Санкт-Петербургский Государственный Университет

Математическое обеспечение и администрирование информационных систем
Кафедра системного программирования

Прошутинский Алексей Владимирович

Разработка инструмента для исследования множества контрактов сети Ethereum

Выпускная квалификационная работа

Научный руководитель:
ст. преп. А. Р. Ханов

Рецензент:
к.т.н., доцент А. А. Воробьева

Санкт-Петербург
2019

SAINT PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering

Aleksei Proshutinskii

Development of a tool for the research of a set of contracts on Ethereum network

Graduation Thesis

Scientific supervisor:
Senior lecturer Arthur Khanov

Reviewer:
Associate professor Alisa Vorobeva

Saint Petersburg
2019

Оглавление

Введение

Ethereum является платформой для создания децентрализованных приложений на базе блокчейна, работающих с помощью смарт-контрактов (или умных контрактов). Блокчейн – это непрерывная цепочка записей, называемых блоками, которые связаны с помощью криптографии. Каждый блок содержит хэш предыдущего, временную метку и данные транзакций. Блокчейн по сути является устойчивой к изменению данных децентрализованной базой данных [1]. Смарт-контракт – это компьютерная программа, в которой заложены соглашения между сторонами. Использование контрактов выгодны в ситуациях, когда классические средства обеспечения выполнения контракта слишком дороги или стороны не имеют доступа к общему арбитру или юридической системе. Основной принцип умного контракта состоит в полной автоматизации и достоверности исполнения договорных отношений. Умные контракты в Ethereum представлены в виде классов, которые могут быть реализованы на различных языках, включая визуальное программирование и компилируются в байт-код Ethereum Virtual Machine (EVM) перед отправкой в блокчейн. Изменение состояния виртуальной машины может быть записано на полном по Тьюрингу языке сценариев.

Помимо множества возможностей для построения приложений смарт-контракты привнесли множество опасностей, связанных с уязвимостями компиляторов и ошибками программистов. Существует множество инструментов для обнаружения уязвимостей смарт-контрактов, но не существует актуальной базы данных со всеми контрактами, их уязвимостями и подробной информацией по ним, а по данным etherscan.io [9] количество верифицированных контрактов очень мало: лишь 53915 [35] из более чем 13 миллионов [6]. Для таких контрактов доступна лишь информация по уязвимостям различных версий компилятора и исходный код.

Исследователям безопасности и разработчикам верификаторов нужен инструмент для исследования рабочей сети Ethereum и построение базы данных с хранением информации и статистики по контрак-

там может решить эту проблему. Такая база может хранить прототипы функций, размеры хранилищ, результаты проверки наиболее популярными верификаторами, сложность ветвления, статистику по ошибкам: вызовы, которые привели к перерасходу газа (комиссия за исполнение операций EVM) [7], откату транзакций. Все вызовы всех контрактов сохранены в блокчейне, поэтому эта информация доступна всем, но нигде не описана.

Благодаря созданию приложения по сбору статистики по контрактам и своевременной передаче информации об уязвимостям через Ethereum Bounty Program [5] можно добиться минимальных потерь как со стороны пользователей, так и со стороны владельцев распределенных приложений. Вознаграждение за уязвимости контрактов в рамках этой программы не назначается, но будет оказана помощь в доведении информации до разработчиков.

1. Постановка задачи и цели

Целью работы является построение инструмента исследования рабочей сети Ethereum.

В рамках данной работы были поставлены следующие задачи:

- Проведение обзора уязвимостей и инструментов анализа контрактов;
- Разработка модуля для получения информации о контрактах в реальном времени;
- Построение базы данных с индексом по контрактам;
- Получение статистики по контрактам в реальном времени;
- Поиск уникальных и похожих контрактов;
- Предсказание уязвимостей;
- Поиск и передача информации об уязвимостях по Ethereum Bounty Program.

