Согласно введённому ранее определению, задать модель означает задать набор $(\Phi^{pre}, E, Addr, I, k, \Sigma_k^{\star}, P)$, после чего мы можем выполнить один из алгоритмов проверки функционального свойства.

Разрабатываемый инструмент будет строить указанные объекты автоматически, по исходному коду контракта, за исключением задания количества участников *Addr*, длины трассы *k* и проверяемого свойства *P*.

Так как инструмент проверки находится в стадии разработки, мы провели построение указанных объектов вручную, получив на выходе модель системы исполнения смарт-контракта и проверяемого функционального свойства, закодированную на языке SMT-решателя. В этой работе мы использовали SMT-

0. NoEvent
1. Deposited, sender = addr4, tokens = 2976
2. Deposited, sender = addr2, tokens = 1672
3. ProposalAdded, sender = addr4, amount = 4648, proposalID = 1
4. Voted, sender = addr4, proposalID = 1, supports=1
5. ProposalExecuted, proposalID = 1

step = 5, investor = addr2

решатель Z3 компании Microsoft [16].

Рис. 2. Трасса, опровергающая функциональное свойство DepositedNotVotedRefund.
Fig. 2. The counter-example for DepositedNotVotedRefund property.

После трансляции исходного кода смарт-контракта MiniDAO в представление SMT-решателя и проведения нескольких оптимизаций, была получена модель, проверка которой за несколько секунд (см. таблицу) синтезировала контр примеры, указывая на ошибку в бизнес-логике контракта.

Так, проверка свойства *DepositedNotVotedRefund* выявила контр пример, представленный на Рис.2. Первое событие NoEvent означает начальное состояние смарт-контракта. Указан набор событий, который ведёт к ошибочному состоянию. Инвестору addr4 не удаётся вызвать refund после события 5.

Проверка свойства *InvDaoBalanceEquTokens* также выявила ошибку. В первоначальной логике смарт-контракта MiniDAO, метод vote обнулял количество токенов на счёту инвестора, выполняя присваивание: daoBalance[msg.sender] = 0. Ошибка была устранена, после чего свойство прошло проверку.

Проверка свойства *RejectedNotExeceted* прошла успешно: не было обнаружено ни одной трассы, опровергающей сформулированное свойство.

Оптимизация модели. Известно, что SAT/SMT-решатели весьма чувствительны к изменению параметров проверяемой системы. После первоначальной трансляции исходного кода смарт-контракта MiniDAO в представление SMT-решателя, без оптимизаций, нам не удалось добиться от решателя результата за приемлемое время, поэтому было решено провести оптимизацию модели.

Оптимизацией мы называем уменьшение мощности множества возможных значений различных параметров системы (σ_i, v, s, t, p). Так, например, вместо множества N_{256} для v и t было положено множество N_{16}. Ещё одним оптимизирующим приёмом можно считать выбор начальной длины трассы, с которой начинается анализ. Очевидно, что при неудачном выборе размеров множеств и начальной длины трассы, есть вероятность пропустить сценарии, ведущие к ошибке.

8. Результаты работы макета

Изменяя размеры множеств, из которых выбираются значения состояний модели, а также длину пути, была составлена таблица с результатами измерений времени работы решателя на соответствующей модели, см. табл. 1.

Следует иметь ввиду, что эти результаты могут давать лишь *приблизительное представление* о том, в каких пределах лежит время поиска контр примера в зависимости от параметров модели: алгоритм, на который опирается SAT/SMT-решатель чувствителен к любым изменениям в модели, а в некоторых сценариях может на одной и той же модели с одинаковыми параметрами выдавать разные результаты, по скорости и контр примеру (недетерминизм).

Эксперименты проводились на машине Intel Core i7-4770, 4 ядра 3.4 GHz , 32 GB RAM под управлением ОС Linux Fedora 28 x64, запущенная в виртуальной машине VirtualBox 5.1.24, под управлением ОС Windows 7. Использовался SMT-решатель Z3 версии 4.7.1, 64 bit, модель записана с использованием расширения Z3Py для Python.

9. Обзор схожих работ

Исследованию различных методов поиска и предотвращения уязвимостей в смарт-контрактах посвящено множество работ, среди прочих [1] [2] [3] [4] [5] [6] [7] [8].

