Insecurity Through Obscurity: Veiled Vulnerabilities in Closed-Source Contracts

Sen Yang*†, Kaihua Qin*†‡§, Aviv Yaish*†, Fan Zhang*†
*Yale University, †IC3, ‡UC Berkeley RDI, §Decentralized Intelligence AG
sen.yang@yale.edu, kaihua@qin.ac, a@yai.sh, f.zhang@yale.edu

***Abstract*—Most blockchains cannot hide the binary code of programs (i.e., smart contracts) running on them. To conceal proprietary business logic and to potentially deter attacks, many smart contracts are closed-source and in many cases exhibit code obfuscation, either intentionally introduced to hide internal logic or unintentionally produced by optimizations. However, we demonstrate that such obfuscation can obscure critical vulnerabilities rather than enhance security, a phenomenon we term *insecurity through obscurity*. To systematically analyze these risks on a large scale, we present SKANF, a novel EVM bytecode analysis tool tailored for closed-source and obfuscated contracts. SKANF combines control-flow deobfuscation, symbolic execution, and concolic execution based on historical transactions to identify and exploit asset management vulnerabilities. Our evaluation on real-world Maximal Extractable Value (MEV) bots reveals that SKANF detects vulnerabilities in 1,030 contracts and successfully generates exploits for 394 of them, with potential losses of $10.6M. Additionally, we uncover 104 real-world MEV bot attacks that collectively resulted in $2.76M in losses.**

1. Introduction

On public blockchains like Ethereum, all smart contracts are deployed as bytecode, publicly visible and accessible to anyone. While transparency is a core feature, it also allows a contract's logic to be examined, reverse-engineered, or copied. As a result, many developers choose not to publish the source code of their smart contracts, either to protect proprietary strategies, reduce the risk of attacks, or simply to raise the barrier of reverse engineering. Furthermore, bytecode *obfuscation* is commonly observed, either intentionally introduced or as a result of optimization [1]–[4].

A prominent example is the Maximal Extractable Value (MEV) bot — an optimized smart contract deployed by MEV searchers to perform arbitrage, sandwich attacks and other on-chain actions to extract MEV [5]. We identify 6,554 closed-source MEV bots as of the time of writing; Six of the top 10, ranked by the number of transaction bundles they sent [6], exhibit some form of obfuscation. There are several plausible reasons along the lines of "security through obscurity," including hiding proprietary MEV extraction strategies and deterring frontrunning attacks, that are deemed "essential" by some practitioners [3].

However, our analysis of the recent "Destroyer Inu" incident (detailed in Sec. A) reveals that obfuscation can also introduce *insecurity*. In this case (detailed in Sec. A), 22 ETH (worth $51,056 at the time) were stolen from a closed-source smart contract equipped with layers of obfuscation.

The attack exploits a *well-known* vulnerable pattern (misuse of tx.origin [7] followed by improper asset management), but existing tools could not detect the full exploit because of control flow obfuscation. E.g., Mythril [8] recognizes the superficial misuse of tx.origin but misses the much more severe asset management vulnerability, only reporting a "low" risk. In this case, obscurity arguably renders the contract *less* secure by hindering analysis, making this a prime instance of insecurity through obscurity [9][1].

This work. Due to the lack of effective analysis tools, little is known about the security of closed-source smart contracts, particularly those handling large volumes of assets and attractive to attackers. In this paper, we aim to develop effective code analysis techniques to address this gap.

We focus on a broad class of security vulnerabilities that we refer to as *asset management vulnerabilities*, issues that allow an adversary to manipulate a smart contract's logic to steal assets (i.e., transferring tokens from the victim to the attacker herself). While our techniques are general, we use MEV bots as a real-world testbed to evaluate the efficacy of our tool and as concrete examples to study the security of closed-source smart contracts in the wild.

Our goals can be summarized as the following three research questions (RQs):

- **RQ1:** How to effectively *deobfuscate* the control flow given smart contract bytecode? Vulnerability detection is only possible if this can be accomplished.
- **RQ2:** Can we detect asset management vulnerabilities in closed-source contracts? Even after deobfuscation, detecting such vulnerabilities can be challenging, given the complexity of MEV bots. Existing tools often timeout when applied to MEV bots, even without obfuscation.
- **RQ3:** How many MEV bot contracts have been exploited in practice, and how much did they lose? Answering this question helps in understanding the severity of asset management vulnerabilities and estimating how much loss could be reduced if SKANF were used by searchers.

We answer these questions using the following methods. First, we analyze notable attacks on closed-source contracts,

1. Compare this with modern cryptographic schemes following Kerckhoffs's principle that a system's security should not be compromised if adversaries uncover its method of operation [10], or, in the words of Shannon: *"the enemy knows the system"* [11].

and distill the vulnerabilities that allowed them and the methods used to execute them. Then, we use our newfound understanding to devise a novel deobfuscator that successfully uncovers crucial parts of closed-source contracts which are opaque to established tools. Making use of our deobfuscator and insights, we build a vulnerability inspector and an automatic attack generator. We combine our deobfuscator, vulnerability inspector and attack generator into a tool we call SKANF. [2]. By applying SKANF to closed-source contracts we collect, we count the number of closed-source contracts that have potential security vulnerabilities and automatically synthesize the exploits against them. We further estimate a lower bound of the amount of funds that can be taken from them using historical Ethereum data.

1.1. Challenges

Numerous tools [8], [12]–[15] are available to identify and exploit smart contract bugs. To illustrate the challenges in answering our research questions, we highlight three core limitations that render existing tools inefficient for our tasks.

Control-flow obfuscation. Languages like Solidity and Vyper rely on jump tables with static jump destinations to manage function dispatch and control flow. However, contracts may employ "indirect" jumps to make the control flow dependent on runtime values to achieve obfuscation.

To the best of our knowledge, no existing tool effectively handles control flow obfuscation. State-of-the-art static analysis tools, such as Gigahorse [15]–[17], classify basic blocks as unreachable when a contract relies on indirect jumps to determine control flow. Decompilation tools such as Dedaub, which claims to "successfully decompile over 99.98% of deployed contracts on the Ethereum blockchain" [12], fail to function effectively when facing control flow obfuscation. For instance, Dedaub can only analyze 4% of code in the top-2 obfuscated MEV bots. Symbolic execution tools like Mythril [8], ETHBMC [13], and Greed [14], [18] raise errors when jump destinations are symbolic.

Lack of fine-grained analysis. Correctly detecting asset management vulnerabilities requires a fine-grained analysis of a smart contract's logic, and simple pattern matching as done by existing tools does not suffice. E.g., JACKAL [14] flags `CALL` instructions with a fixed function selector as safe, overlooking vulnerabilities where function parameters are manipulated (instead of the function selector). As another example, one may flag `tx.origin`-based authentication as insecure (as it appears similar to `tx.origin` phishing [7]), but later steps may rectify the problem.

Performance due to complex logic. In addition to asset management, the contracts we focus on also involve trading, flash loans, swaps on different DEXs, and other operations and interactions. This complexity leads to performance issues in static analysis, as the tools must analyze many control-flow paths and resolve intricate data-flow dependencies, increasing both processing time and memory usage. Moreover, the rapid growth of execution paths and the complexity of constraints make symbolic execution inefficient due to path explosion and the high computational cost of constraint solving. While some tools support concolic execution [8], generating high-quality seed input remains a manual process that does not scale.

2. The name SKANF stands for **S**en, **K**aihua, **A**viv '**n** **F**an.

1.2. Our Methods

Recovering control flow. We propose a method to deobfuscate control flows exploiting the following EVM feature: Unlike traditional instruction sets such as x86-64 or ARM64, where jump instructions can target any executable address, the EVM requires all valid jump destinations to be explicitly marked with a `JUMPDEST` instruction [19]. This allows us to determine the boundaries of basic blocks directly from the bytecode and reconstruct jump tables, even when jump destinations are determined dynamically at runtime. Our tool can instrument the bytecode with a reconstructed jump table to improve the code coverage of subsequent analysis, which is crucial for efficacy.

Detecting asset management vulnerabilities. To not rely on a specific pattern for detecting asset management vulnerabilities, we broadly look for *vulnerable calls* in the bytecode of a smart contract that an adversary can trigger with adversarial parameters. As a naive example, if a smart contract exposes a `CALL` instruction where the adversary can set the target function as well as its input, the adversary can call `WETH.transfer` with the "`to`" parameter set to itself to drain the victim's WETH. Many attacks, however, do not require the adversary to control all inputs.

Our approach draws inspiration from traditional program vulnerability detection, where the combination of *symbolic execution* and *taint analysis* has been used to address similar problems [20]. To detect asset management vulnerability, we first use symbolic execution to identify execution paths leading to a `CALL` statement. Then, we apply taint analysis to examine whether the inputs to the `CALL` statement can be controlled by an adversary. Attacks are possible even if the adversary cannot control the entirety of the inputs, so we carefully enumerate cases where an adversary can manipulate critical parameters, such as the recipient address for `approve` or `transfer`.

Concolic execution based on historical transactions. As mentioned above, symbolic execution by itself does not scale to handle real-world smart contracts of interest. *Concolic execution* solves this issue by combining concrete and symbolic execution to mitigate path explosion [21]. By leveraging concrete *seed inputs* to guide path exploration and simplify constraint solving, concolic execution can reduce the search space, improving efficiency. The "quality" of the seed input is critical, but there are no general algorithms to generate high-quality ones.

We observe, however, that high-quality inputs are readily available on the blockchain: historical transactions are public, and many of them invoke `CALL` instructions through

various paths. This allows us to *automatically* extract high-quality seed inputs from external and internal transactions recorded on the blockchain, significantly improving runtime performance and the efficacy of vulnerability detection.

1.3. Implementation and Evaluation of SKANF

Putting these ideas together, we developed SKANF based on Gigahorse [15] and Greed [14], [18]. At a high level, SKANF takes the bytecode and the historical transactions of a given smart contract as the input and outputs exploits if the smart contract is vulnerable.

The workflow of SKANF consists of three stages. First, it performs static analysis to detect indirect jumps and attempts to deobfuscate the control flow by inserting a branch table that converts all indirect jumps into direct jumps. Second, SKANF parses the contract's historical transactions as concrete seed input for concolic execution and identifies asset management vulnerabilities during the execution. For each identified vulnerability, SKANF synthesizes exploits using constraint-solving, with adversary-controlled parameters — such as the ERC-20 token address and transfer amount — as constraints. The exploit is then validated and reported.

Evaluation results. To evaluate the efficacy of SKANF against real-world smart contracts, we compile a dataset of 6,554 MEV bot contracts and assess SKANF alongside three state-of-the-art tools. For each tool, we limit the execution time to 10 minutes and count the number of vulnerable smart contracts it can detect. Our evaluation shows that SKANF outperforms the other tools: it detects 1,030 vulnerable smart contracts and successfully generates exploits for 394 of them. ***If exploited, the potential losses of these vulnerabilities would exceed 10.6 million USD.*** In comparison, ETHBMC [13], JACKAL [14], and Mythril detected 0, 18, and 89 vulnerable contracts, respectively.

To understand if SKANF can flag attacks that already happened, we conduct an empirical study on MEV bot attacks in the wild. We discovered 104 exploits targeting 37 MEV bot contracts, resulting in a total loss of $2.76M. This confirms that asset management vulnerabilities pose a serious threat to bot security. ***28 of the exploited contracts are flagged by* SKANF*—had* SKANF *been available to developers, a total loss of $2.45M could have been prevented.*** Even after these attacks occurred, only three had been previously reported, probably because no tool was available to automatically scan for such exploits.

