УДК 004.413.4+004.056.5

Ермак М.А.

обучающаяся 4 курса, специальности «10.05.01» ФГБОУ ВО «Донской государственный технический университет», Россия, г. Ростов-на-Дону

МЕТОД ОБНАРУЖЕНИЯ МЕЖФУНКЦИОНАЛЬНЫХ REENTRANCY-УЯЗВИМОСТЕЙ В СМАРТ-КОНТРАКТАХ ETHEREUM НА ОСНОВЕ ДВУХГРАФОВОЙ МОДЕЛИ АНАЛИЗА ЗАВИСИМОСТЕЙ

Аннотация: Статья посвящена задаче обнаружения межфункциональных reentrancy-уязвимостей в смарт-контрактах Ethereum – класса дефектов, при котором уязвимость порождается взаимодействием нескольких функций через разделяемые переменные состояния и не обнаруживается методами локального анализа. Предложена двухграфовая модель, объединяющая межпроцедурный граф вызовов G = (V, E) и граф зависимостей состояния SG = (F, D), ребро в котором (fi, fj) ∈ D фиксирует семантическую зависимость через общее состояние. На основе модели определена функция риска Risk(fi, fj) с четырёхуровневой градацией и разработан алгоритм CrossFunctionReentrancyDetect со сложностью $O(|F|^2 \cdot |S|)$. Для оценки вклада графа SG проведено абляционное исследование на смешанной выборке из 37 контрактов: конфигурации без SG и с одним лишь графом G дают Precision = 0,938 и 0,935 соответственно, тогда как полная модель G + SG достигает Precision = 1,000 при том же Recall = 0,935. Выполнено прямое сравнение с инструментом Slither на идентичном датасете: Recall составляет 1,000 и 0,935 соответственно при Precision 0,969 и 1,000. Различие в профилях метрик отражает разные компромиссы между полнотой и точностью, что делает инструменты дополняющими, а не взаимозаменяемыми. Принципиальное свойство предложенного метода – интерпретируемость: для каждой обнаруженной пары функций формируется формально обоснованный путь атаки с явным указанием разделяемых переменных состояния и ребра в SG.

Ключевые слова: смарт-контракты, Ethereum, reentrancy, межфункциональная уязвимость, граф зависимостей состояния, статический анализ, абляционное исследование, интерпретируемость результатов.

Annotation: The paper addresses the problem of detecting cross-function reentrancy vulnerabilities in Ethereum smart contracts — a class of defects in which the vulnerability arises from the interaction of multiple functions through shared state variables and cannot be identified by local analysis methods. A two-graph model is proposed combining an interprocedural call graph G = (V, E) and a state dependency graph SG = (F, D), where an edge (fi, fj) ∈ D captures semantic dependency through shared state. Based on the model, a risk function Risk(fi, fj) with four-level classification is defined and the CrossFunctionReentrancyDetect algorithm with complexity $O(|F|^2 \cdot |S|)$ is developed. To assess the contribution of the SG graph, an ablation study was conducted on a mixed dataset of 37 contracts: configurations without SG and with the call graph G alone yield Precision = 0.938 and 0.935 respectively, while the full G + SG model achieves Precision = 1.000 at the same Recall = 0.935. A direct comparison with the Slither tool on the identical dataset was performed: Recall is 1.000 and 0.935 respectively at Precision 0.969 and 1.000. The difference in metric profiles reflects different trade-offs between completeness and precision, making the tools complementary rather than interchangeable. The key property of the proposed method is interpretability: for each detected vulnerable pair of functions, a formally grounded attack path is generated with explicit indication of shared state variables and the edge in SG.

Keywords: smart contracts, Ethereum, reentrancy, cross-function vulnerability, state dependency graph, static analysis, ablation study, interpretability of results.

1. ВВЕДЕНИЕ

Смарт-контракты Ethereum представляют собой программы, исполняемые в децентрализованной среде и управляющие финансовыми активами без участия посредников. Необратимость развёртывания – код

контракта не может быть изменён после публикации в сети – определяет принципиально иные требования к безопасности по сравнению с традиционным программным обеспечением: уязвимость не может быть устранена патчем, а её последствия необратимы.

Атака reentrancy, реализованная в 2016 году против контракта The DAO, повлекла хищение криптовалюты эквивалентом около 60 млн долларов США на момент инцидента и послужила одной из причин хардфорка сети Ethereum. Механизм атаки состоит в том, что внешний контракт, получив управление через callback-функцию, повторно входит в уязвимый контракт до завершения первоначальной транзакции, пользуясь несогласованностью состояния. Классическая форма reentrancy – повторный вход в ту же функцию – сегодня надёжно обнаруживается большинством статических анализаторов [2-6]; обзор современного состояния исследований в области безопасности Ethereum представлен в [18].

Однако существует менее изученная разновидность – межфункциональная (cross-function) reentrancy, при которой атака реализуется не через ту же функцию, а через другую публичную функцию контракта, разделяющую переменные состояния с атакуемой. Принципиальная особенность этого класса уязвимостей состоит в том, что ни одна из участвующих функций по отдельности не обнаруживает признаков классической reentrancy: уязвимость порождается исключительно их взаимодействием через общее состояние. Как следствие, методы, ограниченные анализом отдельных функций, не способны выявить данный класс дефектов по определению.

