Enforcing Control Flow Integrity on DeFi Smart Contracts

Zhiyang Chen
University of Toronto
Toronto, Canada
zhiychen@cs.toronto.edu

Sidi Mohamed Beillahi
University of Toronto
Toronto, Canada
sm.beillahi@utoronto.ca

Pasha Barahimi*
University of Tehran
Tehran, Iran
pashabarahimi@gmail.com

Cyrus Minwalla
Bank of Canada
Ottawa, Canada
CMinwalla@bank-banque-canada.ca

Han Du
Bank of Canada
Ottawa, Canada
HDu@bank-banque-canada.ca

Andreas Veneris
University of Toronto
Toronto, Canada
veneris@eecg.toronto.edu

Fan Long
University of Toronto
Toronto, Canada
fanl@cs.toronto.edu

Abstract

Smart contracts power decentralized financial (DeFi) services but are vulnerable to security exploits that can lead to significant financial losses. Existing security measures often fail to adequately protect these contracts due to the composability of DeFi protocols and the increasing sophistication of attacks. Through a large-scale empirical study of historical transactions from the 37 hacked DeFi protocols, we discovered that while benign transactions typically exhibit a limited number of unique control flows, in stark contrast, attack transactions consistently introduce novel, previously unobserved control flows. Building on these insights, we developed CrossGuard, a novel framework that enforces control flow integrity onchain to secure smart contracts. Crucially, CrossGuard does not require prior knowledge of specific hacks. Instead, configured only once at deployment, it enforces control flow whitelisting policies and applies simplification heuristics at runtime. This approach monitors and prevents potential attacks by reverting all transactions that do not adhere to the established control flow whitelisting rules. Our evaluation demonstrates that CrossGuard effectively blocks 35 of the 37 analyzed attacks when configured only once at contract deployment, maintaining a low false positive rate of 0.26% and minimal additional gas costs. These results underscore the efficacy of applying control flow integrity to smart contracts, significantly enhancing security beyond traditional methods and addressing the evolving threat landscape in the DeFi ecosystem.

CCS Concepts

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**.

*Pasha is an incoming Ph.D. student at the University of Southern California. Work done while the author was a remote research intern at the University of Toronto.

ICSE '26, April 12–18, 2026, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773266

Keywords

runtime validation, control flow integrity, dynamic analysis

ACM Reference Format:
Zhiyang Chen, Sidi Mohamed Beillahi, Pasha Barahimi, Cyrus Minwalla, Han Du, Andreas Veneris, and Fan Long. 2026. Enforcing Control Flow Integrity on DeFi Smart Contracts. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773266

1 Introduction

Blockchain technology has revolutionized the creation of global, secure, and programmable ledgers, fundamentally altering how digital transactions are conducted. Central to this innovation are smart contracts, which operate on blockchains, allowing developers to define and enforce complex transactional rules directly on the ledger. This capability has positioned smart contracts as the backbone of various decentralized financial (DeFi) services. As of March 13th, 2025, the total value locked in 3,973 DeFi protocols has surged to approximately $87.82 billion [53], highlighting the substantial economic impact and growth of this technology.

However, by the same date, vulnerabilities in DeFi smart contracts have resulted in financial losses exceeding $11.21 billion USD [54]. In response, researchers have developed program analysis and verification techniques to secure smart contracts [42, 47, 59, 64, 68, 69, 75, 77–80], and developers often commission security audits prior to deployment.

Despite these advancements in security measures, the evolving landscape of smart contracts has continually outpaced traditional defenses. Modern smart contracts are now designed with the flexibility to support the layering of additional contracts, a feature particularly vital in DeFi ecosystem. In DeFi, smart contracts facilitate a diverse array of financial products and services, such as lending and yield farming. These interlinked contracts are often referred to as "DeFi legos," emphasizing their modularity. The ability to combine various DeFi smart contracts, a concept known as "DeFi composability," is widely regarded as one of the key advantages of DeFi [43, 70, 73]. However, this complexity and interdependence introduce significant challenges to securing smart contracts with

conventional methods. The security of a DeFi protocol depends not only on the correct design and implementation of its own contracts but also on the integrity of external contracts it interacts with. Additionally, experienced users or attackers can deploy their own smart contracts to invoke functions across multiple DeFi protocols in arbitrary sequences. Considering all possible interactions a DeFi protocol may encounter prior to deployment is infeasible for traditional security approaches.

A critical observation in hack transaction analysis is that they often *exploit unintended control flows across multiple functions*, deviating from the original design intentions of the developers. For instance, re-entrancy attacks leverage an unforeseen recursive control flow through default handlers in custom contracts, enabling repeated execution of a critical function within one transaction. Similarly, flash loan attacks manipulate multiple functions in a precisely timed sequence, utilizing large asset transfers to coerce the victim contract into executing unfavorable trades. We conducted an empirical study analyzing transaction histories of 37 compromised Ethereum protocols. Our findings reveal that control flows are relatively constrained, with attack transactions introducing novel flows never observed previously in all but one hacked protocol.

CrossGuard: Building on the above observations, we developed CrossGuard, a novel framework to enforce *control flow integrity* to secure smart contracts. Given a DeFi protocol, CrossGuard instruments its existing smart contracts with additional code to track control flow data. CrossGuard also deploys a new guard contract that collects control flow data from these instrumented contracts, and enforces four whitelisting policies at runtime to detect and neutralize any attacks that attempt to exploit unexpected control flows. Unlike many previous invariant enforcement tools [51, 66], CrossGuard does not rely on inferring its security rules from prior benign transaction traces and therefore can apply to smart contracts immediately at their initial deployments, leaving no gap of unprotected periods.

A key challenge arises from CrossGuard's whitelisting-only design. This approach, while enhancing security by adhering strictly to known safe paths, could inherently lead to an increase in false positives if not meticulously managed. Although the number of unique control flows is inherently limited, a naive whitelisting approach could still produce numerous false positives or demand substantial human intervention. To address this issue, CrossGuard employs heuristics to simplify control flows: excluding read-only calls that don't alter state, tracking read-after-write dependencies, and treating independent calls as separate flows. These heuristics effectively simplify control flows, and lower false positive rates.

Experimental Results: We evaluated CrossGuard on the deployed smart contracts and their transactions of the 37 hacked DeFi protocols included in our empirical study. Our results indicate that, when configured only once at deployment, CrossGuard can effectively prevent 35 out of 37 attacks analyzed in our study, maintaining a low average false positive rate of just 0.26%. Unlike traditional methods, CrossGuard does not depend on historical transactions. Despite this, CrossGuard still surpasses the state-of-the-art which instruments the smart contracts with invariants learned from historical transactions. Moreover, after implementing four optimization techniques, CrossGuard achieves a minimal gas consumption overhead of 3.53% on average. These results demonstrate the usefulness of our empirical findings and CrossGuard.

Contributions: This paper makes the following contributions.

- **Empirical Study:** To the best of our knowledge, we conducted the first comprehensive empirical study of control flows in historical transactions of compromised DeFi protocols. Our analysis uncovers critical insights into the control flow patterns prevalent in DeFi protocols and explores various use cases of DeFi composability.
- **CrossGuard Technique:** This paper proposes the first control flow integrity technique for smart contracts with whitelisting policies and simplification heuristics. This paper also details methods for implementing these policies and heuristics through static and dynamic analysis, and describes how they are instrumented in contracts and enforced on the fly.
- **Evaluation and Tools:** This paper evaluates the effectiveness of CrossGuard in preventing attacks. To support ongoing research and facilitate community engagement, we provide open access to the study results, experimental results, and our tool, available at our website [44].

2 Background and Empirical Study

The **Ethereum Virtual Machine (EVM)** executes smart contracts and maintains network state. **Gas** measures computational effort for transaction execution, with storage operations being particularly expensive. The recent **TLOAD** and **TSTORE** opcodes [55, 62] provide temporary transaction-scoped storage at reduced gas costs, with variables automatically initialized to zero at transaction start. **DeFi protocols** are decentralized financial systems using interconnected smart contracts for lending, borrowing, and trading.

Control Flow in software engineering is the order in which program instructions execute. In smart contracts, runtime control-flow analysis can be performed at different granularities, from low-level opcodes to high-level function calls, raising the question of which level offers the best trade-off between security coverage and computational cost.

To ground our framework, we hypothesize that *function-level* control flows are sufficient to distinguish malicious from benign transactions, while remaining practical overheads for runtime monitoring. To evaluate this hypothesis, we analyze each victim protocol's transaction history prior to the incident. Because protocols typically operate for some time before being exploited and attract many interacting actors, benign behavior can evolve over time, diversifying and introducing previously unseen control flows. This motivates the following research question:

RQ1: How do (function level) control flows in hack transactions differ from those in other (benign) transactions prior to a hack?