Contributions

To summarize, we make the following contributions:

- We propose a novel deobfuscation method to remove control flow obfuscation and enable further analysis.
- We propose a concolic execution approach that uses historical transactions of a contract as concrete inputs to identify potential vulnerabilities in smart contracts.
- Building on these two techniques, we implement SKANF. We plan to release the code publicly.
- We evaluate the efficacy of SKANF against real-world closed-source and obfuscated smart contracts. SKANF identifies 1,030 vulnerable contracts and automatically exploits 394 of them. The potential losses of these vulnerabilities exceed $10.6M.
- We discover 104 real-world attacks targeting MEV bots, which we categorize as MEV phishing attacks. These attacks have resulted in an estimated loss of $2.76M. Of the attacked smart contracts, 28 are flagged by SKANF.

Responsible Disclosure. As our exploits can readily compromise existing smart contracts, we responsibly disclosed them to the affected parties through a dedicated blockchain messaging service (Blockscan Chat by Etherscan [22]), following the same disclosure practice adopted by prior research [23]. Moreover, we adhere to several additional responsible practices. All exploits are tested on a local replica of the Ethereum blockchain, isolated from the public network. Unless the attacks have occurred in the wild and the vulnerabilities are publicly known, we do not disclose the specific addresses of vulnerable contracts in the paper.

2. Preliminaries

2.1. Ethereum and Smart Contracts

Ethereum. Ethereum [19] is the second-most popular blockchain, and the foremost smart contract-enabled blockchain. It relies on a Proof-of-Stake (PoS) mechanism, wherein actors can become part of the set of validators operating the blockchain by staking (i.e., depositing) Ethereum's native token, ETH. Ethereum's mechanism sequentially assigns one validator to propose a block, i.e., a batch of transactions, every 12 seconds. The validator chosen to act as a proposer has a free hand in constructing its block.

MEV supply chain. Today's Ethereum block construction pipeline involves several specialized entities. At one end, we have *searchers* who identify profitable opportunities, e.g., transactions that can be front-run [24], and assemble transaction bundles that exploit them. These bundles are then sent to builders, who specialize in constructing profitable blocks, and who compete in an auction-esque process to have their blocks chosen by upcoming proposers.

Smart contract. Smart contracts are computer programs stored and executed on the blockchain. Once deployed, contracts can be invoked and interacted with using transactions. Ethereum contracts are specified in EVM bytecode, but can be written using high-level languages like Solidity [25] and then compiled for deployment [19]. Sophisticated developers can optimize contracts–e.g., to reduce gas costs–by directly writing bytecode or using low-level languages like Huff [1]. While Ethereum smart contracts are stored on-chain using low-level EVM bytecode, developers can open-source their contracts' high-level code on platforms like Etherscan [26].

The Application Binary Interface (ABI) specifies how to encode function calls and decode responses when interacting with a smart contract on Ethereum. If a contract is open-sourced, its ABI becomes publicly accessible; otherwise, the

ABI is unavailable, making it difficult for the public to know how to interact with the contract.

The EVM is the execution environment for Ethereum smart contracts. The EVM is stack-based, meaning operations are performed by manipulating data on a last-in-first-out (LIFO) stack rather than via registers. Contracts are specified using low-level EVM bytecode composed of opcodes such as `CALL`, `JUMP`, and `JUMPI`. Control flow is handled by `JUMP` and `JUMPI`, which provide unconditional and conditional branching, respectively.

Indirect jump. In the EVM, an *indirect jump* occurs when a jump instruction (`JUMP`/`JUMPI`) is executed and the top stack value (i.e., the jump destination) is not a statically known constant, but rather a runtime-computed value. Obfuscation techniques often rely on indirect jumps to hide control flow, making it difficult to statically determine which path may be executed.

External call. During execution, a smart contract can invoke another contract or externally owned account (EOA) using one of four opcodes: `CALL`, `CALLCODE`, `DELEGATECALL`, and `STATICCALL`. These opcodes enable the caller to pass along Ether (value), execution gas, and input data (calldata), facilitating inter-contract interactions. They differ in how the execution context is propagated, particularly with respect to `msg.sender`, `tx.origin`, and storage access. In this paper, we focus on the `CALL` instruction, which underlies common patterns such as calling an ERC-20 token's `transfer` function; we refer to such invocations as *external calls*.

The EVM provides two global variables to track the source of a call: `tx.origin` and `msg.sender` (accessed via `ORIGIN` and `CALLER` opcodes). The variable `tx.origin` always refers to the original EOA that initiated the transaction, remaining constant across the entire call chain. In contrast, `msg.sender` refers to the immediate caller of the current function, which may be an EOA or another contract, and it changes with each external call. For clarity, we refer to `tx.origin` as *origin* and `msg.sender` as *caller* throughout the rest of this paper.

We refer to the target contract that an external call aims to invoke as the *target address*. The calldata consists of two parts: the first four bytes, named *function selector*, which identify the *target function* to be executed in the target contract; and the remaining bytes, which specify the arguments passed to that function.

Asset management. In this paper, we focus on ERC-20 token assets held by smart contracts. The ERC-20 standard defines two methods for transferring tokens [27]. An account can invoke `transfer` on an ERC-20 token contract to send tokens directly to another account. Alternatively, it can use `approve` to authorize another account to transfer tokens from its balance using `transferFrom`.

2.2. Smart Contract Obfuscation

Several obfuscation techniques have been proposed, including layout obfuscation [28], [29], data flow obfuscation [28], [29], control flow obfuscation [4], [28], and preventive transformations [28]. We focus on tackling control flow obfuscation because it prevents existing detection tools from functioning correctly, and it is commonly present among MEV bots (see [4] for empirical evidence).

We confirmed that the layout and data flow obfuscation do not interfere with existing analysis tools. We use the state-of-the-art obfuscator, BiAn [29] to apply layout and data flow obfuscation to 67 vulnerable smart contracts collected from the Smart Contract Weakness Classification registry [30]. Mythril can still detect vulnerabilities in all of the obfuscated contracts. In the interest of space, we relegate details to Sec. B. Preventive transformations, such as self-destructing after execution to make bytecode unavailable, are also less concerning due to blockchain's transparency and immutability; even a self-destructed smart contract's deployment transaction remains permanently recorded and publicly available for analysis.

2.3. Threat Model

Our adversary model captures the ability of a typical attacker targeting smart contracts. They can create transactions, deploy smart contracts, and interact with deployed contracts but cannot access the private keys of other accounts, breach the integrity of the blockchain, or manipulate the block-building process.

Asset management vulnerability. Our focus is on *asset management vulnerabilities*, i.e., vulnerabilities that would allow an adversary to transfer tokens from the contract's account to the adversary's. In particular, weak adversaries such as our own can do so when a smart contract relies on obfuscation to hide its lack of access control checks. Another possibility is if the smart contract verifies that a token transfer is allowed by checking if the invoking transaction originated from a white-listed address; then, adversaries can employ phishing attacks wherein searchers attempt to interact with a seemingly innocent contract, which then performs the malicious transfer on the adversaries' behalf.

3. The design of SKANF

3.1. Overview

The inputs to SKANF are the bytecode of a given smart contract, and, if they exist, historical transactions can also be provided to improve performance. Fig. 1 provides an overview of our system, which consists of several stages:

1) **Control flow deobfuscation.** First, SKANF identifies and eliminates control flow obfuscation by reconstructing the branch table of an input contract to remove indirect jumps, resulting in cleaner bytecode that allows code analysis tools to operate correctly.
2) **Concolic execution powered by historical transactions.** Smart contracts of interest have complex logic and dependencies, requiring concolic execution to speed up vulnerability discovery. Unlike tools that rely on *manual*

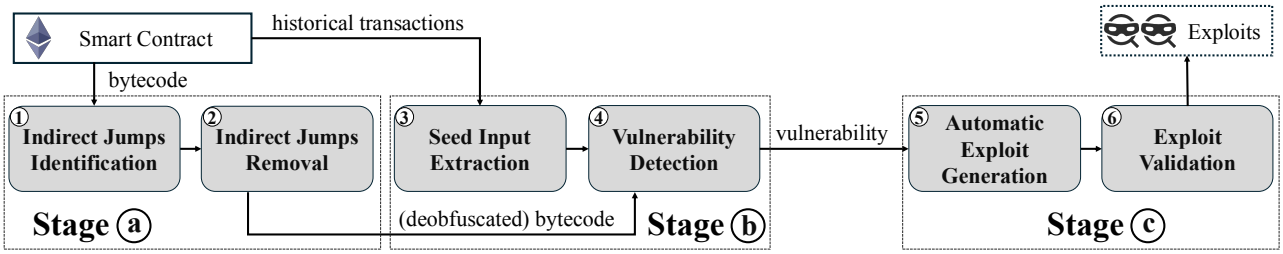

Figure 1. Overview of SKANF. The input consists of the smart contract's bytecode and historical transactions, while the output is the verified exploit. In the first stage (ⓐ), we identify the obfuscated smart contracts (①) and recover their control flow information (②). In the second stage (ⓑ), SKANF parses historical transactions as inputs for concolic execution (③). Then it identifies potential vulnerabilities within the contract and labels adversary-controllable parameters (④). Finally, in the last stage (ⓒ), SKANF generates potential exploits for the identified vulnerabilities (⑤), which are then validated in ⑥.

input [8], SKANF automatically identifies high-quality data from historical transactions to efficiently guide its exploration of the input contract's possibly large number of logic paths. This stage's output is a list of potential vulnerabilities, i.e., insufficiently protected invocations of `CALL` that could be exploited to steal assets.

3) **Exploit generation and validation.** To exploit vulnerabilities identified in the previous stage, SKANF crafts for each one a transaction that triggers the vulnerable `CALL` with specific adversarial inputs. A key challenge in this stage is that an exploit should succeed in executing code *after* the said vulnerable `CALL`. To ensure that exploits are valid, SKANF executes them in a local environment.

3.2. Control Flow Deobfuscation

In SKANF's first stage, our goal is to deobfuscate the target smart contract, thus facilitating further analysis. Technically, we need to identify indirect jumps (i.e., whose destinations are determined at runtime) and convert them to *direct* ones (whose destinations are statically specified) to make the control flow explicit and statically analyzable.

Identifying indirect jumps. We begin by scouring the input bytecode for indirect jumps. A notable difficulty that is faced when "cleaning up" obfuscated control flow logic is that the destination of an indirect jump may be based on dynamic inputs such as `calldata`. To identify such jumps, we extract all jump instructions (`JUMP` and `JUMPI`) and build a control flow graph (CFG). We then analyze whether each jump's destination depends on dynamic inputs (i.e., `calldata`, `value`, `memory`, and `storage`). Indirect jumps are recorded in a set $\mathcal{J} = \{(j_1, d_1), \ldots, (j_n, d_n)\}$, where j_i is the PC of the jump instruction and d_i is the stack variable that determines the jump destination at runtime (e.g., `v58`, `v158` in Fig. 2).

If $\mathcal{J}$ is empty, the contract does not use control flow obfuscation, and SKANF continues to the second stage. Otherwise, we proceed to deobfuscate the indirect jumps.

