A History of GREED: Practical Symbolic Execution for Ethereum Smart Contracts

Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Dongyu Meng, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna

University of California, Santa Barbara
`{ruaronicola,degrigis,robert349,dmeng, grishchenko,chris,vigna}@ucsb.edu`

Abstract. Smart contracts have transformed blockchain applications, enabling decentralized computation and automated asset management without intermediaries. However, with the growth of decentralized finance, the high financial stakes make smart contract vulnerabilities particularly critical. Because vulnerabilities often go undetected, they lead to substantial losses and diminished trust in blockchain systems.
Symbolic execution has emerged as a powerful technique to uncover subtle vulnerabilities by systematically exploring feasible execution paths. However, most existing symbolic execution tools for smart contracts are tailored to specific vulnerability patterns, making them unsuitable for detecting new types of vulnerabilities. In this paper, we introduce GREED, a highly versatile symbolic execution framework for Ethereum (or EVM-based) smart contracts. GREED features a state-of-the-art symbolic execution engine coupled with a suite of supporting analyses and a modular design that allows security researchers to prototype new analyses rapidly. To evaluate the effectiveness and extensibility of GREED, we compare it with the state-of-the-art. We first show that GREED can explore significantly more code paths – reaching 84% of all `CALL` statements, as opposed to 9% on average across existing tools. To demonstrate the ease of use (and extensibility) of GREED, we then implement a novel analysis to detect controllable `JUMPI` instructions and evaluate it against all deployed contracts on Ethereum and Binance Smart Chain (BSC), identifying 390 previously unknown vulnerable contracts.
By releasing GREED to the community, we aim to lower the barrier to developing advanced security analyses for smart contracts, empowering security researchers to rapidly prototype new analyses and contribute to a more secure and resilient blockchain ecosystem.

Keywords: Ethereum · Smart Contract · Symbolic Execution.

1 Introduction

Ethereum [15] is a global, decentralized blockchain that enables the deployment and execution of decentralized programs (smart contracts). Smart contracts are immutable programs that run on the Ethereum Virtual Machine (EVM) and are executed on demand by blockchain users. Smart contracts have transformed the way transactions are executed, enabling decentralized applications and automated asset management without intermediaries.

Ethereum (and other blockchains) have witnessed the explosive growth of a new form of blockchain-based finance that is known as decentralized finance (DeFi) – a rich ecosystem of digital currencies, financial tools, and financial services. Because of the exceptionally high stakes involved [12], identifying and fixing vulnerabilities in smart contracts has become critical. Once deployed, smart contracts cannot be easily patched, and exploits can lead to substantial financial damage and loss of trust in blockchain systems [13]. Therefore, rigorous analysis of smart contracts is necessary to ensure their security.

Symbolic execution [1] (SE) has emerged as a powerful technique for smart contract analysis. SE systematically explores a contract in an emulated environment with symbolic variables representing possible (but unknown) inputs. As the execution progresses, the SE system (or engine) tracks the state of the EVM – e.g., program counter, stack, and memory. At specific points in the execution, the engine queries a constraint solver to determine whether a given state is satisfiable – that is, whether each symbolic variable has a feasible concrete solution. When the execution reaches a conditional branch, and both the condition and its negation are satisfiable, the execution path forks, and both branches are explored separately. This enables the generation of concrete inputs that reproduce specific program behaviors, allowing one to uncover subtle bugs that might evade traditional testing methods (e.g., fuzz testing).

Related work. Over the years, many SE tools have been developed to detect vulnerabilities in smart contracts. Some focus on the formal verification of specific properties [27, 29, 32, 33, 36]. For example, VerX [27] uses SE and induction proofs to study safety properties. Others identify known vulnerability patterns [4, 10, 18, 22–24, 26, 28, 31]. For example, teEther [23] identifies contracts that leak funds to arbitrary users. While existing tools have shown some success in their respective domains, they suffer from two key limitations: First, the symbolic execution engines of existing tools lack critical analysis features – for instance, a precise memory model – that limit their effectiveness. Second, the architecture of existing tools is typically designed around specific vulnerability patterns, making it challenging to adapt them to new vulnerabilities and extend their capabilities beyond the original scope.

Our approach. In this paper, we introduce Greed, a highly versatile SE framework designed for the analysis of EVM-based smart contracts. Greed addresses the limitations of existing tools by providing a novel combination of analysis techniques, including both a state-of-the-art SE engine and a suite of supporting analyses. Unlike traditional tools (with a fixed set of predefined analyses), Greed enables security experts to build new analyses tailored to their needs. Our experiments show that Greed's architecture allows for more efficient path exploration – and superior flexibility – without compromising analysis accuracy.

We implemented Greed in approximately 10,000 lines of Python code and released it as an open-source project[1]. Greed has been met with enthusiasm by the community. After the open-source release, the project attracted hundreds

[1] https://github.com/ucsb-seclab/greed

of new users (in terms of distinct project downloads, GitHub “stars”, and community contributions). We are also aware of several academic institutions and corporations that are either actively using Greed or evaluating it for potential use in their systems.

This paper makes the following contributions:

- We describe Greed, a highly versatile symbolic execution framework designed for EVM-based smart contracts. Greed features a state-of-the-art symbolic execution engine and a novel combination of analysis techniques within a modular and extensible architecture, enabling security experts to tackle complex security challenges.
- We compare Greed against the state-of-the-art and show that it can explore significantly more code paths. Greed outperforms all existing tools, reaching 84% of all `CALL` statements, compared to 9% across alternatives (on average).
- To demonstrate the ease of adding additional security analysis, we implement a novel checker to detect controllable `JUMPI` instructions and evaluate it against all contracts in Ethereum and BSC [3], identifying 390 previously unknown vulnerable contracts.

2 Motivation

Existing symbolic execution systems focus on detecting known classes of vulnerabilities. This specialization has led to two main limitations. First, existing systems often forego implementing comprehensive, robust analyses, opting instead for a subset of features tailored to the targeted vulnerabilities. A precise implementation of all analysis features is sometimes unnecessary for individual security analyses. For example, ERC20 tokens rarely interact with external contracts. Thus, a full-fledged cross-contract analysis may be unnecessary for analyzing ERC20 token contracts [19]. Second, in addition to the lack of analysis features, many existing systems lack any underlying static analysis, such as control-flow graph (CFG) recovery. Yet, a balanced integration of static and dynamic analysis is crucial for building sophisticated security tools. The absence of static analyses makes extending and scaling existing systems (for instance, with exploration strategies) inherently challenging. This underscores the necessity for a versatile unified analysis framework that can be repurposed for complex, evolving security analyses.