To answer RQ1, we conducted a study on a systematically collected benchmark comprising 37 victim protocols involved in security hacking incidents. The scope of this work is described in Section 4, and the details of our benchmark collection methodology are provided in Section 6.1. For each hacking incident, we examine the uniqueness of the control flows in the hack transactions, determining whether they differ from all previously observed transactions. The detailed study results are available in the "RQ1" column in Table 2 in Section 6. In this study, we only collect the top-level

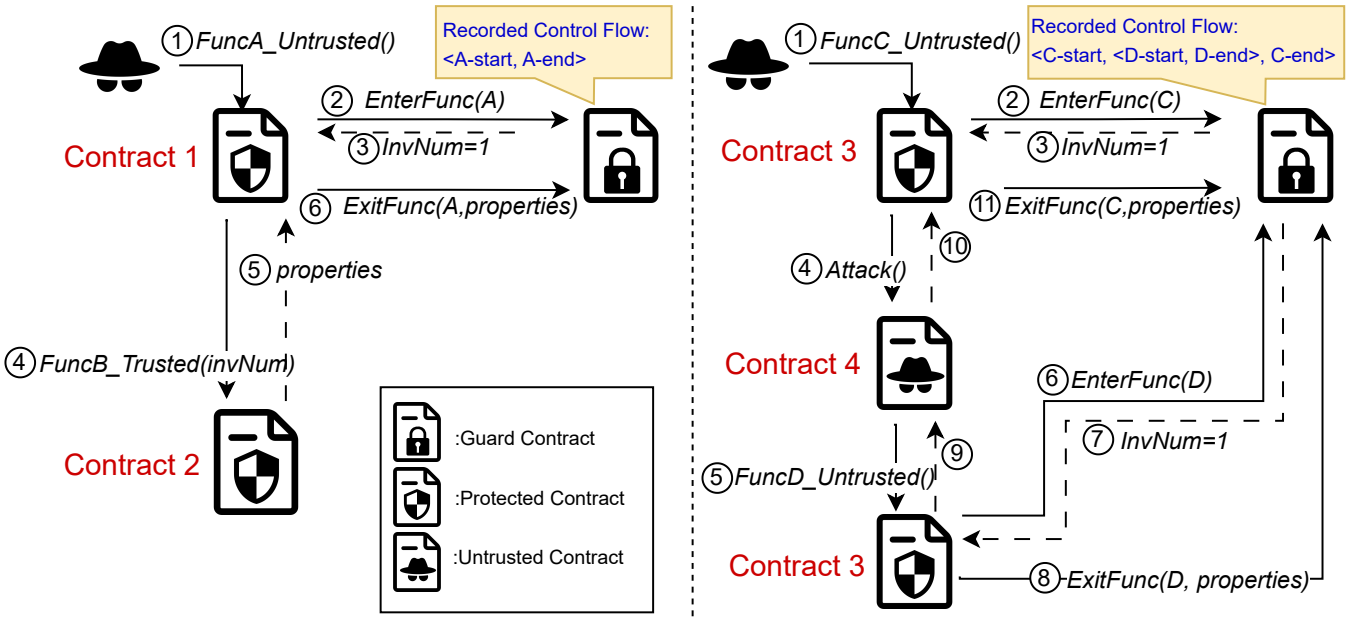

Figure 1: Running example showing CrossGuard's response to simple invocation (left) and re-entrancy attack (right)

function calls to contracts of the victim protocol, ignoring deeper control flows within external calls.

Results: Out of the 37 studied hack incidents, 32 demonstrated control flows that were distinct from any previously observed transaction patterns (marked as ✓in Table 2), highlighting the novel mechanisms by which these exploits were conducted. However, in 5 cases (bZx, VisorFi, Opyn, DODO and Bedrock_DeFi), the control flows had been observed previously in benign transactions. A detailed investigation into these exceptions revealed insightful nuances. In particular, the Bedrock_DeFi hack (marked as ✗) was traced back to an issue which mistakenly set the conversion ratio of ETH/uniBTC to 1:1. This exploit was executed through one function call to steal funds, which, while not novel in terms of control flow pattern, leveraged a specific code vulnerability to breach the protocol [71]. Catching this hack involves combining control flow analysis with data flow analysis. The remaining 4 hacks posed a different complexity: each involved various types of re-entrancy (marked as ✘). Although these attacks appeared to have familiar top-level function calls, they triggered unique and sophisticated control flows at deeper interaction levels. This shows that, even the simple control flow-based analysis can detect 32 hacks, it misses 4 that require a more refined control flow-based approach to identify. Thus, we propose to enhance the control flow analysis framework to more accurately detect and classify attacks with intricate control flows, such as these 4 re-entrancy hacks. The details of this enhanced approach are presented in Section 5.

Finding: Our finding indicates that the vast majority of hack transactions introduce unique function-level control flows that differ from all previously observed benign transactions.

Unless otherwise stated, we use the term "control flow" throughout the rest of this paper to refer to function-level control flow.

3 Running Example

To illustrate how CrossGuard monitors and enforces control flow integrity, we present two contrasting scenarios: a benign transaction with simple invocation and a malicious re-entrancy attack. Figure 1 demonstrates CrossGuard's response to these two different transactions. The left side of Figure 1 illustrates a benign transaction involving two protected contracts. Contract 1 contains `FuncA`, whose code externally invokes `FuncB` in Contract 2. When a user initiates this interaction, because they are not part of the protected protocol, they can only access `FuncA_Untrusted()`, which triggers the monitoring mechanism. Contract 1 first calls `EnterFunc(A)` to notify the guard contract of function entry, receiving an invocation number `invNum=1` that uniquely identifies this execution count. Since `FuncB` is an expected external call within `FuncA`, and both contracts belong to the protected protocol, Contract 1 invokes `FuncB_Trusted(invNum)` directly, bypassing additional guard contract interactions while still passing the invocation number for runtime property tracking. Contract 2 monitors its execution properties and returns them to Contract 1, which then calls `ExitFunc(A, properties)` to report completion along with collected runtime properties. The guard contract ultimately records a simple control flow pattern `<A-start, A-end>`, which represents normal protocol behavior and is classified as benign.

The right side of Figure 1 demonstrates how CrossGuard stops a cross function re-entrancy attack. An attacker deploys Contract 4 and exploits Contract 3, which contains `FuncC` and `FuncD` that together constitute a re-entrancy vulnerability. The attack begins when the attacker invokes `FuncC_Untrusted()`, which subsequently triggers `EnterFunc(C)` and receiving `invNum=1`. During execution, Contract 3 invokes `Attack()` in the attacker's Contract 4, which then re-enters Contract 3 and calls `FuncD_Untrusted()`. This triggers another `EnterFunc(D)` call, which returns the same `invNum=1`, indicating that this function call occurs within the context of the

ongoing invocation. When both functions complete their execution, they call their respective `ExitFunc` methods, resulting in the guard contract recording a nested control flow pattern `<C-start, <D-start, D-end>, C-end>`. This nested structure clearly indicates a same-contract cross function re-entrancy behavior, and unless explicitly whitelisted by protocol administrators, CrossGuard blocks such transactions by default.

The example highlights several key aspects of CrossGuard's design. The dual function variants(defined in Section 4) approach optimizes performance by allowing trusted intra-protocol calls to bypass guard contract interactions and save gas. The invocation numbering system enables correlation of related function calls within complex control flow sequences. Critically, both function variants collect runtime properties that are passed to the guard contract, providing additional context beyond control flow patterns to reduce false positives. The guard contract uses both control flow structure and runtime properties to distinguish between legitimate sequential execution patterns and malicious anomalies.

4 Definitions, Threat Model and Scope

We now formalize the control flow of a transaction which CrossGuard relies on. We use **protected protocol**, denoted as P, to refer to the set of smart contracts protected by our technique. We use C_i to represent the i^{th} **protected contract**, where $P = \{C_1, C_2, \ldots, C_j\}$. Typically, P consists of all the core contracts of the protocol, as specified by the developers of the protocol. We use **external contracts**, denoted as E, to refer to smart contracts that a protected protocol is built on top of but are external to the protected protocol, such as stable coins or oracles. In addition, any contract that is neither part of the protected protocol nor considered external is referred to as an **untrusted contract** (denoted as U). All functions within protected contracts are referred to as **protected functions**.

Definition 1: Tracked Runtime Properties

As shown in Section 3, CrossGuard tracks runtime properties for each execution of a protected function, with the detailed algorithms given in Section 5.4. Specifically, it tracks the following two properties:

- **Runtime Read-Only** (*isRR*): True if the function does not change any blockchain state on the fly.
- **Read-After-Write Dependency** (*isRAW*): True if, within the same invocation count (as tracked by *invNum*), the function reads from any storage location that was written by a previous invocation in the same transaction.

Both properties are maintained as boolean flags and are *mergeable* under intra-protocol composition: for a protected function f calling another protected function g within the same invocation, f aggregates g's runtime properties upon return to reflect the combined behavior. The merged properties at return are updated as $isRR_f \coloneqq isRR_f \wedge isRR_g, \quad isRAW_f \coloneqq isRAW_f \vee isRAW_g$.

CrossGuard instruments each protected contract function with two variants to optimize monitoring overhead while maintaining security guarantees. For each function f in a protected contract $C_i \in P$, CrossGuard creates two variants. Both variants collect and propagates runtime properties.

- **Untrusted variant** f_Untrusted: Accessible to any caller but mandatory triggers `EnterFunc` and `ExitFunc` of guard contract.
- **Trusted variant** f_Trusted: Restricted to calls from other protected contracts via access control, bypasses guard contract interactions.

Definition 2: Call Tree and Invocation

The **call tree** $CT(tx)$ of transaction tx represents the tree structure of function calls, where nodes correspond to function calls and directed edges represent call dependencies. An **invocation** $\iota(tx, P)$ is a sequence of function calls starting from an entry point in protected protocol P. Invocations are classified as **re-entrant** (containing recursive calls to protected contracts through untrusted intermediaries), or **simple** (no re-entrancy to protected contracts).

Definition 3: Control Flow

The **control flow** of an invocation $CF(\iota)$ captures the start and end of function calls within protected contracts, abstracting away calls to external contracts unless they facilitate re-entrancy to protected contracts. The **control flow of a transaction** $CF(tx, P)$ is defined as the sequence $\langle CF(\iota_1), CF(\iota_2), \ldots, CF(\iota_n)\rangle$ of all invocation control flows. A control flow is **trivial** only if it consists of a simple invocation, otherwise it is **non-trivial**.

Consider the re-entrancy scenario in Section 3. The two invocations would be ι_1 (the initial `FuncC` call) and ι_2 (the re-entrant `FuncD` call). The resulting control flow $CF(tx, P) = \langle\langle C_3$.`C-s`, $\langle C_3$.`D-s`, C_3.`D-e`$\rangle, C_3$.`C-e`$\rangle\rangle$ clearly reveals re-entrancy, where notation '-s' and '-e' denote function start and end respectively.

Threat Model: Our threat model assumes sophisticated attackers who can deploy arbitrary untrusted contracts to interact with protected protocols. Given the transparent nature of blockchain systems, attackers have access to CrossGuard's on-chain implementation, enabling them to probe its detection mechanisms and whitelisting policies/heuristics. Additionally, attackers may attempt to evade detection by decomposing complex attacks into multiple simpler transactions or adapting their strategies to minimize detection risks while maximizing profit. We also assume attackers do not have administrative access to the protected protocol.

