Not your Type! Detecting Storage Collision Vulnerabilities in Ethereum Smart Contracts

Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, Giovanni Vigna
University of California, Santa Barbara
{ruaronicola, degrigis, robert349, grishchenko, chris, vigna}@ucsb.edu

***Abstract*—In recent years, the Ethereum blockchain has seen significant growth and adoption. One of the key factors of its success is the possibility to run immutable programs known as *smart contracts*. Smart contracts allow for the automatic manipulation of digital assets and play a central role in the new decentralized finance (DeFi) ecosystem. With the growth of DeFi, the interactions between smart contracts have become increasingly complex, enabling advanced financial protocols and applications. However, bugs in smart contract interactions are also a common cause of critical vulnerabilities that result in considerable financial losses.**

In this paper, we study and detect a type of cross-contract vulnerability known as a *storage collision*. A smart contract uses storage to persistently store its data on the blockchain. Typically, each contract has its own separate storage. However, it is also possible that two smart contracts share their storage (using a delegate call). Unfortunately, when these two contracts have different understandings of the types/semantics of their shared storage, a *storage collision* vulnerability can occur. This may lead to unexpected behavior such as denial of service (frozen funds), privilege escalation, and theft of financial assets.

To detect and investigate the impact of storage collision vulnerabilities at scale, we propose CRUSH, a novel analysis system that discovers these flaws and synthesizes proof-of-concept exploits. We leverage CRUSH to perform a large-scale analysis of 14,237,696 smart contracts deployed on the Ethereum blockchain since its genesis. CRUSH identifies 14,891 potentially vulnerable contracts and automatically synthesizes an end-to-end exploit for 956 of them. Our system uncovers more than $6 million of *novel, previously unreported* potential financial damage caused by storage collision vulnerabilities.

I. Introduction

Ethereum [20] is a global, decentralized platform that uses a blockchain to enable both the transfer of its native currency, ETH [21], and the deployment of programs called *smart contracts*. As opposed to Bitcoin [44], which is mainly used as a decentralized ledger for its native cryptocurrency [13], Ethereum provides a more flexible and programmable alternative, allowing developers to build and deploy decentralized applications (dApp). During recent years, the market capitalization of Ethereum has considerably increased, becoming the second largest chain by market capitalization ($224 billion at the time of writing) [6]. This exceptional growth has been driven by the excitement around a new form of blockchain-based finance known as DeFi (i.e., *decentralized finance*) [19].

Smart contracts are programs implemented on top of the Ethereum Virtual Machine (EVM) [59] and hosted on the blockchain. Smart contracts are executed on-demand by blockchain users, and their execution commonly involves the modification of their persistent storage, the transfer of funds, and the invocation of other contracts' functionality. For example, when a blockchain developer deploys a smart contract that automates the governance of a new financial asset (e.g., a new cryptocurrency), users can directly interact with the contract's logic to manage their holdings. Such a mechanism creates a trustless environment with increased transparency, where the governance logic is clearly defined, public, and *immutable*.

Complex financial protocols commonly involve coordinated interactions among several smart contracts. For instance, a smart contract may consult an on-chain oracle contract [10] to access the current currency exchange rates for USD-EUR, decide to exchange cryptocurrency by interacting with a decentralized exchange (DEX) [14], and use the proceeds to take out a loan [27]. These interactions drastically expand the contract's functionality and allow for the creation of sophisticated dApps. However, these interactions come at the cost of increased complexity and a much wider attack surface. Zhou et al. [63] estimate that between April 2018 and April 2022, DeFi protocols suffered from attacks that resulted in the loss of more than 3 billion dollars, making smart contract vulnerability hunting a hot topic in both industry and academia [8], [9], [15], [30], [35], [36], [38], [40], [43], [47], [50].

An important type of interaction between contracts is via *delegation*. When a source contract invokes a delegate call to a destination contract, the destination contract has direct read-and-write access to the source contract's underlying storage data. In practice, this mechanism is used either when one is interested in including an external contract as a third-party library or, most commonly, when developers want to leverage the *proxy pattern* [45] to enable the upgradeability of a contract's code. In these cases, the two interacting contracts both operate on a single, shared underlying persistent storage, potentially leading to a security issue known as a *storage collision* vulnerability.