2. Типы уязвимостей смарт-контрактов

2.1. Integer Overflow/Underflow

Integer Overflow/Underlow [24] происходит, когда арифметическая операция достигает максимального или минимального размера типа. Без должных проверок возникает опасность получения нулевого или максимального баланса на счете.

2.2. Reentrancy

Reentrancy [4][26] является одной из самых серьезных уязвимостей, основанной на самой крупной из когда-либо совершенных атак [2]. Заключается в перехвате управления при вызове внешнего контракта. При этой атаке вредоносный контракт снова вызывает функцию целевого контракта до завершения первого вызова функции.

2.3. Delegatecall to Untrusted Callee

Delegatecall – это вызов кода внешнего контракта в контексте вызывающего. Это может быть опасно при вызове ненадежных контрактов или контрактов, переданных в качестве аргументов при вызове, так как внешний контракт имеет полный контроль над данными вызывающего и может распоряжаться балансом последнего без каких-либо ограничений. [30]

2.4. Unprotected SELFDESTRUCT Instruction

Unprotected SELFDESTRUCT Instruction [25] заключается в недостаточной защищености деструктора контракта (например, оставление его в модификатором public), что приводит к нежелательному прекращению работы контракта. Одним из самых известных случаев является Parity Multi-Sig Wallet Attack [20].

2.5. Authorization through tx.origin

Авторизация через tx.origin [4][32] позволяет злоумышленникам назначить себя владельцами контрактов. tx.origin – глобальная переменная контракта, которая возвращающая адрес аккаунта, с которого была сделана транзакция. Использование переменной для авторизации может сделать контракт уязвимым, если авторизованный аккаунт вызовет вредоносный контракт. Вызов может быть сделан уязвимому контракту, который проходит авторизацию, поскольку tx.origin возвращает первоначального отправителя транзакции, который в данном случае является авторизованным аккаунтом.

2.6. Timestamp Dependence

Timestamp Dependence [4][33] заключается в использовании злоумышленниками зависимость вывода функции контракта от времени. Данная уязвимость является серьёзной только в критических компонентах контракта.

2.7. Transaction-Order Dependence

Сеть Ethereum обрабатывает транзакции в блоках, и новые блоки подтверждаются каждые несколько секунд. Майнеры просматривают транзакции, которые они получили, и выбирают, какие транзакции включить в блок, основываясь на том, кто заплатил достаточно высокую комиссию, чтобы включить ее. Кроме того, когда транзакции отправляются в сеть Ethereum, они направляются каждому узлу для обработки. Таким образом, человек, который запускает узел Ethereum, может сказать, какие транзакции будут происходить, прежде чем они будут завершены. Transaction-Order Dependence возникает, когда код зависит от порядка транзакций, представленных ему. [4][31]

Самый простой пример – это когда контракт дает вознаграждение за предоставление информации. Скажем, контракт выдаст 1 токен первому человеку, который решит задачу. Алиса решает задачу и отправляет

ответ в сеть со стандартной комиссией. Боб запускает узел Ethereum и может увидеть ответ на математическую задачу в транзакции, которую Алиса отправила в сеть. Таким образом, Боб отправляет ответ в сеть с гораздо более высокой комиссией, и, таким образом, он обрабатывается и передается перед операцией Алисы. Боб получает один жетон, а Алиса - ничего, хотя Алиса работала над решением проблемы.

3. Обзор существующих инструментов

3.1. Mythril Classic

Mythril Classic [12] – инструмент с открытым кодом для анализа безопасности контрактов. Для анализа используется символьное исполнение. Может исследовать как по исходному коду, так и контракты в блокчейне по их адресу. Проверяет контракты на следующие типы уязвимостей:

- Integer Overflow/Underflow. SWC-101 [24];
- Reentrancy. SWC-107 [26];
- Delegatecall to Untrusted Callee. SWC-112. [30];
- Unprotected SELFDESTRUCT Instruction. SWC-106 [25];
- Authorization through tx.origin. SWC-115 [32];
- Assert Violation. SWC-110. [28].