2.1 Basic Analysis Features

Modern smart contracts frequently use cross-contract interactions, memory operations, and hash functions. Not properly supporting these three features leads to significant limitations in the engines’ analysis capabilities. For instance, in Figure 1, we present a contract that – although seemingly simple – cannot be precisely analyzed without implementing the aforementioned analysis features.

```
 1 pragma solidity ^0.8.0;
 2
 3 struct Action {
 4     address router;
 5     bytes data;
 6 }
 7
 8 contract Dispatcher {
 9     address router = 0xROUTER;
10
11     function set_router(Action action) public returns (Action) {
12         if (action.router == address(0)) {
13             action.router = router;
14         }
15         return action;
16     }
17 }
18
19 contract Executor {
20     Dispatcher dispatcher = Dispatcher(0xDISPATCHER);
21     mapping(address => uint256) routerCallCounts;
22
23     function execute(Action[] memory actions) public {
24         // Actions (copied in memory) have symbolic offset and size
25         for (uint256 i = 0; i < actions.length; i++) {
26             // Cross-contract call
27             Action memory action = dispatcher.set_router(actions[i]);
28             // Another (controllable) cross-contract call
29             // BUG: ASSUMES ACTION.ROUTER IS ALWAYS SET BY DISPATCHER
30             action.router.call(action.data);
31             // Increment mapping variable
32             routerCallCounts[action.router] += 1;
33         }
34         // Check mapping variable
35         require(routerCallCounts[dispatcher.router] > 1);
36     }
37 }
```

Fig. 1. Simplified Solidity code of the Executor contract. The contract parses a list of provided actions (CALLDATA), interacts with the Dispatcher contract to fetch the `router` address, then interacts with the router and updates the respective interaction counter. RED: requires a precise memory model. YELLOW: requires cross-contract analysis. GREEN: requires a precise `SHA` model.

Cross-contract interactions. Ethereum allows smart contracts to `CALL` functions of other contracts (Figure 1: Line 27, Line 30), enhancing modularity and code reuse. However, interactions inherently increase the complexity of smart contracts and can introduce unexpected bugs. For instance, the external contract might operate maliciously and inadvertently change its behavior. Without precise cross-contract analysis, it is impossible to detect vulnerabilities arising from such interactions.

Memory model. In the EVM, memory is a volatile, mutable storage area that exists only during the execution of a contract function. Any data stored in memory is freed once the execution terminates. Memory is efficient because it avoids the overhead of writing to persistent blockchain storage. This makes it suitable for intermediate calculations, temporary variables, and data manipulation within a function call. Nonetheless, modeling symbolic memory operations is challenging, and existing systems resort to approximations – such as the strategic concretization of symbolic offsets and lengths. When a symbolic memory buffer (for example, the `actions` array on Line 23) is accessed (Line 25), it is undeniably convenient to concretize its length. However, this prevents the system from detecting vulnerabilities that arise from different configurations. For example, the Executor contract reverts unless we provide an array with at least two actions – since the variable `routerCallCounts` is incremented at most once per array element.

Hash functions. Handling cryptographic hash functions (`SHA`) is crucial due to their pervasive use by dynamic data types – such as arrays and mappings. In Solidity, fixed-size data types have predetermined slots in persistent storage, but dynamic data types grow during execution. To manage this, Solidity computes storage slot offsets dynamically using hash computations: First, all array and mapping variables are assigned a "base slot". Then, the storage slot for an array element with index `i` is calculated as `SHA(base_slot) + i`. Similarly, the storage slot for a mapping element with key `key` is calculated as `SHA(key, base_slot)`. Accurately modeling these hash computations is essential for recognizing data storage patterns (e.g., Line 32 and Line 35) and detecting vulnerabilities related to data access and manipulation.

2.2 Beyond the State-of-the-Art

Robust basic analysis features provide a necessary foundation for smart contract analysis. However, these capabilities alone are insufficient for thoroughly analyzing modern, complex blockchain applications with evolving attack vectors. We argue that it is essential to complement basic analysis features with supporting techniques such as static analysis and exploration strategies. Static analysis techniques – such as control-flow graph recovery and dependency tracking – can isolate critical code regions where vulnerabilities are most likely to reside. Exploration strategies – such as directed search – allow directing the symbolic execution engine toward (previously identified) critical code regions to verify the presence (or absence) of vulnerabilities. Rather than exhaustively exploring all paths, exploration strategies allocate resources to areas with a higher likelihood of revealing subtle bugs, thus addressing long-standing challenges like state explosion. In the following sections, we present our approach to integrating advanced analysis features in our symbolic execution framework.

3 Practical Symbolic Execution with Greed

Figure 2 shows an overview of Greed's architecture. Greed exposes several interfaces that enable both static and dynamic analysis. Initially, the contract is pre-processed using the Gigahorse static analysis framework [20,21]. The contract's intermediate representation (organized in functions, blocks, and statements) is stored in a `project` object. The `project` exposes an interface to all

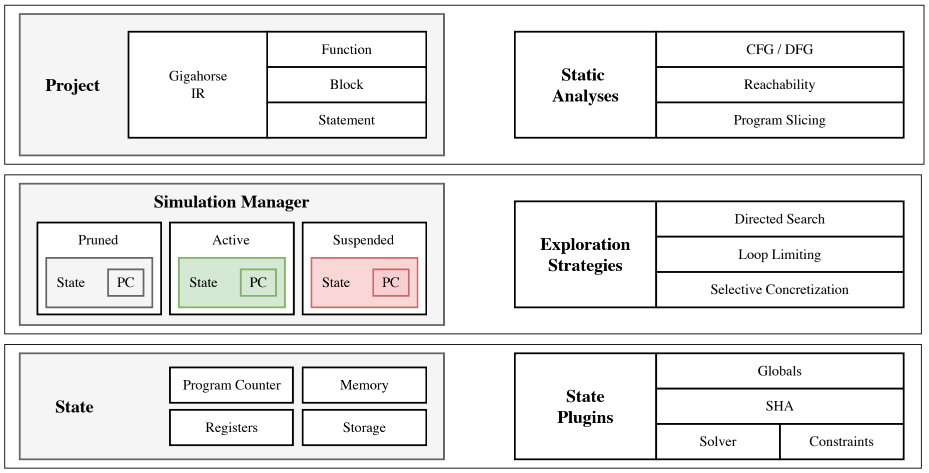

Fig. 2. Overview of Greed. The `project` object exposes static information. The `simulation manager` tracks all execution states and allows one to manipulate them. The `states` store the execution environment and additional context.

