Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика, искусственный интеллект и системы управления

КАФЕДРА Системы обработки информации и управления

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

НА ТЕМУ:

Динамическое обнаружение уязвимостей в смарт-контрактах с использованием машинного обучения

Студент ИУ5-33М (Группа) _____ (Подпись, дата) И.Р. Солохов (И.О.Фамилия)

Руководитель _____ (Подпись, дата) Ю.Е. Гапанюк (И.О.Фамилия)

Консультант _____ (Подпись, дата) _____ (И.О.Фамилия)

2022 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ

Заведующий кафедрой _____
(Индекс)

_____ _____
(И.О.Фамилия)

« _____ » _____ 20 ____ г.

З А Д А Н И Е
на выполнение научно-исследовательской работы

по теме Динамическое обнаружение уязвимостей в смарт-контрактах с использованием машинного обучения

Студент группы ИУ5-33М

Солохов Ильдар Ринатович
(Фамилия, имя, отчество)

Направленность НИР (учебная, исследовательская, практическая, производственная, др.)

исследовательская

Источник тематики (кафедра, предприятие, НИР) кафедра

График выполнения НИР: 25% к 4 нед., 50% к 8 нед., 75% к 12 нед., 100% к 17 нед.

Техническое задание _____

Оформление научно-исследовательской работы:

Расчетно-пояснительная записка на ** листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)

Дата выдачи задания « ___ » _____ 20__ г.

Руководитель НИР _____ Ю.Е. Гапанюк
(Подпись, дата) (И.О.Фамилия)

Студент _____ И.Р. Солохов
(Подпись, дата) (И.О.Фамилия)

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

Содержание

стр.

Введение

Блокчейн — это распределенная книга, которая управляет активами между пользователями. Смарт-контракт кодирует правила для обработки передачи этих активов. Переводы происходят внутри транзакций, которые хранятся в блокчейне и являются постоянными. Таким образом, смарт-контракты могут применяться в широком диапазоне вариантов использования, включая финансовые и управленческие приложения. Например, договор может действовать как автономное соглашение между несколькими сторонами о переводе активов на желаемые счета при выполнении определенных условий.

Одной из самых популярных блокчейн-платформ, поддерживающих смарт-контракты, является Ethereum. Для выполнения операций со смарт-контрактом требуется плата за выполнение, которая называется газом. Плата за газ оплачивается в собственной валюте Ethereum, эфире (ETH).

Новая семантика и модель программирования смарт-контрактов затрудняют обеспечение их правильного поведения. Это делает их восприимчивыми к ошибкам или уязвимостям, которые могут быть использованы другими учетными записями в сети Ethereum.

Фактически, на основную сеть Ethereum было совершено несколько атак, которые привели к потере миллионов ETH. На сегодняшний день самой известной атакой на Ethereum была атака на децентрализованную автономную организацию (DAO), которая была проведена с использованием уязвимости повторного входа. В результате этой атаки было украдено 3,5 млн ETH.

Повторный вход включает в себя повторные вызовы одной и той же функции (или набора функций) до завершения первого вызова. Такие вложенные вызовы могут привести к неожиданному поведению смарт-контракта, что может быть использовано злоумышленником, обычно для перевода средств из контракта-жертвы. Повторный вход известен как одна из самых опасных уязвимостей в смарт-контрактах Ethereum.

Существующие инструменты для обнаружения уязвимостей повторного входа используют сложный анализ кода и созданные вручную правила для тщательного анализа потока управления и передачи активов в смарт-контрактах. Однако на уровне транзакций такие атаки явно не наблюдаются. В данной статье рассматривается совершенно новое направление:

- отслеживаются транзакции во время выполнения на уровне блокчейна Ethereum. Этот мониторинг не требует сложной проверки самих смарт-контрактов и позволяет развернуть технику непосредственно на клиенте блокчейна Ethereum, без каких-либо дополнительных действий. модификация смарт-контрактов или вовлеченного клиента.
- используется машинное обучение метаданных контролируемых транзакций. Это позволяет избежать необходимости разрабатывать правила (возможно, ошибочные), а также прокладывает путь к распознаванию новых типов уязвимостей в будущем.

Dynamit предназначен для анализа транзакций в смарт-контрактах и сообщения о вредоносных. Наша методика при использовании с моделью случайного леса показала высокую точность (96 %) на 105 транзакциях. Мы усреднили наши экспериментальные результаты по десяти итерациям настройки, в которой использовалась десятикратная перекрестная проверка для этапов обучения и тестирования всех моделей машинного обучения на итерацию.