Deobfuscation. The EVM's specification requires that every valid jump destination must be explicitly marked with a `JUMPDEST` instruction [19]. This property allows us to statically scan the disassembled bytecode, collect all legal jump destinations, and instrument the bytecode with a

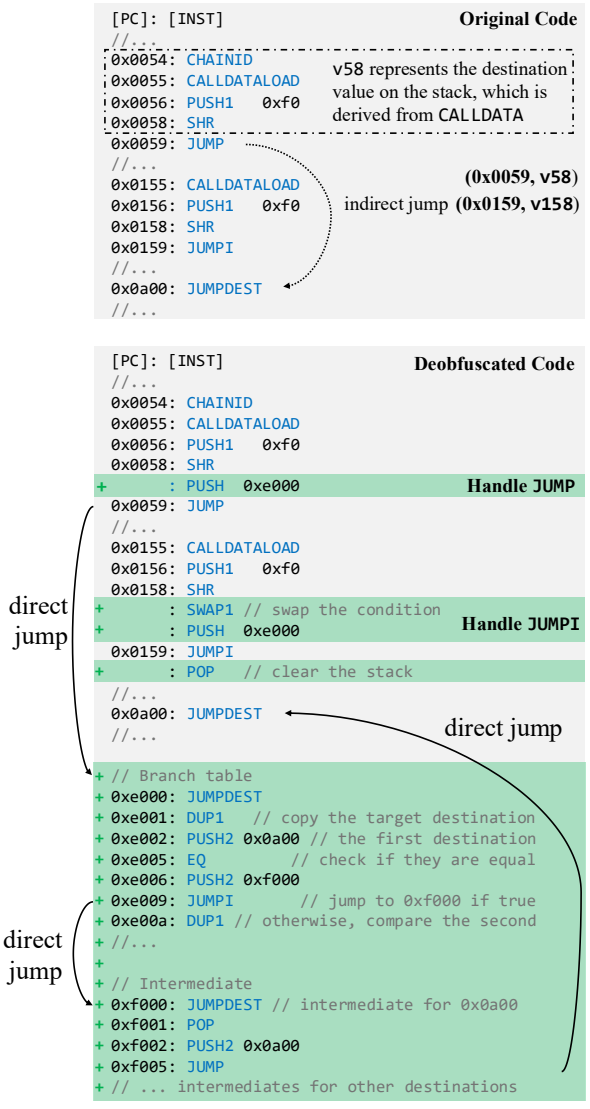

Figure 2. Assembly code for the obfuscated smart contract and how to rewrite the indirect jump as a direct jump by inserting a crafted branch table. The added lines do not carry a PC value, nor do they affect the PC of existing ones, as explained in Sec. 3.2.

branch table that replaces indirect jumps with direct jumps to restore analyzable control flow. This transformation preserves the original control flow while ensuring that jump destinations are constants encoded in the bytecode.

The idea is best illustrated with an example. Suppose the original contract contains an indirect jump to an address stored in variable `v`. Most static analysis tools struggle to handle this since there are too many possible branches. Our idea is to rewrite the code as follows:

```
1  if v == 0x0a00 : jump 0x0a00
2  else if v == 0x0b00 : jump 0x0b00
3  else if v == ...
```

where `0x0a00`, `0x0b00`, and so on, are jump destinations extracted from the bytecode. The number of branches is exactly the number of valid jump destinations.

Replacing every jump instruction with a copy of the above code is inefficient, so we reuse the code by adding a *branch table*. A branch table is a structure commonly used to implement multi-way branching based on a runtime value. We implement the table as a sequence of conditional checks against the variable that the input code uses to hold jump destination addresses (e.g., `v58` in Fig. 2). For each valid destination address, we populate the table with an entry that jumps to the address if the variable is equal to it; otherwise, it proceeds to the next entry. In doing so, we take the contract's "implicit" and opaque control flow where destinations are "tucked away" in a variable, and make it *explicit*: all possible valid destinations are now clearly enumerated in the text. To ensure that the added code does not interfere with existing logic, we cleverly rely on the 24KB (0x6000 bytes) size limit of contract bytecode [31], and insert our tables at PC `0xe000`. Note that any position after `0x6000` could work.

If the original jump instruction is `JUMPI`, special handling is required because it involves two stack values: the jump destination (on top of the stack) and the condition (just below it). To preserve the original semantics when redirecting execution to the branch table, we must maintain the correct stack layout. Before inserting the new jump destination, we insert a `SWAP1` to move the condition to the top of the stack. This ensures that when the rewritten `JUMPI` executes, it evaluates the condition correctly while using the branch table entry as the destination. We also insert a `POP` immediately after `JUMPI` to remove the unused destination value from the stack in case the condition is false and the jump does not occur, thereby avoiding stack imbalance.

To appreciate the subtleties in replacing indirect jumps in a seamless fashion, we need to dive into the technicalities involved. For illustration, we refer at each step to the corresponding lines in a corresponding example given in Fig. 2. For every indirect jump instruction at PC $j \in \mathcal{J}$, we "redirect" the jump destination to our table by rewriting the bytecode to push `0xe000` onto the stack. Each table entry is implemented in a manner which accounts for the stack-based semantics of the EVM: First, the runtime value is duplicated (see PC `0xe001`), and a known destination is pushed (e.g., `0x0a00` in Fig. 2), allowing us to compare the two using the `EQ` instruction (PC `0xe005`). If both are equal, the program jumps to an intermediate "gadget" which cleans up the stack and performs the final jump (see PC `0xf000`–`0xf005`). This intermediate step is necessary because the comparison only consumes the duplicated value, leaving the original runtime value on the stack. If the comparison fails, execution continues to the next table entry (e.g., at PC `0xe00a`), which repeats the process for another possible destination. This continues until a matching entry is found.

In order to prevent bytecode offset shifts, we do not directly modify the raw bytecode when instructions need to be injected in PC values lower than `0xe000` (e.g., consider the `PUSH` instruction between PC values `0x0058` and `0x0059`). Instead, we apply such changes only when the analysis of the program operates on a CFG rather than concrete bytecode offsets. As a result, injected instructions do not carry PC values. In addition, the branch table includes all possible jump destinations within the contract. Although some destinations may not be used at runtime, the evaluation in Sec. 4.2 shows that this does not introduce significant overhead.

After this transformation, all jump targets in the contract are encoded as immediate values, and the control flow becomes fully explicit and statically analyzable.

3.3. Vulnerability Detection

In this stage, we identify potential asset management vulnerabilities in the smart contract — specifically, opportunities for an adversary to steal assets. We define a *vulnerability oracle* that flags any *vulnerable `CALL` instruction* as one that is (1) reachable from a public entry point and (2) accepts inputs that are either adversary-controlled or fixed but risky (e.g., a known ERC-20 token address).

Note that a vulnerability does not guarantee a successful exploit. Akin to traditional vulnerabilities like buffer overflows, where a flaw alone does not guarantee arbitrary code execution, as additional conditions may be required. In this paper, we define an exploit as a transaction that transfers assets from the victim contract to the adversary's account.

We detect vulnerabilities in this stage and generate exploits in the next. To this end, we first extract concrete seed inputs from historical transactions to perform concolic execution, thereby speeding up the exploration. Additionally, we perform taint analysis during the path exploration leading to a vulnerable `CALL` instruction. The taint analysis results are then used to determine whether the reachable `CALL` instruction is one of the vulnerabilities of interest.

Extracting seed input from historical transactions. A key challenge in analyzing real-world smart contracts lies in their complex logic, especially when involving multiple `CALL` instructions to interact with other smart contracts, such as DEXs, NFTs, and lending protocols. To efficiently explore such code, we adopt *concolic execution* [21], a hybrid analysis technique that uses concrete *seed inputs* to guide path exploration and simplify constraint solving, thereby reducing the search space and improving efficiency.

In our setting, an ideal seed input is a set of parameters that trigger an external call to transfer assets out of the smart contract. This prioritizes the exploration of "high-risk" paths involving asset transfers. Crafting seed inputs for closed-source smart contracts is hard because their ABI (Application Binary Interface) is unknown. Randomly generated seed inputs lack insight into the code's logic, and thus may have a low probability of reaching vulnerable paths. Fortunately, high-quality seed inputs are already readily available on-chain in historical transactions. Given a target contract, we retrieve the associated transactions that are executed successfully and extract the relevant inputs, which serve as concrete seeds for concolic execution.

Focusing on asset-transferring paths may introduce false negatives, since not all potentially vulnerable paths are explored. However, this trade-off is acceptable — discovering even a single asset management vulnerability suffices to demonstrate that the contract's assets are at risk. As shown in Sec. 4.3, this approach allows us to uncover more vulnerabilities than existing tools.

Specifically, we collect historical transactions of a given contract C and extract external calls that involve ERC-20 transfers by inspecting standardized *transfer event logs* [27]. For relevant transactions, we extract all associated external calls, keeping those where the target is C. We further filter out calls with empty calldata (e.g., plain ETH transfers), which typically do not involve asset management logic.

Concolic execution. From the parsed historical transactions, we extract all external calls to the target contract, including the origin, caller, calldata, and value. These, along with the block number which reflects the correct blockchain state, are used as concrete inputs for concolic execution.

While concolic execution can help identify reachable `CALL` instructions, further analysis is required to determine whether the inputs to these instructions are vulnerable. During concolic execution, we perform *on-the-fly taint propagation* with *byte-level* granularity to trace how tainted calldata bytes, representing adversarial inputs, propagate to critical input to `CALL` instructions. Specifically, we decompose the input to the `CALL` instruction into different *parameters*, including the target address, the function selector, and individual function arguments.

Taint propagation is initiated when calldata is accessed (e.g., via `CALLDATALOAD` or `CALLDATACOPY`) and proceeds dynamically through the EVM's stack, memory, and storage. Our policy is conservative: taint is propagated through all arithmetic, logical, and data copy instructions. For each such instruction, if any operand is tainted, the resulting value is also marked as tainted. We treat external calls as taint sinks and examine if the target address, function selector, or any arguments are influenced by tainted data.

When concolic execution encounters a `CALL`, it performs two key operations. First, we inspect the instruction's parameters to determine whether it represents a potential asset management vulnerability. Then, based on taint analysis, we selectively replace calldata bytes that influence the `CALL` parameters with symbolic variables, while preserving concrete values for all other bytes. In stage ⓒ, we generate exploits by restarting concolic execution with these modified calldata bytes, which allows us to construct an exploit based on the values specified by the adversary.

Identifying asset management vulnerability. After the taint analysis, SKANF identifies `CALL` parameters that can be determined by the adversary via their input. We refer to such parameters as *adversary-controllable*. However, even when some parameters are fixed (thus cannot be changed by an adversary), the `CALL` instruction may still be vulnerable. For example, the function selector (0xa9059cbb, corresponding to `transfer`) is fixed, yet the adversary still controls the target address of the call and can construct the calldata to transfer all WETH to their account. Therefore, we also analyze fixed parameters to assess whether they pose a risk.

In the context of asset management vulnerabilities, a fixed target address may still be problematic if it corresponds to an ERC-20 token contract (e.g., WETH). Similarly, a fixed function selector may be risky if it maps to functions like `transfer` or `approve`. We refer to such parameters as *risky*, since they can still lead to asset loss even without full adversarial control.

Once we determine that both the target address and function selector of the vulnerable `CALL` instruction are either adversary-controllable or risky, we examine if bytes 5 to 36 of its calldata, which are interpreted as the first argument if the function is *transfer* or *approve*, can be arbitrarily set by the adversary, noting that if the function selector is controllable, the adversary can choose *transfer* or *approve*. If this is the case, we classify the corresponding `CALL` instruction as a potential asset management vulnerability. Additionally, we record whether the second argument (i.e., the transfer amount) is fixed or under adversarial control.

Fallback mode. Given that not all contracts are highly active and may lack sufficient historical transactions, we incorporate symbolic execution as a fallback when concolic execution fails to identify potential vulnerabilities. In fallback mode, SKANF conducts a broader analysis to identify all potentially vulnerable `CALL` instructions and assess their reachability. Specifically, we first enumerate all `CALL` instructions in the contract. For each instruction, we use the call graph to identify functions that are reachable from public entry points, and the CFG to determine whether those functions can eventually reach the `CALL` instruction. If such a path is found, we initiate symbolic execution from the corresponding public function using a symbolic byte string as calldata.

