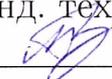


Министерство науки и высшего образования Российской Федерации  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ  
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)  
Институт прикладной математики и компьютерных наук  
Кафедра компьютерной безопасности

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК  
Руководитель ООП  
канд. техн. наук, доцент  
 В. Н. Тренькаев  
“ 23 ” января 20 23 г.

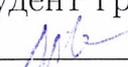
**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА СПЕЦИАЛИСТА  
(ДИПЛОМНАЯ РАБОТА)**

**СИМВОЛЬНОЕ ИСПОЛНЕНИЕ СМАРТ-КОНТРАКТОВ  
В АКТОРНЫХ БЛОКЧЕЙНАХ**

по специальности 10.05.01 Компьютерная безопасность,  
специализация (профиль) «Анализ безопасности компьютерных систем»

Лебедев Владимир Витальевич

Руководитель ВКР  
зав. лаб. компьютерной криптогра-  
фии  
 Панкратова И. А.  
“ 16 ” января 20 23 г.

Автор работы  
студент группы № 931724  
 Лебедев В. В.  
“ 16 ” января 20 23 г.

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

УТВЕРЖДАЮ

Руководитель ООП

канд. техн. наук, доцент

В. Н. Тренькаев

“ 27 ” июня 20 22 г.

ЗАДАНИЕ

по выполнению выпускной квалификационной работы специалиста обучающемуся

Лебедеву Владимиру Витальевичу  
*Фамилия Имя Отчество обучающегося*

по специальности 10.05.01 Компьютерная безопасность, специализация (профиль)  
«Анализ безопасности компьютерных систем»

1 Тема дипломной работы

Символьное исполнение смарт-контрактов в акторных блокчейнах

2 Срок сдачи студентом выполненной дипломной работы:

а) в учебный офис / деканат — 16.01.2023 б) в ГЭК — 27.01.2023

3 Исходные данные к работе:

Объект исследования — методы анализа безопасности смарт-контрактов

Предмет исследования — символьное исполнение

Цель исследования — разработка библиотеки символьного исполнения для смарт-контрактов акторного блокчейна TON

Задачи:

— разработать библиотеку символьного исполнения для смарт-контрактов акторного блокчейна TON;

— проверить библиотеку символьного исполнения, верифицировав смарт-контракт кошелька.

Методы исследования:

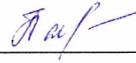
формальная верификация программного обеспечения, символьное исполнение, методы компьютерной безопасности

Организация или отрасль, по тематике которой выполняется работа, —  
05.13.19

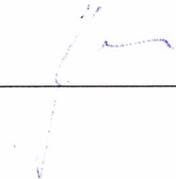
4 Краткое содержание работы

Описание состояния виртуальной машины смарт-контрактов TON на языке SMT-LIB,  
реализация функции перехода в следующее состояние, реализация алгоритма символического  
исполнения смарт-контрактов, верификация смарт-контракта кошелька.

Научный руководитель выпускной  
квалификационной работы,

 / Панкратова И. А. /

Задание принял к исполнению,  
студент группы № 931724

 / Лебедев В. В. /

# АННОТАЦИЯ

**Актуальность.** Существуют категории программ, в которых каждая исполненная инструкция стоит относительно больших денег – идеальный случай для символьного исполнения, ведь разработчики максимально их оптимизируют, позволяя символьному исполнителю за короткое время обойти всю программу целиком. К этой категории относятся смарт-контракты – программы, исполняющиеся в блокчейне. Их результат всегда детерминирован и перепроверяется сотнями валидаторов, а сами смарт-контракты позволяют управлять криптовалютами внутри блокчейна – это значит, что уязвимости, найденные символьным исполнением, могут оцениваться в сотни миллионов долларов. Тем не менее, в классических блокчейнах смарт-контракты достаточно сильно связаны между собой. Существует глобальное состояние блокчейна, которое блокируется каждый раз, когда кто-то вызывает смарт-контракт. В этот момент смарт-контракт может вызывать функции любых других смарт-контрактов, причём в любом контракте из цепочки вызовов может измениться состояние. Это вызывает проблемы, так как в символьном исполнении мы очень сильно завязаны на состояниях сразу большого множества контрактов и не можем их эффективно моделировать. Существует альтернативная модель блокчейна, решающая проблему глобального состояния, она называется акторной моделью. Акторная модель была предложена 50 лет назад для разработки параллельных систем, в которых независимые объекты (акторы) асинхронно обмениваются между собой сообщениями без необходимости глобальной блокировки состояния. Впервые акторный блокчейн был описан Н. Дуровым в работе Telegram Open Network.