1 Background

1.1 Ethereum

Смарт-контракты воплощают новую модель программирования, которая включает глобальное общее состояние (управляемое децентрализованным способом в блокчейне). Глобальное состояние, в котором хранятся все активы, автоматически управляется смарт-контрактами, которые представляют собой небольшие программы, выраженные в определенном формате, таком как байт-

код Ethereum. Этот байт-код обычно компилируется из языка высокого уровня, например Solidity. Код выполняется виртуальной машиной. Каждая инструкция также требует затрат, измеряемых в газе, которые должен заплатить инициатор (пользователь) смарт-контракта. Виртуальная машина управляет влиянием инструкций и их стоимостью на все активы в блокчейне. Потенциальные уязвимости могут возникать на разных уровнях этой архитектуры; реентерабельность обычно считается одной из самых тяжелых.

1.2 Работа программы

Методы анализа программы для обнаружения потенциальных уязвимостей можно разделить на статический анализ, который анализирует структуру кода без его запуска, и динамический анализ, который анализирует поведение выполняемой программы во время выполнения. Преимущество статического анализа в том, что он не требует тестового примера для выявления ошибки; и наоборот, у него есть недостаток, заключающийся в том, что анализ может быть слишком строгим и выявить ложные проблемы, которые на самом деле не являются ошибками, которые можно использовать во время выполнения программы. Динамический анализ, с другой стороны, всегда приводит к фактическому выполнению (и, таким образом, является свидетелем реальной проблемы), но может оказаться безуспешным при поиске правильных входных данных, чтобы это произошло. Также существуют комбинации этих методов, обычно в форме статического анализа для выявления частей программы, которые могут потребовать более тщательного изучения во время выполнения.

Фреймворк Dynamit обнаруживает уязвимости повторного входа в развернутых смарт-контрактах, не нуждаясь в их исходном коде. Dynamit учитывает только динамическое поведение смарт-контракта; это поведение извлекается из метаданных, описывающих транзакции между контрактами. Этот мониторинг основан на существующем интерфейсе прикладного программирования (API) немодифицированного клиента блокчейна Ethereum.

Dynamit состоит из двух частей (рисунок 1):

1. Монитор, который наблюдает за транзакциями в блокчейне.
2. Детектор, который классифицирует поведение как доброкачественное или злонамеренное.

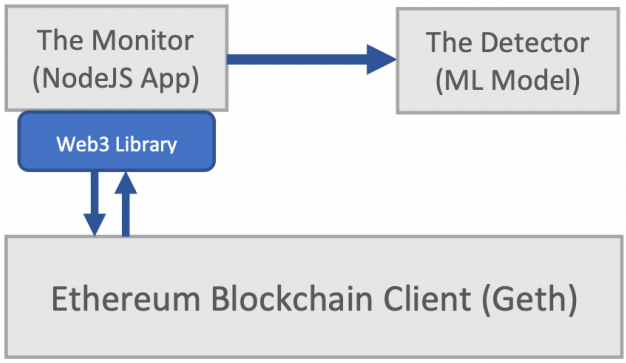

Рисунок 1 – Схема системы фреймворка Dynamit

Детектор можно настроить на различные классификаторы, которые сначала обучаются на обучающем наборе, прежде чем инструмент будет использоваться для обнаружения вредоносных транзакций в производственной среде.

Монитор подключается к клиенту блокчейна Ethereum для сбора информации о желаемых транзакциях. Он использует последнюю версию Web3js, которая является официальным API Javascript Ethereum для подключения и исследования сети Ethereum. Мониторинг получает данные следующим образом:

- подписка на события, генерируемые клиентом Ethereum. Эти события генерируются, когда выполняется транзакция, связанная с учетной записью. pendingTransactions используется для наблюдения за любыми новыми транзакциями, связанными с нашими счетами.
- проверка блокчейна через определенные промежутки времени до тех пор, пока не будет получена нужная информация. Это подходит для получения информации об уже добытой транзакции или получения состояния контракта после события.

1.3 Детектор

Детектор — это часть системы, которая отличает вредоносные транзакции от безопасных. Он состоит из части, которая обрабатывает и очищает данные, полученные монитором, и модели машинного обучения, которая обучается по мере того, как монитор вводит данные.

Чтобы найти лучшую модель, были обучены и протестированы в детекторе следующие модели:

- Random Forest,
- Naive Bayes classifier,
- Логистическая регрессия
- Метод k-ближайших соседей
- Метод опорных векторов: использовались как линейные, так и полиномиальные ядра. Модель с линейным ядром превзошла другую.