Scope: CrossGuard blocks attacks that exhibit non-trivial control flows(those utilizing multiple function invocations), or re-entrancy patterns. CrossGuard cannot prevent vulnerabilities exploitable through single function calls with trivial control flows, such as access control bypasses or bridge exploits.

5 Approach

In this section, we present the design of CrossGuard. We begin by introducing four whitelisting policies, followed by two heuristics aimed at reducing false positives. Finally, we describe how these policies and heuristics are implemented within a smart contract, and how they enable real-time control flow tracking and simplification.

5.1 Control Flow Whitelisting Policies

We propose four control flow whitelisting policies.

Policy 1: Simple Independent Invocations. An invocation in a transaction is considered benign if it is *simple* and independent of any prior invocations executed within the transaction. The rationale for this policy is that a simple, independent invocation mirrors the behavior of a function being invoked by an EOA in a single, standalone call. This represents the fundamental usage of a function [1]. As such, this type of invocation is considered benign.

Policy 2: Read-Only Invocations. An invocation is considered **benign** if it is *read-only* and does not modify any blockchain state. Specifically, an invocation is read-only if it performs no write operations to the storage (i.e., no *SSTORE* operations), does not call

[1] If a simple invocation is flagged as malicious, it points to a single function access control issue, which falls outside the scope of this paper.

other state-changing functions, and does not transfer Ether [2]. This is because invocations not altering blockchain state have no effect on the final state. Therefore, those invocations can be omitted from the transaction without influencing the overall outcome.

Policy 3: Runtime Read-Only (RR) Function Calls. A function call is considered **runtime read-only** if it does not have a storage write (SSTORE), and it performs no Ether transfers. Some functions may not be marked as read-only in their source code but behave as such on the fly. An invocation is runtime read-only if all function calls within it are runtime read-only. By tracking runtime behavior and identifying such function calls and invocations, we can prune these invocations from the control flow, thereby simplifying it.

Policy 4: Restore-on-Exit (RE) Storage Writes. This heuristic permits to safely ignore storage writes that temporarily alter values but restore the original state at the end of execution, i.e., the value returned in the first read operation (SLOAD) equals the one written by its last write operation, with no write preceding the first read. This increases the likelihood of classifying function calls as runtime read-only. A typical example is re-entrancy guard, where a function restores the original state of the guard before exiting to prevent re-entrancy attacks. In our system, while these re-entrancy guards remain active and function as intended, any storage writes to them are disregarded when determining whether a function call is runtime read-only.

5.2 Control Flow Simplification Heuristics

To reduce the complexity of control flows and minimize false positives, we propose two heuristics that allow CrossGuard to safely ignore certain function calls.

Heuristic 1: ERC20 Function Calls. *ERC20* [1] is a widely used smart contract standard for implementing tokens in DeFi protocols. ERC20 contracts perform token management, and several functions within these contracts modify user properties without impacting the overall protocol state. For example, the functions `transfer` and `transferFrom` change only the balances of the sender and receiver, while `approve`, `increaseAllowance`, and `decreaseAllowance` simply modify the allowance granted by the sender to the spender. We consider these five ERC20 functions to be benign and safe, as they have been extensively tested and are widely used across numerous DeFi projects. Therefore, calls to those functions within invocations are safely ignored.

Heuristic 2: Read-After-Write (RAW) Dependency. An invocation ι_2 is considered **storage read-after-write (RAW)-dependent** on an earlier invocation ι_1 if: ι_2 reads from a storage location that ι_1 writes to (with an exception of storage writes classified in Policy 4). In the absence of such dependencies, invocations are treated as independent, and a control flow consisting only simple and independent invocations is whitelisted. Note that re-entrant invocations, no matter if it is dependent, will never be whitelisted by this heuristic.

5.3 Soundness and Limitation Analysis

As mentioned in Section 4, CrossGuard is a general-purpose defense designed to prevent *all* smart contract attacks, with the only exception of those exploitable via a single function call to a protected contract (e.g., integer overflow or access control problems).

[2]Other operations, such as *SELFDESTRUCT* or *CREATE*, could also alter the blockchain state but are rare in high-profile DeFi protocols. If such operations are present in a function, that function should not be classified as read-only.

Policies 1-4 are sound by design as they only whitelist control flows that preserve security properties: Policy 1 ensures whitelisted transactions are equivalent to direct EOA calls, Policies 2-3 are inherently safe since read-only operations cannot alter blockchain state, and Policy 4 maintains soundness by ensuring temporary state changes are reverted, still not altering blockchain states.

However, our heuristics operate on a best-effort basis with specific limitations. Heuristic 1 assumes standard ERC20 functions maintain expected behavior, which could be violated by non-standard implementations. Heuristic 2 tracks read-after-write dependencies only within protected contracts and may miss external state manipulations such as oracle manipulations. However, as shown in Section 6, this limitation impacted only 1 out of 37 benchmarks evaluated (i.e. Bedrock_DeFi).

5.4 System Overview

In this section, we explain how CrossGuard is integrated into a DeFi protocol pre-deployment by instrumenting the original code. The system consists of two main components: instrumentation within the protected contracts and a guard contract. Each function in the protected contracts is instrumented and assigned a unique positive integer `funcID` as its identifier.

Algorithm 1 EnterFunc and ExitFunc in guard contract

```
1:  State Variables (accessed via tload/tsstore):
2:    sum, invCount : int
3:    callTrace : int[]
4:    isCFRAW, isCFReEntrancy : bool
5:    _allowedPatterns : mapping(int → bool)
6:  function EnterFunc(funcID: int)
7:      if sum = 0 then
8:          invCount ← invCount + 1
9:      else
10:         isCFReEntrancy ← true
11:     sum ← sum + funcID
12:     callTrace.push(funcID)
13:     return invCount
14: function ExitFunc(funcID: int, isRR, isRAW: bool)
15:     sum ← sum − funcID
16:     if isRR then
17:         callTrace.pop()
18:     else
19:         callTrace.push(−funcID)
20:     CFHash ← 0
21:     if sum = 0 then
22:         for each id in callTrace do
23:             CFHash ← keccak256(id, CFHash)
24:         callTrace.clear()
25:     if isRAW then
26:         isCFRAW ← true
27:     if ¬ _allowedPatterns[CFHash] ∧ (isCFReEntrancy ∨
    isCFRAW) then
28:         revert "Unsafe pattern detected"
```

When an instrumented function is invoked, it sends the guard contract its `funcID` to record its function entry and receives a

positive integer `invCount` indicating the invocation count. The instrumented function then tracks storage accesses, records storage writes using `invCount`, and sends tracked runtime properties to the guard contract upon exit. The guard contract collects control flows and tracked runtime properties from the protected contracts, simplifies control flows using the defined policies and heuristics, evaluates whether the control flow is whitelisted, and reverts the transaction if it is not. [3] Alg. 1 outlines the implementation within the guard contract, while Alg. 2 outlines the execution of the instrumented protected functions. Furthermore, Table 1 details the instrumentation applied to the original source code.

Algorithm in the guard contract. Alg. 1 implements the control flow tracking in the guard contract. The function `EnterFunc` is invoked by protected contracts when one of their functions is called by an untrusted contract, taking a unique function identifier (`funcID`) as input for each function in the protected contracts. If the sum of function identifiers is zero, it signifies the start of a new invocation, and `invCount` is incremented (line 8). Otherwise, it indicates a re-entrancy condition, and `isCFReEntrancy` is set to `true` (line 10). The `sum` and `callTrace` are updated (lines 11-12) to record `funcID`. The `EnterFunc` returns `invCount` to the protected contracts, enabling them to internally track runtime properties.

The `ExitFunc` (lines 14-28) [4] is called by the protected contracts at the exit of the same function that triggered `EnterFunc`. It takes three arguments: `funcID`, `isRR` (isRuntimeReadOnly), and `isRAW` (isRead-After-Write dependent on a previous invocation). If the invocation is runtime read-only, it removes the `funcID` added by `EnterFunc` from the `callTrace` (line 17). Otherwise, it pushes the negated `funcID` (which represents the end of the function call) onto the stack (line 19). When the `sum` equals zero, signaling the end of an invocation, the `CFHash` is computed over the entire `callTrace` (lines 22-23) to summarize the control flow of the invocation as a hash. Additionally, if any invocation has a RAW dependency, Alg. 1 sets the `isCFRAW` flag to `true` (lines 25-26). Finally, if the computed hash does not match an allowed pattern, and a read-after-write condition or a re-entrancy condition is detected, the transaction is reverted to prevent unsafe behavior, enforcing the policy to block malicious control flows (lines 27-28). Without any pre-approved control flow patterns, `_allowedPatterns` only include simple invocations by default. But administrators can add more patterns to this mapping to whitelist more control flows.

Algorithm for Protected Functions. Alg. 2 implements the storage access tracking within instrumented protected functions. Table 1 provides a detailed breakdown of the instrumentation made to the original functions. Given an original function implementation, `Func`, it is replicated into two functions: `FuncTrusted` and `FuncUntrusted`. `FuncTrusted` can only be invoked by protected contracts [5], where the `invNum` is passed by its caller. In contrast, `FuncUntrusted` is designed to handle invocations from untrusted contracts or external wallets. It fetches the `invNum` from the guard contract by calling `EnterFunc`, tracks the storage accesses, and determines whether the function is runtime read-only or involves RAW (read-after-write) dependencies before sending the function identifier to the guard contract via `ExitFunc`. Both functions call `ExecuteInstrumentedCode`, which tracks state changes and evaluates whether the invocation is runtime read-only and whether any read-after-write dependencies exist.