**Объект исследования:** смарт-контракты

**Предмет исследования:** символьное исполнение

**Цель работы:** разработка библиотеки символьного исполнения смарт-контрактов в акторном блокчейне TON

**Методы исследования:** формальная верификация программного обеспечения, символьное исполнение, методы компьютерной безопасности

**Результаты:** реализована библиотека символьного исполнения смарт-контрактов в акторном блокчейне TON, с помощью библиотеки проанализирован смарт-контракт кошелька.

# ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ . . . . .	3
1 Состояние TVM . . . . .	5
2 Моделирование состояния TVM на языке SMT-LIB . . . . .	7
2.1 Integer . . . . .	7
2.2 Cell . . . . .	7
2.3 Slice . . . . .	10
2.4 Builder . . . . .	10
2.5 Tuple и StackEntry . . . . .	11
3 Реализация символьного состояния TVM . . . . .	12
4 Формирование начального состояния TVM . . . . .	14
5 Реализация символьного исполнения TVM . . . . .	15
5.1 Алгоритм символьного исполнения . . . . .	15
6 Верификация смарт-контракта кошелька . . . . .	17
6.1 Модель смарт-контракта-кошелька . . . . .	17
6.2 Символьное исполнение смарт-контракта кошелька . . . . .	18
ЗАКЛЮЧЕНИЕ . . . . .	21
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ	22

# ВВЕДЕНИЕ

Символьное исполнение [1] является инструментом преобразования программного кода в математическую модель, которую можно затем применять для тестирования и поиска уязвимостей. Тем не менее, такие проблемы, как экспоненциальный рост числа возможных путей исполнения и большая длина самих путей, в большинстве случаев не позволяют применять символьное исполнение целиком на всю программу. Чаще всего для его использования нужно вводить ограничения: на входные и выходные параметры программы, на максимальную длину пути, а также использовать эвристики, которые откидывают «неинтересные» пути на самых ранних стадиях.

Существуют категории программ, в которых каждая исполненная инструкция стоит относительно больших денег – идеальный случай для символьного исполнения, ведь разработчики максимально их оптимизируют, позволяя символьному исполнителю за короткое время обойти всю программу целиком. К этой категории относятся смарт-контракты [2] – программы, исполняющиеся в блокчейне. Их результат всегда детерминирован и перепроверяется сотнями валидаторов, а сами смарт-контракты позволяют управлять криптовалютами внутри блокчейна – это значит, что уязвимости, найденные символьным исполнением, могут оцениваться в сотни миллионов долларов [3].

Тем не менее, в классических блокчейнах смарт-контракты достаточно сильно связаны между собой. Существует глобальное состояние блокчейна, которое блокируется каждый раз, когда кто-то вызывает смарт-контракт. В этот момент смарт-контракт может вызывать функции любых других смарт-контрактов, причём в любом контракте из цепочки вызовов может измениться состояние. Это вызывает проблемы, так как в символьном исполнении мы очень сильно завязаны на состояниях сразу большого

множества контрактов и не можем их эффективно моделировать. Существует альтернативная модель блокчейна, решающая проблему глобального состояния, она называется акторной моделью. Акторная модель была предложена 50 лет назад для разработки параллельных систем, в которых независимые объекты (акторы) асинхронно обмениваются между собой сообщениями без необходимости глобальной блокировки состояния. Впервые акторный блокчейн был описан Н. Дуровым в работе Telegram Open Network [4].

**Целью** данной работы является разработка библиотеки символьного исполнения смарт-контрактов акторного блокчейна TON и её проверка на смарт-контракте кошелька.

Для достижения данной цели необходимо решить следующие **задачи**:

- разработать библиотеку символьного исполнения смарт-контрактов акторного блокчейна TON;
- проверить библиотеку символьного исполнения, верифицировав смарт-контракт кошелька.