Настоящая работа направлена на устранение этого ограничения. Предлагается двухграфовая модель, в которой наряду с традиционным графом вызовов G = (V, E) вводится граф зависимостей состояния SG = (F, D) – самостоятельная структура анализа, явно представляющая семантические зависимости между функциями через разделяемые переменные. Вклад SG в снижение числа ложноположительных срабатываний подтверждается

экспериментально посредством абляционного исследования. Метод реализован в виде инструмента статического анализа с веб-интерфейсом и апробирован путём прямого экспериментального сравнения с инструментом Slither на стандартном датасете SmartBugs .

Новизна работы состоит не в построении графа зависимостей состояния как такового, а в использовании SG = (F, D) как обязательного критерия классификации кандидатов на межфункциональную reentrancy-уязвимость. В отличие от рассмотренных инструментов, применяющих зависимости по данным как вспомогательную структуру трассировки, предложенный метод использует ребро (fi, fj) ∈ D как необходимое условие, без выполнения которого пара не рассматривается как кандидат независимо от структуры G. Экспериментально показано, что исключение SG из алгоритма приводит к росту FP с 0 до 2 при неизменном Recall. Дополнительно предложена функция риска Risk(fi, fj) с четырёхуровневой градацией и разработан алгоритм CrossFunctionReentrancyDetect.

2. ПОСТАНОВКА ПРОБЛЕМЫ

Рассмотрим типичный сценарий межфункциональной reentrancy. Пусть контракт содержит функции withdraw() и transfer(), обе модифицирующие отображение balances. Функция withdraw() выполняет внешний вызов msg.sender.call{value: amount}("") до обновления balances[msg.sender]; функция transfer() изменяет balances произвольных адресов. При отсутствии блокировки повторного входа атакующий контракт из callback-функции вызывает transfer(), перераспределяя средства до того, как withdraw() зафиксирует факт вывода. Контрактный инвариант нарушается.

Существенно, что ни withdraw(), ни transfer() в отдельности не содержат признаков классической reentrancy: уязвимость возникает только при их совместном рассмотрении в контексте разделяемого состояния. Из этого непосредственно следует, что инструменты, анализирующие функции изолированно или использующие только граф управляющего потока без

семантических зависимостей по состоянию, не располагают информацией, необходимой для обнаружения данного класса уязвимостей.

Задача, решаемая в настоящей работе, состоит в построении такого формального представления контракта, которое явно кодирует зависимости между функциями через переменные состояния, и в разработке алгоритма обнаружения опасных конфигураций в этом представлении. Таксономия уязвимостей смарт-контрактов Ethereum, включающая reentrancy как один из ключевых классов, представлена в работе. Вместе с тем эмпирические данные показывают, что не все задокументированные уязвимости являются практически эксплуатируемыми; данное наблюдение согласуется с различием между Critical и High в предлагаемой функции риска.

3. ОБЗОР СУЩЕСТВУЮЩИХ МЕТОДОВ

3.1. Методы символьного выполнения

Mythril и Manticore применяют символьное выполнение на уровне байткода EVM совместно с SMT-решателями. Детекция reentrancy в Mythril основана на анализе последовательности опкодов CALL → SSTORE в рамках одной функции, что достаточно для классического паттерна, однако не охватывает зависимости между функциями. Масштабирование символьного выполнения на межпроцедурный анализ сопряжено с экспоненциальным ростом пространства состояний и на практике применимо лишь для небольших контрактов. ZEUS опирается на абстрактную интерпретацию с кодированием свойств безопасности в виде Horn-клауз; детекторы reentrancy также сфокусированы на внутрифункциональном уровне. Подход к верификации функциональных свойств смарт-контрактов методом проверки модели рассмотрен в. Направление фаззинга, управляемого символьным выполнением, применительно к смарт-контрактам исследовано в; масштабируемый статический анализ байткода реализован в системе Vandal.

3.2. Методы статического анализа

Slither преобразует исходный код Solidity в промежуточное представление SlithIR (форма SSA) и применяет набор детекторов на основе

анализа потоков данных. Детектор reentrancy-eth выявляет нарушения паттерна Checks-Effects-Interactions внутри одной функции. На тестовой подвыборке SmartBugs reentrancy (31 уязвимый контракт) Slither корректно идентифицирует все 31 контракт (Recall = 1,000), однако на авторской выборке из 6 безопасных контрактов, разработанных для целей настоящего исследования, даёт одно ложноположительное срабатывание. SmartCheck реализует сигнатурный анализ синтаксического дерева без семантического анализа зависимостей по состоянию. Securify формулирует шаблоны безопасности в Datalog, однако ограничен поддержкой современных версий Solidity и закрытым исходным кодом. Oyente – один из первых инструментов символьного анализа смарт-контрактов – не обновляется с 2018 года. Среди специализированных инструментов следует отметить MadMax, ориентированный на анализ условий переполнения газа, и Osiris – на обнаружение целочисленных переполнений; оба не нацелены на обнаружение reentrancy-уязвимостей. Для динамической защиты от reentrancy-атак предложены подходы на основе мониторинга времени выполнения [9, 10].

3.3. Анализ ограничений рассмотренных подходов

Общей чертой описанных инструментов является отсутствие явного представления семантических зависимостей между функциями через разделяемые переменные состояния в роли самостоятельного критерия обнаружения. Межпроцедурные графы зависимостей программ (PDG) [26, 27] применяются в ряде инструментов статического анализа [22, 25], однако в контексте обнаружения reentrancy они используются как вспомогательные структуры трассировки, а не как первичные фильтры кандидатов на уязвимость. Data-flow анализ Slither учитывает потоки данных, однако не конструирует отдельного объекта, фиксирующего пары (fi, fj) с непустым пересечением областей записи и использующего это пересечение как необходимое условие классификации. Методологические аспекты разработки безопасного системного программного обеспечения рассмотрены в [24, 25].