available static analyses (e.g., CFG, Slicing). During execution, the `simulation manager` orchestrates all the execution states, which are organized in "stashes" that indicate whether they are active, pruned, suspended, etc. The `simulation manager` also accepts various `exploration strategies`. At a high level, exploration strategies allow one to programmatically manipulate the execution states and determine which state should be executed next – or which states are uninteresting to explore. Each `state` represents a snapshot of the execution at a specific program location, which stores both the execution environment and additional context. *This is where the basic analysis features live* (see Section 2.1). Finally, `state plugins` track additional context (e.g., `SHA` operations and constraints) that allows for checking the satisfiability of an execution state. The modularity of Greed allows one to easily write new static analyses, exploration strategies, and state plugins – or experiment with different memory models and solvers.

3.1 Static Analysis

Greed operates on the Gigahorse IR, which provides its foundational static analysis capabilities: decompilation, IR lifting, constant folding, basic control-flow and data-flow modeling, and loop analysis. This allows Greed to instead focus on advanced static analyses (e.g., backward and forward program slicing, reachability analysis) and symbolic execution, which are highly valuable for building complex security tools. Below, we discuss some examples of static analyses available in Greed.

Control-flow graph (CFG). Gigahorse provides state-of-the-art CFG and call-graph reconstruction for EVM bytecode. This is automatically available in

Greed. The CFG encodes control-flow relationships, enabling reasoning about reachability between statements. For instance, this is essential for directing the execution toward a desired statement.

Data-flow graph (DFG). Similarly, Gigahorse also provides state-of-the-art DFG reconstruction.The DFG captures data dependencies, allowing one to track how variables are assigned and manipulated throughout the contract.

Reachability. Greed's reachability analysis allows one to automatically determine whether an execution path might exist between two program points. For blocks within the same function, Greed directly analyzes their relationships in the CFG. For blocks in different functions, Greed identifies possible sequences of function calls that connect them. When available, Greed also examines the call stack to identify additional paths that connect the two program points.

Program slicing. Leveraging the CFG and DFG, Greed can calculate a "slice" of statements that affect (backward) or are affected by (forward) a given variable. For instance, this is essential for implementing under-constrained execution, which enables an approximate but lightweight analysis of local properties.

3.2 Exploration Strategies

Exploration strategies allow for the orchestration of execution states and typically employ a combination of state pruning, prioritization, and manipulation. Pruning allows one to discard states that are unfit for the desired analysis goals. Prioritization allows one to prioritize the exploration of certain states. Manipulation allows one to alter (the execution environment of) certain states. Below, we discuss some examples of exploration strategies available in Greed.

Directed search. Directed search is an example of an exploration strategy that can leverage both state pruning and prioritization to direct the symbolic execution toward a desired (target) statement. This strategy is supported by a CFG-driven reachability analysis. States closest to the target statement are prioritized. States unfit to reach the target statement are (optionally) discarded. This allows one to focus the analysis on specific execution paths that are relevant to a desired property.

Under-constrained search. Under-constrained search allows executing arbitrary program slices by first creating a symbolic state at a specific program location and then manipulating the execution states to manage undefined behavior. First, Greed creates a symbolic execution state at the first program location in the slice. Then, the under-constrained search rewrites all undefined variables to assign them fresh 256-bit symbolic variables. Optionally, the under-constrained search can guide (force) the execution along a predetermined, statically observed path – even if that path is unfeasible in a fully constrained context. This allows one to effectively study the (security) properties of arbitrary program slices without incurring the overhead of fully-constrained symbolic execution.

Loop limiting. Loop limiting is an essential technique for mitigating state explosion during symbolic execution. In Greed, a counter-based strategy monitors the number of times a given program point is reached. Once a predefined threshold is exceeded, we prune the corresponding execution state. This approach effectively controls redundant loop iterations, ensuring that excessive unrolling does not overwhelm the analysis.

State monitoring and rewriting. State rewriting enables the dynamic modification of execution states to incorporate external information – such as concrete execution data, observed blockchain states, or freshly generated symbolic variables. Through this process, one can refine the analysis context to reflect relevant properties or to simulate any desired execution state. For example, a symbolic variable representing an asset's price can be replaced with its actual value retrieved from a live oracle, thereby allowing the analysis to mirror realistic market conditions. Additionally, by coupling state rewriting with state monitoring, Greed can collect valuable metrics (e.g., constraint-solving time) that can be used to identify or prune paths with a desired property – for example, computationally expensive paths.

Selective concretization. Selective concretization is an example of state manipulation, where a heuristic determines whether any environment variable should be concretized. This is helpful to enforce a specific property ("the value of variable `X` must be exactly 42 to trigger the vulnerability") or to simplify the analysis when the constraints are too complex (at the cost of possible false negatives).

Classic prioritization. Depth-first search (DFS) and breadth-first search (BFS) are classic examples of state prioritization. Execution states are never pruned. Instead, a heuristic determines which states should be explored first. In DFS, deep execution states are explored first. In BFS, shallow execution states are explored first. Exhaustive strategies such as BFS or DFS are often impractical for large contracts. In fact, even simple loops or repeated subroutine calls can rapidly inflate the state space. For this reason, exhaustive search strategies are often paired with additional strategies for state pruning.

3.3 Additional Analysis Features and Implementation Details

In the following paragraphs, we discuss important implementation details beyond the analysis features detailed above.