Algorithm 2 State Access Tracking in Protected Functions (Each step within in this algorithm is executed as part of the instrumented code, in accordance with the modifications outlined in Table 1.)

```
1:  State Variables (accessed via tload/tsstore):
2:    storageWrites : mapping(mapping(bytes → int) → bool)
3:    tempReads, tempWrites : mapping(bytes → bytes)
4:  function ExecuteInstrumentedCode(invNum: int)
5:    readElements, writeElements : arrays of int ← []
6:    isRR : bool ← true
7:    isRAW : bool ← false
8:    Execute the original source code with instrumentation outlined in Table 1.
9:    for each slot in writeElements do
10:     storageWrites[invNum][slot] ← true
11:     if tempWrites[slot] ≠ tempReads[slot] then
12:       isRR ← false
13:   for each slot in readElements do
14:     for i ← 1 to invNum − 1 do
15:       if storageWrites[i][slot] then
16:         isRAW ← true
17:         break
18:   clear tempReads and tempWrites
19:   return isRR, isRAW
20: function FuncUntrusted
21:   invNum ← EnterFunc(funcID)
22:   isRR, isRAW ← ExecuteInstrumentedCode(invNum)
23:   ExitFunc(unique funcID, isRR, isRAW)
24: function FuncTrusted(invNum: int)
25:   isRR, isRAW ← ExecuteInstrumentedCode(invNum)
26:   return isRR, isRAW
```

Alg. 2 initializes `readElements` and `writeElements` arrays to store accessed storage slots, alongside two boolean flags, `isRR` (runtime read-only) and `isRAW` (read-after-write dependencies), as described in lines 5-7 of the implementation. Then Alg. 2 proceeds to execute the instrumented code (line 8), with specifics provided in Table 1. After each `SLOAD` operation, instrumentation appends the accessed slot to `readElements` and records it in `tempReads` if it hasn't previously been written to. Correspondingly, each `SSTORE` operation results in the slot being added to `writeElements` and its value stored in `tempWrites`. If any EVM opcode or function call modifies the blockchain state or if a subsequent protected contract call is not runtime read-only, `isRR` is set to `false`. Additionally, if any subsequent call to protected contracts involves a read-after-write dependency, `isRAW` is set to `true`.

Finally, Alg. 2 checks and updates `isRR` and `isRAW`, as well as `storageWrites` for future invocations (lines 9-19). For each written

[3] Note when protocol administrators execute a transaction, CrossGuard includes a straightforward mechanism (not detailed but trivial to implement) that allows administrators to deactivate the guard contract at the beginning of a transaction, perform actions without interference from the control flow integrity checks, and reactivate the guard contract at the end.

[4] Both `EnterFunc` and `ExitFunc` have access control to check if the caller is a protected contract, which is omitted for brevity in Alg. 1.

[5] `FuncTrusted` has access control to check whether the `msg.sender` is in the whitelist of protected contracts, which is not detailed in Alg. 2 for simplicity.

Table 1: Instrumentation for Protected Functions

Original Code	Instrumentation Needed
After every SLOAD ($sload(slot) \rightarrow value$):	1: $readElements.append(slot)$ 2: **if** $slot \notin tempWrites$ **then** 3: $\quad tempReads[slot] \leftarrow value$
After every SSTORE ($sstore(slot, value)$):	1: $writeElements.append(slot)$ 2: $tempWrites[slot] \leftarrow value$
After other state-changing opcodes or external non-read-only calls:	1: Set $isRR \leftarrow$ **false**
After every call to other protected contracts ($funcCall() \rightarrow isSubRR, isSubRAW$):	1: $isRR \leftarrow isRR \wedge isSubRR$ 2: $isRAW \leftarrow isRAW \vee isSubRAW$

storage slot, the slot is recorded in `storageWrites` (line 10). If a slot written is not restored-on-exit, the function is marked as not runtime read-only(lines 11-12). For each slot read, Alg. 2 checks whether the slot was written to in a previous invocation, marking the invocation as RAW-dependent if so (lines 13-17). The temporary mappings are cleared (line 18) before returning `isRR` and `isRAW` to the guard contract (line 19).

5.5 Gas Optimizations

We have implemented four optimizations to reduce gas costs.

Optimization 1: Bypassing Validation for Simple Invocations from EOAs. When a protected function is not designed to invoke arbitrary untrusted contracts given by users(a prerequisite for re-entrancy attacks), and is directly invoked by an EOA, the transaction will consistently follow a single, straightforward invocation path, which conforms to the criteria set by Policy 1 (Section 5.1). We use Slither [59] to identify these functions and manually verify to mitigate occasional false positives. For the verified functions, we insert an EOA check at the beginning of execution. When the caller is an EOA, control-flow validation can be safely bypassed, substantially reducing gas overhead.

Optimization 2: Detecting Restore-on-Exit Storage Slots Statically. Another optimization involves statically detecting restore-on-exit storage slots. By analyzing a function's control flow graph, we can identify certain storage slots that are restored to their original values at the end of every execution branch. If such restore-on-exit slots are detected statically, they do not need to be tracked at runtime, reducing the overhead of monitoring storage reads and writes. We implemented a prototype of this optimization on top of the open-source EVM bytecode analysis tool Heimdall [45]. When applied to the protected contracts analyzed in Section 2, we identified 4 contracts and 50 functions that utilize re-entrancy guards.

Optimization 3: Merging Guard Contract. We merge the guard contract with the most frequently used protected contract to convert external `EnterFunc` and `ExitFunc` calls into internal calls, reducing gas overhead. Since it is hard to predict usage patterns before deployment, we employ a simple heuristic: after deployment, we merge with the contract having the largest bytecode size. [6] Larger contracts typically implement core protocol logic and are more likely to be frequently invoked by users.

Optimization 4: Bypassing Validation for Administrator Transactions. Transactions originated from administrators are exempted from control flow validation under the assumption that protocol administrators act benignly. In our evaluation in Section 6, we identify administrators as the deployers of protected contracts, allowing their transactions to bypass validation entirely. This optimization eliminates gas overhead for protocol maintenance operations while preserving security against external threats.

6 Evaluation

Our evaluation aims to answer the following research questions:

RQ 2: How accurately does CrossGuard stop hack transactions, considering both true positives and false positives?

RQ 3: How do various actors, aside from hackers, introduce new control flows, and what causes false positives in CrossGuard?

RQ 4: Can informed hackers bypass CrossGuard?

RQ 5: What are the gas overheads of CrossGuard?

RQ 6: How does the performance of CrossGuard compare to that of the state-of-the-art tool Trace2Inv?

6.1 Methodology

Hacked Protocol Selection: We systematically selected hacked DeFi protocols that experienced significant financial losses (exceeding $300k) on Ethereum. Protocols were selected from three complementary sources: (1) victim protocols identified by [51] between February 14, 2020 and August 1, 2022; (2) hacked protocols reported by [79] within the same period; and (3) protocols compromised between February and July 2024, as documented by DeFiHackLabs [52]. Cross-chain bridge hacks were excluded.

Target Contract Selection and Protocol Filtering: For each selected protocol, we identified all relevant victim protocol contracts by analyzing deployer addresses and labels on Etherscan [10]. We then manually examined the control flow of each hack transaction with respect to these contracts. We filtered out hacks involving offchain components and hacks with only trivial control flows w.r.t. the protocol contracts, in line with our study's scope (see Section 4). After this filtering process, our final dataset consists of 37 hack incidents: 21 from source (1), 8 from source (2), and 8 from source (3). Table 2 presents our selected benchmarks, including their *Protocol Type*, hack transaction links(*Hack*), exploited vulnerability category (*Hack Type*), and number of affected contracts (*#C*). The vulnerabilities span diverse attack vectors including oracle manipulation, re-entrancy, DAO governance attack, access control failures, and input validation errors. Our selected protocols represent a comprehensive spectrum of DeFi categories, ensuring broad applicability of our findings.

Transaction History Retrieval: We then retrieved the transaction history of these identified contracts from their deployment until the hack. This complete transaction history, including the hack transaction itself, constitutes the dataset of each victim protocol. [7] Then we evaluate CrossGuard on this dataset.

[6] For proxy contracts, we use the sum of the bytecode sizes of all implementation contracts. For contracts implementing the ERC20 token standard [1], we discount the size contribution of ERC20 functions, following Heuristic 1 in Section 5.2.

[7] Although we might miss a few affected protocol contracts due to varying deployers or indirect involvement, having more contracts to protect would only introduce more complexity to the control flows, not reduce it. Thus, our analysis remains valid even with a subset of core contracts.

Table 2: Summary of Benchmarks and Control Flow Analysis Results for Victim DeFi Protocols.