3.2. MAIAN

MAIAN [10] – инструмент с открытым кодом для анализа трех типов уязвимостей:

- Prodigal contracts (”расточительные” контракты, т.е. могут отослать токены кому угодно);
- Suicidal contracts (”суицидальные” контракты, т.е. могут деструктор контракта может быть вызван любым пользователем). SWC-106 [25];
- Greedy contracts (”жадные” контракты, т.е. никто не может вывести токены).

Также может анализировать как на основе исходного кода, так и адреса контракта.

3.3. Porosity

Porosity [3] – проект с исходным кодом для декомпиляции и дезассемблирования контрактов с одновременным анализом на уязвимости. Более не поддерживается и не обновляется.

3.4. Manticore

Manticore [11] – инструмент для символьного исполнения контрактов Ethereum и Linux-ELF исполняемых файлов.

Проверяет контракты на следующие типы уязвимостей:

- Integer Overflow/Underflow. SWC-101 [24];
- Reentrancy. SWC-107 [26];
- Delegatecall to Untrusted Callee. SWC-112. [30];
- Unprotected SELFDESTRUCT Instruction. SWC-106 [25];
- Uninitialized Storage Pointer. SWC-109 [27];
- Use of Deprecated Solidity Functions. SWC-111 [29].

3.5. Securify

Securify [34] – веб-сервис для анализа контрактов. Анализ производится только по исходному коду. Содержит краткую статистику по анализированным контрактам. По данным на середину декабря 2018 года 31649 контрактов проанализированно из более миллиона [6] контрактов блокчейна.

3.6. Oyente

Oyente – инструмент для анализа уязвимостей смарт-контрактов, работает с кодами операций путем символьного исполнения. [16] Анализирует следующие типы уязвимостей:

- Integer Overflow/Underflow. SWC-101 [24];
- Reentrancy. SWC-107 [26];
- Timestamp Dependence. SWC-116. [33];
- Transaction-Order Dependence. SWC-114 [31];
- Uninitialized Storage Pointer. SWC-109 [27].

3.7. Выводы

Существует множество инструментов для обнаружения уязвимостей, но нет полной базы по всем контрактам блокчейна и время анализа даже одного контракта может занимать значительное время.

4. Архитектура решения

Архитектура решения изображена на рисунке 1.