As symbolic execution progresses, we keep track of symbolic expressions for all values in the EVM stack, memory, and storage. These symbolic values are updated with each instruction to reflect how they depend on input variables. When the execution encounters a conditional branch (i.e., `JUMPI`), we fork the execution into two paths — one for each possible outcome — and add the corresponding condition to the path constraints. To manage complexity, we prune infeasible paths. This can happen in two cases: either the accumulated constraints become unsatisfiable (i.e., no input could result in that path), or the CFG shows that

```
{"caller": "0xdead...beef",
 "origin":"0xdead...beef",
 "blockNumber":20000000,
 "callPC":"0xac5", // PC of the vulnerable CALL
 "calldata":"12345678SS...", // SS represents the
     symbolic data
 "targetAddress":"*", // target address is adversary-
     controllable
 "functionSelector":"0xa9059cbb", // fixed but risky
 "destination":"*",
 "amount":"*"}
```

Figure 3. Example output of SKANF for a vulnerability.

the path cannot reach the target CALL instruction. Handling access control checks — like those involving *msg.sender* or *tx.origin* — can be particularly tricky. To improve the chance of passing these checks, we employ two configurations. First, we set both the caller and the origin to a predefined adversary address. If that fails (for instance, if the contract requires a specific *tx.origin*), we try again with the origin set to the one recorded from a previously observed transaction during concolic execution.

Once symbolic execution discovers a feasible path to a CALL instruction, we analyze it using the same criteria as in concolic execution — checking whether any of its parameters are adversary-controllable or otherwise risky — to determine whether it constitutes a vulnerability.

Preliminary validation. For each vulnerability, we perform a preliminary validation step before attempting to exploit it, as concrete exploit generation requires additional constraints to be satisfied. This validation is worthwhile because symbolic execution may incorrectly identify vulnerabilities even when these cannot be exploited, due to imprecise modeling of external calls and incomplete representations of storage or execution context [14]. For instance, a CALL instruction may appear reachable in symbolic analysis but cannot be triggered under any concrete execution.

Our preliminary validation is done using a local blockchain instance to simulate a concrete execution environment. We supply concrete calldata, solved from any symbolic constraints, as well as caller and origin addresses to verify whether the CALL instruction is actually executed. Unlike exploit generation, we do not require the transaction to succeed or result in asset transfer; we only check whether the vulnerable CALL is triggered. This is sufficient to confirm the vulnerability's existence.

At the end of this stage, SKANF produces a "vulnerability report" (an example is given in Fig. 3), recording the PC of the CALL instruction, the risky or adversary-controllable parameters, and other contextual information such as the caller, origin, block height, and the concrete calldata used. This is then forwarded to the next stage for exploit generation.

3.4. Exploit Generation and Validation

At the final stage, SKANF attempts to synthesize and validate a concrete exploit for each identified vulnerability.

Automatic exploit generation. Given a vulnerability identified in the previous stage, SKANF attempts to construct a transaction that successfully exploits it. The transaction is built at the specific block height where the vulnerability is observed. The "to" address is set to the victim contract. The "gas" and "gas price" fields are configured to exceed the minimum requirements to ensure execution and inclusion in a block. For the "value" field, if any value is acceptable, we set it to zero; otherwise, we use the required amount. The "from" address is set to either the adversary's address (if no constraints are specified) or to a specific origin address if one was recorded during vulnerability detection. The latter case is typically caused by tx.origin-based authorization, which can be bypassed through phishing attacks, as is the case with the Destroyer Inu attack. Given how easy such phishing attacks are in practice (see Sec. 4.4 for real-world evidence), we conservatively assume that the attacker can always set the "from" value to the required one.

The key step in exploit generation is constructing the correct calldata to ensure the transaction executes successfully. Recall from stage ⓑ that for each vulnerability, we record which CALL parameters — the target address, function selector, and arguments — are adversary-controllable. If the target address is controllable, we constrain it to the address of an ERC-20 token held by the victim contract with a non-zero balance. By repeating the attack with different addresses, we can systematically target each token. Similarly, if the function selector and arguments are controllable, we constrain them to values that mimic a realistic attack: we set the selector to `transfer`, the recipient (`to`) to the adversary's address. For the transfer amount, we query the token balance of the victim contract at the specified block height and set this balance as the transfer amount, aiming to steal the maximum possible amount as an adversary would.

According to the EVM specification [19], a transaction is considered successful only if execution halts at a valid stop instruction such as STOP or RETURN. To ensure this condition is met, we resume concolic execution from the vulnerable CALL and symbolically execute forward until a stop instruction is reached. This introduces additional path constraints, which are solved to adjust the synthesized calldata and complete exploit generation.

Exploit validation. Finally, SKANF validates the exploit in a local execution environment to ensure that it executes successfully and produces the expected behavior.

To do this, we simulate the blockchain environment at the block height specified in the exploit. This includes loading the contract code, account balances, and persistent storage state. We then execute the synthesized transaction.

Finally, we check whether the transaction succeeded, and further assess exploit success by inspecting the emitted event logs: for ERC-20 exploits, we verify the presence of a `Transfer` or `Approval` event consistent with the expected asset movement, following the ERC-20 standard [27]. If both checks pass, SKANF confirms the exploit as valid.

4. Evaluation

4.1. Experimental Setup

Implementation. We implement SKANF on top of Gigahorse [15], [16] and Greed [14], [18], comprising 2.2K lines of Python code and 210 lines of Soufflé code. In more technical detail, we develop a plugin for Gigahorse to analyze bytecode and recover the control flow of obfuscated contracts. We then use Gigahorse to lift the (deobfuscated) bytecode to its register-based Intermediate Representation (IR). Vulnerability detection and exploit generation are implemented based on Greed, which attempts to automatically synthesize exploits through constraint solving given the following inputs: the register-based IR of a contract along with the CFG and call graph generated by Gigahorse. For a given contract, we fetch its recent transactions using Etherscan's API [26]. To validate exploits, we simulate their execution using the Python implementation of the EVM [32], [33].

Dataset. We use real-world MEV bots as the target to evaluate the efficacy and performance of SKANF. We first compile a dataset of the addresses of a total of 6,554 MEV bots from multiple sources, including existing works [34]–[36], public dashboards [6], [37], and results from MEV analysis tools such as MEV inspect [38].

Testbed. We perform our experiments on Ubuntu 22.04, with an Intel Xeon Platinum 8380 (80 cores), 128GB RAM and 8TB SSD.

4.2. RQ1: Deobfuscation Effectiveness

The effectiveness of SKANF in addressing control-flow obfuscation is measured using the *code coverage* metric. Specifically, a tool's coverage of a given smart contract is the fraction of reachable code blocks. Low code coverage indicates that a significant portion of the smart contract cannot be analyzed by the tool.

We compare Gigahorse's coverage with and without deobfuscation for all smart contracts in our dataset. To zoom in on real-world obfuscation practices used by major actors, we repeat the comparison on the top 10 and top 100 most active MEV bots since the Merge (ranked by the number of bundles they sent), as reported by the "libMEV" online dashboard [6]. We choose these bots because sophisticated MEV searchers who remain active are strongly motivated to keep their business logic private, making them more likely to adopt control flow obfuscation in their code.

Fig. 4 presents the cumulative distribution function (CDF) of the code coverage for the top 10, top 100, and all smart contracts in the dataset, respectively. Dashed lines represent Gigahorse, and solid lines represent SKANF. We observe that among the top 10 MEV bots, six are highly obfuscated (with code coverage below 10%), and SKANF successfully increases the code coverage to 100% for five of them. Note that SKANF fails on one bot contract because it relies on the preliminary output from Gigahorse for bytecode analysis. Although Gigahorse generally produces

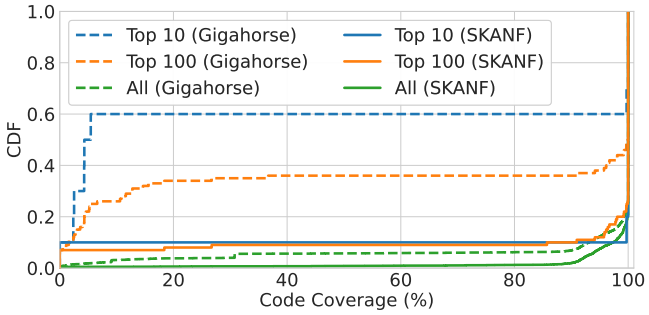

Figure 4. Cumulative Distribution Function (CDF) of the code coverage of smart contracts in our dataset for both Gigahorse and our tool, SKANF. As can be observed, SKANF's CDF is concentrated near 100%, while that of Gigahorse is spread out and is consistently higher (i.e., *worse*) than SKANF, indicating that SKANF outperforms Gigahorse.

preliminary output even for highly obfuscated contracts, in this particular case, it fails to do so, which prevents SKANF from proceeding. Comparing the code coverage of three MEV bot groups (top 10, top 100, and all bots), we find that control flow obfuscation is more prevalent among the most active MEV bots, underscoring our approach's importance.

For all bots, we observe that for 5% of the smart contracts, the original code coverage is below 50%. In contrast, fewer than 0.4% of smart contracts have deobfuscated code coverage below 50%. This comparison highlights the effectiveness of SKANF in mitigating obfuscation. For the small fraction (0.4%) of contracts where deobfuscation does not significantly improve code coverage, our manual analysis reveals two reasons: either Gigahorse fails to produce any preliminary output before deobfuscation, or some blocks within the contracts appear to be unreachable, rather than hidden by control flow obfuscation.

Answer to RQ-1

SKANF effectively addresses control flow obfuscation in smart contracts, for 90% of contracts with an initial coverage below 50%, we can successfully increase their code coverage.

4.3. RQ2: Vulnerability Detection Effectiveness

We now evaluate the effectiveness of SKANF in detecting vulnerabilities in smart contracts, as measured by the number of identified vulnerabilities. To benchmark our tool, we conduct a comparative evaluation against three state-of-the-art bytecode-based symbolic execution tools: Mythril (commit:9e9ee39) [8], ETHBMC (commit: e887f33) [13], and JACKAL (commit: 3993e5c) [14]. In the comparative evaluation, we run each tool for a maximum of 10 minutes per contract and impose a memory limit of 20 GB.

4.3.1. Vulnerability detection by SKANF. To illustrate how concolic execution enhances the effectiveness of vulnerability detection, we conduct an ablation study by running

SKANF in full symbolic execution mode without any concrete inputs, which we refer to as the *baseline mode*. In our evaluation, SKANF detects 721 vulnerable smart contracts in the baseline mode. When executed in concolic mode, the number of detected vulnerable smart contracts increases to 1,030. 54% of the unique vulnerable contracts identified by concolic execution timed out in the baseline mode. In the remaining cases where symbolic execution fails, we find that the corresponding path constraints are incorrectly deemed unsatisfiable. In contrast, concolic execution, guided by concrete inputs, can avoid this limitation and successfully explore the vulnerable paths. This highlights a significant improvement in vulnerability detection achieved through concolic execution, which simplifies constraint solving and prioritizes exploration of the vulnerable `CALL` instruction within the limited time.

Comparison with state-of-the-art tools. **Mythril** alerts when it detects a publicly reachable `CALL` instruction where the adversary can control the target address of the `CALL`. In the evaluation, Mythril flags 574 potentially vulnerable smart contracts. We validate these vulnerabilities identified by Mythril by executing the exploit transaction it generates in a local blockchain instance, similar to our preliminary validation for SKANF. Specifically, we check whether the vulnerable `CALL` instruction is triggered by the adversary and whether the adversary can control other parameters to steal the contract's assets.

In the end, we only reach the `CALL` instructions in 89 of the flagged contracts (SKANF identifies 1,030). Among these, 68 are also covered by SKANF. For the rest not identified by SKANF, we manually inspect them and find that identifying vulnerabilities requires an accurate CFG. However, we construct the CFG using Gigahorse, which fails on these contracts (e.g., due to out-of-memory errors during bytecode lifting).