- **Cross-contract interactions:** To handle cross-contract interactions, Greed defaults to concretizing both the target address and the parameters of the `CALL` instruction. This allows approximating the execution state without incurring the overhead of symbolically executing an external (possibly undetermined) contract. Nonetheless, Greed can be easily configured to support fully symbolic `CALL` parameters – in fact, this feature (symbolic cross-contract interactions) has been used in the context of other academic works.
- **Memory and storage modeling:** Greed implements a precise memory model [16] that tracks EVM memory as a byte-addressable array supporting symbolic reads, writes, and memcopy-style operations. Our design employs an instantiation-based approach, where memory updates are lazily instantiated (on demand) during reads. To avoid redundant constraint instantiation,

we also integrate a caching mechanism such that when a read is performed at a concrete address, the corresponding value (indexed by both the address and read width) is cached. For storage, GREED uses a hybrid model based on array theory, treating storage as an array of 256-bit words keyed by either concrete or symbolic values. Optionally, concrete storage reads (`SLOAD`s) can retrieve actual on-chain data at a specified block number, and our design allows the use of these concrete values in symbolic operations.

– **Hash functions:** We employ a two-phase strategy for handling symbolic hash operations such as `SHA`. During symbolic execution, `SHA` instructions are captured as symbolic expressions that record the input parameters (offset, size, and memory contents) in order. When operating in "greedy" mode, GREED first attempts to concretize these parameters. If a unique solution is found, GREED computes its `SHA` hash value [2] and adds constraints that link the symbolic expression to this concrete value. Otherwise, it instantiates Ackermann constraints [5] to link multiple `SHA` operations as non-interpretable functions. After execution, a dedicated resolver plugin steps through the observed `SHA` operations in chronological order and fixes their outcomes by re-evaluating the memory and enforcing the appropriate constraints.
– **Solver integration:** GREED interfaces with an SMT solver – by default, Yices [14] – to query the satisfiability of path constraints. As with most components in our architecture, alternative SMT solvers can be substituted. During development, we evaluated various solvers, such as Z3 [11] and Boolector [8] – and found that Yices consistently offered the best performance.

Finally, as briefly mentioned above, GREED also offers high-level APIs for implementing custom vulnerability checks, exploration strategies (for state pruning, prioritization, and manipulation), and domain-specific analyses, simplifying the development of new smart contract security tools.

4 Evaluation

We evaluate the performance, analysis features, and versatility of GREED through a series of experiments. First, we qualitatively compare its analysis capabilities against existing systems (see Table 1), highlighting comprehensive support for basic analysis features, static analysis, and advanced exploration strategies. Second, we quantitatively compare GREED's targeted exploration capabilities against other state-of-the-art systems. Our results show that GREED reaches significantly more (10x) `CALL` statements in a sample of (randomly chosen) smart contracts. Third, we study the effect of different configuration settings on the performance of GREED. Finally, to demonstrate the extensibility of GREED, we implement a novel analysis to detect controllable `JUMPI` instructions. GREED identifies 390 previously unknown vulnerable contracts on Ethereum and BSC.

Experimental setup. For all our experiments, we use a server equipped with 512GB of RAM and dual Intel Xeon Gold 6330 CPUs. We use GNU Parallel [34] to parallelize our tasks, and always limit each task to 5GB of RAM and 60 seconds of CPU time. We compare against the latest available version of all

Table 1. Comparison of the features of existing systems. ○ Not implemented. ◐ Partially implemented. ● Fully implemented.

Tool	CROSS	MEM	HASH	STATIC	API
Oyente [24]	○	◐	○	○	○
Maian [26]	○	◐	◐	○	○
teEther [23]	○	◐	◐	◐	○
Manticore [25]	◐	◐	◐	○	○
Mythril [10]	◐	◐	◐	○	○
EthBMC [18]	●	●	●	○	○
Greed	◐	●	●	●	●

tools at the time of writing: Maian [26] at commit `3965e30`, teEther [23] at commit `04adf56`, Manticore [25] at commit `8861005`, Mythril [10] at commit `125914a`, and EthBMC [18] at commit `e887f33`. Notably, integrating Maian in our evaluation environment required significant modifications – due to syntax errors, broken dependencies, and missing implementations for several key opcodes. Similarly, we were unable to run Oyente [24] in our environment, and thus, we have excluded it from our evaluation.

4.1 Analysis Features

In Table 1, we show a comparison between existing systems and Greed, with a focus on basic analysis features (similar to Frank et al. [18]), availability of static analyses, and availability of a high-level API to develop ad hoc static and dynamic analyses. In our comparison, we only include symbolic execution tools that are both publicly available and operate on EVM bytecode – even in the absence of source code.

Cross-contract interactions. Greed, Manticore, Mythril, and EthBMC are the only systems that support some form of cross-contract analysis. Manticore and Mythril only support `CALL` instructions with concrete (or concretized) parameters. Greed also supports concrete `CALL` parameters and handles symbolic parameters through concretization. Nonetheless, Greed can be configured to handle fully symbolic parameters (see Section 3.3). EthBMC supports concrete or fully symbolic `CALL` parameters.

Memory model. Maian supports symbolic memory reads (not writes). teEther, Manticore, and Mythril support simple symbolic memory operations, but must concretize all symbolic memcopy-like operations (e.g., `CALLDATACOPY`). Oyente does not support any memcopy-like operation. Greed and EthBMC implement a precise memory model [16] and can handle symbolic memory reads, writes, and memcopy-style operations. As discussed in Section 3.3, Greed also implements a caching mechanism to avoid redundant constraint instantiation.

Hash functions. Maian, teEther, and Mythril support the hashing of memory buffers with fully concrete offsets, lengths, and values. Oyente does not support symbolic hashing operations, and approximates the result of concrete

Table 2. Number (Percentage) of reached `CALL` instructions across different analysis tools. We run each system for 60 seconds (per contract) to assess its exploration capabilities, highlighting the coverage differences.

	Small	**Medium**	**Large**	**Total**
Maian[2]	127 (13%)	20 (2%)	4 (0%)	151 (5%)
teEther	247 (25%)	58 (6%)	1 (0%)	306 (10%)
Manticore	157 (16%)	14 (1%)	2 (0%)	173 (6%)
Mythril	294 (29%)	118 (12%)	12 (1%)	424 (14%)
EthBMC	224 (22%)	87 (9%)	31 (3%)	342 (11%)
Greed	960 (96%)	821 (82%)	745 (75%)	2,526 (84%)

hashing operations. Greed, Manticore, and EthBMC support the hashing of arbitrary (symbolic or concrete) memory buffers.

Static analysis. To complement our robust basic analysis features, Greed integrates advanced static analyses that allow us to focus symbolic execution on critical code regions. Greed inherits a number of static analyses from Gigahorse [20, 21] (e.g., CFG recovery) and implements additional static analyses (such as program slicing). Among the other tools, only teEther incorporates static analysis – specifically, CFG recovery and backward slicing.