A storage collision occurs when two contracts read and write the same underlying storage slot (variable), but they have different understandings regarding the data stored in that slot (e.g., different types). For example, one contract might write an integer value to a specific storage slot, whereas the other accesses *the same storage slot* as a Boolean value.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24713
www.ndss-symposium.org

Not surprisingly, such (colliding) accesses can lead the two contracts to operate on unexpected values. This can result in abnormal behavior and security vulnerabilities. In fact, storage collision vulnerabilities have been previously exploited, causing extensive damage and loss of funds. For example, the decentralized music platform AUDIUS [3] was attacked in July 2022 and lost more than 6 million dollars [17] due to a storage collision vulnerability. More specifically, a type misinterpretation (integer to Boolean) of a specific storage slot disabled an important security check in one of their contracts. This eventually let the attacker re-initialize the owner of the AUDIUS protocol and seize control of their treasury.

Storage collision vulnerabilities are often easy to overlook without a deep understanding of both the low-level storage operations in the EVM and the storage layout of all the contracts involved. Currently, the most effective way to prevent storage collision vulnerabilities is to rely on compiler warnings. Unfortunately, such warnings are *only* raised when using certain development frameworks, such as Truffle [55]. Moreover, for these compile-time warnings to be complete, it is essential to know the source code of *all* contracts that will be used – and have been used – in a delegation operation. In practice, this is not always possible, and, therefore, such warnings are not always effective.

To address the limited scope of such warning mechanisms, we implement CRUSH, a system that can identify storage collision vulnerabilities and automatically synthesize a proof-of-concept end-to-end exploit against a vulnerable contract. To the best of our knowledge, CRUSH represents the first systematic approach to detect storage collision vulnerabilities.

In summary, we make the following contributions:

- We describe a technique to automatically identify groups of contracts that share the same underlying storage by observing their interactions on-chain.
- We propose and implement CRUSH, a novel approach that leverages symbolic execution and program slicing to detect and exploit storage collisions among such contract groups.
- We evaluate CRUSH over more than 14 million smart contracts on the Ethereum blockchain, detecting 14,891 potentially vulnerable contracts, and automatically generating a proof-of-concept attack for 956 of them. We estimate the potential financial damage to be over $12 million, with more than $6 million in *previously unreported* attack opportunities.

In the spirit of open science, we share our dataset upon request, and open-source CRUSH.

II. BACKGROUND

A. Smart Contracts

Smart contracts are programs, usually written in a high-level programming language such as Solidity [53] or Vyper [54], and then compiled into EVM bytecode. The bytecode is then deployed on the Ethereum blockchain and executed on-demand by the Ethereum Virtual Machine (EVM). Contracts execute when a user (or, more precisely, an externally owned account) invokes one of their functions. Practically, a user signs and sends a transaction to the contract along with input data (i.e., calldata), the EVM interprets the transaction, and the smart contract code is executed. Contracts use two forms of *temporary* storage to hold and manipulate ephemeral data during their execution: addressable memory and a stack. In addition, contracts use one form of *persistent* storage, simply referred to as "storage," which is made permanent on the blockchain at the end of each transaction. In the following paragraphs, we describe the dynamics of smart contract execution, and the interplay of calldata, storage, stack, and memory.