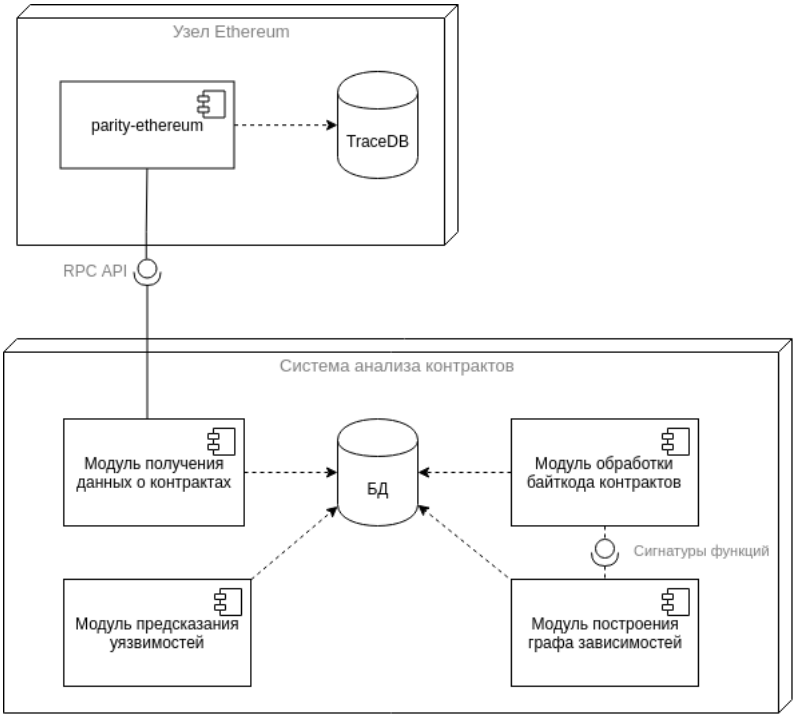


Рис. 1: Архитектура системы

4.1. Узел Ethereum

В качестве узла для доступа к блокчейну Ethereum изначально предполагалось использовать официальный клиент geth [15], но после подробного изучения RPC-API [8] не было найдено удобного способо для получения полной информации о контрактах и выбор пал на проект parity-ethereum [17]. В его RPC-API присутствует модуль trace [22] и база данных вызовов, с помощью которых можно получить все данные по всем вызовам в блокчейне, включая транзакции создания контрактов. Узел был запущен на основном блокчейне Ethereum (mainnet) с флагом –tracing=on.

4.2. База данных

В качестве базы данных была выбрана PostgreSQL[19], как проект с открытым исходным кодом и поддержкой Python. В ней содержатся следующие таблицы:

- contracts – содержит данные по все контрактам блокчейна:
  - Адрес контракта;
  - Номер блока создания;
  - Хеш(идентификатор) транзакции создания;
  - Байткод;
  - Поле isDuplicated, указывающее на оригинальность кода контракта;
  - Адрес оригинала, если isDuplicated истина.
- func_stats – содержит сигнатуры функций и частоту встречаемости в коде контрактов;
- unique_contacts – содержит данные только по уникальным контрактам.

В последней таблице используется индекс на основе хешей от байткода, позволяющий быстро определять уникальность контракта.

4.3. Модуль получения данных

Модуль получения данных посредством RPC-API выделяет из блоков транзакции создания контрактов, проверяет их успешность и добавляет в базу данных байткод, адрес, блок создания, хеш транзакции.

Для реализации логики использованы следующие RPC-запросы:

- eth_blockNumber – для получения номера текущего последнего блока;

- trace_block – для получения списка вызовов по номеру блока.

Модуль хранит у себя номер последнего обработанного блока (инициализированный изначально нулем) и периодически опрашивает узел о получении новых блоков. При получении списка вызовов производится поиск вызовов категории create с последующим добавлением информации в базу данных. Модуль реализован на языке Python с использованием библиотеки requests [23] для взаимодействия с узлом и библиотеки psycopg2 [21] для взаимодействия с базой данных.

4.4. Модуль обработки байткода

Задача этого модуля состоит в дизассемблировании кода контракта и выделении сигнатур функций с помощью модуля анализатора Mythril Classic[13] с последующим обновлением статистики и добавлении данных в базу данных.

4.5. Модуль построения графа зависимостей

Данный модуль предназначен для нахождения близости контрактов. В качестве близости была выбрана функция схожести набора сигнатур функций. Этот граф позволяет устанавливать как близость нового контракта к уже существующим уязвимым, так и показывает возможные уязвимые контракты, близкие к уже взломанным. Кроме того, изображение такого графа покажет основные кластеры контрактов по сигнатурам.

4.6. Модуль предсказания уязвимостей

Модуль осуществляет предсказание наличия конкретных уязвимостей на основе кортежа сигнатур функций контракта. В качестве источника данных об уязвимостям был выбран вывод анализатора Mythril Classic, как самого популярного и постоянно развивающегося проекта.

Моделью служит персептрон[18] из пакета scikit-learn[36]. Модель обучалась на основе сигнатур некоторого множества функций всех контрактов для каждого типа уязвимостей отдельно.