Benchmarks					RQ1	RQ3									
					Unique CF	Total		#P-Tx		#S-Tx		#O-Tx		#E-Tx	
Victim Protocol	Protocol Type	Hack	Hack Type	#C	in Hack?	#Txs	#nCF	#Tx	#nCF	#Tx	#nCF	#Tx	#nCF	#Tx	#nCF
bZx	Lending	[15]	oracle manipulation	7	✗	28712	39	1389	22	16060	18	7729	14	3534	14
Warp	Lending	[39]	oracle manipulation	17	✓	416	1	11	0	404	0	0	0	1	1
CheeseBank	Lending	[16]	oracle manipulation	12	✓	2377	1	98	0	1690	0	543	0	46	1
InverseFi	Lending	[26]	oracle manipulation	12	✓	121433	60	705	11	51453	16	46452	1	22823	33
CreamFi1	Lending	[17]	re-entrancy	6	✓	190797	77	320	21	181676	20	5794	16	3007	33
CreamFi2	Lending	[18]	oracle manipulation	24	✓	220417	218	484	31	204481	40	5046	28	10406	141
RariCapital1	Lending	[33]	read-only re-entrancy	8	✓	7137	12	54	1	6831	5	208	9	44	5
RariCapital2	Lending	[34]	re-entrancy	12	✓	84423	12	393	1	45217	4	27674	2	11139	5
XCarnival	Yield Earning	[40]	logic flaw	6	✓	875	3	59	0	800	0	0	0	16	3
Harvest	Yield Earning	[23]	oracle manipulation	11	✓	30997	7	618	5	30338	1	2	0	39	2
ValueDeFi	Yield Earning	[37]	oracle manipulation	9	✓	356	1	93	0	262	0	0	0	1	1
Yearn	Yield Earning	[11]	logic flaw	7	✓	133161	14	2860	10	80618	4	44951	1	4732	3
VisorFi	Yield Earning	[38]	re-entrancy	3	✗	85420	2	52	1	26480	1	47232	0	11656	1
PickleFi	Yield Earning	[29]	fake tokens	4	✓	7511	7	1402	0	4850	5	1231	1	28	1
Eminence	DeFi	[21]	logic flaw	6	✓	21345	1	24	0	9402	0	9930	0	1989	1
Opyn	DeFi	[28]	logic flaw	4	✗	3921	2	16	1	607	0	1	0	3297	1
IndexFi	DeFi	[25]	logic flaw	6	✓	70325	5	84	0	14961	4	28182	0	27098	1
RevestFi	Yield Earning	[35]	re-entrancy	5	✓	2176	4	30	1	2127	3	11	0	8	1
DODO	DeFi	[19]	access control	2	✗	1519	2	2	1	1285	0	195	0	37	1
Punk	NFT	[32]	access control	3	✓	108	1	11	0	96	0	0	0	1	1
BeanstalkFarms	DAO	[12]	DAO governance attack	8	✓	58555	12	238	8	43713	1	8860	0	5744	4
DoughFina	Lending	[20]	no input validation	2	✓	18	2	17	1	0	0	0	0	1	1
Bedrock_DeFi	Restaking	[13]	price miscalculation	3	✗	3414	2	4	0	2127	0	893	0	390	2
OnyxDAO	DAO	[27]	fake market	8	✓	157106	3	212	0	97210	0	357	0	59327	3
BlueberryProtocol	Yield Earning	[14]	decimal difference	5	✓	464	1	113	0	347	0	0	0	4	1
PrismaFi	Restaking	[31]	no input validation	3	✓	43564	31	123	1	24722	20	4185	0	14534	23
PikeFinance	Lending	[30]	uninitialized proxy	1	✓	8408	1	15	0	7026	0	0	0	1367	1
GFOX	Game Fi	[22]	access control	2	✓	12433	1	28	0	9392	0	12	0	3001	1
UwULend	Lending	[36]	oracle manipulation	1	✓	19681	90	126	0	18296	43	6	3	1253	72
Audius	Lending	[6]	initialization bug	3	✓	6923	3	8	0	6850	1	4	0	61	2
OmniNFT	NFT	[9]	re-entrancy	2	✓	95	4	9	2	72	1	0	0	14	1
MetaSwap	DEX	[8]	logic flaw	3	✓	7027	4	22	1	6641	2	4	0	360	1
Auctus	DeFi	[5]	access control	1	✓	35	1	1	0	33	0	0	0	1	1
BaconProtocol	Yield Earning	[7]	re-entrancy	1	✓	745	3	5	0	680	0	42	1	18	2
MonoXFi	DEX	[2]	logic flaw	4	✓	1037	1	52	0	921	0	0	0	64	1
NowSwap	DEX	[3]	arbitrary call	1	✓	2290	3	11	1	0	0	0	0	2279	2
PopsicleFi	Yield Earning	[4]	logic flaw	6	✓	2648	2	117	0	2499	0	0	0	32	2
					Avg. Ratio	100	100	6.11	18.35	70.14	20.01	10.58	5.97	13.17	65.64

CrossGuard Evaluation(RQ2): To evaluate the effectiveness of CrossGuard, we conducted experiments under four configurations: (1) **Baseline**: A prototype implementing only whitelisting policies 1 and 2 (see Section 5.2). (2) **Baseline+RR**: Baseline augmented with Policy 3. (3) **Baseline+RR+RE**: Baseline augmented with Policy 3 and 4. (4) **Baseline+RR+RE+ERC20**: Baseline augmented with Policy 3 and 4, and Heuristic 1. (5) **CrossGuard**: Integrates all 4 policies and 2 heuristics into the Baseline. Note that these configurations are instrumented pre-deployment and operate autonomously post-deployment without manual intervention. CrossGuard enforces predefined policies and heuristics without relying on past transaction data. However, if an unseen control flow is mistakenly blocked, CrossGuard provides an administrative feedback mechanism that allows protocol administrators to manually approve and whitelist it. To evaluate this mechanism, we tested CrossGuard under three CrossGuard+Feedback settings, assuming administrators could approve new control flows within 3 days (19,200 blocks), 1 day (6,400 blocks), and 1 hour (267 blocks).

We assessed these configurations using historical transactions from 37 benchmarks collected in Section 2. To further evaluate CrossGuard under extreme conditions, we applied it to another 3 widely adopted DeFi protocols(AAVE, Lido, and Uniswap) which serve as fundamental DeFi building blocks. These protocols attract many DeFi developers and feature the most complex and continuously evolving control flows due to their high composability and extensive integrations. To conduct this evaluation, we collected the core smart contracts for these protocols from their official websites [41, 63, 76]. Next, we retrieved the most recent 100,000 transactions interacting with these contracts. We then applied CrossGuard to these transactions, measuring its false positive rate (FP%) under real-world extreme conditions. The experimental results are summarized in Table 3.

DeFi Actors Identification(RQ3): To deeply understand the results of CrossGuard and the diversity of control flows in DeFi protocols, we categorize transactions according to their origins and initiators. We focus explicitly on transactions with non-trivial control flows (nCFs), as these represent complex and less predictable interactions, offering deeper insights into protocol dynamics. We identify four primary actor groups capable of introducing unique, non-trivial control flows: **1. Privileged Transactions (P-Tx)**: Originated by protocol deployers or administrators via privileged functions (e.g., constructors, administrative operations), typically reflecting protocol setup or administrative management activities. **2. Same Protocol (S-Tx)**: Transactions initiated by other contracts within the same protocol, developed internally to enhance operational coherence and overall functionality. **3. Other DeFi Protocols (O-Tx)**: Transactions initiated by externally deployed DeFi protocols (commonly labeled on Etherscan), often through open-source collaboration, enriching the broader DeFi ecosystem. **4. External Actors (E-Tx)**: Transactions initiated by contracts deployed by external actors such as arbitrageurs, advanced DeFi users, or malicious entities (hackers), generally employing closed-source contracts to execute complex strategies or exploit vulnerabilities.

6.2 RQ2: Effectiveness of CrossGuard

The columns "RQ2" in Table 3 present the results for RQ2. Each configuration is evaluated using two key metrics: "Block?" indicates whether the hack was successfully blocked; "FP%" represents the false positive rate for that configuration. Two sets of benchmarks, 37 hacked protocols and 3 popular protocols, both include a "Summary" row at the bottom, showing the total number of blocked hacks and the average false positive rate per protocol.

When the ERC20 and RAW heuristics are enabled, CrossGuard blocks 35 out of 37 hacks. The only exceptions are Bedrock_DeFi and Auctus. In the case of Bedrock_DeFi, the attacker exploited a missing input validation vulnerability by invoking a single function; while two ERC20 functions were also called, they were not essential to the core exploit mechanism. Similarly, for Auctus, the attacker exploited an access control vulnerability by repeatedly calling the same function to drain funds. In both instances, these operations could have been executed as independent transactions without relying on the complex control flows that CrossGuard is designed to detect. Consequently, CrossGuard did not block these transactions. Overall, for the 35 blocked attacks, the exploits involved intricate control flows that were effectively captured by CrossGuard, demonstrating its robustness against sophisticated hacks."

6.3 RQ3: Control Flows Introduced by Different Actors and False Positives

The columns labeled *RQ3* in Table 2 summarize our analysis of non-trivial control flows introduced by each transaction category. The last row provides the average ratios of transactions and control flows for each transaction category.

A key insight from this analysis is that a significant number of protocols (27, as highlighted in gray) exhibit a relatively low number of non-trivial control flows (≤ 9) throughout their operational lifetimes prior to being hacked. Notably, in 17 protocols (Warp, CheeseBank, XCarnival, Harvest, ValueDeFi, VisorFi, Eminence, IndexFi, RevestFi, Punk, DoughFina, BlueberryProtocol, PikeFinance, GFOX, OmniNFT, Auctus, MonoXFi), the hack was the first external non-trivial control flow introduced, beyond those generated internally (as indicated by gray cells showing 0 nCF from O-Tx but exactly 1 nCF from E-Tx) [8]. This insight suggests that 17 hacks could be simply prevented with zero false positive rates by restricting **ALL** external ("non-trusted") developers, allowing only the protocol deployers to create new control flows. The analysis reveals that E-Tx and P-Tx are significant sources of non-trivial control flows. Although external transactions account for only 13.17% of the total, they introduce 65.64% of non-trivial control flows, often bringing unexpected interactions and potential vulnerabilities. In contrast, S-Tx, despite comprising 70.14% of the total, contribute to only 20.01% of non-trivial flows, underscoring the fact that most users interact directly with the protocol rather than through other intermediate contracts.

We also utilize the above identified transaction categories to conduct a deeper analysis of false positives in RQ2. We examined the three protocols with the highest false positive rates: bZx2 (3.57%), UwULend (2.38%), and RariCapital1 (1.22%)—all of which are lending protocols. Our investigation reveals that false positives predominantly originate from two specific actor categories: Other DeFi Protocols (O-Tx) accounting for 827 out of 1025 false positives in bZx2, 3 out of 468 in UwULend, and 74 out of 87 in RariCapital1; and External Actors (E-Tx) responsible for 198 out of 1025 in bZx2, 465 out of 468 in UwULend, and 13 out of 87 in RariCapital1. Additionally, we identified that 9 out of 13 false positives in RariCapital1 stem from MEV operations.

Further investigation into the nature of these false positives reveals two primary categories of legitimate use cases that introduce novel control flows. The first category involves helper contracts designed to streamline user operations, such as a contract that facilitates depositing three different tokens into UwULend by sequentially invoking the deposit function three times within a single transaction. The second category encompasses innovative functionality extensions, exemplified by patterns observed in UwULend where users deploy contracts that invoke deposit and borrow functions multiple times in sophisticated sequences to maximize their borrowing power and optimize capital efficiency. These legitimate but complex interaction patterns highlight opportunities for future refinement of control flow policies to better accommodate common DeFi usage patterns while maintaining security guarantees.