# 1 СОСТОЯНИЕ TVM

TVM [5] – виртуальная машина, исполняющая смарт-контракты в блокчейне TON. Представляет из себя стековую виртуальную машину без линейной памяти (отсутствуют инструкции, которые могут загружать/выгружать данные в памяти по заданному адресу; все инструкции работают только со стеком, как в Java). Основная особенность – специальный тип *Cell* (клетка), в котором хранятся все нечисловые данные (битовые векторы). Каждая клетка может хранить в себе до 1023 бит информации и до 4 ссылок на другие клетки. Граф зависимостей между клетками должен быть ациклическим. Важно отметить, что значения типа *Cell* иммутабельны (их невозможно изменить) и, более того, из них невозможно без предварительной подготовки прочитать данные или ссылки. Для создания новых клеток используется тип *Builder*, для чтения из клеток существует тип *Slice*. За счёт иммутабельности ориентированный ациклический граф зависимостей можно свести к дереву (его проще моделировать), копируя клетку во все поддеревья, где есть зависимость от неё.

Состояние TVM состоит из следующих компонентов:

- код смарт-контракта;
- стек;
- регистры специального назначения.

Код смарт-контракта также представлен в виде древовидной структуры клеток. Каждая инструкция представляет из себя префиксный битовый код, в качестве операндов может иметь как битовую последовательность, так и ссылки на поддеревья – например, при ветвлении или безусловном переходе (в TVM нет такого понятия как *Instruction Pointer*). Стек представляет из себя неограниченный список из типизированных значений. Всего существует семь типов: *Integer*, *Cell*, *Tuple*, *Null*, *Slice*, *Builder*, *Continuation*. *Integer* представляет собой обычное знаковое 257-битное

число в диапазоне  $-2^{256} \dots 2^{256} - 1$ . Существует также специальное значение NaN, которое обрабатывается арифметическими инструкциями специальным образом. Важно заметить, что в данном типе запрещено переполнение, во всех операциях с данным типом нужно это проверять. *Tuple* – это кортеж, позволяющий хранить до 255 значений разных типов. *Null* представляет из себя специальный тип с единственным значением  $\perp$ , используется для обозначения пустых ветвей в деревьях, пустых списков, отсутствия возвращаемого значения функции и т.д. *Continuation* это специальная форма типа *Cell*, обозначающая исполняемую клетку (код смарт-контракта).

Регистры специального назначения:

- $c_0$  – клетка возврата, используется при возврате из функции (аналог адреса возврата в традиционных архитектурах);
- $c_1$  – альтернативная клетка возврата, может использоваться инструкциями управления потоком исполнения как вторая возможная клетка возврата;
- $c_2$  – обработчик исключений, вызывается при выбрасывании исключения. Если  $c_2 = Null$ , работа TVM завершается ошибкой;
- $c_3$  – каталог функций, используется инструкциями управления потоком исполнения при вызове функций по их идентификатору;
- $c_4$  – постоянное хранилище смарт-контракта, используется для хранения состояния между запусками смарт-контракта;
- $c_5$  – сообщения к отправке в конце работы смарт-контракта. В TON смарт-контракты обмениваются сообщениями, содержащими в себе отправителя, получателя, какое-то количество криптовалюты и произвольные данные;
- $c_7$  – конфигурация блокчейна. Содержит такую информацию, как текущие комиссии на операции в блокчейне и список валидаторов блокчейна.

## 2 МОДЕЛИРОВАНИЕ СОСТОЯНИЯ TVM НА ЯЗЫКЕ SMT-LIB

### 2.1 INTEGER

Integer – это (`_ BitVec 257`) со встроенными арифметическими операциями `bvadd`, `bvsub` и т.д. Однако важно отметить, что в TVM запрещено переполнение, поэтому при реализации арифметических инструкций нужно добавлять ограничения:

```
1 (define-sort Int257 () (_ BitVec 257))
2 (declare-const a Int257)
3 (declare-const b Int257)
4 (assert (implies (and (bvsgt a (_ bv0 257)) (bvsgt b
    (_ bv0 257))) (bvsgt (bvadd a b) (_ bv0 257))))
```

### 2.2 CELL

Определим клетку как 1023-битный вектор со списком ссылок на другие клетки. Как мы отметили выше, в данном случае ориентированный ациклический граф можно моделировать в виде дерева. Именно поэтому в списке ссылок на клетки мы можем хранить сами клетки. Также важно отметить, что длина данных в клетке может варьироваться от 0 до 1023 бит, поэтому мы должны внести в модель и её. При создании клетки эта длина будет складываться из длин всех битовых векторов, которые были записаны в эту клетку. При чтении очередной части битового вектора необходимо добавлять ограничение, что длина читаемой части меньше или равна длине всего битового вектора.

```
1 (declare-datatypes () (
2   (Cell (cell (data (_ BitVec 1023)) (data_len (_
3     BitVec 10)) (refs RefsList)))
4   (RefsList nil (cons (hd Cell) (tl RefsList))))
```