ETHBMC also raises an alert when it detects a `CALL` where the adversary controls the target address and transfer amount [13], [14]. Note that this captures only a subset of asset management vulnerabilities, as an adversary can still steal assets even if the target and amount are fixed, as long as they control the recipient.

However, in our evaluation, ETHBMC did not detect *any* vulnerabilities. Our further analysis shows that low code coverage and timeouts are the two main reasons. For about 36% of the contracts, ETHBMC achieved less than 25% code coverage during execution. Additionally, for approximately 27% of the contracts, ETHBMC encountered timeout errors while solving constraints. These findings indicate limitations in ETHBMC's ability to analyze obfuscated smart contracts and identify asset management vulnerabilities in contracts with complex logic.

JACKAL detects reachable `CALL` instructions where the adversary can control the target address and the function selector of the `CALL`. For the same reason as with ETHBMC, this pattern represents a subset of asset management vulnerabilities. In our evaluation, JACKAL detects 18 vulnerable contracts, all of which are included in the set of 1,030 vulnerable contracts we identified.

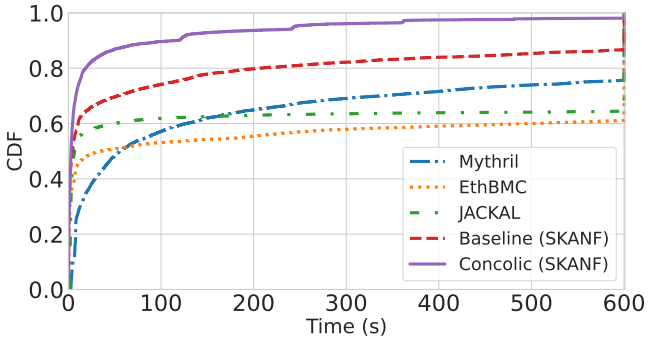

Figure 5. CDF of the time taken by SKANF, Mythril, ETHBMC, and JACKAL to analyze the smart contract.

4.3.2. Runtime of SKANF. Beyond its effectiveness in identifying vulnerabilities, we further evaluate SKANF's runtime. As shown in Fig. 5, we observe that SKANF achieves shorter analysis time in the concolic mode than in the baseline mode — it can analyze about 90% of the dataset's contracts within 100 seconds. This demonstrates the effectiveness of concolic execution. For other tools, our runtime advantage may result from multiple factors. For example, these tools were originally designed with different goals, and the additional analysis overhead from checking issues such as unprotected `SELFDESTRUCT` may affect their performance.

4.3.3. Exploit generation and loss estimation. For each contract with identified vulnerabilities, we generate and validate exploits, and compute the loss for realized exploits.

In total, we successfully exploited 394 out of 1,030 vulnerable contracts. We validated all 394 exploits locally. These results show that SKANF already significantly outperforms existing tools; for instance, JACKAL generates only 31 exploits from 529 potentially vulnerable smart contracts.

We manually inspected 100 randomly selected contracts for which exploit generation failed. In 65 of them, the failure was due to additional `CALL` instructions that required more precise parameter constraints. We leave this for future work. In the remaining 35 cases, the failure is likely due to a bug in Greed, which introduces incorrect constraints on the parameters of `CALL` instructions.

Among the 394 exploits, 141 require attackers to control `tx.origin` (e.g., through phishing attacks), while the rest have no such requirement. Additionally, 162 of the exploits target contracts with control flow obfuscation (i.e., Gigahorse measures their code coverage as below 50%).

We further calculate a lower bound on the potential loss if the synthesized exploits were carried out. Since the balance of the victim contract changes over time, we use two reference points: the balances at the time of writing (June 5th, 2025), and the historical high balances (including vulnerable historical versions; see Sec. C), which represent a worst-case attack. For simplicity of comparison, we use the token price on June 5th, 2025, to convert the loss to USD. We focus on seven major ERC-20 tokens: WETH, WBTC, USDC, USDT, DAI, UNI, and LINK, so our results

are a lower bound. If a contract's vulnerability is limited to specific tokens, we only consider the affected tokens.

The results are shown in Tab. 1. The "current" and "maximal" columns refer to the current balances and historical high balances, respectively. As shown in the table, the total estimated economic loss caused by the synthesized exploits amounts to approximately 10.6 million USD. Among them, WETH accounts for the largest portion, with an estimated loss of approximately 7.6 million USD. The significant loss suggests that these profitable MEV bots are vulnerable and exposed to substantial financial risks.

TABLE 1. A CONSERVATIVE ESTIMATE OF THE POTENTIAL LOSS IF THE SYNTHESIZED EXPLOITS WERE CARRIED OUT.

Token	Current		Maximal	
	Amount	Loss ($)	Amount	Loss ($)
WETH	114.4	299,626.9	2,895.1	7,585,162.0
WBTC	0.2	20209.0	20.0	2,100,000
USDC	55,730.7	55,730.7	296,716.3	296,716.3
USDT	9,441.1	9,441.1	282,295.5	282,295.5
DAI	50,386.6	50,386.6	185,609.3	185,609.3
UNI	16,131.7	101,952.6	16,131.7	101,952.6
LINK	2,076.4	28,799.4	3,061.2	42,458.8
Total	—	**566,146.3**	—	**10,594,194.5**

Answer to **RQ-2**

SKANF identifies 1,030 potentially vulnerable MEV bot contracts and successfully generates effective exploits for 394 of them, with a potential loss of $10.6M. Its vulnerability detection outperforms other tools.

In closing, we would like to clarify that the limitations or bugs in Greed and Gigahorse should be considered separately from the evaluation of SKANF. SKANF itself can be implemented on top of any existing tool with some additional engineering effort. The goal of this evaluation is to show that existing tools fail to address control flow obfuscation and suffer from performance issues on complex contracts. More importantly, they miss a significant number of asset management vulnerabilities. When integrated with SKANF, these tools can identify more potential vulnerabilities that would otherwise be missed.

4.4. Attacks in the Wild

Previous evaluation confirms that SKANF can identify *new* vulnerabilities not reported or exploited in practice. In this section, we collected data on *real-world attacks* and 1) we evaluate whether SKANF can discover vulnerabilities in attacked smart contracts; 2) we quantify the potential loss that could have been saved by SKANF; 3) we gain an understanding of the prevalence of real-world exploits against closed-source smart contracts.

Detecting attacks. We start with 65,758,934 Ethereum transactions sent to 6,554 MEV bot contracts from January 2021 to May 2025, from which we aim to identify attack transactions. We note that our goal is not to conduct a comprehensive measurement study (in particular, we do not aim to be exhaustive). Thus, we employ a relatively simple method supplemented by manual verification.

First, based on how MEV bots work, we observe that external calls from a strange smart contract that an MEV bot had no prior interaction with are likely attack attempts. Thus, we narrow down to transactions that involve strange callers, followed by ERC-20 asset transfer from the bot's account (or approval to transfer), a necessary action to steal assets. These two rules result in 164,404 attack candidates, which are still too numerous to be manually verified. We further narrow down by looking for a specific malicious pattern: callbacks from an unintended caller. E.g., `uniswapV3SwapCallback` is intended to be called by Uniswap, and calls by others are likely an attack attempt. Finally, this brings the number down to 104.

We manually verify all of them by checking whether the assets were transferred to the attacker's accounts, following a strict rule to determine if an account belongs to the adversary: we consider an account to belong to adversary if it has no prior interaction with the victim contracts, and almost all of its transactions are related to the attack, and created malicious contracts explicitly for the attack. All results have been cross-validated by multiple authors to ensure accuracy.

We confirmed that all of the 104 detected transactions were adversarial, involving 51 malicious contracts and 37 victim MEV bot contracts. Our method is tailored to specified patterns and thus has no false positives, but may miss certain attacks. Without ground truth, one cannot evaluate false negatives. Our dataset, nonetheless, is the largest real-world dataset of phishing-based exploits against asset management vulnerabilities to the best of our knowledge.

SKANF's ability to re-discover attacks. We apply SKANF to victim contracts and find that it successfully identifies 28 as vulnerable. If the searchers had used SKANF, most of the attacks could have been prevented.

For the nine vulnerable smart contracts not detected by SKANF, our further investigation shows that four are due to failures in Gigahorse — similar to the issue described in Sec. 4.3 — and the remaining five are caused by Greed, which incorrectly classifies vulnerable paths as infeasible. These findings suggest that, if Gigahorse and Greed functioned as intended, SKANF could potentially detect all of these vulnerabilities.

The earliest observed attack occurred in July 2021, resulting in a loss of 30 ETH ($76K), while the most recent attack took place in April 2025. The single largest attack caused a loss of 250 ETH ($636K). The total losses from these incidents amount to about $2.76M. The losses are calculated using the token price at the time of each attack. We notice that only three of these attacks have been previously reported. Our analysis indicates that $2.45M of the total loss could have been mitigated if SKANF had been used by searchers before these incidents. This highlights that vulnerabilities in asset management smart contracts remain an ongoing security concern and often go unnoticed by developers.

Answer to RQ-3

We identify 104 MEV phishing attacks in the wild, causing about $2.76M in losses to searchers. Only three of them have been covered in prior analyses and reports. SKANF detects asset management vulnerabilities in most of the exploited contracts and could have prevented losses totaling $2.45M.

These attacks all involve the attacker bypassing tx.origin checks, supporting the assumption made in Sec. 3.3.

5. New Attack Pattern: MEV Phishing Attack

In this section, we delve into the attacks identified in Sec. 4.4 to understand how they occurred. Similar to the "Destroyer Inu" Attack (Sec. A), in all these incidents, the adversary lures the victim into interacting with their malicious contract by creating a special MEV opportunity. Once searchers try to capture this MEV opportunity via their MEV bot, they fall into the phishing trap, as the malicious contract is now embedded in their MEV supply chain.

We call these *MEV phishing attacks*, and further categorize them into two types based on the attack vectors: *token-based* and *pool-based* MEV phishing attacks, comprising 101 and 3 incidents, respectively. A typical example of the first is the "Destroyer Inu" Attack discussed in Sec. A. For pool-based attacks, a malicious pool (e.g., 0x0EF...3Fa) can exploit searchers when a swap occurs within it.

Beyond malicious tokens and pools, the adversary can also launch attacks from other components in the MEV supply chain. To investigate the existence of such attacks, we modified existing heuristics to identify components that, in theory, should not invoke the bot contract but do so in practice. This led us to identify the refund address — a component theoretically supposed to only receive refunds from searchers in exchange for providing MEV opportunities [39], and that should not interact with the bot contract. We refer to this as *refund-based* attacks.

To provide a better understanding of how real-world attackers exploit asset management vulnerabilities through sophisticated MEV phishing strategies, we present a case study of refund-based attacks. This novel attack vector is exemplified by a real-world transaction 0x263...df0.

Refund-based MEV phishing attack. Most MEV phishing attacks rely on malicious tokens as the attack vectors. These tokens store specially crafted calldata associated with specific target accounts. When a transaction originates from that targeted account, the token automatically triggers the attack against the MEV bot contract using the prepared calldata. However, this method has a key limitation: after repeated targeting, searchers may detect the issue and adopt countermeasures, such as validating that tokens strictly follow the ERC-20 standard. A recent variant departs from using malicious tokens and instead exploits the mechanics of MEV refund services. The attack flow is illustrated in Fig. 6.

The attacker first deploys a malicious contract to serve as the refund address (①) and then creates an MEV opportunity, involving a swap, which is submitted to the BackRunMe service [39] (②). The tokens used in the swap follow the ERC-20 standard, avoiding detection mechanisms targeting non-compliant tokens.