В работе [2] авторы приводят исчерпывающий список известных уязвимостей языка Solidity и виртуальной машины EVM, детально описывается механизм известной атаки на контракт TheDAO. Работа не стремится предложить какие-либо решения для верификации смарт-контрактов, но отлично освещает все известные на момент написания уязвимости языка и платформы.

Табл. 1. Левая таблица - длительность проверки свойства DepositedNotVotedRefund. Результат >N означает, что мы прервали работу решателя по истечении N секунд. Средняя таблица - длительность проверки свойства RejectedNotExeceted. Правая таблица - длительность проверки свойства InvDaoBalanceEquTokens.

Table 1. Left section: a duration of checking DepositedNotVotedRefund property. Abbr. «> N» denotes the fact that we have stopped the SMT solver after N seconds. Middle section: a duration of checking RejectedNotExecuted property. Right section: a duration of checking InvDaoBalanceEquTokens property.

Начальная длина трассы	Ширина целого числа, в битах	Время проверки, сек
6	16	34
8	16	7
12	16	474
6	32	9
8	32	115
12	32	> 660

Начальная длина трассы	Ширина целого числа, в битах	Время проверки, сек
6	16	12.9
8	16	9.4
6	32	46
8	32	30

Ширина целого числа, в битах	Время проверки, сек
16	243
24	997

Работа [3] предлагает способ статического анализа смарт-контрактов, записанных на языке Solidity* (упрощённая версия Solidity без циклов), с помощью трансляции в язык F* вместе со специально введёнными типами (монада Eth), помогающими отслеживать отсутствие должной обработки ошибок после вызова функций, а также обработке результата функции send. Предложенный в статье подход нацелен на устранение типовых ошибок в реализации смарт-контрактов на языке Solidity, при этом высокоуровневые функциональные свойства никак не проверяются.

В [1] исследуется возможность программирования смарт-контрактов на функциональном языке Idris; используя выразительную силу системы типов данного языка, авторы вводят несколько алгебраических типов, помогающих устранить определённый класс операционных ошибок на этапе компиляции. Более того, представлен backend для компилятора Idris, транслирующий код контрактов в исполняемый EVM байткод. Аналогично, в работе никак не адресован вопрос проверки высокоуровневых функциональных свойств смарт-контракта.

Работы [4], [7] описывают инструменты статического анализа смарт-контрактов, основанных на символьном исполнении программы смарт-контракта с поиском условий, выполнение которых влечёт переход управления на потенциально опасные участки кода, а также поиску в исходном коде смарт-контракта паттернов некоторых типовых уязвимостей.

В работе [6] описывается инструмент статического анализа смарт-контрактов ZEUS, способный искать нарушение заданных пользователем политик. Политики описываются как утверждения о состоянии переменных контракта с поддержкой арифметики (пропозиционные высказывания с арифметикой). Кроме этого, ZEUS ищет типичные уязвимости в программах на языке Solidity. В отличие от нашей работы, ZEUS не способен анализировать логику работы контракта, растянутую во времени, а нацелен исключительно на свойства безопасности достижимых состояний (safety).

Пожалуй, первой попыткой использовать интерактивную среду построения доказательств для проверки корректности смарт-контракта является работа [5]. Автор закодировал семантику инструкций EVM сначала в среде Coq, а затем и в Isabelle, что позволяет верифицировать свойства смарт-контрактов на уровне байт-кода EVM. Предоставляя наивысшие гарантии на корректность проверенного артефакта, подход обладает недостатками: пользователь должен иметь специальную квалификации и сам процесс построения доказательств может занять длительное время.

В работе [8], автор формализует часть языка Solidity, описывая операционную семантику некоторых конструкций. Формализация ведётся в среде построения доказательств Coq.

10. Заключение

В статье описан подход к проверке некоторых видов функциональных свойств подмножества языка Solidity методом символьной верификации модели. Описана модель исполнения смарт-контракта, позволяющая проверять функциональные свойства, заданные 4-мя возможными способами. Впервые представлен способ описания поведения смарт-контракта, заданный цепочкой генерируемых событий.