Именно эту роль выполняет вводимый в следующем разделе граф зависимостей состояния SG = (F, D).

4. ФОРМАЛЬНАЯ МОДЕЛЬ

4.1. Базовые определения

Определение 1 (смарт-контракт). Смарт-контракт C = (F, S), где F = {f1, ..., fn} – конечное множество функций, S = {s1, ..., sm} – конечное множество переменных состояния. Для каждой функции fi ∈ F определяются следующие атрибуты:

- R(fi) ⊆ S – переменные состояния, читаемые функцией fi;
- W(fi) ⊆ S — переменные состояния, изменяемые функцией fi;
- Ext(fi) ∈ {0,1} – наличие внешнего вызова (call, send, transfer, delegatecall);
- Protected(fi) ∈ {0,1} – наличие per-function защиты от повторного входа (модификатор nonReentrant или аналог);
- vis(fi) ∈ {public, external, internal, private} – уровень видимости функции;
- lineExt(fi) – номер строки первого внешнего вызова в fi; lineW(fi) – минимальный номер строки обновления переменных из W(fi). Формальная семантика операций EVM, лежащая в основе понятий Ext и W, описана в [13, 17]. Формальные аспекты конкурентной семантики смарт-контрактов рассмотрены в ; определение виртуальной машины Ethereum для целей интерактивного доказательства теорем – в [29].

4.2. Граф вызовов G

Определение 2. Граф вызовов G = (V, E): V = F – множество функций контракта; ребро (fi, fj) ∈ E существует тогда и только тогда, когда fi содержит вызов fj. Функция fj называется достижимой из fi (обозначение: fi →* fj), если в G существует путь из fi в fj. Граф G описывает структурные возможности управляющего потока, но не несёт информации о том, какие функции обращаются к общим переменным состояния.

4.3. Граф зависимостей состояния SG

Граф вызовов G фиксирует управляющие зависимости, не отражая семантических. Для явного представления последних вводится самостоятельная структура.

Определение 3. Граф зависимостей состояния SG = (F, D): ребро (fi, fj) ∈ D существует тогда и только тогда, когда

$$W(fi) \cap (R(fj) \cup W(fj)) \neq \emptyset.$$

Иными словами, некоторая переменная, изменяемая fi, читается или изменяется fj. Наличие ребра (fi, fj) ∈ D означает: если выполнение fi прервать после внешнего вызова и передать управление fj, последняя работает с состоянием, несогласованность которого может быть наблюдаема и эксплуатируема.

Граф G определяет структурно достижимые для атакующего функции; SG – семантически опасные среди них: те, что разделяют переменные состояния с атакуемой функцией. Межфункциональная reentrancy предполагает наличие рёбер в обоих графах одновременно.

Соотношение SG с классическими представлениями зависимостей по данным. Граф SG концептуально связан с межпроцедурным графом зависимостей программы (Program Dependence Graph, PDG) [26, 27]. Тем не менее между ними имеются существенные различия. Первое: SG строится исключительно над переменными состояния контракта – подмножеством данных, изменяемых персистентно между транзакциями Ethereum – а не над всеми переменными программы; это принципиально для задачи reentrancy, поскольку атака эксплуатирует именно персистентное состояние. Второе, и более важное: в классических PDG-подходах граф зависимостей используется как вспомогательная структура для нахождения путей и слайсинга, тогда как SG в предложенном методе выполняет роль самостоятельного критерия принятия решения – пара (fi, fj) не рассматривается алгоритмом как кандидат на уязвимость при отсутствии ребра в SG, независимо от структуры G. Третье: абляционное исследование (раздел 7.2) экспериментально подтверждает, что

граф G без SG-фильтра не улучшает Precision относительно сигнатурного подхода (FP = 2 в обоих случаях); следовательно, выигрыш в точности не воспроизводим через граф управляющего потока без явного представления зависимостей по состоянию. Таким образом, отличие SG от классических PDG-представлений состоит не в самой конструкции ребра, а в его функциональной роли в алгоритме: не вспомогательный инструмент трассировки, а первичный фильтр кандидатов.

4.4. Функция риска и критерий обнаружения

Функция риска Risk: F × F → {Critical, High, Medium, None} задаётся через следующие вспомогательные условия:

cond_ext := Ext(fi) = 1

cond_prot := Protected(fi) = 0

cond_rei := (fi →* fj) ∨ vis(fj) ∈ {public, external}

cond_dep := (fi, fj) ∈ D

cond_cei := lineExt(fi) < lineW(fi) [нарушение паттерна CEI]

Значения функции риска:

- Risk(fi, fj) = Critical: выполнены condext ∧ condprot ∧ condrei ∧ conddep ∧ cond_cei — все пять условий, включая явное нарушение паттерна Checks-Effects-Interactions;
- Risk(fi, fj) = High: выполнены condext ∧ condprot ∧ condrei ∧ conddep ∧ ¬cond_cei – CEI формально соблюдён в fi, однако fj разделяет переменные состояния с fi и доступна атакующему. Данная категория представляет потенциальный вектор; конкретная возможность эксплуатации зависит от семантики операций в fj. В реализации High трактуется как предупреждение, требующее ручной проверки;
- Risk(fi, –) = Medium (вырожденный случай): выполнены condext ∧ condprot, и при этом для всех fj ∈ F выполнено ¬cond_dep – граф SG не содержит рёбер из fi ни в одну другую функцию. Запись Risk(fi, –) подчёркивает, что уязвимость рассматривается без явной целевой

функции; наличие незащищённого внешнего вызова при нескольких функциях с Ext расширяет поверхность атаки;

– Risk(fi, fj) = None: Protected(fi) = 1 либо Ext(fi) = 0.