Также зададим рекурсивно функцию длины списка, чтобы затем ограничить длину четырьмя ссылками:

```
1 (define-fun-rec length ((ls (RefsList))) Int
2   (if ((_ is nil) ls) 0 (+ 1 (length (tl ls)))))
```

Пример клетки с четырьмя ссылками:

```
1 (declare-const cell1 Cell)
2 (declare-const cell2 Cell)
3 (declare-const cell3 Cell)
4 (declare-const cell4 Cell)
5 (declare-const cell5 Cell)
6 (assert (= (data cell1) #b1101000...000000))
7 (assert (= (data_len cell1) #b00000000100))
8 (assert (= (refs cell1) (cons cell2 (cons cell3 (cons
   cell4 (cons cell5 nil))))))
```

Необходимо убедиться, что количество ссылок не превышает лимита:

```
1 (assert (<= (length (refs cell1)) 4))
```

Проверим выполнимость модели:

```
1 (check-sat)
2 (get-model)
```

Модель выполнима:

```
1 sat
```

```

2 (
3 (define-fun cell2 () Cell
4 (cell #b000...000
5 #b0000000000
6 nil))
7 (define-fun cell4 () Cell
8 (cell #b000...000
9 #b0000000000
10 nil))
11 (define-fun cell11 () Cell
12 (let ((a!1 (cons (cell #b000...000
13 #b0000000000
14 nil)
15 (cons (cell #b000...000
16 #b0000000000
17 nil)
18 (cons (cell #b000...000
19 #b0000000000
20 nil)
21 nil))))))
22 (cell #b11010000...000
23 #b0000000100
24 (cons (cell #b000...000
25 #b0000000000
26 nil)
27 a!1))))
28 (define-fun cell15 () Cell
29 (cell #b000...000
30 #b0000000000
31 nil))
32 (define-fun cell13 () Cell
33 (cell #b000...000
34 #b0000000000
35 nil))
36 (define-fun cell16 () Cell

```

```
37     (cell #b000...000
38         #b00000000000
39         nil))
40 )
```

При добавлении пятой ссылки SMT-решатель сообщает, что модель невыполнима – не выполняется условие о лимите количества ссылок.

## 2.3 SLICE

Для чтения первых  $x$  бит клетки используется оператор логического сдвига:

```
1 (declare-const cell1 Cell)
2
3 (declare-const x Int)
4
5 (simplify (bv1shr (data cell1) ((_ int2bv 1023) (-
    1023 x))))
```

Также, при формировании новой клетки (из которой уже прочитали какую-то часть данных) необходимо вычислять новую длину без прочитанного куска:

```
1 (simplify (bvsub (data_len cell1) ((_ int2bv 10) x)))
```

## 2.4 BUILDER

Для записи в клетку используются обратные операции: сдвиг влево на  $x$  бит и дизъюнкция записываемых данных с битовым вектором, а также увеличение длины на  $x$ .

## 2.5 TUPLE И STACKENTRY

Определим StackEntry как произвольный тип данных на стеке. Его конструкторами будут являться все остальные типы, в том числе и Tuple, который будет определён рекурсивно через StackEntry.

```
1 (define-fun itobv257 ((x Int)) Int257 ((_ int2bv 257)
    x))
2
3 (declare-datatypes ((StackEntry 0))
4   ((
5     (int (int_val Int257))
6     (cell (cell_val Cell))
7     (builder (builder_val Cell))
8     (slice (slice_val Cell))
9     (null)
10    (tuple (tuple_val (Seq StackEntry)))
11   ))
12 )
13
14 (declare-const entry StackEntry)
15 (declare-const cell1 Cell)
16 (assert (is-tuple entry))
17 (assert (= (seq.nth (tuple_val entry) 0) (int
    (itobv257 123))))
18 (assert (= (seq.nth (tuple_val entry) 1) (cell cell1)))
19 (assert (>= (seq.len (tuple_val entry)) 4))
20 (assert (<= (seq.len (tuple_val entry)) 255))
21 (check-sat)
22 (get-model)
```