6.4 RQ4 & 5: Bypassability and Gas Overheads

Case Studies. Given that CrossGuard operates as a fully on-chain runtime system, it is transparent, allowing attackers to study its implementations and whitelisted control flows. A prevalent concern is whether informed attackers could bypass CrossGuard by splitting complex hack transactions into simpler ones. To address this, we perform in-depth studies of the 35 hacks blocked by CrossGuard. Our analysis involves scrutinizing the control flows, underlying vulnerabilities, and the potential outcomes if attackers were to split their transactions.

Results for Case Studies. The *RQ4* column in Table 3 summarizes our findings: 31 out of 35 hacks cannot be bypassed by attackers. Specifically, 18 hacks inherently require complex control flows to exploit vulnerabilities (marked as ✓).

Additionally, 13 hacks rely on executing multiple capital-intensive functions to carry out the exploit. Historically, these attacks have used flash loans, which require all steps to be completed within a single transaction. Without flash loans, the attackers have to risk their own capital while competing with arbitrage bots, a scenario we deem as non-bypassable due to the high financial risks (marked as ✓∗). Only five hacks Punk, DoughFina, PikeFinance, Audius, and Auctus (marked as ✗) show a bypass chance for attackers. The root causes of these exploits are access control, missing input validation, and initialization bugs. Each of these vulnerabilities can be exploited by invoking a single function without requiring complex control flows. These attacks were initially caught by CrossGuard because the hackers included additional preparatory operations in their transactions. However, these preparatory steps can indeed be split and executed separately in separate transactions, allowing attackers to bypass CrossGuard.

Experiment. To measure the gas overhead of CrossGuard, we instrumented smart contracts in a template-based manner. We insert specific code snippets at key points such as function entry and exit,

[8] Two exceptions are XCarnival and Harvest. They were hacked in multiple hack transactions which introduced 3 and 2 non-trivial control flows, respectively.

Table 3: Ablation study, Gas Consumption and Bypassability of CrossGuard

Victim Protocol	RQ2										RQ5	RQ2				RQ4	
	Baseline		Baseline +RR		Baseline +RR+RE		Baseline +RR+RE+ERC20		Baseline+RR+RE+ERC20 +RAW (a.k.a, CrossGuard)			CrossGuard+Feedback				Not-Bypassable	Flash-Loan
													3 days	1 day	1 hour		
	Block?	FP%	Block?	FP%	Block?	FP%	Block?	FP%	Block?	FP%	Gas OH(%)	Block?	FP%	FP%	FP%		
bZx	✓	4.51	✓	4.51	✓	4.5	✓	3.88	✓	3.57	6.29	✓	0.4	0.23	0.06	✓	✓
Warp	✓	0	✓	0	✓	0	✓	0	✓	0	0.03	✓	0	0	0	✓*	✓
CheeseBank	✓	2.06	✓	2.06	✓	2.06	✓	0	✓	0	0.05	✓	0	0	0	✓*	✓
InverseFi	✓	14.96	✓	14.94	✓	14.91	✓	0.08	✓	0.08	2.17	✓	0.04	0.03	0.03	✓*	✓
CreamFi1	✓	4.30	✓	2.28	✓	1.16	✓	1.15	✓	0.39	3.63	✓	0.13	0.07	0.03	✓	✓
CreamFi2	✓	4.32	✓	3.1	✓	2.27	✓	1.12	✓	0.84	9.23	✓	0.21	0.15	0.09	✓*	✓
RariCapital1	✓	1.92	✓	1.25	✓	1.25	✓	1.23	✓	1.22	2.14	✓	0.62	0.5	0.36	✓*	✓
RariCapital2	✓	1.42	✓	0.33	✓	0.09	✓	0.09	✓	0.02	5.23	✓	0.02	0.02	0.01	✓	✓
XCarnival	✓	0	✓	0	✓	0	✓	0	✓	0	0.15	✓	0	0	0	✓	✗
Harvest	✓	0	✓	0	✓	0	✓	0	✓	0	0.02	✓	0	0	0	✓*	✓
ValueDeFi	✓	0	✓	0	✓	0	✓	0	✓	0	0.06	✓	0	0	0	✓*	✓
Yearn	✓	2.47	✓	2.47	✓	2.43	✓	0.33	✓	0	1.01	✓	0	0	0	✓*	✓
VisorFi	✓	9.56	✓	9.56	✓	9.56	✓	0	✓	0	0.01	✓	0	0	0	✓	✗
PickleFi	✓	0.71	✓	0.71	✓	0.71	✓	0.01	✓	0.01	0.54	✓	0.01	0.01	0.01	✓	✗
Eminence	✓	3.21	✓	3.21	✓	3.21	✓	0	✓	0	0.7	✓	0	0	0	✓*	✓
Opyn	✓	0.05	✓	0.05	✓	0	✓	0	✓	0	11.5	✓	0	0	0	✓	✗
IndexFi	✓	5.67	✓	5.64	✓	5.64	✓	0	✓	0	13.48	✓	0	0	0	✓	✓
RevestFi	✓	0	✓	0	✓	0	✓	0	✓	0	0.09	✓	0	0	0	✓	✓
DODO	✓	0.07	✓	0.07	✓	0	✓	0	✓	0	1.58	✓	0	0	0	✓	✓
Punk	✓	0	✓	0	✓	0	✓	0	✓	0	0.03	✓	0	0	0	✗	✗
BeanstalkFarms	✓	5.83	✓	5.83	✓	5.83	✓	0.06	✓	0.06	19.89	✓	0.01	0.01	0.01	✓	✓
DoughFina	✓	0	✓	0	✓	0	✓	0	✓	0	0.09	✓	0	0	0	✗	✓
Bedrock_DeFi	✓	15.41	✓	15.41	✓	15.41	✗	0.26	✗	0.06	10.51	✗	0.06	0.06	0.06	N/A	✓
OnyxDAO	✓	4.5	✓	4.48	✓	4.48	✓	0	✓	0	0.1	✓	0	0	0	✓	✓
BlueberryProtocol	✓	0.65	✓	0	✓	0	✓	0	✓	0	0.29	✓	0	0	0	✓*	✓
PrismaFi	✓	31.49	✓	31.49	✓	31.46	✓	1.2	✓	0.7	0.68	✓	0.05	0.02	0.02	✓	✓
PikeFinance	✓	0	✓	0	✓	0	✓	0	✓	0	0.91	✓	0	0	0	✗	✗
GFOX	✓	3.40	✓	3.4	✓	3.39	✓	0	✓	0	0.02	✓	0	0	0	✓*	✗
UwULend	✓	2.89	✓	2.55	✓	2.55	✓	2.55	✓	2.38	10.51	✓	0.41	0.33	0.17	✓*	✓
Audius	✓	0.03	✓	0.01	✓	0.01	✓	0.01	✓	0.01	0.94	✓	0.01	0.01	0.01	✗	✗
OmniNFT	✓	0	✓	0	✓	0	✓	0	✓	0	2.33	✓	0	0	0	✓	✓
MetaSwap	✓	0.01	✓	0.01	✓	0.01	✓	0.01	✓	0	3.63	✓	0	0	0	✓	✓
Auctus	✓	0	✓	0	✓	0	✓	0	✗	0	0.02	✗	0	0	0	N/A	✗
BaconProtocol	✓	0.27	✓	0.27	✓	0.27	✓	0.27	✓	0.27	0.75	✓	0.27	0.27	0.27	✓	✓
MonoXFi	✓	0	✓	0	✓	0	✓	0	✓	0	3.52	✓	0	0	0	✓	✗
NowSwap	✓	0.31	✓	0.31	✓	0.31	✓	0.04	✓	0.04	18.08	✓	0.04	0.04	0.04	✓	✗
PopsicleFi	✓	0.42	✓	0.42	✓	0.42	✓	0.08	✓	0.08	0.24	✓	0.08	0.04	0.04	✓*	✓
Summary	37	3.26	37	3.09	37	3.03	36	0.33	35	0.26	3.53	35	0.06	0.05	0.03	31	26
AAVE	N/A	9.69	N/A	8.53	N/A	8.52	N/A	8.52	N/A	7.32	14.13	N/A	1.10	0.53	0.22	N/A	N/A
Lido	N/A	13.09	N/A	13.09	N/A	13.07	N/A	7.29	N/A	7.27	6.84	N/A	4.38	2.04	0.43	N/A	N/A
Uniswap	N/A	1.17	N/A	1.04	N/A	1.00	N/A	0.49	N/A	0.24	1.59	N/A	0.24	0.23	0.05	N/A	N/A
Summary	N/A	7.98	N/A	7.55	N/A	7.53	N/A	5.43	N/A	4.94	7.52	N/A	1.91	0.93	0.23		

as well as during storage access operations. These snippets execute at runtime to capture the additional gas consumption introduced by CrossGuard. We used Foundry [60] to compare the gas costs between the original and instrumented contracts, recording overhead differences in various execution phases. This process involved detailed assessments of each instrumentation type, functions within the guard contract, and EOA checks. During transaction replays, we logged the additional gas costs incurred at these key execution points to compute the overall gas overhead.

To benchmark against industry standards, we evaluated the Hyperithm protocol [24, 56–58] deployed on May 22, 2025, which is protected by SphereX, an industry control flow restriction solution discussed in Section 8. We measured the additional gas costs during transaction replays to compute overhead for both CrossGuard and SphereX implementations on identical transaction history.

Results for Gas Overheads. The RQ5 column in Table 3 presents the gas overhead introduced by CrossGuard for each benchmark. On average, the overall gas overhead is 3.53%. Notably, 28 protocols exhibit a gas overhead below 5%, primarily due to the high proportion of EOA transactions within these benchmarks. Since EOA transactions benefit from Optimization 1 (Section 5.5), which reduces unnecessary gas consumption, the resulting gas overhead remains minimal. Even for the three widely used DeFi protocols, which feature a significant number of contract-initiated transactions, the average gas overhead remains 7.52%. This is a reasonable tradeoff considering the strong security guarantees provided by CrossGuard. The higher overhead in these cases stems from the need to track function calls and storage accesses, which are essential for securing protocols with complex execution flows.