Свойство (в рамках принятой модели). Для контрактов, не использующих модификаторы как основной механизм контроля повторного входа и не содержащих транзитивных внешних вызовов через цепочки внутренних функций, необходимым условием классификации пары (fi, fj) как уязвимой данным методом является Risk(fi, fj) ∈ {Critical, High}. Необходимость условия cond_dep следует из того, что в рамках принятой модели, без ребра (fi, fj) ∈ D атакующий, получив управление через callback, не располагает функцией fj, через которую мог бы изменить состояние fi в наблюдаемой форме; случаи косвенного изменения состояния через вспомогательные структуры или делегирующие вызовы выходят за рамки модели. Контракты, нарушающие данные предположения, образуют FN-случаи метода и рассмотрены в разделе 9.

Пример вычисления Risk. Рассмотрим пару (withdraw, transfer) из контракта reentrancycrossfunction.sol. Функция withdraw() содержит внешний вызов на строке 20 (lineExt = 20), обновление userBalances – на строке 22 (lineW = 22). Функция transfer() изменяет ту же переменную и имеет видимость public. Проверка: condext = 1; condprot = 1 (Protected(withdraw) = 0, то есть модификатор защиты отсутствует, условие выполнено); condrei = 1 (transfer публична); conddep = 1, поскольку W(withdraw) ∩ W(transfer) = {userBalances} ≠ ∅, то есть (withdraw, transfer) ∈ D; condcei = 1, так как lineExt(20) < lineW(22). Все пять условий выполнены → Risk(withdraw, transfer) = Critical. Для контраста: в safecei.sol обновление balances стоит на строке 14, вызов – на строке 15, поэтому condcei = 0 → Risk = High; реализация дополнительно проверяет updatesafter = ∅ и снижает до None, корректно признавая контракт безопасным.

Введённая формальная модель C = (F, S), граф вызовов G и граф зависимостей состояния SG совместно определяют пространство кандидатов на уязвимость. В следующем разделе на основе этих объектов строится алгоритм обнаружения, последовательно применяющий G и SG как фильтры.

5. АЛГОРИТМ ОБНАРУЖЕНИЯ

5.1. Конвейер преобразований

Метод реализуется в виде последовательного конвейера, каждый этап которого соответствует одному из теоретических объектов модели:

Исходный код Solidity

↓ Лексико-синтаксический анализ → {F, S, Ext, W, R, vis, lineExt, lineW}

↓ Построение G = (V, E) – граф вызовов

↓ Построение SG = (F, D) – граф зависимостей состояния

↓ BFS(G, fi): Reach(fi) для каждой fi с Ext(fi) = 1

↓ SG-фильтр: отбор пар (fi, fj) с (fi, fj) ∈ D

↓ Вычисление Risk(fi, fj) для отобранных пар

↓ Формирование отчёта: severity, attackpath, sharedvars

5.2. Псевдокод алгоритма CrossFunctionReentrancyDetect

Вход: src – исходный код Solidity-контракта

Выход: V_set – множество обнаруженных уязвимостей

```
1.  C  ← LexSyntaxParse(src)
2.  G  ← BuildCallGraph(C)
3.  SG ← BuildStateDependencyGraph(C)   // O(|F|² · |S|)
4.  V_set ← ∅
5.  for each fi ∈ F do
6.      if Ext(fi) = 0  or  Protected(fi) = 1 then continue
7.      Reach ← BFS(G, fi)
8.      Pub   ← { fj ∈ F | vis(fj) ∈ {public, external},  fj ≠ fi }
```

```
9.    for each fj ∈ Reach ∪ Pub do
10.       if (fi, fj) ∉ D then continue          // SG-фильтр
11.       r ← Risk(fi, fj)
12.       if r ≠ None: Vset ← Vset ∪ {(fi, fj, r, W(fi) ∩ W(fj))}
13.   if ∀ fj: (fi, fj) ∉ D  and  |{fk | Ext(fk)}| > 1:
14.       Vset ← Vset ∪ {(fi, fi, Medium, ∅)}
15. return V_set
```

Временная сложность алгоритма определяется этапом построения SG – $O(|F|^2 \cdot |S|)$, где $|F|$ число функций контракта, $|S|$ – число переменных состояния. BFS-обход графа G добавляет слагаемое $O(|F|^2)$. Итоговая сложность $O(|F|^2 \cdot |S|)$ является полиномиальной и практически применима для контрактов промышленного масштаба: анализ контракта с 5-10 функциями выполняется за 3-8 мс, наиболее сложного контракта в тестовой выборке (spankchainpayment.sol, 22 функции) – за ~50 мс.

Описанный алгоритм реализован в виде программного инструмента. В следующем разделе рассмотрена архитектура реализации, инженерные решения и свойства, обеспечивающие практическую применимость метода.

6. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ

Метод реализован на языке Python 3.10 с веб-интерфейсом на основе Flask. Система структурирована в три последовательных модуля: SolidityParser (parser.py) — лексико-синтаксический разбор Solidity без привлечения компилятора solc, совместимость с версиями 0.4.x–0.8.x; CallGraphBuilder (graph.py) — построение G = (V, E) и SG = (F, D), per-function определение Protected(fi); ReentrancyDetector (detector.py) — реализация алгоритма раздела 5.2 с формированием объекта AnalysisReport. Исходный код будет опубликован в открытом репозитории после завершения рецензирования.

Ключевое инженерное решение — отказ от компилятора solc — обеспечивает время анализа 3–8 мс для контрактов с 5–10 функциями и ~50 мс для наиболее сложного контракта выборки (22 функции), что на порядки

быстрее символьного выполнения [6, 12] и делает инструмент пригодным для интеграции в CI/CD-конвейеры. Для классификации High в реализации дополнительно проверяется отсутствие обновлений состояния после внешнего вызова (updates_after = ∅): если CEI соблюдён, пара не получает срабатывания, что устраняет ложноположительные результаты на корректных контрактах. Ограничение: inline assembly, множественное наследование и вызовы через library могут обрабатываться некорректно.

Отличительным свойством реализации является интерпретируемость: для каждой уязвимости формируется путь атаки вида «fi() → call() → [attacker.fallback()] → fj()», множество разделяемых переменных W(fi) ∩ W(fj) и рекомендация по исправлению. Веб-интерфейс содержит четыре вкладки: отчёт по уязвимостям, сводная таблица, SVG-визуализация G с выделением рёбер SG, бенчмарк на датасете SmartBugs.

7. ЭКСПЕРИМЕНТАЛЬНАЯ ОЦЕНКА

7.1. Тестовая выборка

Для корректной оценки как полноты обнаружения (Recall), так и точности (Precision) сформирована смешанная выборка из 37 контрактов. Уязвимые контракты (31) взяты из датасета SmartBugs Curated , категория reentrancy — общепринятого бенчмарка для сравнения инструментов анализа смарт-контрактов, включающего 11 именованных эталонных примеров и 20 реальных контрактов Ethereum. Безопасные контракты (6) разработаны авторами и охватывают основные паттерны защиты от reentrancy: соблюдение порядка CEI (Checks-Effects-Interactions), ручной мьютекс (nonReentrant), паттерн pull payment, отсутствие внешних вызовов, применение OpenZeppelin ReentrancyGuard. Корректность разработанных контрактов дополнительно верифицирована инструментом Slither: ни один из шести контрактов не получил предупреждений категории reentrancy, что подтверждает их принадлежность к отрицательной подвыборке. Необходимо, тем не менее, сделать существенную оговорку: выборка из шести безопасных контрактов является критически малой для статистически значимой оценки Precision.

Значение Precision = 1,000 не может интерпретироваться как свидетельство общего отсутствия ложноположительных срабатываний; оно лишь показывает, что на шести специально разработанных контрактах FP отсутствует. Для статистически обоснованных выводов необходима выборка из 50–100 безопасных контрактов реальных репозиториев (OpenZeppelin, Uniswap, Aave, Compound).

Настоящее исследование представляет собой пилотную экспериментальную валидацию метода. Использование стандартного датасета SmartBugs Curated обеспечивает воспроизводимость результатов и их сопоставимость с опубликованными работами по анализу смарт-контрактов.

Таблица 1. Состав смешанной тестовой выборки

Подвыборка	**N**	**Ground truth**	**Источник**
Уязвимые	31	Positive	SmartBugs Curated [7]
Безопасные	6	Negative	Авторская разработка
Итого	37	–	–

7.2. Абляционное исследование: вклад графа SG

Для экспериментальной проверки гипотезы о том, что именно граф зависимостей состояния SG обеспечивает снижение числа ложноположительных срабатываний, проведено абляционное исследование на смешанной выборке из 37 контрактов. Сравниваются три конфигурации детектора:

- Без SG — сигнатурный подход: каждая функция с Ext(fi) = 1 и Protected(fi) = 0 рассматривается как уязвимая без анализа зависимостей по состоянию;
- Только G — используется BFS-обход графа вызовов для нахождения достижимых функций, однако SG-фильтр (строка 10 алгоритма) отключён;

– G + SG — полный метод с обязательным условием (fi, fj) ∈ D перед вычислением Risk.

Таблица 2. Абляционное исследование (смешанная выборка, 37 контрактов; доверительные интервалы по методу Уилсона, 95%)

Конфигурация	**TP**	**FN**	**FP**	**TN**	**Precision**	**95% ДИ**	**Recall**	**95% ДИ**	**F1**
Без SG	30	1	2	4	0,938	[0,799;0,983]	0,968	[0,838;0,994]	0,952
Только SG	29	2	2	4	0,935	[0,793;0,982]	0,935	[0,793;0,982]	0,935
G+SG	29	2	0	6	1,000	[0,883;1,000]	0,935	[0,793;0,982]	0,966

Из таблицы 2 следует несколько наблюдений. Асимметрия TP = 30 в строке «Без SG» против TP = 29 в строках «Только G» и «G + SG» объясняется следующим: сигнатурный подход обнаруживает один дополнительный контракт (reentrancy_bonus.sol), поскольку не требует наличия ребра в SG и срабатывает на любую функцию с внешним вызовом; полные конфигурации с SG-фильтром или BFS-проверкой не классифицируют его как уязвимый из-за специфики транзитивного вызова. При этом «Без SG» даёт FP = 2, то есть более высокий TP достигается ценой ложных срабатываний. Конфигурация «Только G» (BFS-обход без SG-фильтра) даёт то же число ложноположительных срабатываний (FP = 2), что и «Без SG». Таким образом, включение графа вызовов само по себе не улучшает Precision: именно условие (fi, fj) ∈ D в графе SG является механизмом устранения ложных срабатываний. Снижение FP с 2 до 0 при переходе к конфигурации «G + SG» достигается без уменьшения Recall. Это подтверждает предположение о том, что SG несёт семантическую информацию, недоступную из одного лишь графа управляющего потока.