На примере смарт-контракта MiniDAO показана практическая применимость описываемого подхода к верификации. В качестве дальнейших работ по данной теме мы хотели бы особенно выделить такие направления:

1. Описание формального языка спецификации поведения смарт-контракта, основанный на событиях, генерируемых смарт-контрактом. Сейчас такие свойства записываются на языке логики первого порядка - это неудобно, и требует внимания.
2. Оптимизация представлений модели на языке SMT решателя. От оптимальности представления системы зависит быстрота поиска контр примера.

Список литературы

[1]. Pettersson J., Edström R. Safer smart contracts through type-driven development. Master's thesis, Chalmers University of Technology, Department of Computer Science and Engineering, Sweden, 2016.
[2]. Atzei N., Bartoletti M., Cimoli T. A survey of attacks on ethereum smart contracts (sok). Lecture Notes in Computer Science, vol. 10204, 2017, pp. 164-186.
[3]. Bhargavan K. et al. Formal verification of smart contracts. In Proc. of the 2016 ACM Workshop on Programming Languages and Analysis for Security, 2016, pp. 91-96.
[4]. Luu L. et al. Making smart contracts smarter. In Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 254-269.
[5]. Hirai Y. Defining the ethereum virtual machine for interactive theorem provers. Lecture Notes in Computer Science, vol. 10323, 2017, pp. 520-535.
[6]. Kalra S. et al. Zeus: Analyzing safety of smart contracts. In Proc. of the Network and Distributed System Security Symposium, 2018.
[7]. Mueller B. Smashing Ethereum Smart Contracts for Fun and Real Profit. In Proc. of the 9th Annual HITB Security Conference, 2018.
[8]. Zakrzewski J. Towards verification of Ethereum smart contracts: a formalization of core of Solidity, Lecture Notes in Computer Science, vol. 11294, 2018, pp. 229-247.
[9]. Solidity github. Доступно по ссылке: https://github.com/ethereum/solidity/blob/develop/docs/grammar.txt, дата обращения 20.11.2018.
[10]. Nakamoto S. Bitcoin. A peer-to-peer electronic cash system. 2008. Доступно по ссылке: https://bitcoin.org/bitcoin.pdf, дата обращения 20.11.2018, дата обращения 20.11.2018..
[11]. Gideon Greenspan. Smart contracts and the dao implosion. 2016. Доступно по ссылке: https://www.multichain.com/blog/2016/06/smart-contracts-the-dao-implosion/, дата обращения 20.11.2018.
[12]. S. Palladino. The Parity Wallet Hack Explained, https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7, дата обращения 20.11.2018.
[13]. J.D. Alois. Ethereum Parity Hack May Impact ETH 500,000 or $146 Million. 2017. Доступно по ссылке: https://www.crowdfundinsider.com/2017/11/124200-ethereum-parity-hack-may-impact-eth-500000-146-million/, дата обращения 20.11.2018.
[14]. Jentzsch C. Decentralized autonomous organization to automate governance. 2016. Доступно по ссылке: https://download.slock.it/public/DAO/WhitePaper.pdf, дата обращения 20.11.2018.
[15]. Е. Шишкин. О построении среды для конструирования гарантированно надёжных смарт-контрактов. Материалы конференции РусКрипто'2018, 2018. Доступно по ссылке: https://www.ruscrypto.ru/resource/archive/rc2018/files/03_Shishkin.pdf, дата обращения 20.11.2018.
[16]. De Moura L., Bjørner N. Z3: An efficient SMT solver. Lecture Notes in Computer Science, vol. 4963, 2008, pp. 337-340.
[17]. Manna Z., Pnueli A. The temporal logic of reactive and concurrent systems: Specification. Springer-Verlag, 1992, 427 p/
[18]. Ethereum project. Доступно по ссылке: https://www.ethereum.org/, дата обращения 20.11.2018.
[19]. Szabo N. Smart contracts. 1994. Доступно по ссылке: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOT

winterschool2006/szabo.best.vwh.net/smart.contracts.html, дата обращения 20.11.2018.
[20]. Sheeran M., Singh S., Stålmarck G. Checking safety properties using induction and a SAT-solver. Lecture Notes in Computer Science, vol. 1954, 2000, pp. 127-144.