## 3 РЕАЛИЗАЦИЯ СИМВОЛЬНОГО СОСТОЯНИЯ TVM

Символьное состояние TVM отличается от конкретного тем, что каждая его составляющая может быть представлена SMT-формулой. Кроме того, вместе с самим состоянием (набором формул) необходимо также хранить и условия его достижимости (набор ограничений). Если у конкретного состояния может быть только одно следующее состояние, то у символьного состояния ветвей исполнения может быть одновременно несколько. Почти каждая инструкция при определённых условиях может выбросить исключение, кроме того, существуют инструкции условного ветвления (IF, WHILE и т.д.), которые в зависимости от входного значения могут выбирать разные ветви исполнения. При символьном исполнении зачастую невозможно определить, какая ветвь была выбрана, поэтому выбираются обе ветви, однако в их символьные состояния добавляются противоположные ограничения (при ветвлении с условием  $x > 0$  очевидно, что после выбора ветвления это условие будет выполнено, в то же время на другой ветви будет выполняться противоположное условие  $x \leq 0$ ).

Для реализации инструкций реализованы следующие операции над состоянием:

- проверка типа значения на стеке (проверка использования конкретного конструктора типа *StackEntry*);
- типизированные операции над стеком (положить значение определённого типа на стек, снять; выбросить исключение при ошибке проверки типов);
- нетипизированные операции над стеком (обмен, дубликация, удаление значений на стеке);
- выброс исключения;

- ветвление (дублирование исходного состояния с применением противоположных ограничений к каждому);
- операции с клетками (чтение/запись битовых векторов и ссылок, проверка на переполнения и выброс исключений).

## 4 ФОРМИРОВАНИЕ НАЧАЛЬНОГО СОСТОЯНИЯ TVM

Запуск смарт-контракта происходит при получении им входящего сообщения. Сообщение может быть внешним или внутренним. Внутренние сообщения могут отправляться только от контракта к контракту и обладают полем баланса, тогда как внешние сообщения отправляются внешним пользователем блокчейна в смарт-контракты и не могут иметь баланс. Также существуют так называемые *get*-методы, которые присутствуют в коде смарт-контракта, однако выполняются не в блокчейне. Они представляют из себя интерфейс для получения пользователем информации о контракте. Для того, чтобы отличать все эти вызовы, используется специальный параметр *method\_id*, который кладётся в начальное состояние TVM. *method\_id* = 0 при получении внутреннего сообщения, *method\_id* = 1 при получении внешнего сообщения; любые другие *method\_id* представляют из себя вызовы определённых *get*-методов.

На стек передаются следующие параметры:

- баланс смарт-контракта;
- баланс входящего сообщения;
- само входящее сообщение;
- тело входящего сообщения;
- селектор функции (*method\_id*).

# 5 РЕАЛИЗАЦИЯ СИМВОЛЬНОГО ИСПОЛНЕНИЯ TVM

## 5.1 АЛГОРИТМ СИМВОЛЬНОГО ИСПОЛНЕНИЯ

1. Вход алгоритма: начальное состояние  $S_0$ ;
2. Положить  $S_0$  в список активных состояний;
3. Взять из списка активных состояний следующее состояние  $S_i$ ;
4. Читать биты из  $S_i.code$  до тех пор, пока не будет найдено однозначное соответствие с префиксом какой-либо инструкции. Выбросить исключение, если инструкция с данным префиксом не найдена;
5. Прочитать из  $S_i.code$  операнды обнаруженной инструкции. Они могут быть как битовыми последовательностями, так и ссылками на другие клетки;
6. Выполнить инструкцию:
  - проверить операнды;
  - проверить глубину стека и произвести ветвление на ошибку с условием недостаточного количества входных значений на стеке (ошибка StackUnderflow);
  - взять входные значения со стека;
  - проверить корректность значений со стека;
  - выполнить операцию;
  - положить на стек результат;
  - сформировать множество следующих состояний  $S_m \dots S_{m+k}$ , где  $k \geq 0$  (среди них могут быть как активные, так и конечные состояния).
7. Проверить выполнимость ограничений для состояний  $S_m \dots S_{m+k}$ . Пометить невыполнимые активные состояния. Выполнимые активные состояния положить в список активных состояний;

8. Продолжать с пункта 3 до тех пор, пока существуют выполнимые активные состояния;
9. Результат алгоритма: множество конечных выполнимых состояний, которые можно достигнуть из  $S_0$ ; множество невыполнимых состояний, которые достигнуть невозможно для заданного  $S_0$ .