5. Анализ

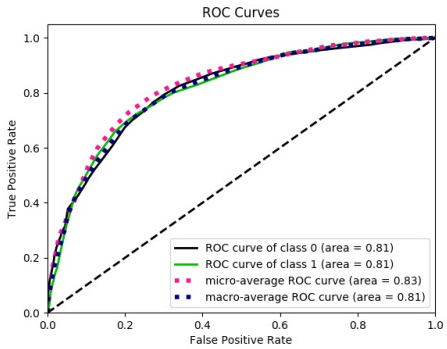
5.1. Общие сведения

В процессе анализа контрактов было выяснено, что множество из них являются точными копиями каких-то других контрактов. Так из 13 миллионов контрактов было выделено лишь 180 тысяч уникальных, что существенно уменьшает общее количество контрактов для анализа. Общее количество различных сигнатур функций – 220 тысяч.

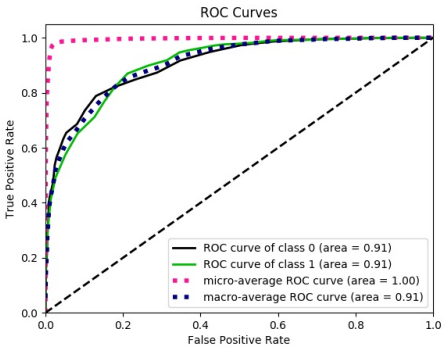
5.2. Предсказание уязвимостей

Персептрон был обучен с использованием сигнатур функций, которые встречались не реже, чем в 20 уникальных контрактах. Входные данные представляли собой вектор 0 и 1, означающих отсутствие или наличие сигнатуры в данном контракте. Результаты кросс-валидации на 90 тысячах контрактов представлены в таблице и графиках ниже(именование уязвимостей согласно репозиторию mythril-classic[14]):

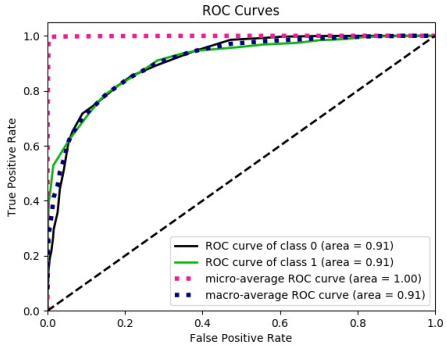
Код уязвимости	AUC	ACC
SWC-101	0.81	0.79
SWC-104	0.91	0.97
SWC-105	0.91	0.99
SWC-106	0.94	0.99
SWC-107	0.91	0.87
SWC-110	0.86	0.85
SWC-111	0.89	0.99
SWC-112	0.95	0.99
SWC-113	0.90	0.98
SWC-116	0.89	0.99
SWC-120	0.94	0.99