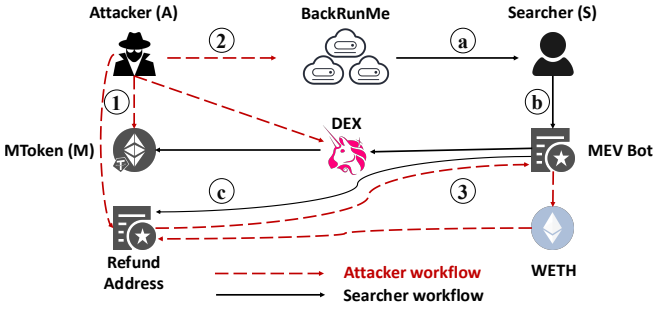

Figure 6. The workflow of refund-based MEV phishing attack.

A searcher receives the opportunity information via BackRunMe (ⓐ) and invokes its MEV bot to extract it (ⓑ). As part of the BackRunMe protocol, the searcher must send a refund to the attacker-specified address (ⓒ). Upon receiving the refund, the malicious contract is triggered and calls the MEV bot contract using crafted calldata (③). Since the transaction originates from the searcher, it bypasses origin-based access control and enables unauthorized token transfers to the attacker.

6. Related Works

Static analysis. Static analysis is a method of examining code without executing it to detect vulnerabilities in software. Static analysis tools for smart contracts, such as Vandal [40], Securify [41], MadMax [42], Slither [43], and Ethainter [44], typically aim for completeness but often produce false positives, requiring manual effort to verify warnings. Moreover, a well-known challenge for static analysis is control flow obfuscation [45], which often prevents them from constructing accurate control flow graphs, thus reducing their effectiveness. Specifically, potentially vulnerable logic may be treated as unreachable, leading to false negatives. The deobfuscation technique in SKANF complements these tools by improving accuracy and efficiency.

Symbolic execution. Symbolic execution explores program execution paths by treating inputs as symbolic variables rather than concrete values. Symbolic execution tools for smart contracts, such as Oyente [46], Mythril [8], teEther [47], Manticore [48], ETHBMC [13], and Greed [14], [18], systematically explore possible paths to identify issues like reentrancy and integer overflows. However, like static analysis, symbolic execution struggles with symbolic jump destinations caused by control flow obfuscation. SKANF extends Greed with a de-obfuscation module, enabling it to handle control flow obfuscation and detect vulnerabilities in obfuscated contracts. Although JACKAL [14] and CRUSH [18] are also built on Greed, they target different types of vulnerabilities — confused deputy

and storage collision, respectively. In contrast, SKANF targets asset management vulnerabilities in closed-source contracts and improves performance by guiding concolic execution with historical transactions.

Fuzzing. Fuzzing is a dynamic testing technique that mutates inputs to uncover software vulnerabilities. For smart contracts, fuzzing applies random and unexpected inputs, such as crafted calldata and reentrant calls, to test contract behavior. Tools developed for EVM smart contracts include ContractFuzzer [49], Echidna [50], Smartian [51], Confuzzius [52], ItyFuzz [53], and MAU [54]. However, while state-of-the-art fuzzing tools have improved the mutation efficiency, they remain insufficient for our problem. The vulnerability logic we focus on is often triggered only by specifically crafted calldata, which differs across contracts. As a result, fuzzers may treat these inputs as arbitrary byte mutations, making the exploration process highly inefficient.

Attacks against searchers. When searchers send transactions to the public mempool, their transactions and execution logic become visible to adversaries, enabling real-time imitation and front-running. Qin et al. [23] generalize this attack through Ape, a framework that uses dynamic program analysis to automatically generate adversarial contracts. Interestingly, the same technique can defend against malicious searchers. Zhang et al. propose STING, a defense mechanism that identifies attacking transactions and instantly synthesizes counterattack smart contracts [55].

However, as private mempools become more common [34], [56], previous solutions lose effectiveness because searchers' pending transactions are invisible when sent to a private mempool. Various works attempt to circumvent these limitations. The GhostTX attack by Yaish et al. [57] tricks searchers into bundling adversarial transactions that appear profitable but are invalid, lowering their standing in reputation-based private mempools. Shou et al. [58] introduce BACKRUNNER, which exploits (1) the delay between exploit deployment and execution, and (2) incomplete asset drainage, enabling backrunning to recover funds. These works target searchers by analyzing their logic and competing for the same MEV. In contrast, we exploit vulnerabilities within searchers' contracts directly.

MEV attacks and defenses. Although MEV searchers are considered victims in this paper, prior work often considers them attackers. They perform various MEV activities to extract profit [5], [59]–[61], including front-running [62], sandwiching [63], liquidations [64], and arbitrages [36], [65]–[67]. Some of these activities, notably front-running and sandwich attacks, constitute direct attacks on users by manipulating their transaction order to extract value. To mitigate the negative effects of MEV on users, both academia and industry have proposed various countermeasures, including private transaction channels [34], [68], time-based order fairness protocols [69]–[71], and front-running-resistant AMM designs [72]–[74].

7. Discussion

7.1. Tradeoff Between Cost and Security

Unlike traditional programs, computation on EVMs is expensive: users must pay high fees for complex transactions. This implies that, if rigorously enforced, access control can be costly. To illustrate this, we show in Fig. 10 (in Sec. D) how access control for a Uniswap V3 pool requires computing the expected address of the pool according to the `CREATE2` specification [75], consuming 449 gas per interaction. Assume that each transaction sent to the MEV bot involves an interaction with a Uniswap V3 pool, requiring one verification. If a searcher sends 10K transactions to the MEV bot contract per month, the total gas usage amounts to 4.5M. In contrast, if the contract employs an *incorrect but cheap* verification by checking `tx.origin`, the total gas usage is reduced to 80K.

The cost could even be zero if the contract does not employ any access control mechanism. For example, SKANF also identified an 1inch contract does not implement any access control to protect its asset management logic, allowing any account to transfer ERC-20 tokens from its account. We disclosed this issue to the 1inch team. Their response indicated that the contract only holds residual tokens from executed transactions and does not affect user-owned assets. In this case, the cost of a rigorous access control mechanism seems less justified, considering that the exploit only extracts residual tokens. It may be more reasonable to save gas for legitimate users of the contract.

This raises an interesting question: What is the best tradeoff between cost and security? Many smart contracts that are driven by economic profits may be sensitive to cost, while a rigorous approach would definitely increase their transaction costs. Therefore, a promising research direction would be to propose designs that provide good security guarantees while keeping costs low.

7.2. SKANF as Defense Mechanism

The evaluation in Sec. 4 shows that SKANF can detect vulnerable smart contracts and thus prevent attacks. In this section, we further discuss how SKANF serves as a defense, with developers benefiting more from it than attackers.

Developers can run SKANF before deployment to detect vulnerabilities in the bytecode, even if the code is written directly in low-level languages and employs control flow obfuscation. In particular, they can craft transactions to leverage SKANF's concolic execution. Even after deployment, SKANF can be used before each new interaction with the contract by first passing new transactions to verify the relevant code paths they use, before broadcasting them to the network. In contrast, the attackers can only leverage the limited on-chain transactions, which limits their ability to utilize SKANF effectively.

8. Conclusion

We have presented SKANF, an EVM bytecode analysis tool optimized for closed-source smart contracts such as MEV bots. SKANF can effectively deobfuscate the control flow and identify asset management vulnerabilities, an ability that existing tools do not offer. SKANF also employs concolic execution to improve efficacy. We evaluated SKANF against real-world MEV bots, using historical transactions to facilitate concolic execution. Among 6,554 MEV bots we studied, SKANF detects vulnerabilities in 1,030 of them. Further, SKANF automatically generates exploits against 394 of them, with a potential loss exceeding $10.6M. Furthermore, we discovered 104 attacks in the wild that exploited asset management vulnerabilities against 37 MEV bots, resulting in a total loss of $2.76M.

References

[1] H. Team, "Huff language," https://huff.sh, 2025, accessed: 2025-03-13.

[2] DeGatchi, "Smart contract obfuscation techniques," February 2023, accessed: 2025-04-22. [Online]. Available: https://degatchi.com/articles/smart-contract-obfuscation

[3] DeGatchi, "Swimming safely in the public mempool: Mev smart contract obfuscation techniques," January 2024, accessed: 2025-04-22. [Online]. Available: https://degatchi.com/articles/mev-smart-contract-obfuscation

[4] Z. Ma, M. Jiang, F. Luo, X. Luo, and Y. Zhou, "Surviving in dark forest: Towards evading the attacks from front-running bots in application layer," in *34th USENIX Security Symposium*. Seattle, WA, USA: USENIX Association, 2025.

[5] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *2020 IEEE symposium on security and privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2020, pp. 910–927.

[6] libMEV, "libMEV Leaderboard," https://libmev.com/leaderboard, 2025, accessed: 2025-03-05.

[7] S. by Example, "Phishing with tx.origin," 2025, accessed: 2025-02-19. [Online]. Available: https://solidity-by-example.org/hacks/phishing-with-tx-origin

[8] ConsenSys, "Mythril: A security analysis tool for evm bytecode," 2025, accessed: 2025-02-22. [Online]. Available: https://github.com/ConsenSys/mythril

[9] P. Meunier, "Cwe-656: Reliance on security through obscurity," Common Weakness Enumeration, Jan. 2008. [Online]. Available: https://cwe.mitre.org/data/definitions/656.html

[10] F. A. P. Petitcolas, *Kerckhoffs' Principle*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, ch. K, pp. 1–2.

[11] C. E. Shannon, "Communication theory of secrecy systems," *The Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[12] Dedaub, "Bytecode decompiler," 2025. [Online]. Available: https://dedaub.com/feature/bytecode-decompiler

[13] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *29th USENIX Security Symposium (USENIX Security)*. Boston, MA, USA: USENIX Association, 8 2020, pp. 2757–2774. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/frank

[14] F. Gritti, N. Ruaro, R. McLaughlin, P. Bose, D. Das, I. Grishchenko, C. Kruegel, and G. Vigna, "Confusum Contractum: Confused Deputy Vulnerabilities in Ethereum Smart Contracts," in *32nd USENIX Security Symposium (USENIX Security)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1793–1810. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/gritti

[15] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 1176–1186.

[16] N. Grech, S. Lagouvardos, I. Tsatiris, and Y. Smaragdakis, "Elipmoc: Advanced decompilation of ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–27, 2022.

[17] S. Lagouvardos, Y. Bollanos, N. Grech, and Y. Smaragdakis, "The incredible shrinking context... in a decompiler near you," 2024.

[18] N. Ruaro, F. Gritti, R. McLaughlin, I. Grishchenko, C. Kruegel, and G. Vigna, "Not your type! detecting storage collision vulnerabilities in ethereum smart contracts," in *31st Annual Network and Distributed System Security Symposium (NDSS)*. Reston, VA: The Internet Society, 2024. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/not-your-type-detecting-storage-collision-vulnerabilities-in-ethereum-smart-contracts

[19] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," 2014. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[20] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE, 2010, pp. 317–331.

[21] K. Sen, "Concolic testing," in *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 571–572.

[22] Etherscan Team, "Blockscan chat: Wallet-to-wallet messaging for web3," https://chat.blockscan.com, 2023, accessed: 2025-04-14.

[23] K. Qin, S. Chaliasos, L. Zhou, B. Livshits, D. Song, and A. Gervais, "The blockchain imitation game," in *32nd USENIX Security Symposium (USENIX Security)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 3961–3978. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/qin

[24] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, " SoK: Decentralized Finance (DeFi) Attacks ," in *2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 2444–2461. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.10179435

[25] S. Team, "Solidity programming language," 2025, accessed: 2025-03-13. [Online]. Available: https://soliditylang.org

[26] Etherscan, "Etherscan api documentation," https://docs.etherscan.io, 2025, accessed: 2025-03-08.

[27] Ethereum Foundation, "ERC-20 Token Standard," 2025, accessed: 2025-02-27. [Online]. Available: https://ethereum.org/en/developers/docs/standards/tokens/erc-20

[28] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," 1997.