# 6 ВЕРИФИКАЦИЯ СМАРТ-КОНТРАКТА КОШЕЛЬКА

## 6.1 МОДЕЛЬ СМАРТ-КОНТРАКТА-КОШЕЛЬКА

Смарт-контракт кошелька представляет из себя сущность блокчейна, которая хранит баланс пользователя. Сам пользователь не может иметь баланс, так как он не является сущностью блокчейна. Вместо этого он создаёт в блокчейне смарт-контракт кошелька, который аутентифицирует внешние сообщения пользователя по паре асимметричных ключей `ed25519` и отправляет внутренние сообщения, присланные пользователем внутри своего внешнего сообщения.

Алгоритм работы кошелька

1. Прочитать 512-битную подпись из входящего сообщения;
2. Прочитать  $(K_p, stored\_seqno)$  из постоянного хранилища смарт-контракта, где  $K_p$  – 256-битный публичный ключ владельца кошелька,  $stored\_seqno$  – счётчик транзакций;
3. Проверить присланную подпись на ключе  $K_p$ . Прочитать из входящего сообщения 32-битное число  $received\_seqno$ , выдать ошибку, если не совпадает с  $stored\_seqno$ ;
4. Если аутентификация прошла успешно, прочитать из входящего сообщения ссылку на внутреннее сообщение и поместить его в регистр  $c_5$ ;
5. Увеличить  $stored\_seqno$  на 1, сохранить в регистре  $c_4$  новое значение.

Кошелёк работает корректно, если выполнены следующие условия:

1. Владелец приватного ключа  $K_s$ , соответствующего публичному ключу  $K_p$  может отправить произвольное внутреннее сообщение  $M$ ;
2. Не существует такого внутреннего сообщения  $M'$ , которое можно отправить, не зная приватного ключа  $K_s$ .

## 6.2 СИМВОЛЬНОЕ ИСПОЛНЕНИЕ СМАРТ-КОНТРАКТА КОШЕЛЬКА

Исходный код смарт-контракта кошелька на языке ассемблера приведён в листинге. Он был скомпилирован в байткод, после чего из него было сформировано начальное состояние  $S_0$  с произвольным внешним сообщением на входе и произвольными данными в постоянном хранилище.

```
1 "Asm.fif" include
2
3 <{ SETCPO DUP IFNOTRET // return if recv_internal
4   DUP 85143 INT EQUAL IFJMP:<{ // "seqno" get-method
5     DROP c4 PUSHCTR CTOS 32 PLDU // cnt
6   }>
7   INC 32 THROWIF // fail unless recv_external
8   512 INT LDSLICEX DUP 32 PLDU // sign cs cnt
9   c4 PUSHCTR CTOS 32 LDU 256 LDU ENDS // sign cs cnt
   cnt' pubk
10  s1 s2 XCPU // sign cs cnt pubk cnt' cnt
11  EQUAL 33 THROWIFNOT // ( seqno mismatch? )
12  s2 PUSH HASHSU // sign cs cnt pubk hash
13  s0 s4 s4 XC2PU // pubk cs cnt hash sign pubk
14  CHKSIGNU // pubk cs cnt ?
15  34 THROWIFNOT // signature mismatch
16  ACCEPT
```

```
17 SWAP 32 LDU NIP 8 LDU LDREF ENDS // pubk cnt
mode msg
18 SWAP SENDRAWMSG // pubk cnt ; ( message sent )
19 INC NEWC 32 STU 256 STU ENDC c4 POPCTR
20 }>c
21
22 2 boc+>B "simple-wallet.boc" tuck B>file
```

В результате символического исполнения был получен граф потока исполнения (Рисунок 1).