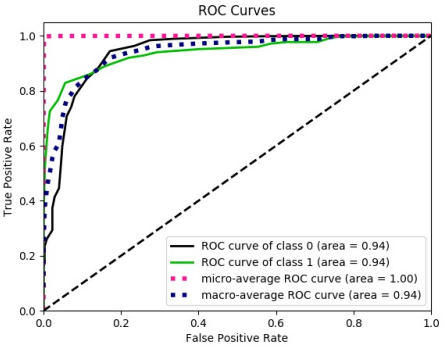
(a) SWC-101



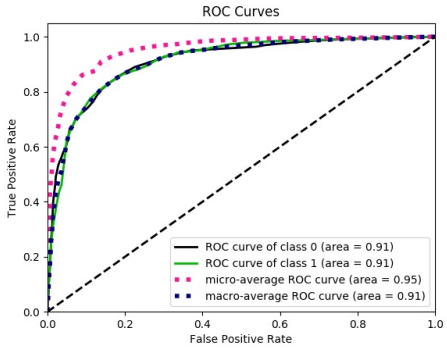
(b) SWC-104



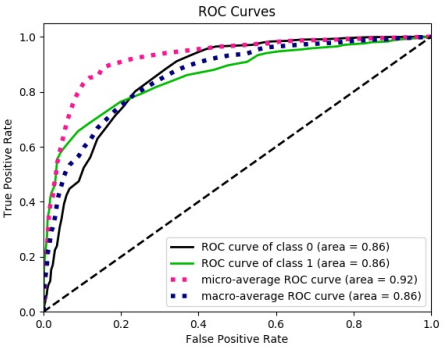
(c) SWC-105



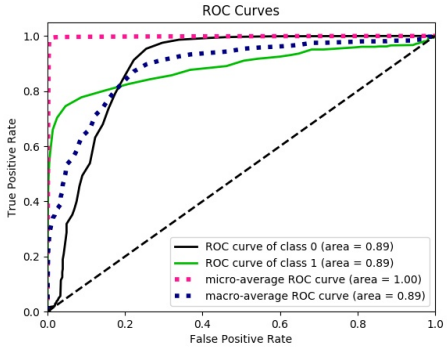
(d) SWC-105



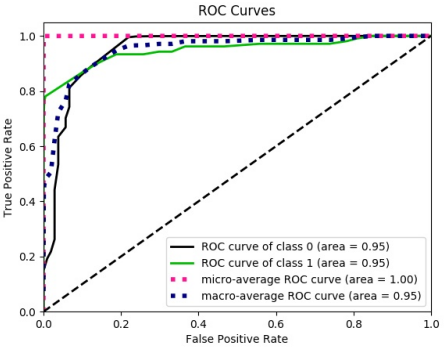
(e) SWC-107



(f) SWC-110

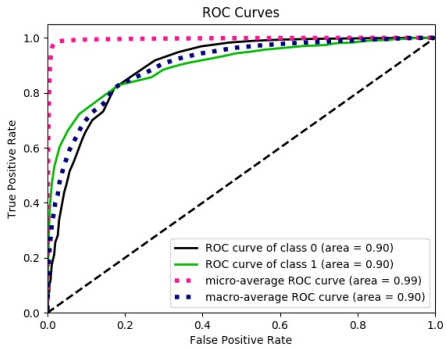


(g) SWC-111

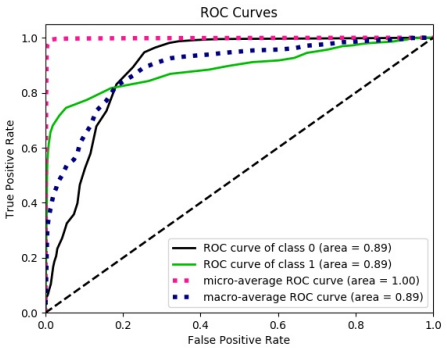


(h) SWC-112

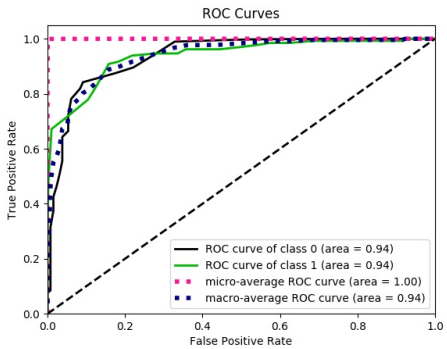
Рис. 2: ROC Curves



(a) SWC-113



(b) SWC-116



(c) SWC-120

Рис. 3: ROC Curves

Данные графики и таблица показывают точность и эффективность использования персептрона для анализа контрактов.

Заключение

В ходе работы были получены следующие результаты:

- Создана архитектура инструмента для анализа контрактов сети Ethereum;
- Собраны данные по всем контрактам и выделены уникальные;
- Реализована возможность получения списка копий и похожих контрактов;
- Реализован компонент предсказания уязвимостей и проверен на 90 тысячах уникальных контрактов;

Из полученных данных можно сказать, что с помощью обученных моделей можно с успехом предсказывать уязвимости на основе кортежа сигнатур функций.

Список литературы

[1] Blockchain definition. — Access mode: https://en.wikipedia.org/wiki/Blockchain.