7.3. Результаты на смешанной выборке

Таблица 3. Результаты анализа (смешанная выборка, 37 контрактов)

Метрика	**Значение**	**Примечание**
True Positive (TP)	29	Уязвимые контракты, верно идентифицированы
False Negative (FN)	2	modifierreentrancy.sol, reentrancybonus.sol
False Positive (FP)	0	Все 6 безопасных верно классифицированы

True Negative (TN)	6	–
Recall	0,935 [0,793; 0,982]	29 / 31; 95% ДИ по Уилсону
Precision	1,000 [0,883; 1,000]	29 / 29; 95% ДИ по Уилсону
F1	0,966	–
Severity Critical	25	Нарушение паттерна CEI
Severity High	3	Cross-function через граф SG (засчитаны как TP, см. примечание)
Severity Medium	1	Отсутствие защиты при нескольких внешних вызовах

Примечание о категории High. В таблице 3 случаи Severity High включены в TP, поскольку все три контракта SmartBugs, получивших данную классификацию, задокументированы в датасете как уязвимые . Таким образом, High трактуется как предупреждение, требующее ручной проверки со стороны разработчика, однако в контексте оценки на датасете с известной разметкой ground truth оно засчитывается как корректное обнаружение. При использовании метода на контрактах без разметки рекомендуется рассматривать High отдельно от Critical и проводить ручной аудит соответствующих пар функций.

Два ложноотрицательных результата обусловлены ограничениями модели. Контракт modifierreentrancy.sol реализует уязвимость через взаимодействие двух модификаторов Solidity: данный паттерн принципиально выходит за рамки анализа тел функций и потребует расширения формальной модели на уровень модификаторов. Контракт reentrancybonus.sol содержит транзитивный внешний вызов — функция getFirstWithdrawalBonus() вызывает withdrawReward(), которая выполняет вызов вовне, тогда как обновление состояния claimedBonus происходит в вызывающей функции после возврата. Оба случая идентифицированы как направления дальнейших исследований.

Полученные результаты приобретают содержательный смысл только в сопоставлении с существующими инструментами. Раздел 8 посвящён прямому экспериментальному сравнению со Slither на идентичной выборке.

8. СРАВНЕНИЕ СО SLITHER

Для получения сопоставимых результатов Slither запущен на идентичной смешанной выборке из 37 контрактов. Версия компилятора выбиралась через solc-select в соответствии с директивой pragma solidity каждого контракта (0.4.24 для контрактов версии 0.4.x, 0.8.0 для прочих).

Таблица 4.

Сравнение предложенного метода и Slither (смешанная выборка, 37 контрактов)

Метрика	**Предложенный метод**	**95% ДИ**	**Slither**	**95% ДИ**
TP/FN/FP/TN	29/2/0/6	–	31/0/1/5	–
Recall	0,935	[0,793;0,982]	1,000	[0,890;1,000]
Precision	1,000	[0,883;1,000]	0,969	
F1	0,966	–	0,984	–
Явный граф SG=(F,D)	Да	–	Нет	–
Путь атаки с W(fi) ∩ W(fj)	Да	–	Нет	–
Независимость от компилятора	Да	–	Нет	–

Slither демонстрирует более высокий Recall (1,000 против 0,935): SSA-анализ потоков данных охватывает более широкий класс паттернов reentrancy, в том числе уязвимости в модификаторах и транзитивные цепочки вызовов. Различие по Recall устойчиво наблюдается на исследуемой выборке. Различие

по Precision (1,000 против 0,969) вследствие малого размера отрицательной подвыборки (n = 6) не может считаться статистически значимым: доверительные интервалы существенно перекрываются ([0,883; 1,000] и [0,843; 0,994] соответственно), а сам показатель основан на недостаточном числе наблюдений для обобщающих выводов. Указанное ограничение в равной мере распространяется на интерпретацию обоих инструментов.

Различие в профилях метрик отражает различие в целевых задачах. Несмотря на более низкий Recall на данной выборке (0,935 против 1,000), предложенный метод ориентирован не на максимизацию полноты обнаружения всех разновидностей reentrancy, а на интерпретируемое выявление межфункциональных зависимостей через явное представление состояния. Slither позиционируется как инструмент общего аудита с приоритетом полноты; предложенный метод решает более специализированную задачу — интерпретируемое обнаружение межфункциональных reentrancy-уязвимостей (explainable cross-function reentrancy detection). Отличительной особенностью предложенного подхода, отсутствующей в стандартном выводе Slither, является формально обоснованный путь атаки: для каждой обнаруженной пары (fi, fj) метод явно указывает ребро $(fi, fj) \in D$ в графе SG и конкретное множество $W(fi) \cap W(fj)$, обосновывая механизм уязвимости в терминах модели. Это делает результаты пригодными для автоматической генерации объяснений для разработчика и потенциальной интеграции в формальные процессы верификации. Инструменты решают смежные, но не тождественные задачи и могут применяться совместно: Slither — для первичного скрининга, предложенный метод — для детального анализа обнаруженных кандидатов с формальным обоснованием.