High-level APIs. Finally, Greed is the only system that offers a high-level API to develop ad hoc static and dynamic analyses. Greed also offers a number of (built-in) exploration strategies such as directed search, loop limiting, state rewriting, and selective concretization.

4.2 Exploration Capabilities

While existing systems excel at detecting specific vulnerabilities, they prove lacking when evaluated on slightly different tasks. We demonstrate the performance of Greed with a basic code reachability experiment. First, we select a target smart contract with a `CALL` statement x. Then, we alter all existing systems to simply emit a report and terminate when successfully executing (reaching) the chosen statement x. To do this, we leverage the ability of Maian, teEther, and EthBMC to detect "prodigal" contracts – i.e., `CALL` statements with positive Ether value and controllable target address [26]. We (slightly) modified that analysis so that when a `CALL` statement is reached, instead of verifying the prodigal property, we just check whether the instruction address matches that of the chosen statement x. If so, the analysis simply terminates. Similarly, we modify the execution engine of Manticore and Mythril to terminate when executing the chosen statement x. As mentioned above, we were unable to run Oyente in our environment, and thus, we excluded it from our qualitative evaluation. Finally, we run Greed in its default configuration (with directed search).

[2] Integrating Maian in our evaluation environment required significant modifications.

We evaluate all tools on a sample of 3,000 (randomly chosen) Ethereum contracts[3] and report our findings in Table 2. In summary, we find that Greed outperforms all existing tools, reaching 84% of all `CALL`s – whereas others reach 9% on average. We attribute the performance gap observed in related work to a combination of (1) limited basic analysis features and (2) lack of (robust) exploration strategies. In fact, existing systems perform reasonably well on small contracts but struggle to handle the complexity of larger contracts. For example, teEther is the only system with a (CFG-driven) exploration strategy, but its CFG recovery often fails on larger contracts. Moreover, we observe that all tools have several failures related to misimplemented instructions and mishandling of external or symbolic data. For example, teEther discards any execution path that includes instructions such as `RETURNDATACOPY` or `RETURNDATASIZE`, whereas Maian fails to model instructions such as `SELFBALANCE`.

4.3 Ablation Study

In the following paragraphs, we study the effect of different analysis configurations on the performance of Greed. In its default configuration, Greed uses full support for symbolic memory operations (including read, write, and memcopy-like operations), symbolic hash operations, and a directed search strategy that uses prioritization – without pruning.

Directed Search. Table 3 shows the number of reached `CALL` instructions under different directed search configurations. Disabling pruning (while keeping prioritization active) results in a slight increase in reached targets across all contract sizes (from 958 to 960 for small contracts, from 780 to 821 for medium contracts, and from 708 to 745 for large contracts). We attribute this to imprecisions in the recovered control-flow graph that may incorrectly rule out reachable targets: When this happens, the lack of pruning allows Greed to explore these additional paths. However, this gain comes with an increased memory footprint (rising from an average of 180MB to 260MB per contract). In contrast, disabling prioritization leads to a notable drop in performance, as the execution engine wastes resources exploring paths that are farther from the target state. When both pruning and prioritization are disabled, the deterioration in performance is even more pronounced, especially for medium and large contracts. Importantly, even in this worst-case configuration, Greed still outperforms all other tools by a wide margin.

Memory Model. We further investigate the impact of our precise symbolic memory model on Greed's performance by replacing it with (gradually) simplified variants – that is, disabling our caching layer and symbolic memory operations. We observe that disabling our caching layer results in a sharp drop in analysis performance across all contract sizes, although this is more evident in larger contracts. As detailed in Table 4, disabling our symbolic memory model

[3] We compute the size distribution of all deployed contracts and sample 1,000 small contracts (smallest 25%), 1,000 medium contracts, and 1,000 large contracts (largest 25%) with distinct code.

Table 3. Number of reached `CALL` instructions under different directed search configurations. Disabling pruning yields a slight coverage increase but raises memory usage, whereas disabling prioritization leads to a notable drop in performance – more pronounced in large contracts.

	Prioritization			**No Prioritization**		
	S	M	L	S	M	L
Pruning	958	780	708	953	745	646
No Pruning	960	821	745	947	513	352

Table 4. Comparison of the number of reached `CALL` instructions under different memory model configurations. Using a concrete memory model results in faster analysis times at the cost of decreased precision. Our caching layer allows for boosting performance without compromising precision.

	Symbolic Memory			**Concrete Memory**		
	S	M	L	S	M	L
Memory Cache	960	821	745	937	866	757
No Memory Cache	912	707	633	925	745	720

(and instead using a concrete one) results in a modest overall boost in performance. We observe that, although approximating symbolic memory operations with their concrete counterparts may result in faster analysis times, *this comes at the cost of a much-decreased analysis accuracy.* In fact, we argue that Greed achieves the best analysis results by combining our symbolic memory model with our caching layer: this configuration yields robust performance without compromising analysis accuracy.

Hash Functions. We find that disabling our precise handling of (symbolic) hash operations results in a slight boost in analysis performance (from 960 to 962 for small contracts, from 821 to 824 for medium contracts, and from 745 to 755 for large contracts). Similar to the observations above, while approximating hash operations might result in faster analysis times, this comes at the cost of a much-decreased analysis accuracy.

Cross-contract interactions. Finally, in the context of this experiment, our reachability analysis stops when it encounters an external interaction (`CALL`). Therefore, any configuration change in our handling of external interactions does not lead to any change in performance.

Overall, our results underscore the importance of incorporating advanced analysis features – such as exploration strategies and a precise symbolic memory model – in Greed. We observe that while disabling pruning can reveal additional reachable targets, prioritization is essential to guide the search efficiently and keep the state space manageable. Similarly, our precise memory model and caching layer enable Greed's accurate analysis of complex memory operations, thus contributing to its overall superior performance.

```python
# Discard statements with concrete jump destination
stmts = set()
for s in proj.stmts:
    if s.op == "JUMPI" and not s.dest_val:
        stmts.add(s)

# Analyze each JUMPI statement
for s in stmts:
    # Set up directed symbolic execution
    simgr.use_strategy(DirectedSearch(s))

    # Explore each state until we reach the target statement
    for found in simgr.findall():
        # Jump condition must be satisfied
        found.solver.add_constraint(Equal(s.cond_val, TRUE))

        # Jump destination must be controllable
        found.solver.add_constraint(Equal(s.dest_val, ARBITRARY))

        # Check if the state (with the new constraints) is satisfiable
        if found.solver.is_sat():
            yield found.solver.eval_memory(found.calldata, CALLDATASIZE)
```

Fig. 3. Simplified Python code for the controllable `JUMPI` analysis.