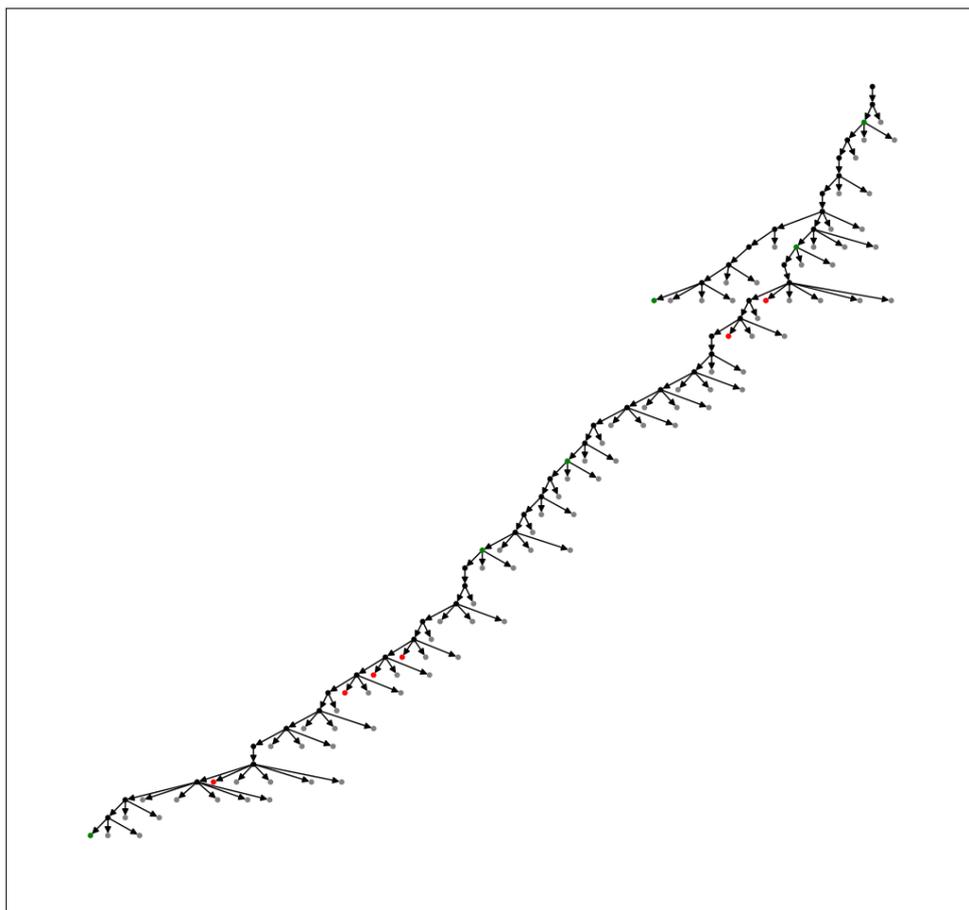


Рисунок 1 — Граф потока исполнения

Далее был проведён анализ конечных состояний. Было обнаружено 97 недостижимых ошибочных состояний, 6 достижимых ошибочных и 6 достижимых успешных. Под успешными состояниями подразумеваются любые предусмотренные разработчиком выходы (в том числе с ненулевым кодом ошибки).

Для доказательства невозможности отправить сообщение без проверки подписи добавим к единственному успешному состоянию, в котором отправляется сообщение, условие «проверка подписи не проходит». SMT-решатель выдаёт ответ «модель невыполнима», из чего следует, что при любых входных значениях проверка подписи должна проходить успешно, а значит и обойти её нельзя.

Для доказательства возможности отправить произвольное внутреннее сообщение необходимо проанализировать ограничения во всех достижимых состояниях. Достижимые ошибочные состояния включают в себя 5 ошибок CellUnderflow, связанных с недостаточностью данных на входе. Ещё одна ошибка, OutOfRange, возникает из-за переполнения типа Integer при сериализации в 32-битное число. При более детальном ручном анализе было обнаружено, что при подаче на вход сообщения под номером  $2^{32} - 1$  счётчик *stored\_seqno* более не может быть увеличен, таким образом дальнейшее использование кошелька невозможно.

# ЗАКЛЮЧЕНИЕ

В рамках данной дипломной работы были выполнены следующие задачи:

- построена модель символьного состояния TVM;
- реализована библиотека символьного исполнения смарт-контрактов акторного блокчейна TON;
- проведён анализ безопасности смарт-контракта кошелька с помощью символьного исполнения;
- в реализации смарт-контракта кошелька были выявлены отличия от модели: возможно переполнение счётчика, следовательно, аутентификация сообщений становится невозможной при значении счётчика равном  $2^{32} - 1$ .