9. ОБСУЖДЕНИЕ

Абляционное исследование устанавливает причинно-следственную связь: нулевое значение FP в конфигурации «G + SG» обеспечивается именно SG-фильтром (строка 10 алгоритма), а не BFS-обходом G. Два механизма

совместно обеспечивают точность: условие (fi, fj) ∈ D отсекает пары функций без реальной зависимости по состоянию, а требование cond_cei для классификации High дополнительно исключает контракты, соблюдающие паттерн CEI при наличии рёбер в SG.

Оба ложноотрицательных результата обусловлены принципиальными ограничениями формальной модели, а не ошибками реализации. Контракт modifierreentrancy.sol реализует уязвимость через взаимодействие двух модификаторов: ни один из них не является функцией в смысле модели, поэтому ни Ext, ни W не определены для них. Контракт reentrancybonus.sol содержит транзитивный внешний вызов: getFirstWithdrawalBonus() вызывает withdrawReward(), которая выполняет вызов вовне, тогда как обновление claimedBonus происходит в вызывающей функции после возврата. Оба случая требуют расширения модели: первый — введением анализа модификаторов как отдельного уровня, второй — транзитивным замыканием атрибута Ext по рёбрам G. Такое расширение планируется в продолжении работы.

Практическая применимость. Полиномиальная сложность $O(|F|^2 \cdot |S|)$ и отсутствие зависимости от компилятора делают инструмент пригодным для интеграции в конвейеры непрерывной интеграции (CI/CD). Задачи автоматизированного анализа защищённости программных систем в более широком контексте рассмотрены в . Время анализа — 3–50 мс — не создаёт значимых задержек даже при запуске при каждом коммите. Формируемые отчёты содержат конкретные пути атак и переменные состояния, что позволяет разработчику немедленно локализовать и устранить уязвимость без дополнительного ручного аудита для случаев Critical. Для случаев High рекомендуется дополнительная проверка семантики fj, поскольку эксплуатируемость зависит от конкретных операций чтения.

Ограничения эксперимента. Выборка из шести безопасных контрактов представляет собой необходимый минимум для демонстрации нулевого FP, однако не является достаточной основой для статистически значимых выводов о Precision на произвольных контрактах. Для получения таких выводов

потребуется смешанная выборка объёмом не менее 50–100 безопасных контрактов из реальных репозиториев (в частности, из библиотек OpenZeppelin, Uniswap v2 и аналогичных).

10. ЗАКЛЮЧЕНИЕ

В работе предложен и реализован метод обнаружения межфункциональных reentrancy-уязвимостей в смарт-контрактах Ethereum, основанный на двухграфовой модели (G, SG). Граф зависимостей состояния SG = (F, D) используется как самостоятельный критерий классификации кандидатов: наличие ребра (fi, fj) ∈ D является обязательным условием отнесения пары к потенциально уязвимой в рамках принятой модели. Алгоритм CrossFunctionReentrancyDetect имеет полиномиальную вычислительную сложность $O(|F|^2 \cdot |S|)$.

Проведённое абляционное исследование показало, что конфигурация без SG-фильтра (включая вариант с одним лишь BFS-обходом по G) даёт FP = 2, тогда как полная модель G + SG достигает FP = 0 при неизменном Recall = 0,935. Этот результат подтверждает вклад графа SG как самостоятельного элемента метода, недублируемого графом вызовов. Экспериментальное сравнение со Slither показало различие профилей Recall и Precision (0,935 против 1,000 и 1,000 против 0,969 соответственно), обусловленное различием целевых задач инструментов. Отличительной особенностью предложенного подхода является интерпретируемость — формально обоснованный путь атаки с явным указанием разделяемых переменных и ребра в SG.

Среди приоритетных направлений развития метода следует выделить: расширение формальной модели на анализ модификаторов Solidity; поддержку транзитивных внешних вызовов; межконтрактный анализ цепочек delegatecall; верификацию Precision на расширенной смешанной выборке из реальных репозиториев.

Литература:

1. Mehar M.I., Shier C.L., Giambattista A., Gong E., Fletcher G., Sanayhie R., Kim H.M., Laskowski M. Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack // Journal of Cases on Information Technology. — 2019. — Vol. 21, No. 1. — P. 19–32. DOI: 10.4018/JCIT.2019010102
2. Feist J., Grieco G., Groce A. Slither: A Static Analysis Framework for Smart Contracts // Proc. IEEE/ACM 2nd Intl. Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). — 2019. — P. 8–15. DOI: 10.1109/WETSEB.2019.00008
3. Tsankov P., Dan A., Drachsler-Cohen D., Gervais A., Bunzli F., Vechev M. Securify: Practical Security Analysis of Smart Contracts // Proc. 2018 ACM SIGSAC CCS. — 2018. — P. 67–82. DOI: 10.1145/3243734.3243780
4. Tikhomirov S., Voskresenskaya E., Ivanitskiy I., Takhaviev R., Marchenko E., Alexandrov Y. SmartCheck: Static Analysis of Ethereum Smart Contracts // Proc. IEEE/ACM 1st Intl. Workshop WETSEB. — 2018. — P. 9–16. DOI: 10.1145/3194113.3194115
5. Luu L., Chu D.-H., Olickel H., Saxena P., Hobor A. Making Smart Contracts Smarter // Proc. 2016 ACM SIGSAC CCS. — 2016. — P. 254–269. DOI: 10.1145/2976749.2978309
6. Mueller B. Smashing Ethereum Smart Contracts for Fun and Real Profit // 9th Hack In The Box Security Conference (HITB SECCONF). — Amsterdam : HITB, 2018.
7. Durieux T., Ferreira J.F., Abreu R., Cruz P. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts // Proc. 42nd ICSE. — 2020. — P. 530–541. DOI: 10.1145/3377811.3380364
8. Atzei N., Bartoletti M., Cimoli T. A Survey of Attacks on Ethereum Smart Contracts (SoK) // Proc. POST 2017. — P. 164–186. DOI: 10.1007/978-3-662-54455-6_8