[29] P. Zhang, Q. Yu, Y. Xiao, H. Dong, X. Luo, X. Wang, and M. Zhang, "Bian: Smart contract source code obfuscation," *IEEE Transactions on Software Engineering*, vol. 49, no. 9, pp. 4456–4476, 2023.

[30] Software Carpentry, "Smart Contract Weakness Classification (SWC)," https://swcregistry.io, 2024, accessed: 2025-06-01.

[31] V. Buterin, "EIP-170: Contract code size limit," November 2016, accessed: 2025-02-27. [Online]. Available: https://eips.ethereum.org/EIPS/eip-170

[32] E. Foundation, "Py-evm: A python implementation of the ethereum virtual machine," 2025, accessed: 2025-02-27. [Online]. Available: https://github.com/ethereum/py-evm

[33] ethpwn, "ethpwn: The swiss army knife for smart contract hacking," 2025, accessed: 2025-02-27. [Online]. Available: https://github.com/ethpwn/ethpwn

[34] S. Yang, K. Nayak, and F. Zhang, " Decentralization of Ethereum's Builder Market ," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2025, pp. 1456–1474.

[35] B. Öz, D. Sui, T. Thiery, and F. Matthes, "Who Wins Ethereum Block Building Auctions and Why?" in *6th Conference on Advances in Financial Technologies (AFT)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 316. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 22:1–22:25. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.AFT.2024.22

[36] L. Heimbach, V. Pahari, and E. Schertenleib, "Non-atomic arbitrage in decentralized finance," in *IEEE Symposium on Security and Privacy (SP), San Francisco, USA*. Los Alamitos, CA, USA: IEEE Computer Society, 2024, pp. 3866–3884.

[37] Etherscan, "MEV Bot Accounts," https://etherscan.io/accounts/label/mev-bot, 2025, accessed: 2025-03-05.

[38] Flashbots, "mev-inspect-py: An mev inspector for ethereum," https://github.com/flashbots/mev-inspect-py, 2024, accessed: 2025-03-08.

[39] bloXroute Labs, "Backrunme api documentation," 2025, accessed: 2025-03-12. [Online]. Available: https://docs.bloxroute.com/evm-networks-bsc-eth/apis/backrunme

[40] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," 2018. [Online]. Available: https://arxiv.org/abs/1809.03981

[41] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[42] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[43] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. Los Alamitos, CA, USA: IEEE Computer Society, 2019, pp. 8–15.

[44] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 454–469.

[45] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 421–430.

[46] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 254–269.

[47] J. Krupp and C. Rossow, "{teEther}: Gnawing at Ethereum to Automatically Exploit Smart Contracts," in *27th USENIX security symposium (USENIX Security)*. Baltimore, MD: USENIX Association, 2018, pp. 1317–1333.

[48] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, 2019, pp. 1186–1189.

[49] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 259–269.

[50] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 557–560.

[51] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic dataflow analyses," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, 2021, pp. 227–239.

[52] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. Los Alamitos, CA, USA: IEEE Computer Society, 2021, pp. 103–119.

[53] C. Shou, S. Tan, and K. Sen, "Ityfuzz: Snapshot-based fuzzer for smart contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. New York, NY, USA: Association for Computing Machinery, 2023, pp. 322–333.

[54] W. Chen, X. Luo, H. Cai, and H. Wang, "Towards smart contract fuzzing on gpus," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2024, pp. 2255–2272.

[55] Z. Zhang, Z. Lin, M. Morales, X. Zhang, and K. Zhang, "Your exploit is mine: Instantly synthesizing counterattack smart contract," in *32nd USENIX Security Symposium (USENIX Security)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1757–1774. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhuo-exploit

[56] A. Yaish, M. Dotan, K. Qin, A. Zohar, and A. Gervais, "Suboptimality in defi," 2023. [Online]. Available: https://ia.cr/2023/892

[57] A. Yaish, K. Qin, L. Zhou, A. Zohar, and A. Gervais, "Speculative denial-of-service attacks in ethereum," in *33rd USENIX Security Symposium (USENIX Security)*. Philadelphia, PA: USENIX Association, 8 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/yaish

[58] C. Shou, Y. Ke, Y. Yang, Q. Su, O. Dadosh, A. Eli, D. Benchimol, D. Lu, D. Tong, D. Chen *et al.*, "Backrunner: Mitigating smart contract attacks in the real world," 2024.

[59] K. Qin, L. Zhou, and A. Gervais, "Quantifying blockchain extractable value: How dark is the forest?" in *2022 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2022, pp. 198–214.

[60] Z. Li, J. Li, Z. He, X. Luo, T. Wang, X. Ni, W. Yang, X. Chen, and T. Chen, "Demystifying defi mev activities in flashbots bundle," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. New York, NY, USA: Association for Computing Machinery, 2023, pp. 165–179.

[61] C. Ferreira Torres, A. Mamuti, B. Weintraub, C. Nita-Rotaru, and S. Shinde, "Rolling in the shadows: Analyzing the extraction of mev across layer-2 rollups," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS)*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 2591–2605.

[62] C. F. Torres, R. Camino, and R. State, "Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain," in *30th USENIX Security Symposium (USENIX Security 21)*. Online: USENIX Association, 2021, pp. 1343–1359. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/torres

[63] L. Zhou, K. Qin, C. F. Torres, D. V. Le, and A. Gervais, "High-frequency trading on decentralized on-chain exchanges," in *2021 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2021, pp. 428–445.

[64] K. Qin, L. Zhou, P. Gamito, P. Jovanovic, and A. Gervais, "An empirical study of defi liquidations: Incentives, risks, and instabilities," in *Proceedings of the 21st ACM Internet Measurement Conference (IMC)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 336–350.

[65] L. Zhou, K. Qin, A. Cully, B. Livshits, and A. Gervais, "On the just-in-time discovery of profit-generating transactions in defi protocols," in *2021 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2021, pp. 919–936.

[66] R. McLaughlin, C. Kruegel, and G. Vigna, "A large scale study of the ethereum arbitrage ecosystem," in *32nd USENIX Security Symposium (USENIX Security)*. Anaheim, CA: USENIX Association, 2023, pp. 3295–3312.

[67] B. Öz, C. F. Torres, J. Gebele, F. Rezabek, B. Mazorra, and F. Matthes, "Pandora's box: Cross-chain arbitrages in the realm of blockchain interoperability," 2025.

[68] S. Yang, F. Zhang, K. Huang, X. Chen, Y. Yang, and F. Zhu, "Sok: Mev countermeasures," in *Proceedings of the Workshop on Decentralized Finance and Security (DeFi)*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 21–30.

[69] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels, "Order-fairness for byzantine consensus," in *40th Annual International Cryptology Conference (CRYPTO)*. Cham: Springer International Publishing, 2020, pp. 451–480.

[70] Y. Zhang, S. Setty, Q. Chen, L. Zhou, and L. Alvisi, "Byzantine Ordered Consensus without Byzantine Oligarchy," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Online: USENIX Association, 2020, pp. 633–649.

[71] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, Strong Order-Fairness in Byzantine Consensus," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. New York, NY, USA: Association for Computing Machinery, 2023, pp. 475–489.

[72] L. Zhou, K. Qin, and A. Gervais, "A2mm: Mitigating frontrunning, transaction reordering and consensus instability in decentralized exchanges," 2021.

[73] S. Wadhwa, L. Zanolini, A. Asgaonkar, F. D'Amato, C. Fang, F. Zhang, and K. Nayak, "Data independent order policy enforcement: Limitations and solutions," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS)*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 378–392.

[74] M. Zhang, S. Yang, and F. Zhang, "Rediswap: Mev redistribution mechanism for cfmms," 2024.

[75] V. Buterin, "EIP-1014: Skinny CREATE2," https://eips.ethereum.org/EIPS/eip-1014, 2018, accessed: 2025-03-13.

[76] 0xprincess, "One arbitrage bot was drained by a token called destroyer inu," 2024, accessed: 2025-02-19. [Online]. Available: https://twitter.com/0x9212ce55/status/1808233634522095809

[77] PeterBorah, "Remove tx.origin · issue #683 · ethereum/solidity · github," 2016. [Online]. Available: https://web.archive.org/web/20250415170758/https://github.com/ethereum/solidity/issues/683

[78] S. Contributors, "Contract abi specification," https://docs.soliditylang.org/en/latest/abi-spec.html, 2025, accessed: 2025-02-19.

[79] Uniswap, "IUniswapV3SwapCallback Interface," https://docs.uniswap.org/contracts/v3/reference/core/interfaces/callback/IUniswapV3SwapCallback, 2025, accessed: 2025-02-21.

[80] Q. Yu, P. Zhang, H. Dong, Y. Xiao, and S. Ji, "Bytecode obfuscation for smart contracts," in *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2022, pp. 566–567.

[81] B. Zhang, N. He, X. Hu, K. Ma, and H. Wang, "Following Devils' Footprint: Towards Real-time Detection of Price Manipulation Attacks," in *34th USENIX Security Symposium*. Seattle, WA, USA: USENIX Association, 2025.

[82] Software Weakness Classification (SWC) Registry, "SWC-110: Missing Release-Critical Information," https://swcregistry.io/docs/SWC-110, 2021, accessed: 2025-06-01.

[83] P. Ma, N. He, Y. Huang, H. Wang, and X. Luo, "Abusing the ethereum smart contract verification services for fun and profit," 2023.

[84] Binance, "Binance data collection," https://data.binance.vision, 2025, accessed: 2025-06-05.

[85] A. Beregszaszi, P. Bylica, A. Maiboroda, M. Garnett, and P. Dobaczewski, "EIP-3540: EOF - EVM Object Format v1," https://eips.ethereum.org/EIPS/eip-3540, 2021, accessed: 2025-04-11.

[86] pcaversaccio, Matt, Moody, and Ramana, "EOF: When Complexity Outweighs Necessity," https://hackmd.io/@pcaversaccio/eof-when-complexity-outweighs-necessity, 2025, accessed: 2025-04-11.

Appendix A. Case Study: the Destroyer Inu Attack

A notable attack on a closed-source smart contract that relies on external inputs for its operation took place on July 1st, '24, causing a loss of 22 ETH (worth $51,056 at the time) [76]. As the attack involves an attacker-created ERC-20 token called "Destroyer Inu", we use the same name for the attack itself. This case study shows that control-flow obfuscation adopted by the victim contract ends up hiding vulnerabilities from analysis tools — without stopping real-world attackers — arguably making contracts less secure.

tx.origin phishing attack. The first issue is a classic vulnerability where `tx.origin` is compared to a hard-coded trusted address. This vulnerability is known as the *tx.origin phishing attack* [7], dating back to at least 2016 [77]. In such attacks, an attacker tricks the victim into calling a malicious contract, which then calls the victim's contract; since both calls have the same `tx.origin`, the malicious smart contract can execute the victim smart contract with the same privileges as the victim herself.