Обучили и протестированы пять различных типов классификаторов и сравнили их на основе средней частоты ложноположительных результатов (FPR) и ложноотрицательных результатов (FNR), а также точности, оценки F1 и отзыва (рисунок 2 и 3). FPR варьируется от 1,48 % (логистическая регрессия) до 5,74 % (наивный байесовский метод), в то время как FNR является самым низким для модели случайного леса (RF) и составляет 12,37 %.

Классификатор RF достигает наивысшей точности (93%). Большую часть неточностей моделей можно отнести к FNR. Другими словами, детектор помечает значительное количество вредоносных транзакций как безопасные (даже с использованием RF). И наоборот, низкий FPR делает Dynamit полезным в качестве инструмента мониторинга в сценариях, где стоимость ложных срабатываний довольно высока, например, при тестировании или приостановке проблемных контрактов в производстве для ручной проверки.

2 Экспериментальная часть

Было подобрано 25 контрактов с открытым исходным кодом для экспериментов, которые реализуют определенную функциональность, которую обозначена здесь как сервисные контракты.

Таблица 1 – Сет контрактов

Сервисный контракт	Пользовательский контракт
13 надежных контрактов	11 рабочих контрактов
12 уязвимых контрактов	9 вредоносных контрактов

Контракт на обслуживание может быть надежным (неэксплуатируемым) или содержать уязвимость; аналогично, пользовательский контракт может быть доброкачественным или злонамеренным. Только комбинация уязвимого сервисного контракта со злонамеренным пользователем может фактически выявить уязвимость в сервисном контракте.

В рамках эксперимента мы отслеживали в общей сложности 105 транзакций, созданных на основе этих контрактов, из них 53 безопасных и 52 вредоносных. Все эти транзакции были помечены вручную перед началом эксперимента, поэтому их можно использовать как для обучения, так и для тестирования контролируемой модели. Мы передаем помеченные данные о транзакциях в наш классификатор (в автономном режиме) для этапа обучения; в производстве могут использоваться онлайновые (немаркированные) данные.

Из 105 транзакций 25 транзакций были взяты из 25 сервисных контрактов с открытым исходным кодом, которые мы дополнили 20 вариантами пользовательских контрактов. Остальные 80 транзакций генерируются с использованием двух пар шаблонов контрактов (четыре контракта), которые случайным образом генерируют как вредоносные, так и безопасные транзакции. Contract Vulnerable2 — один из таких вариантов сервисного контракта, который жертвует случайную сумму пользователю (приложение А). Для генерации этих

случайных транзакций и служба, и пользовательский контракт (таблица 1) фаззируют свое поведение, чтобы представить различные варианты поведения в реальных сценариях. Другая причина случайного поведения (фаззинга) как в служебных, так и в пользовательских контрактах заключается в том, что могут выполняться сложные внутренние вычисления с определенной глубиной стека вызовов или использованием газа. Это потенциально может затруднить обнаружение атаки. Мы хотели бы, чтобы такое поведение было включено в наши данные, чтобы иметь менее предвзятый классификатор в детекторе. Следовательно, эти транзакции генерируются таким образом, чтобы предотвратить переоснащение модели. Например, мы фаззируем использование газа, вводя случайный цикл с вероятностью 50 % в уязвимый шаблон контракта (см. строки 12–18 на рис. 6). Каждое использование счетчика расходует дополнительный газ. Точно так же мы рандомизируем сумму, пожертвованную пользователю, и количество раз, когда злоумышленник фактически использует повторный вход, чтобы затруднить распознавание атак.

Поскольку каждое взаимодействие между сервисом и его пользователем может быть либо благотворным, либо вредным, могут возникнуть следующие результаты:

- Пользовательский контракт успешно использует уязвимость повторного входа: вредоносная транзакция.
- Пользовательский контракт пытается использовать уязвимость входа (которая может существовать, а может и не существовать в сервисном контракте), но безуспешно. Это приведет к одной из следующих ситуаций:
    - Транзакция и, соответственно, ее влияние на состояние целевого контракта отменяются средой выполнения Ethereum. Такие неудачные (отмененные) транзакции не видны через API мониторинга в Ethereum и, следовательно, не учитываются в нашем анализе.
    - Транзакция не отменяется и принимает предполагаемый первоначальный эффект: доброкачественная транзакция.

- Одноранговый контракт вообще не пытается использовать повторный вход: безопасная транзакция.

Как упоминалось ранее после того, как данные будут собраны монитором, они будут переданы детектору для классификации. Мы обучили и протестировали модели в детекторе, используя вышеупомянутые данные. Для всех наших моделей мы использовали стратифицированные 10-кратные перекрестно проверенные обучающие и тестовые наборы, чтобы получить последовательные и надежные результаты. Для каждого числа на графиках весь эксперимент (включая перекрестную проверку) проводился 10 раз, и бралась средняя производительность. Количество соседей в модели K-NN и количество деревьев в нашей модели RF выбираются на основе эмпирических наблюдений, чтобы максимизировать производительность модели.