Verifying functional properties of smart contracts using symbolic model-checking

E.S. Shishkin <evgeniy.shishkin@gmail.com>
Infotecs, Scientific Research Department
1/23, Petrovsko-Razumovskiy Proezd, Moscow, 127287, Russia

Abstract. We describe our efforts towards building a tool that automatically verify high-level functional properties of Ethereum smart contracts against its formal specification that can be given using four different methods: an invariant over contract state or three different types of trace properties. A model of runtime system, the source code of smart contract together with its specification is translated into SMT-solver formula and checked for counter example. We tested the method on simplified version of notorious TheDAO smart-contract, called Mini-DAO. Our proof-of-concept tool was able to find a functional property violation of MiniDAO in just several seconds. We believe that the proposed method is indeed useful and deserves deeper investigation.

Keywords: symbolic model-checking; smart contracts; blockchain; formal specification;

DOI: 10.15514/ISPRAS-2018-30(5)-16

For citation: Shishkin E.S. Verifying functional properties of smart contracts using symbolic model checking. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 5, 2018, pp. 265-288 (in Russian). DOI: 10.15514/ISPRAS-2018-30(5)-16

References

[1]. Pettersson J., Edström R. Safer smart contracts through type-driven development. Master's thesis, Chalmers University of Technology, Department of Computer Science and Engineering, Sweden, 2016.
[2]. Atzei N., Bartoletti M., Cimoli T. A survey of attacks on ethereum smart contracts (sok). Lecture Notes in Computer Science, vol. 10204, 2017, pp. 164-186.
[3]. Bhargavan K. et al. Formal verification of smart contracts. In Proc. of the 2016 ACM Workshop on Programming Languages and Analysis for Security, 2016, pp. 91-96.
[4]. Luu L. et al. Making smart contracts smarter. In Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 254-269.
[5]. Hirai Y. Defining the ethereum virtual machine for interactive theorem provers. Lecture Notes in Computer Science, vol. 10323, 2017, pp. 520-535.

[6]. Kalra S. et al. Zeus: Analyzing safety of smart contracts. In Proc. of the Network and Distributed System Security Symposium, 2018.
[7]. Mueller B. Smashing Ethereum Smart Contracts for Fun and Real Profit. In Proc. of the 9th Annual HITB Security Conference, 2018.
[8]. Zakrzewski J. Towards verification of Ethereum smart contracts: a formalization of core of Solidity, Lecture Notes in Computer Science, vol. 11294, 2018, pp. 229-247.
[9]. Solidity github. Available at: https://github.com/ethereum/solidity/blob/develop/docs/grammar.txt, accessed 20.11.2018.
[10]. Nakamoto S. Bitcoin. A peer-to-peer electronic cash system. 2008. Available at: https://bitcoin.org/bitcoin.pdf, accessed 20.11.2018, accessed 20.11.2018..
[11]. Gideon Greenspan. Smart contracts and the dao implosion. 2016. Available at: https://www.multichain.com/blog/2016/06/smart-contracts-the-dao-implosion/, accessed 20.11.2018.
[12]. S. Palladino. The Parity Wallet Hack Explained, https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7, accessed 20.11.2018.
[13]. J.D. Alois. Ethereum Parity Hack May Impact ETH 500,000 or $146 Million. 2017. Available at: https://www.crowdfundinsider.com/2017/11/124200-ethereum-parity-hack-may-impact-eth-500000-146-million/, accessed 20.11.2018.
[14]. Jentzsch C. Decentralized autonomous organization to automate governance. 2016. Available at: https://download.slock.it/public/DAO/WhitePaper.pdf, accessed 20.11.2018.
[15]. Shishkin E. Towards building an environment for reliable smart contracts construction. RusCrypto, 2018.
[16]. De Moura L., Bjørner N. Z3: An efficient SMT solver. Lecture Notes in Computer Science, vol. 4963, 2008, pp. 337-340.
[17]. Manna Z., Pnueli A. The temporal logic of reactive and concurrent systems: Specification. Springer-Verlag, 1992, 427 p/
[18]. Ethereum project. Available at: https://www.ethereum.org/, accessed 20.11.2018.
[19]. Szabo N. Smart contracts. 1994. Available at: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html, accessed 20.11.2018.
[20]. Sheeran M., Singh S., Stålmarck G. Checking safety properties using induction and a SAT-solver. Lecture Notes in Computer Science, vol. 1954, 2000, pp. 127-144.