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. *King J. C.* Symbolic Execution and Program Testing // Commun. ACM. — New York, NY, USA, 1976. — июль. — т. 19, № 7. — с. 385—394. — ISSN 0001-0782. — DOI: 10.1145/360248.360252. — URL: <https://doi.org/10.1145/360248.360252>.
2. An Overview of Smart Contract: Architecture, Applications, and Future Trends / S. Wang [и др.] // 2018 IEEE Intelligent Vehicles Symposium (IV). — 2018. — с. 108—113. — DOI: 10.1109/IVS.2018.8500488.
3. *Samreen N. F., Alalfi M. H.* A Survey of Security Vulnerabilities in Ethereum Smart Contracts. — 2021. — DOI: 10.48550/ARXIV.2105.06974. — URL: <https://arxiv.org/abs/2105.06974>.
4. *Durov N.* — 2019. — URL: <https://ton.org/ton.pdf>.
5. *Durov N.* — 2020. — URL: <https://ton.org/tvm.pdf>.

# Отчет о проверке на заимствования №1



Автор: Лебедев Владимир  
Проверяющий: Лебедев Владимир

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

## ИНФОРМАЦИЯ О ДОКУМЕНТЕ

№ документа: 6  
Начало загрузки: 22.01.2023 12:53:48  
Длительность загрузки: 00:00:00  
Имя исходного файла:  
Лебедев\_диплом\_v5.pdf  
Название документа: Лебедев\_диплом\_v5  
Размер текста: 25 кБ  
Символов в тексте: 25487  
Слов в тексте: 3072  
Число предложений: 150

## ИНФОРМАЦИЯ ОБ ОТЧЕТЕ

Начало проверки: 22.01.2023 09:53:49  
Длительность проверки: 00:00:10  
Комментарии: не указано  
Модули поиска: Интернет Free

### СОВПАДЕНИЯ

4,15%

### САМОЦИТИРОВАНИЯ

0%

### ЦИТИРОВАНИЯ

0%

### ОРИГИНАЛЬНОСТЬ

95,85%

Совпадения — доля всех найденных текстовых пересечений, за исключением тех, которые система отнесла к цитированиям, по отношению к общему объему документа.  
Самоцитирования — доля фрагментов текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника, автором или соавтором которого является автор проверяемого документа, по отношению к общему объему документа.

Цитирования — доля текстовых пересечений, которые не являются авторскими, но система посчитала их использование корректным, по отношению к общему объему документа. Сюда относятся оформленные по ГОСТу цитаты; общепотребительные выражения; фрагменты текста, найденные в источниках из коллекций нормативно-правовой документации.

Текстовое пересечение — фрагмент текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника.

Источник — документ, проиндексированный в системе и содержащийся в модуле поиска, по которому проводится проверка.

Оригинальность — доля фрагментов текста проверяемого документа, не обнаруженных ни в одном источнике, по которому шла проверка, по отношению к общему объему документа.

Совпадения, самоцитирования, цитирования и оригинальность являются отдельными показателями и в сумме дают 100%, что соответствует всему тексту проверяемого документа.

Обращаем Ваше внимание, что система находит текстовые пересечения проверяемого документа с проиндексированными в системе текстовыми источниками. При этом система является вспомогательным инструментом, определение корректности и правомерности заимствований или цитирований, а также авторства текстовых фрагментов проверяемого документа остается в компетенции проверяющего.

№	Доля в отчете	Источник	Актуален на	Модуль поиска	Комментарии
[01]	1,73%	<a href="http://vital.lib.tsu.ru/vital/access/services/Download/vital:6890/SOURCE01">http://vital.lib.tsu.ru/vital/access/services/Download/vital:6890/SOURCE01</a> <a href="http://vital.lib.tsu.ru">http://vital.lib.tsu.ru</a>	07 Сен 2020	Интернет Free	
[02]	1,74%	Тестовый клиент TON (Telegram Open Network) и новый язык Fift для смарт-контрактов / Habr <a href="https://habr.com">https://habr.com</a>	09 Июл 2019	Интернет Free	
[03]	0,67%	<a href="http://vital.lib.tsu.ru/vital/access/services/Download/vital:8382/SOURCE01">http://vital.lib.tsu.ru/vital/access/services/Download/vital:8382/SOURCE01</a> <a href="http://vital.lib.tsu.ru">http://vital.lib.tsu.ru</a>	24 Янв 2020	Интернет Free	

Еще источников: 6  
Еще совпадений: 0%

*С результатами ознакомлена  
Руководитель И.А. Панкратова*

*И.А.*