[2] The DAO attack. — Access mode: https://www.coindesk.com/understanding-dao-hack-journalists.

[3] Decompiler and Security Analysis tool for Blockchain-based Ethereum Smart-Contracts. — Access mode: https://github.com/comaeio/porosity.

[4] Dika Ardit. Ethereum Smart Contracts: Security Vulnerabilities and Security Tools. 3.1.3 Vulnerabilities Explained. — Access mode: https://www.comp.nus.edu.sg/~loiluu/papers/oyente.pdf.

[5] Ethereum Bounty Program. — Access mode: https://bounty.ethereum.org/.

[6] Ethereum Contracts listing. — Access mode: https://blockchair.com/ethereum/calls?q=type(create)#.

[7] Ethereum Gas definition. — Access mode: https://www.investopedia.com/terms/g/gas-ethereum.asp.

[8] Ethereum JSON RPC API. — Access mode: https://github.com/ethereum/wiki/wiki/JSON-RPC.

[9] Etherscan. Block Explorer and Analytics. — Access mode: https://etherscan.io.

[10] MAIAN: automatic tool for finding trace vulnerabilities in Ethereum smart contracts. — Access mode: https://github.com/MAIAN-tool/MAIAN.

[11] Manticore. Symbolic execution tool. — Access mode: https://github.com/trailofbits/manticore.

[12] Mythril Classic: Security analysis tool for Ethereum smart contracts. — Access mode: https://github.com/ConsenSys/mythril-classic.

[13] Mythril Disassembler package. — Access mode: https://mythril-classic.readthedocs.io/en/master/mythril.disassembler.html.

[14] Mythril Vulnerabilities Naming. — Access mode: https://github.com/ConsenSys/mythril-classic/blob/develop/mythril/analysis/swc_data.py.

[15] Official Go implementation of the Ethereum protocol. — Access mode: https://github.com/ethereum/go-ethereum.

[16] Oyente. Making Smart Contracts Smarter. — Access mode: https://www.comp.nus.edu.sg/~loiluu/papers/oyente.pdf.

[17] Parity-Ethereum. The fast, light, and robust EVM and WASM client. — Access mode: https://github.com/paritytech/parity-ethereum.

[18] Perceptron. — Access mode: https://en.wikipedia.org/wiki/Perceptron.

[19] PostgreSQL: Open Source Database. — Access mode: https://www.postgresql.org/.

[20] A Postmortem on the Parity Multi-Sig Library Self-Destruct. — Access mode: https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/.

[21] Psycopg library. — Access mode: http://initd.org/psycopg/.

[22] RPC-API. Parity-Ethereum. trace module. — Access mode: https://wiki.parity.io/JSONRPC-trace-module.html.

[23] Requests library. — Access mode: https://2.python-requests.org/en/master/.

[24] SWC-101. Integer Overflow and Underflow. — Access mode: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-101.

[25] SWC-106. Unprotected SELFDESTRUCT Instruction. — Access mode: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-106.

[26] SWC-107. Reentrancy. — Access mode: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-107.

[27] SWC-109. Uninitialized Storage Pointer. — Access mode: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-109.

[28] SWC-110. Assert Violation. — Access mode: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-110.

[29] SWC-111. Use of Deprecated Solidity Functions. — Access mode: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-111.

[30] SWC-112. Delegatecall to Untrusted Callee. — Access mode: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-112.

[31] SWC-114. Transaction Order Dependence. — Access mode: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-114.

[32] SWC-115. Authorization through tx.origin. — Access mode: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-115.

[33] SWC-116. Timestamp Dependence. — Access mode: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-116.

[34] Securify. Security Scanner for Ethereum Smart Contracts. — Access mode: https://securify.chainsecurity.com/.

[35] Verified Contracts listing. — Access mode: https://etherscan.io/contractsVerified.

[36] scikit-learn: Machine Learning in Python. — Access mode: https://scikit-learn.org/stable/.