Our industry comparison reveals that SphereX introduces 6.09% overhead for the Hyperithm protocol, while CrossGuard achieves only 0.22% overhead on the same contracts. Analysis of 166 transactions shows that 127 are simple invocations optimized by Optimization 1, and 38 are deployer-originated transactions optimized by Optimization 4. In contrast, SphereX lacks these optimizations and triggers external calls for every function invocation, resulting in substantially higher gas overhead. This demonstrates CrossGuard's superior efficiency through targeted optimization strategies.

6.5 RQ6: Comparative Analysis with Trace2Inv

Experiment. We compare CrossGuard against Trace2Inv [51]. Trace2Inv relies on historical transaction data to generate invariants, which are then instrumented into smart contracts to prevent hacks. This approach requires a training set (TS) of past transactions to learn security rules. In contrast, CrossGuard operates without historical data, making it applicable to new contracts before deployment. Additionally, CrossGuard allows protocol administrators to

explicitly whitelist control flows they deem safe. To evaluate their effectiveness, we apply both tools to our 37 benchmark protocols. As required by Trace2Inv, we use 70% of transaction history as the training set and evaluate both tools on the remaining 30% of transactions as the testing set. We assess two versions of CrossGuard: one operating without training data and another that assumes all control flows from the training set are whitelisted. We compare CrossGuard against Trace2Inv using its two most effective security invariants: EOA∧GC∧DFU and EOA∧(OB∨DFU) [51].

Table 4: Comparison of CrossGuard and Trace2Inv

	CrossGuard (w/o TS)	CrossGuard (w TS)	Trace2Inv (w TS)	
			EOA∧GC∧DFU	EOA∧(OB∨DFU)
# Hacks Blocked	35	35	34	29
Avg. FP%	1.19	0.15	3.14	0.23

Results. Table 4 presents the analysis results that demonstrate that CrossGuard, even without training data, effectively blocks 35 out of 37 while maintaining an average FP% rate of 1.19%. This is a significant achievement compared to Trace2Inv, which no only requires training data to function but only blocks at most 34 hacks. When trained, CrossGuard maintains its effectiveness in blocking hacks, reducing its FP% rate dramatically to 0.15%, which is superior to the FP% rates achieved by Trace2Inv's invariants (3.14% and 0.23%). These results underline CrossGuard's potential as a state-of-the-art solution providing robust security for DeFi applications. Moreover, CrossGuard and Trace2Inv can be used in conjunction to provide a more comprehensive security solution.

7 Discussion and Threats to Validity

Generalization. CrossGuard can generalize through two mechanisms. First, it can evolve and reduce false positives by learning from blocked benign transactions, enabling administrators to whitelist new control flow patterns. Second, it records all non-trivial control flows in the guard contract. If novel attacks bypass current conditions (Alg. 1, Line 27), developers can update the guard contract to add new detection logic and automatically revert such transactions. This framework is extensible to track additional runtime properties for emerging threats.

Integrating CrossGuard Pre- and Post-deployment. For new protocols, CrossGuard's integration involves three steps: (1) instrument contracts (Section 5), using Slither and Heimdall to omit unnecessary instrumentation (Section 5.5); (2) deploy a guard contract to enforce policies and heuristics; and (3) configure access controls for all components. For existing deployed upgradable protocols [46], CrossGuard can be retrofitted by upgrading implementation contracts to new instrumented versions, thereby enabling CrossGuard without disrupting existing protocol logic. Following deployment, CrossGuard operates autonomously. Administrators can further improve CrossGuard by whitelisting blocked legitimate control flows (false positives). This adaptive process ensures that CrossGuard evolves alongside protocol growth, maintaining strong security guarantees over time.

Threats to Validity. The internal threat to validity concerns potential human errors in identifying protected contracts. As discussed in Section 6.1, we rely on Etherscan labels to identify these core contracts to protect. The labels might be incomplete, leading to missing protected contracts. However, with more protected contracts identified, our approach will only become more effective in blocking hacks, as the control flows of hack transactions recorded by CrossGuard will be more complex but still unique. Our results may also face external threats due to the reliance on Trace2Inv [51] and Sting [79] benchmarks, which focus on hacks up to 2022. We mitigate this threat by including additional 8 hacks from February to July 2024. Additionally, we also include three major DeFi protocols representing the most current protocols and user transactions.

8 Related Works

Invariant Generation and Enforcement. Prior research generates and enforces invariants for contract security. Cider [65] derives arithmetic overflow invariants via deep reinforcement learning, while InvCon [66] and InvCon+ [67] combine dynamic inference with static verification for function-level invariants. Trace2Inv [51] learns invariants from transaction history to prevent attacks. Unlike these approaches, which focus on individual contracts or specific bugs, CrossGuard is a general-purpose framework that targets control flows across multiple protocol contracts with a one-time pre-deployment configuration.

Control Flow Restriction. SphereX [74] in industry offers services to manually restrict control flows, requiring developers to explicitly whitelist or blacklist paths. In contrast, CrossGuard automates the whitelisting of unseen control flows and simplifies the overall structure, significantly reducing developer burden and increasing system adaptability compared to manual intervention.

Re-entrancy Attack Defense and Secure Type System. Numerous tools restrict control flows to combat re-entrancy attacks. Static analysis tools [42, 47, 59, 64, 68, 69, 75, 77, 78, 80] identify re-entrancy vulnerabilities and apply guards. Runtime frameworks like Sereum [72] and Grossman et al. protect deployed contracts, while Callens et al. prevent duplicate function calls within transactions. Unlike these re-entrancy-focused approaches, CrossGuard adopts broader control flow restriction targeting comprehensive vulnerability classes. Cecchetti et al. propose a security type system to enforce information flow and re-entrancy controls [49, 50]. Conversely, CrossGuard is a purely runtime system built entirely on the EVM, operating without supplementary type systems or language modifications.

9 Conclusion

In this paper, we presented CrossGuard, a novel control flow integrity framework specifically designed to secure DeFi smart contracts at runtime. Configured only once at deployment, CrossGuard prevents malicious transactions from executing risky control flows on the fly, effectively mitigating a wide range of attacks. Our comprehensive evaluation demonstrates that CrossGuard blocks the vast majority of benchmark attacks, significantly reduces false positives without relying on a pre-collected training set of benign transactions, and maintains a manageable gas overhead. Furthermore, integrating manual feedback enhances its accuracy. Together, these results establish CrossGuard as a practical solution for securing smart contracts.

Acknowledgments

We thank anonymous reviewers for their insightful comments on the early version of the paper. This work was supported by Mitacs through the Mitacs Accelerate program.