3 Результаты

Обучили и протестировали пять различных типов классификаторов и сравнили их на основе средней частоты ложноположительных результатов (FPR) и ложноотрицательных результатов (FNR), а также точности, оценки F1 и отзыва (см. рис. 2 и 3). FPR варьируется от 1,48 % (логистическая регрессия) до 5,74 % (наивный байесовский метод), в то время как FNR является самым низким для модели случайного леса (RF) и составляет 12,37 %.

Классификатор RF достигает наивысшей точности (93%). Большую часть неточностей моделей можно отнести к FNR. Другими словами, детектор помечает значительное количество вредоносных транзакций как безопасные (даже с использованием RF). И наоборот, низкий FPR делает Dynamit полезным в качестве инструмента мониторинга в сценариях, где стоимость ложных срабатываний довольно высока, например, при тестировании или приостановке проблемных контрактов в производстве для ручной проверки.

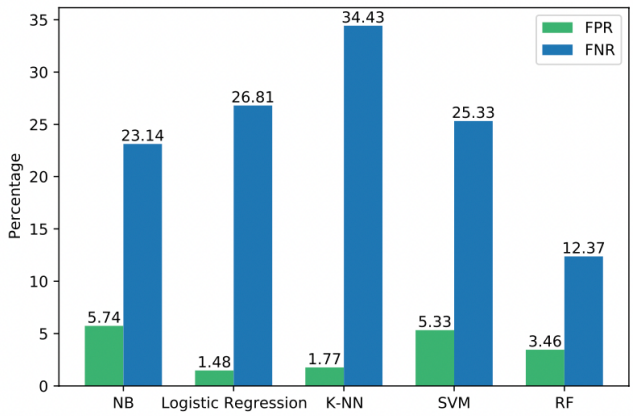

Рисунок 2 – Средняя доля ложноположительных и ложноотрицательных результатов для обнаружения уязвимых транзакций с различными моделями классификации.

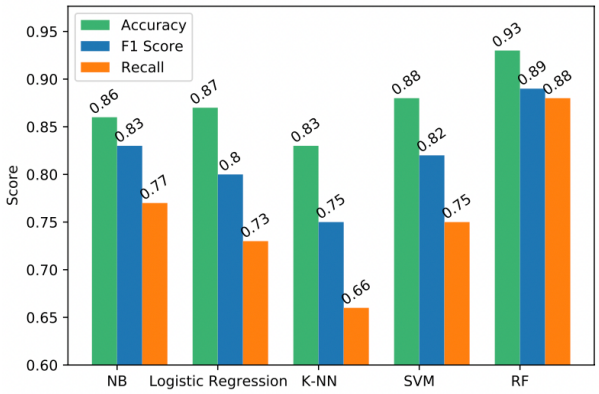

Рисунок 3 – Средняя точность, оценка F1 и полнота для обнаружения уязвимых транзакций с различными моделями классификации.

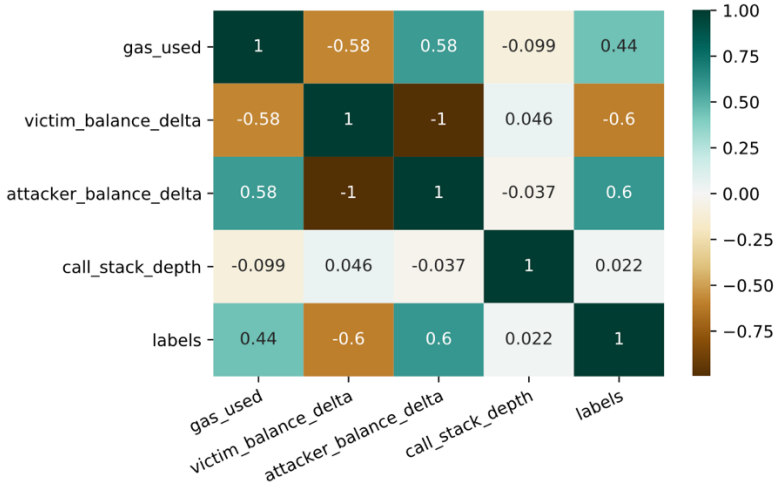

Рисунок 4 – Тепловая карта корреляции признаков для детекторов с меткой 1 для вредоносных транзакций и 0 для безопасных.