4.4 Detecting Controllable `JUMPI`s

In this section, we demonstrate that Greed can be easily tailored to novel security analyses. To this end, we implement a novel analysis to detect controllable `JUMPI` instructions – i.e., conditional `JUMP` instructions. A controllable `JUMPI` allows an attacker to hijack the program counter, and thus take control of the program execution. This vulnerability has been recently reported [37] in a highly profitable MEV bot and could have resulted in hundreds of thousands of US dollars of financial damage. We implement this analysis in 50 lines of Python code. Figure 3 presents the core of our analysis script.

First, we (statically) inspect all contract statements to identify any `JUMPI` instructions with a non-constant destination (target) addresses. This lightweight analysis reduces the number of contracts in scope from 4.1M (all contracts with distinct bytecode across Ethereum and BSC) to 1,141. We symbolically execute these contracts and use directed search to reach the target statement. We add additional constraints to enforce that (1) the guarding condition for the `JUMPI` instruction is satisfied, and (2) the `JUMPI` destination is controllable. If our engine reaches the `JUMPI` instruction and the two aforementioned constraints are satisfied, we synthesize a concrete attack and verify it against a private fork of the respective chain. We evaluate our analysis on all deployed contracts in Ethereum and BSC and identify 134 and 256 previously unknown vulnerabilities, respectively, as well as one known vulnerability [37]. We manually confirmed that 130 of the 134 Ethereum contracts are still vulnerable at the time of writing (block 22,279,016). Three of the contracts were vulnerable in the past but have since been destructed and redeployed. One of the contracts contains an invalid `JUMPI` destination derived from a memory operation that does not appear to be controllable. We confirmed that all 256 BSC contracts are still vulnerable at the time of writing (block 48,398,024). We reported all issues to the Cybersecurity and Infrastructure Security Agency [9].

```
pragma solidity ^0.8.0;

contract TradingBot {
    // Public execute function lacking proper access control
    function execute(address target, bytes calldata data) public {
        // Forwards untrusted input to the target contract
        target.call(data);
    }
}

contract Token {
    mapping(address => uint256) public balances;

    // The transfer function deducts tokens based on msg.sender
    function transfer(address recipient, uint256 amount) public {
        [...]
    }
}
```

Fig. 4. Simplified Solidity code of the TradingBot and Token contracts. The TradingBot contract is vulnerable to a confused deputy attack.

5 Case Studies

In this section, we illustrate how Greed has been successfully applied to build advanced program analysis systems for Ethereum smart contracts. We focus on two representative case studies: (a) detecting confused deputy vulnerabilities and (b) detecting storage collision vulnerabilities. Both studies leverage Greed's symbolic execution capabilities – augmented with domain-specific rules – to analyze real-world contracts at scale and automatically generate proof-of-concept exploits.

5.1 Detecting Confused Deputy Vulnerabilities

Confused deputy vulnerabilities occur when an attacker hijacks a smart contract's privileged operations via an inter-contract call (e.g., `CALL`) that is not intended to handle untrusted input. This can lead to unauthorized actions such as transferring assets or modifying critical state variables. For example, the `TradingBot` contract in Figure 4 exposes a public `execute` function that forwards untrusted input directly to any target contract. As a result, an attacker can craft a transaction that redirects this call to the `Token` contract's `transfer` function, effectively leveraging the `TradingBot`'s identity (and privileges) to initiate unauthorized asset transfers.

Implementation Overview. While we provide a high-level summary of the approach here, the complete system, Jackal, is detailed in a separate paper [22]. Jackal is built on top of Greed's core symbolic execution engine and incorporates several analysis stages tailored to detecting confused deputy vulnerabilities:

- **Confused Contract Discovery.** Jackal leverages directed symbolic execution to inspect inter-contract calls where untrusted input might influence (control) the target address or function selector. As a result, contracts with controllable `CALL` instructions are flagged as confused contract "candidates."

```
pragma solidity ^0.8.0;

contract Proxy {
    // Slot 0 -> implementation
    address public implementation;
    fallback() external payable {
        implementation.delegatecall(msg.data);
    }
}

contract Implementation {
    // Slot 0 -> owner (collides with Proxy)
    address public owner;
    function setOwner(address _owner) public {
        owner = _owner;
    }
}
```

Fig. 5. Simplified Solidity code of the Proxy and Logic contracts. The interaction of such contracts results in a storage collision.

– **Target Contract Discovery.** For each confused contract candidate, JACKAL examines historical blockchain transactions to identify interesting external interactions and determines whether such interactions could lead to state modifications (e.g., via `SSTORE`) that exploit the confused contract's identity. When JACKAL determines that an external interaction could lead to the exploitation of the confused contract's identity, the respective external contract is flagged as a "target" contract.
– **Exploit Generation.** For each target contract, JACKAL leverages GREED to synthesize a transaction that forces the confused contract to invoke sensitive functions in the target contract, thereby demonstrating the exploit. The synthesized transaction is then replayed in a local blockchain simulator to confirm that the attack does not unexpectedly revert.

Through these stages, JACKAL enables end-to-end detection and exploitation of confused deputy vulnerabilities. JACKAL's analysis of over 2.3 million smart contracts identified 529 vulnerable instances and synthesized 31 working end-to-end exploits. All 31 exploits have been manually verified, demonstrating that attackers could potentially compromise digital assets valued at over one million US dollars.

5.2 Detecting Storage Collision Vulnerabilities

Storage collision vulnerabilities arise in proxy-based architectures, where a "proxy" contract delegates calls to separate "logic" contracts via the `DELEGATECALL` instruction. In this context, although the proxy and logic contracts execute independently, they both share the same underlying persistent storage. As a result, when the two contracts have conflicting interpretations of their storage slots, they might inadvertently overwrite such slots with the wrong value. This allows an attacker to overwrite privileged variables, potentially leading to unauthorized access (privilege escalation) and loss of funds. For example, in Figure 5, the `Proxy` contract reserves storage slot zero for its `implementation` variable. Instead, the `Implementation` contract reserves the same storage slot for its `owner` variable.