References

[1] [n. d.]. ERC-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20.
[2] 2021. MonoXFi Attack Transaction. https://etherscan.io/tx/0x9f14d093a2349de08f02fc0fb018dadb449351d0cdb7d0738ff69cc6fef5f299.
[3] 2021. NowSwap Attack Transaction. https://etherscan.io/tx/0xf3158a7ea59586c5570f5532c22e2582ee9adba2408eabe61622595197c50713.
[4] 2021. PopsicleFi Attack Transaction. https://etherscan.io/tx/0xcd7dae143a4c0223349c16237ce4cd7696b1638d116a72755231ede872ab70fc.
[5] 2022. Auctus Attack Transaction. https://etherscan.io/tx/0x2e7d7e7a6eb157b98974c8687fbd848d0158d37edc1302ea08ee5ddb376befea.
[6] 2022. Audius Attack Transaction. https://etherscan.io/tx/0xfefd829e246002a8fd061eede7501bccb6e244a9aacea0ebceaecef5d877a984.
[7] 2022. BaconProtocol Attack Transaction. https://etherscan.io/tx/0x7d2296bcb936aa5e2397ddf8ccba59f54a178c3901666b49291d880369dbcf31.
[8] 2022. MetaSwap Attack Transaction. https://etherscan.io/tx/0x2b023d65485c4bb68d781960c2196588d03b871dc9eb1c054f596b7ca6f7da56.
[9] 2022. OmniNFT Attack Transaction. https://etherscan.io/tx/0x264e16f4862d182a6a0b74977df28a85747b6f237b5e229c9a5bbacdf499ccb4.
[10] 2024. Etherscan. https://etherscan.io.
[11] 2024. Yearn Attack Transaction. https://etherscan.io/tx/0x59faab5a1911618064f1ffa1e4649d85c99cfd9f0d64dcebbc1af7d7630da98b.
[12] 2025. BeanstalkFarms Attack Transaction. https://etherscan.io/tx/0xcd314668aaa9bbfebaf1a0bd2b6553d01dd58899c508d4729fa7311dc5d33ad7.
[13] 2025. Bedrock DeFi Attack Transaction. https://etherscan.io/tx/0x725f0d65340c859e0f64e72ca8260220c526c3e0ccde530004160809f6177940.
[14] 2025. BlueberryProtocol Attack Transaction. https://etherscan.io/tx/0xf0464b01d962f714eee9d4392b2494524d0e10ce3eb3723873afd1346b8b06e4.
[15] 2025. bZx Attack Transaction. https://etherscan.io/tx/0x762881b07feb63c436dee38edd4ff1f7a74c33091e534af56c9f7d49b5ecac15.
[16] 2025. CheeseBank Attack Transaction. https://etherscan.io/tx/0x600a869aa3a259158310a233b815ff67ca41eab8961a49918c2031297a02f1cc.
[17] 2025. CreamFi Attack Transaction 1. https://etherscan.io/tx/0x0016745693d68d734faa408b94cdf2d6c95f511b50f47b03909dc599c1dd9ff6.
[18] 2025. CreamFi Attack Transaction 2. https://etherscan.io/tx/0xab486012f21be741c9e674ffda227e30518e8a1e37a5f1d58d0b0d41f6e76530.
[19] 2025. DODO Attack Transaction. https://etherscan.io/tx/0x395675b56370a9f5fe8b32badfa80043f5291443bd6c8273900476880fb5221e.
[20] 2025. DoughFina Attack Transaction. https://etherscan.io/tx/0x92cdcc732eebf47200ea56123716e337f6ef7d5ad714a2295794fdc6031ebb2e.
[21] 2025. Eminence Attack Transaction. https://etherscan.io/tx/0x3503253131644dd9f52802d071de74e456570374d586ddd640159cf6fb9b8ad8.
[22] 2025. GFOX Attack Transaction. https://etherscan.io/tx/0x12fe79f1de8aed0ba947cec4dce5d33368d649903cb45a5d3e915cc459e751fc.
[23] 2025. Harvest Attack Transaction 1. https://etherscan.io/tx/0x0fc6d2ca064fc841bc9b1c1fad1fbb97bcea5c9a1b2b66ef837f1227e06519a6.
[24] 2025. Hyperithm — Digital Asset Gateway for Institutions. https://www.hyperithm.com/. Accessed: 2025-07-18.
[25] 2025. IndexFi Attack Transaction. https://etherscan.io/tx/0x44aad3b85386646816 1735496a5d9cc961ce5aa872924c5d78673076b1cd95aa.
[26] 2025. InverseFi Attack Transaction. https://etherscan.io/tx/0x600373f67521324c8068cfd025f121a0843d57ec813411661b07edc5ff781842.
[27] 2025. OnyxDAO Attack Transaction. https://etherscan.io/tx/0x46567c731c4f4f7e27c4ce591f0aebdeb2d9ae1038237a0134de7b13e63d8729.
[28] 2025. Opyn Attack Transaction. https://etherscan.io/tx/0x56de6c4bd906ee0c067a332e64966db8b1e866c7965c044163a503de6ee6552a.
[29] 2025. PickleFi Attack Transaction. https://etherscan.io/tx/0xe72d4e7ba9b5af0cf2a8cfb1e30fd9f388df0ab3da79790be842bfbed11087b0.
[30] 2025. PikeFinance Attack Transaction. https://etherscan.io/tx/0xe2912b8bf34d561983f2ae95f34e33ecc7792a2905a3e317fcc98052bce66431.
[31] 2025. PrismaFi Attack Transaction. https://etherscan.io/tx/0x00c503b595946bccaea3d58025b5f9b3726177bbdc9674e634244135282116c7.
[32] 2025. Punk Attack Transaction. https://etherscan.io/tx/0x597d11c05563611cb4ad4ed4c57ca53bbe3b7d3fefc37d1ef0724ad58904742b.
[33] 2025. RariCapital Attack Transaction 1. https://etherscan.io/tx/0x4764dc6ff19a64fc1b0e57e735661f64d97bc1c44e026317be8765358d0a7392.
[34] 2025. RariCapital Attack Transaction 2. https://etherscan.io/tx/0x0fe2542079644e107cbf13690eb9c2c65963ccb79089ff96bfaf8dced2331c92.
[35] 2025. RevestFi Attack Transaction. https://etherscan.io/tx/0xe0b0c2672b760bef4e2851e91c69c8c0ad135c6987bbf1f43f5846d89e691428.
[36] 2025. UwULend Attack Transaction. https://etherscan.io/tx/0x242a0fb4fde9de0dc2fd42e8db743cbc197ffa2bf6a036ba0bba303df296408b.
[37] 2025. ValueDeFi Attack Transaction. https://etherscan.io/tx/0x46a03488247425f845e444b9c10b52ba3c14927c687d38287c0faddc7471150a.
[38] 2025. VisorFi Attack Transactions. https://etherscan.io/tx/0x69272d8c84d67d1da2f6425b339192fa472898dce936f24818fda415c1c1ff3f and https://etherscan.io/tx/0x6eabef1bf310a1361041d97897c192581cd9870f6a39040cd24d7de2335b4546.
[39] 2025. Warp Attack Transaction. https://etherscan.io/tx/0x8bb8dc5c7c830bac85fa48acad2505e9300a91c3ff239c9517d0cae33b595090.
[40] 2025. XCarnival Attack Transaction. https://etherscan.io/tx/0x51cbfd46f21afb44da4fa971f220bd28a14530e1d5da5009cfbdfee012e57e35.
[41] Aave. 2024. Aave Protocol. https://aave.com/. Accessed: 2024-12-18.
[42] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020. Taming callbacks for smart contract modularity. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
[43] Hendrik Amler, Lisa Eckey, Sebastian Faust, Marcel Kaiser, Philipp Sandner, and Benjamin Schlosser. 2021. Defi-ning defi: Challenges & pathway. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 181–184.
[44] Anonymous Authors. 2024. CrossGuard Website. https://sites.google.com/view/crossguard/home.
[45] Jon Becker. 2023. heimdall-rs. https://github.com/Jon-Becker/heimdall-rs GitHub repository.
[46] William E Bodell III, Sajad Meisami, and Yue Duan. 2023. Proxy hunting: understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1829–1846.
[47] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 161–178.
[48] Valerian Callens, Zeeshan Meghji, and Jan Gorzny. 2024. Temporarily Restricting Solidity Smart Contract Interactions. *arXiv preprint arXiv:2405.09084* (2024).
[49] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C Myers. 2020. Securing smart contracts with information flow. In *International Symposium on Foundations and Applications of Blockchain*.
[50] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C Myers. 2021. Compositional security for reentrant applications. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1249–1267.
[51] Zhiyang Chen, Ye Liu, Sidi Mohamed Beillahi, Yi Li, and Fan Long. 2024. Demystifying Invariant Effectiveness for Securing Smart Contracts. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1772–1795.
[52] Many Contributors. 2025. DeFi Hacks Reproduce - Foundry. https://github.com/SunWeb3Sec/DeFiHackLabs.
[53] DefiLlama. 2025. DefiLlama. https://defillama.com/. Accessed: 2025-03-13.
[54] DefiLlama. 2025. DefiLlama Hacks. https://defillama.com/hacks. Accessed: 2025-03-13.
[55] Ethereum Improvement Proposals. 2023. EIP-1153: Transient Storage Opcodes. https://eips.ethereum.org/EIPS/eip-1153. Accessed: 2024-08-30.
[56] Etherscan. 2025. Ethereum Address 0x6231a192089fb636e704d2c7807d7a79c2457b07. https://etherscan.io/address/0x6231a192089fb636e704d2c7807d7a79c2457b07. Accessed: 2025-07-18.
[57] Etherscan. 2025. Ethereum Address 0xc92b021ff09ae005cb3fccb66af8db01fc4cdf90. https://etherscan.io/address/0xc92b021ff09ae005cb3fccb66af8db01fc4cdf90. Accessed: 2025-07-18.
[58] Etherscan. 2025. Ethereum Address 0xf5d35b9e95f6842a2064a2dd24f8deede9d58f97. https://etherscan.io/address/0xf5d35b9e95f6842a2064a2dd24f8deede9d58f97. Accessed: 2025-07-18.
[59] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.
[60] Foundry Contributors. 2023. Foundry. https://github.com/foundry-rs/foundry/. Accessed: 2024-08-31.
[61] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–28.
[62] Kraken Exchange. 2024. Everything You Need to Know About the Ethereum Cancun Upgrade. https://blog.kraken.com/news/everything-you-need-to-know-about-the-ethereum-cancun-upgrade. Accessed: 2024-08-30.
[63] Lido DAO. 2024. Lido - Liquid Staking for Ethereum 2.0. https://lido.fi/. Accessed: 2024-12-18.
[64] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 65–68.
[65] Junrui Liu, Yanju Chen, Bryan Tan, Isil Dillig, and Yu Feng. 2022. Learning Contract Invariants Using Reinforcement Learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–11.
[66] Ye Liu and Yi Li. 2022. Invcon: A dynamic invariant detector for ethereum smart contracts. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–4.
[67] Ye Liu, Chengxuan Zhang, et al. 2024. Automated Invariant Generation for Solidity Smart Contracts. *arXiv preprint arXiv:2401.00650* (2024).
[68] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC*

conference on computer and communications security. 254–269.

[69] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. 2021. Pluto: Exposing vulnerabilities in inter-contract scenarios. *IEEE Transactions on Software Engineering* 48, 11 (2021), 4380–4396.

[70] Andrei-Dragoş Popescu. 2020. Decentralized finance (defi)–the lego of finance. *Social Sciences and Education Research Review* 7, 1 (2020), 321–349.

[71] QuillAudits Team. 2025. Decoding What Went Wrong with Bedrock: $2M Exploit. https://www.quillaudits.com/blog/hack-analysis/bedrock-2million-exploit Accessed: 2025-12-06.

[72] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting Existing Smart Contracts against Re-Entrancy Attacks. *arXiv preprint arXiv:1812.05934* (2018).

[73] Fabian Schär. 2021. Decentralized finance: On blockchain-and smart contract-based financial markets. *FRB of St. Louis Review* (2021).

[74] Spherex. 2024. About Spherex. https://www.spherex.xyz/about Accessed: 2024-11-12.

[75] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th annual computer security applications conference*. 664–676.

[76] Uniswap Labs. 2024. Uniswap Protocol. https://uniswap.org/. Accessed: 2024-12-18.

[77] Zexu Wang, Jiachi Chen, Yanlin Wang, Yu Zhang, Weizhe Zhang, and Zibin Zheng. 2024. Efficiently detecting reentrancy vulnerabilities in complex smart contracts. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 161–181.

[78] William Zhang, Sebastian Banescu, Leonardo Pasos, Steven Stewart, and Vijay Ganesh. 2019. Mpro: Combining static and symbolic analysis for scalable testing of smart contract. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 456–462.

[79] Zhuo Zhang, Zhiqiang Lin, Marcelo Morales, Xiangyu Zhang, and Kaiyuan Zhang. 2023. Your exploit is mine: Instantly synthesizing counterattack smart contract. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1757–1774.

[80] Peilin Zheng, Zibin Zheng, and Xiapu Luo. 2022. Park: accelerating smart contract vulnerability detection via parallel-fork symbolic execution. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 740–751.