A more severe issue that follows the incorrect access-control check is improper asset management. However, existing tools miss the full exploit because of control-flow obfuscation. Existing tools such as Mythril flag it only as a "low-risk" issue. We will discuss the full vulnerability next.

tx.origin phishing attack, obfuscated. In the Destroyer Inu Attack, the victim contract is obfuscated. We use pseudo-Yul code to illustrate the smart contract's logic in Fig. 7. First, at the entry point of the MEV bot's smart contract in lines 3 to 5, it implements an access control mechanism based on the transaction's origin to prevent unauthorized addresses from interacting with the contract. Then, in lines 7 to 9, the contract proceeds by jumping

```
1  assembly {
2  entry 0x0:
3     let from := origin() // tx.origin
4     let expected := 0xdead...beef
5     if iszero(eq(from, expected)) {revert(0, 0)}
        // access check
6     // read two bytes from calldata as the jump
        destination.
7     let data := calldataload(0x84) // suppose it
        's 0x0a00...
8     let dst := shr(0xf0, data)
9     jump dst // jump 0x0a00
10
11 jumpdest 0x0a00:
12    let token := calldataload(0x86)
13    let to := calldataload(0xa6)
14    let value := calldataload(0xc6)
15    let ptr := mload(0x40)
16    mstore(ptr, 0xa9059cbb) // ERC-20 transfer
17    mstore(add(ptr, 4), to)
18    mstore(add(ptr, 36), value)
19    // call target.transfer(to, value)
20    let success := call(gas(), token, 0, ptr,
        68, 0, 0)
21    if iszero(success) { revert(0,0) }
22
23 jumpdest 0x0b00: ...
24 }
```

Figure 7. Pseudo-Yul MEV bot smart contract. The code starting at `0x0a00` contains a vulnerable function call whose input is determined by the calldata, allowing an adversary to transfer any amount of ERC-20 tokens from the MEV bot's account to their own. Note that `jump` and `jumpdest` are not part of Yul syntax but are included for better readability.

to a specific place in code, as determined by the calldata supplied by the transaction. This is unlike "standard" smart contracts that match function calls using the so-called *function selector* defined as the first 4 bytes of the calldata [78]. In this MEV bot smart contract specifically, when the two calldata bytes at positions `0x85` and `0x86` are `0x0a00`, the program jumps to the code segment starting at `jumpdest 0x0a00`. The functionality of this segment is to manage the transfer of a specific amount of an ERC-20 token to a target address. Therefore, it loads the address of the ERC-20 token, the recipient address, and the amount from the calldata (lines 12 to 14). Then, in lines 15 to 21, it constructs the input based on these parameters and invokes the ERC-20 transfer.

Exploit. To an attacker who can remove control-flow obfuscation, the vulnerability is immediate: the victim contract allows transferring any amount of any ERC-20 tokens held by this contract to another address if an attacker can bypass the tx.origin check via many forms of *tx.origin phishing*, which is precisely what happened.

Attack analysis. A direct cause of the Destroyer Inu attack is that the victim's contract relies on a vulnerable comparison to block unintended calls by adversaries. A strawman mitigation is to use a hard-coded comparison against `msg.sender` instead of `tx.origin`. However, this solution does not meet real-world requirements when verifying if the caller is the searcher (the contract's creator and owner), as, in practice, the contract may be called by

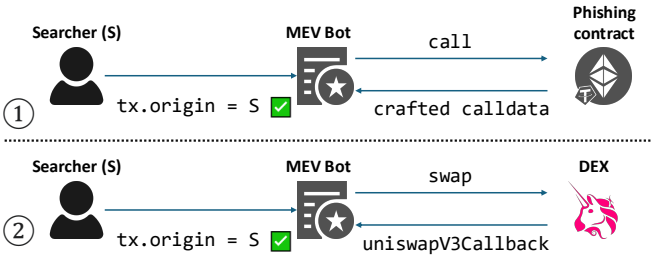

Figure 8. Two scenarios where an MEV bot contract receives a callback: a phishing contract (①) and a DEX (②).

other different contracts. For example, Uniswap V3 pools require any contract that calls their `swap` function to implement a function called `uniswapV3SwapCallback` [79], which is invoked by pools at the end of executing `swap` to ensure that the interacting contract pays the pool the tokens owed for the swap (as illustrated in Fig. 8, scenario ②)

Other DeFi protocols, such as Uniswap V2, Sushiswap, and AAVE, also adopt similar callback designs. In these cases, a hard-coded comparison against `msg.sender` is not effective because the caller may be any contract deployed by these protocols. Compared to this, verifying `tx.origin` is not restricted by a specific caller and can prevent direct calls to the contract, but it introduces security risks. Therefore, more precise protection is needed to prevent potential adversaries, which can be a high requirement for contracts, as real-world attack incidents have shown.

Appendix B. Other Obfuscation Techniques

We identify four contract obfuscation techniques based on previous works [28], [29], [80], [81], summarized as:

- **Layout obfuscation** typically aims to reduce information available to a human reader, e.g., by removing comments and renaming functions and variables [28], [29].
- **Data flow obfuscation** obscures how data is processed and accessed within the smart contract [28], [29].
- **Control flow obfuscation** alters the execution path of the smart contract to make its logical flow difficult to follow [4], [28], which is our focus.
- **Preventive transformations** aim to hinder reverse engineering by exploiting weaknesses in current decompilers and deobfuscators [28]. E.g., contracts can self-destruct after execution to render their bytecode unavailable, preventing future inspection [81].

We empirically verified that the layout and data flow obfuscation do not interfere with existing analysis tools.

Two mature obfuscation tools are available for EVM smart contracts [81]: BOSC [80] and BiAn [29]. Since BOSC cannot guarantee that the obfuscated contracts remain deployable, it is not suitable for our setting. Following prior work [81], which also only used BiAn due to BOSC's limitations, we adopt BiAn for our evaluation.

We collected vulnerable smart contracts from the Smart Contract Weakness Classification Registry [30] and applied

BiAn to each contract to generate an obfuscated version. Specifically, we apply two types of obfuscation: layout obfuscation and data flow obfuscation. We then compile the original vulnerable contracts and the obfuscated ones, and apply Mythril to detect their vulnerabilities.

Interestingly, we found that for 65 of the vulnerable smart contracts, layout and data flow obfuscation do not prevent the vulnerabilities from being detected; instead, the obfuscation even introduces new "Assert Violation" issues [82]. For the remaining two contracts, we observed that Mythril could still detect issues in the obfuscated contracts at the source code level, but not at the bytecode level. We speculate this might be related to compiler optimizations. In summary, our findings indicate that existing analysis tools like Mythril can still work effectively on smart contracts with both layout and data flow obfuscation.

Appendix C. Historical Versions of MEV Bots

Although smart contracts on Ethereum are generally immutable once deployed, their bytecode at a given address may change over time through the use of `SELFDESTRUCT` and `CREATE2` opcodes [19]. Specifically, `SELFDESTRUCT` removes the contract code from the state, and `CREATE2` allows redeployment of a new contract at the same address using a fixed deployer address, salt, and modified initialization code. As a result, a smart contract may have different bytecode at different points in time [83].

We define a *version* of a smart contract as the specific bytecode deployed at a given address during a particular period. If a contract is destroyed and re-deployed with different bytecode at the same address, each instance is considered a distinct version.

Version statistics of MEV bots. To identify all versions of an MEV bot contract, we analyze the historical transactions associated with its address. Specifically, we parse transaction traces to check whether the address was the target of a `CREATE` or `CREATE2` deployment. For each instance of bytecode deployment at the address, we record the corresponding version. This allows us to reconstruct the full sequence of code changes for MEV bot contracts that may have been redeployed multiple times.

We apply this method to all MEV bots in our dataset. As shown in Fig. 9, about 96% of them are deployed only once, indicating no code change over time. Meanwhile, about 0.9% of MEV bots have more than five versions, suggesting that their code has been repeatedly updated.

Vulnerability detection in historical versions. For MEV bots with multiple historical versions, we also apply SKANF to each historical version to identify potential asset management vulnerabilities and attempt exploit generation. This analysis allows us to detect vulnerabilities that may exist only in earlier versions of the MEV bot contract but not in the current one. In total, we identify 114 vulnerable historical versions across 44 MEV bot contracts and successfully generate 41 exploits targeting these versions.

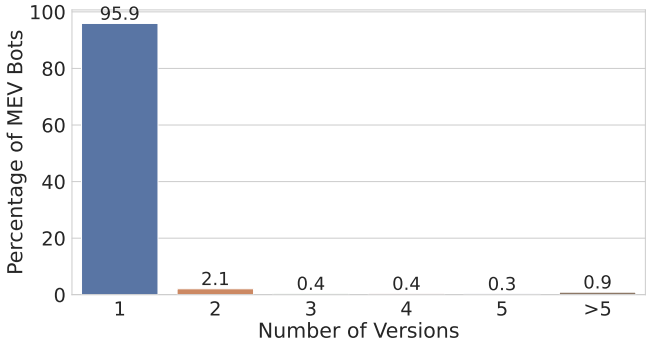

Figure 9. Distribution of the number of versions per MEV bot in our dataset.

Loss estimation. To approximate the maximum possible loss, we analyze the highest ERC-20 token balance held by each exploitable version of an MEV bot. Specifically, for each exploitable version, we identify the block range during which that version was active and observe the maximum ERC-20 token balance within that range. If the version is the latest one, we consider the range from its deployment up to block 22,635,000 (corresponding to June 5th, 2025).

Consistent with the estimation for current loss, if a contract's vulnerability is limited to a specific ERC-20 token, we consider only the potential loss associated with that token. If an MEV bot has multiple exploitable versions, we report the loss corresponding to the version with the highest observed ERC-20 balance. To simplify the estimation, we assume that all tokens were sold on June 5th, 2025. This allows us to use the token prices on Binance [84] on that day to estimate the potential loss for seven major tokens: WETH, WBTC, USDC, USDT, DAI, UNI and LINK.

Appendix D. Solidity Example of Checking Caller

Fig. 10 shows a Solidity example of how to rigorously check whether the caller is a Uniswap V3 pool. The `verifyPoolAccess` function checks whether the caller is a valid Uniswap V3 pool by reconstructing the expected pool address using the `CREATE2` scheme. It first computes a salt by hashing the tuple `(token0, token1, fee)` with `keccak256`, consistent with how Uniswap V3 encodes pool parameters. Then, it derives the expected pool address using the standard `CREATE2` formula, which combines the factory address, the salt, and the pool's initialization code hash. If the derived address matches `msg.sender`, the function returns `true`, confirming that the caller is a legitimate pool contract created by the Uniswap V3 factory with the given parameters.

Appendix E. Implications of EVM Object Format

The EVM Object Format (EOF) [85] is a proposed redesign of the EVM that introduces structured sections,

```
contract UniswapV3PoolAccessControl {
    address FACTORY = 0
    x1F98431c8aD98523631AE4a59f267346ea31F984;
    bytes32 POOL_INIT_CODE_HASH =
    0xe34f36d28c5efcd7c58e2e84af79e2a
    dffbe52f705d05dca7e6a181f8a19baf1;
    function verifyPoolAccess(
        address token0,
        address token1,
        uint24 fee
    ) public view returns (bool) {
        bytes32 salt = keccak256(abi.encode(
        token0,token1,fee));
        address computedAddress = address(uint160
        (uint256(
            keccak256(abi.encodePacked(
                bytes1(0xff),
                FACTORY,
                salt,
                POOL_INIT_CODE_HASH
            )))));
        return computedAddress == msg.sender;
    }
}
```

Figure 10. A Solidity implementation to check if the caller is a Uniswap V3 pool. It verifies whether a given pool address is valid by deriving its expected address using `CREATE2`.

enforces static control flow, and removes dynamic jumps, aiming to improve analyzability and enable future EVM upgrades. However, such an upgrade also tends to constrain the expressiveness of smart contracts and introduces additional complexity for developers [86]. These concerns have led to EOF being excluded from the next Ethereum upgrade.

In EOF, indirect jumps and dynamic dispatch are not allowed [85], making control-flow obfuscation more difficult to implement, and static analysis easier to perform. Nevertheless, the concolic execution approach we present remains effective. Even in the absence of control-flow obfuscation, detecting asset management vulnerabilities requires reasoning about attacker-controlled inputs and complex data flows, tasks that benefit from the techniques used in SKANF.

Moreover, legacy (non-EOF) smart contracts will continue to exist and interact with new contracts, preserving the relevance of our techniques. Until EOF adoption is complete, and perhaps even beyond, tools like SKANF will remain essential for analyzing the security of deployed bytecode across formats.