9. Liu C., Liu H., Cao Z., Chen Z., Chen B., Roscoe B. ReGuard: Finding Reentrancy Bugs in Smart Contracts // Proc. 40th ICSE Companion. — 2018. DOI: 10.1145/3183440.3183495
10. Rodler M., Li W., Karame G.O., Davi L. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks // Proc. NDSS 2019. DOI: 10.14722/ndss.2019.23413
11. Kalra S., Goel S., Dhawan M., Sharma S. ZEUS: Analyzing Safety of Smart Contracts // Proc. NDSS 2018. DOI: 10.14722/ndss.2018.23082
12. Mossberg M., Manzano F., Hennenfent E., Groce A., Grieco G., Feist J., Brunson T., Dinaburg A. Manticore: A User-Friendly Symbolic Execution Framework // Proc. ASE 2019. — P. 1186–1189. DOI: 10.1109/ASE.2019.00133
13. Grishchenko I., Maffei M., Schneidewind C. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts // Proc. POST 2018. — P. 243–269. DOI: 10.1007/978-3-319-89722-6_10
14. Perez D., Livshits B. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited // Proc. USENIX Security 2021. — P. 1325–1341.
15. Grech N., Kong M., Jurisevic A., Brent L., Scholz B., Smaragdakis Y. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts // PACMPL (OOPSLA). — 2018. — Vol. 2. DOI: 10.1145/3276486
16. Torres C.F., Schütte J., State R. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts // Proc. ACSAC 2018. — P. 664–676. DOI: 10.1145/3274694.3274737
17. Wood G. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Yellow Paper. — Ethereum Foundation, 2014.
18. Tikhomirov S.A. Ethereum: State of Knowledge and Research Perspectives // Proc. FTC 2017. — P. 206–212. DOI: 10.1109/FTC.2017.8239518

19. He J., Balunovic M., Ambroladze N., Tsankov P., Vechev M. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts // Proc. ACM CCS 2019. — P. 531–548. DOI: 10.1145/3319535.3363230
20. Brent L., Jurisevic A., Kong M. et al. Vandal: A Scalable Security Analysis Framework for Smart Contracts // arXiv:1809.03981. — 2018.
21. Шишкин Е.С. Проверка функциональных свойств смарт-контрактов методом символьной верификации модели // Труды ИСП РАН. — 2018. — Т. 30, № 5. — С. 265–288. DOI: 10.15514/ISPRAS-2018-30(5)-15
22. Захаров В.А., Подловченко Р.И. Полиномиальный по сложности алгоритм, распознающий коммутативную эквивалентность схем программ // Доклады Российской Академии наук. — 1998. — Т. 362, № 6. — С. 744–747.
23. Котенко И.В., Чечулин А.А. Активный анализ защищённости компьютерных сетей // Вопросы кибербезопасности. — 2013. — № 1. — С. 22–34.
24. Девянин П.Н., Тележников В.Ю., Хорошилов А.В. Формирование методологии разработки безопасного системного программного обеспечения на примере операционных систем // Труды ИСП РАН. — 2021. — Т. 33, № 5. — С. 25–40. DOI: 10.15514/ISPRAS-2021-33(5)-2
25. Бакулин М.Г., Гайсарян С.С., Курмангалеев Ш.Ф., Падарян В.А. Комбинированный статический и динамический анализ бинарного кода // Труды ИСП РАН. — 2012. — Т. 23. — С. 257–276. DOI: 10.15514/ISPRAS-2012-23-16
26. Ferrante J., Ottenstein K.J., Warren J.D. The Program Dependence Graph and Its Use in Optimization // ACM Trans. Programming Languages and Systems. — 1987. — Vol. 9, No. 3. — P. 319–349. DOI: 10.1145/24039.24041

27. Reps T., Horwitz S., Sagiv M. Precise Interprocedural Dataflow Analysis via Graph Reachability // Proc. 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). — 1995. — P. 49–61. DOI: 10.1145/199448.199462
28. Sergey I., Hobor A. A Concurrent Perspective on Smart Contracts // Proc. 1st Workshop on Trusted Smart Contracts. — 2017. DOI: 10.1007/978-3-319-70278-0_12
29. Hirai Y. Defining the Ethereum Virtual Machine for Interactive Theorem Provers // Proc. 1st Workshop on Trusted Smart Contracts. — 2017. DOI: 10.1007/978-3-319-70278-0_10
30. Antonopoulos A.M., Wood G. Mastering Ethereum: Building Smart Contracts and DApps. — Sebastopol, CA: O'Reilly Media, 2018. — 452 p.