As a result, when the `Proxy` delegates a call (Line 6) to the `Implementation`'s `setOwner` function, the `owner` value overwrites the `implementation` variable in the `Proxy` contract, leading to a storage collision.

Implementation Overview. While we provide a high-level summary of the approach here, the complete system, Crush, is presented in a separate paper [28]. Crush builds on Greed to automatically detect and exploit storage collision vulnerabilities through the following analysis stages:

- **Component Discovery.** Crush analyzes on-chain transactions to identify clusters of contracts – namely, proxies and their corresponding logic contracts – that interact via `DELEGATECALL`.
- **Collision Discovery.** For each pair of proxy-logic contracts, Crush leverages Greed to symbolically execute their bytecode and infer the type of their storage variables. More precisely, after identifying all `SLOAD` and `SSTORE` instructions, Crush leverages (1) Greed's backward slice analysis to determine how each storage slot is computed and (2) Greed's forward slice analysis to deduce the accessed byte ranges. Then, Crush compares the inferred types of the proxy and logic contracts to detect collisions.
- **Exploit Generation.** Once a collision is detected, Crush verifies whether an attacker can exploit it by writing to a critical slot in one contract and reading it in another. To do this, Crush leverages Greed to synthesize concrete transactions that demonstrate the exploit.

By leveraging Greed's precise modeling of EVM instructions and storage access patterns, Crush uncovered critical storage collision vulnerabilities. These vulnerabilities could have led to serious incidents in practice: Crush's analysis of over 14 million smart contracts identified 14,891 vulnerable instances and synthesized 956 working end-to-end exploits. All profitable exploits have been manually verified, demonstrating that attackers could potentially compromise digital assets valued at over 6 million US dollars.

6 Discussion and Limitations

Greed inevitably inherits some limitations that arise from our design choices. First, we choose to build Greed directly on top of Gigahorse's IR, rather than extending an existing binary–analysis framework – such as `angr` [30]. This decision significantly simplifies our modeling of blockchain-specific concepts – e.g., blockchain state, transactions, persistent storage, cross-contract interactions. However, it also implies that sophisticated analyses that already exist in other frameworks, such as taint analysis, are not available out-of-the-box in Greed and must be re-implemented. While this creates unfortunate duplication of effort in the short term, it ultimately enables a more flexible, extensible, and research-friendly framework for smart contract security analysis.

Second, Greed's reliance on Gigahorse's intermediate representation (IR), provides robust static analysis capabilities, but makes Greed's effectiveness partly dependent on Gigahorse's accuracy. For example, inaccuracies such as missing JUMP destinations can cause pruning of paths that are in fact reachable. In Section 4.3 we show that this occasionally happens in practice: For some contracts, disabling pruning yields marginal coverage gains at the cost of a sharp increase in memory usage. Although we limit this dependency to well-tested features of Gigahorse (lifting, constant folding, and control-flow analysis), it remains a potential source of inaccuracies.

Other Limitations. Beyond the limitations discussed above, Greed shares modeling limitations common to similar symbolic execution systems. First, our handling of gas costs is deliberately simplified and may potentially miss vulnerabilities that arise from gas-specific behaviors. Second, by default, Greed employs a simplified handling of CALL instructions, which may miss vulnerabilities that require symbolic modeling of contract interactions. Additionally, the blockchain state (e.g., block number, timestamp, difficulty) remains symbolic by default, although one can optionally constraint such a state to actual (concrete) values when needed for more precise analysis. Addressing the aforementioned limitations, including the modeling of gas costs and cross-contract interactions, presents promising avenues for future research.

7 Related Work

Static Analysis. Early research in smart contract security focused on static analysis of the source code. Tools such as SmartCheck [35] and Slither [17] detect common vulnerabilities (e.g., re-entrancy, integer overflows) by scanning Solidity source code using rule-based approaches, offering quick insights to developers. Their availability and ease of use lowered the barrier for preliminary security audits. For example, Slither converts Solidity code into an intermediate representation for detailed data-flow and control-flow analysis, providing both vulnerability detection and potential code optimization insights.

In parallel, other efforts focused on direct analysis of EVM bytecode. Brent et al. proposed Vandal [7] and Ethainter [6], two tools that perform control-flow and data-flow analyses post-compilation, enabling insight even when source code is unavailable. In a similar vein, Grech et al. proposed Gigahorse [20] and Elipmoc [21] – a decompilation framework for EVM bytecode that also provides several rule-based vulnerability analyses. However, these tools often rely on fixed heuristics – such as rigid slicing rules or pattern matching – which may be insufficient to fully capture complex state interactions during execution.

Formal Verification. To provide stronger correctness guarantees, researchers have developed verification frameworks for smart contracts. For instance, Securify [36] operates on EVM bytecode and extracts predicates via a domain-specific language to capture compliance and violation patterns. Similarly, eThor [29] frames safety specifications in terms of reachability and uses an off-the-shelf SMT solver to reason about property violations. On the Solidity side, VerX [27] employs symbolic execution with induction and predicate abstraction to verify safety properties across multiple transactions, while VeriSmart [32] focuses on arithmetic safety through counterexample-guided invariant refinement. Extending these approaches further, Stephens et al. [33] incorporate liveness specifications to broaden the range of verifiable properties. Although these methods promise high-assurance security, they often incur significant engineering overhead, limiting their widespread adoption.

Symbolic Execution. Symbolic execution has emerged as a powerful technique for systematically exploring a contract's execution paths. One of the pioneering systems in this area, Oyente [24], demonstrated that symbolically executing EVM bytecode could effectively uncover vulnerabilities such as re-entrancy and transaction-ordering dependence. Mythril [10] is a symbolic execution-based tool that detects issues including integer overflows, unhandled exceptions, and unprotected self-destruct instructions. Similarly, Teether [23] and Maian [26] also leverage symbolic execution to identify vulnerable states. Manticore [25] and EthBMC [18] further advanced the state-of-the-art by integrating precise memory models and supporting cross-contract analysis. Nonetheless, Manticore does not integrate static analysis techniques – such as control-flow graph recovery or program slicing – limiting its ability to dynamically target critical code regions. Similarly, although EthBMC supports fully symbolic handling of cross-contract calls and a precise memory model, its monolithic design enforces rigid exploration strategies, making it difficult to extend to novel attack vectors.