Наборы контрактов, которые были использованы для генерации случайных транзакций, пытаются скрыть свое поведение. Данная мера была использована, чтобы построить реалистичную модель и уменьшить погрешность. В результате корреляция средней глубины стека вызовов и метки транзакции очень низкая. Поэтому мы решили также построить те же модели без функции средней глубины стека вызовов. Результаты этой версии моделей показаны на рисунках 4 и 5.

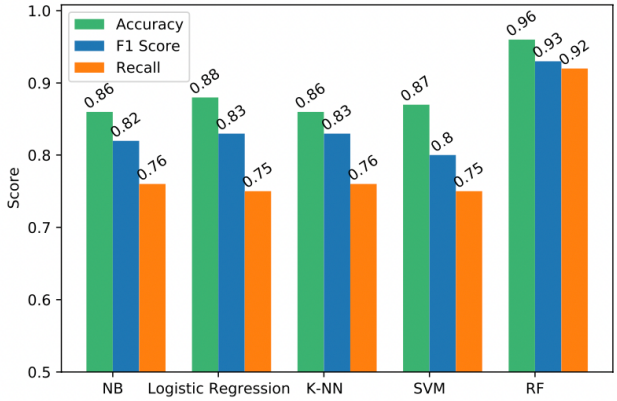

Рисунок 4 – Средняя доля ложноположительных и ложноотрицательных результатов для обнаружения уязвимых транзакций с различными моделями классификации без функции средней глубины стека вызовов.

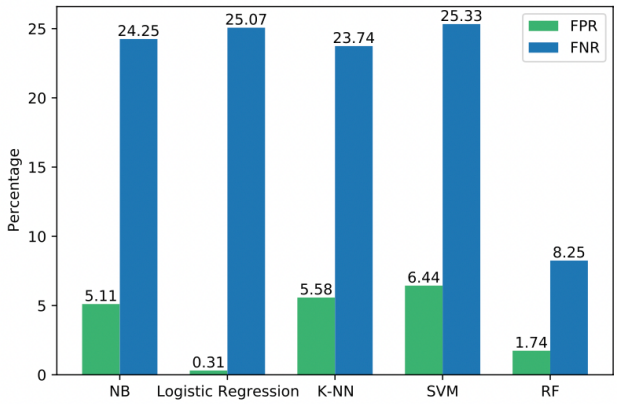

Рисунок 5 – Средняя точность, оценка F1 и отзыв для обнаружения уязвимых транзакций с различными моделями классификации без функции средней глубины стека вызовов.

Общее поведение всех моделей согласуется с результатами на рисунках 2 и 3. Однако есть несколько интересных изменений. Хотя RF по-прежнему является наиболее точной моделью и даже более точной, чем раньше, относительное уменьшение FPR для RF выше, чем для FNR. Самая высокая средняя точность в этом эксперименте принадлежит RF (96 %).

Заключение

В этой работе Dynamit был представлен, как динамическая среда обнаружения уязвимостей для смарт-контрактов Ethereum. Dynamit обнаруживает уязвимые смарт-контракты, классифицируя вредоносные транзакции в блокчейне, используя машинное обучение метаданных транзакций. Была достигнута точность 96 % на наборе данных из 105 транзакций.

Список использованных источников

1. Web3.js // web3.js - Ethereum JavaScript API URL: https://web3js.readthedocs.io (дата обращения: 22.05.2022).
2. Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning // Papers with code URL: https://paperswithcode.com (дата обращения: 22.05.2022).
3. Badruddoja S. et al. Making smart contracts smarter //2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). – IEEE, 2021. – С. 1-3.
4. Bailis P. et al. Research for practice: cryptocurrencies, blockchains, and smart contracts; hardware for deep learning //Communications of the ACM. – 2017. – Т. 60. – №. 5. – С. 48-51.
5. Momeni P., Wang Y., Samavi R. Machine learning model for smart contracts security analysis //2019 17th International Conference on Privacy, Security and Trust (PST). – IEEE, 2019. – С. 1-6.

Приложение А

Листинг смарт-контракта

```
contract Vulnerable2 {
    uint public gasFuzzingCounter = 0;
    uintpublicc=0;
    uint public d_binary = 0;
    uint public amnt;
    function random(uint num) private view returns (uint8) {
        return uint8(uint256(keccak256(block. timestamp, block.difficulty))%num);
}
        constructor() public payable {}
            function donate(address to_) public payable {
            d_binary = random_binary();
            c = random(10);
            if (d_binary == 1) {
                for (uint i = 0; i < c; i++) {
                gasFuzzingCounter++;
                }
        }
        amnt = random(1000) * 500000000000000;
        require(to_.call.value(amount)());
        }
}
```