In contrast to approaches that rely exclusively on static or dynamic methods, our framework Greed integrates static analyses (such as control-flow graph recovery and program slicing) with a flexible suite of symbolic exploration strategies – including directed search, loop limiting, state rewriting, and selective concretization. This unified approach preserves the core advantages of existing systems while adapting more readily to novel attack vectors.

8 Conclusion

We introduce Greed, a versatile open-source symbolic execution framework for EVM-based smart contracts. Greed addresses the limitations of existing tools by providing a novel combination of analysis techniques, including both a state-of-the-art SE engine and a suite of supporting analyses. Our experiments show that Greed reaches significantly more (10x) `CALL` statements in a sample of (randomly chosen) smart contracts. As a result, Greed enables more efficient path exploration – and superior flexibility – without compromising on the accuracy of the analysis. To demonstrate Greed's flexibility and ease of use, we implement a novel analysis to detect controllable `JUMP` instructions and evaluate it against all contracts in Ethereum and BSC [3], identifying 390 previously unknown vulnerable contracts. By releasing Greed to the community, we aim to lower the barrier to developing advanced security analyses for smart contracts, empowering security researchers to contribute to a more secure blockchain ecosystem.

Acknowledgments

This material is based upon work supported by NSF under Award No. CNS-2334709. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the NSF.

References

1. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) (2018)
2. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Annual international conference on the theory and applications of cryptographic techniques. Springer (2013)
3. Binance: Binance Smart Chain. `https://binance.com/en` (2024)
4. Bose, P., Das, D., Chen, Y., Feng, Y., Kruegel, C., Vigna, G.: Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In: 2022 IEEE Symposium on Security and Privacy (SP). IEEE (2022)
5. Bradley, A.R., Manna, Z.: The calculus of computation: decision procedures with applications to verification. Springer Science & Business Media (2007)
6. Brent, L., Grech, N., Lagouvardos, S., Scholz, B., Smaragdakis, Y.: Ethainter: a smart contract security analyzer for composite vulnerabilities. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (2020)
7. Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B.: Vandal: A scalable security analysis framework for smart contracts. arXiv preprint (2018)
8. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer (2009)
9. CISA: Cybersecurity and Infrastructure Security Agency. `https://www.cisa.gov` (2024)
10. ConsenSys: Mythril. `https://github.com/ConsenSys/mythril` (2022)
11. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer (2008)
12. DefiLlama: Ethereum. `https://defillama.com/chain/Ethereum` (2024)
13. DefiLlama: Hacks. `https://defillama.com/hacks` (2024)
14. Dutertre, B., De Moura, L.: The yices smt solver (2006)
15. Ethereum: Ethereum. `https://ethereum.org/en` (2024)
16. Falke, S., Merz, F., Sinz, C.: Extending the theory of arrays: memset, memcpy, and beyond. In: Verified Software: Theories, Tools, Experiments (VSTTE). Springer (2014)
17. Feist, J., Grieco, G., Groce, A.: 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 (2019)
18. Frank, J., Aschermann, C., Holz, T.: ETHBMC: A bounded model checker for smart contracts. In: Proceedings of the 29th USENIX Conference on Security Symposium (2020)
19. Fröwis, M., Fuchs, A., Böhme, R.: Detecting token systems on ethereum. In: Financial Cryptography and Data Security (FC). Springer (2019)
20. Grech, N., Brent, L., Scholz, B., Smaragdakis, Y.: Gigahorse: thorough, declarative decompilation of smart contracts. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE (2019)
21. Grech, N., Lagouvardos, S., Tsatiris, I., Smaragdakis, Y.: Elipmoc: advanced decompilation of Ethereum smart contracts. Proceedings of the ACM on Programming Languages (2022)

22. Gritti, F., Ruaro, N., McLaughlin, R., Bose, P., Das, D., Grishchenko, I., Kruegel, C., Vigna, G.: Confusum contractum: confused deputy vulnerabilities in ethereum smart contracts. In: 32nd USENIX Security Symposium (USENIX Security 23) (2023)
23. Krupp, J., Rossow, C.: teether: Gnawing at ethereum to automatically exploit smart contracts. In: 27th USENIX Security Symposium (USENIX Security 18) (2018)
24. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: 2016 ACM SIGSAC conference on computer and communications security (2016)
25. Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A.: Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE (2019)
26. Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th annual computer security applications conference (2018)
27. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: Verx: Safety verification of smart contracts. In: 2020 IEEE symposium on security and privacy (SP). IEEE (2020)
28. Ruaro, N., Gritti, F., McLaughlin, R., Grishchenko, I., Kruegel, C., Vigna, G.: Not your Type! Detecting Storage Collision Vulnerabilities in Ethereum Smart Contracts. In: Network and Distributed Systems Security (NDSS) Symposium 2024 (2024)
29. Schneidewind, C., Grishchenko, I., Scherer, M., Maffei, M.: ethor: Practical and provably sound static analysis of ethereum smart contracts. In: 2020 ACM SIGSAC Conference on Computer and Communications Security (2020)
30. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE symposium on security and privacy (SP). IEEE (2016)
31. So, S., Hong, S., Oh, H.: SmarTest: Effectively hunting vulnerable transaction sequences in smart contracts through language Model-Guided symbolic execution. In: 30th USENIX Security Symposium (USENIX Security 21) (2021)
32. So, S., Lee, M., Park, J., Lee, H., Oh, H.: Verismart: A highly precise safety verifier for ethereum smart contracts. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE (2020)
33. Stephens, J., Ferles, K., Mariano, B., Lahiri, S., Dillig, I.: SmartPulse: automated checking of temporal properties in smart contracts. In: 2021 IEEE Symposium on Security and Privacy (SP). IEEE (2021)
34. Tange, O.: Gnu parallel-the command-line power tool. Usenix Mag (2011)
35. Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: Smartcheck: Static analysis of ethereum smart contracts. In: Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain (2018)
36. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: Practical security analysis of smart contracts. In: 2018 ACM SIGSAC Conference on Computer and Communications Security (2018)
37. Zellic: Your Sandwich Is My Lunch: How to Drain MEV Contracts V2. `https://zellic.io/blog/your-sandwich-is-my-lunch-how-to-drain-mev-contracts-v2` (2023)