B. Smart Contract Interactions

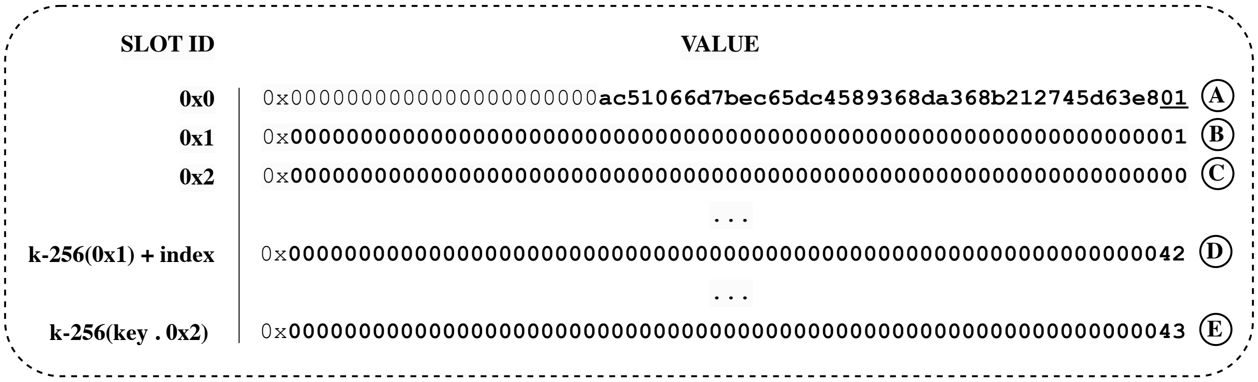
In the Ethereum blockchain, transactions involving smart contracts can be of two types. *External transactions* – or simply "transactions" – are generated by any externally-owned account (EOA) that interacts with a contract on the blockchain. On the other hand, *internal transactions* are generated when a smart contract directly calls another contract's functions. To fully understand contract interactions in the Ethereum blockchain, one must first understand the dynamics of such function calls.

Public Functions. A smart contract's bytecode executes as a single monolithic blob of code, with jump instructions directing the control flow. Public functions serve as entry points for contract execution and play a crucial role in understanding a smart contract's operation. A transaction's calldata encodes public function calls as a sequence of bytes, which consist of a function selector (*4 bytes*) plus the function's arguments. The function selector is generated by hashing the target function's signature with the `keccak256` hashing function [58] and truncating its output after the first four bytes. For example, the function selector for `initialize(address)` is `c4d66de8`. The function selector is followed by a byte-string that represents the function arguments, which are encoded according to the compiler convention [53], [54]. Compiled contracts start the execution at a generated "dispatcher" routine. The dispatcher first extracts the function selector and, if the extracted function selector matches one of the public functions, the contract jumps to the function's start. The function then parses its arguments and executes the function body. Otherwise, if a matching function selector is not found, the dispatcher executes a `fallback` method [16].

Contract Communication Opcodes. Public functions can be directly invoked by a user, or invoked by a contract through one of the following EVM opcodes:

(1) **CALL** is the most basic function call and allows to both invoke a function of a target contract and transfer ETH.
(2) **STATICCALL** invokes a function in the target contract in a view-only fashion – that is, preventing any modifications to the storage of the target contract and of any contract that is called by the target itself.
(3) **DELEGATECALL** allows a contract to call another contract's functions as if they were its own – that is, the code of the called contract is directly operating on the caller's persistent storage and uses its ETH balance[1].

[1]The EVM specification also includes a `CALLCODE` opcode, which provides a functionality equivalent to `DELEGATECALL`. We will not discuss its use, as it has been deprecated in favor of `DELEGATECALL` [18].



SLOT ID	VALUE	
0x0	0x0000000000000000000000ac51066d7bec65dc4589368da368b212745d63e801	Ⓐ
0x1	0x0000000000000000000000000000000000000000000000000000000000000001	Ⓑ
0x2	0x0000000000000000000000000000000000000000000000000000000000000000	Ⓒ
	...	
k-256(0x1) + index	0x0000000000000000000000000000000000000000000000000000000000000042	Ⓓ
	...	
k-256(key . 0x2)	0x0000000000000000000000000000000000000000000000000000000000000043	Ⓔ

Fig. 1: The storage layout of the Logic contract in Figure 2. In Ⓐ the Boolean variable `initialized` (01) packed in the same slot with the address variable `admin`. In Ⓑ the `BASE` slot of the array variable `artworkIDs`, and in Ⓓ one of its elements. Finally, in Ⓒ the `BASE` slot of the `artworkHolders` mapping, and in Ⓔ one of its elements.

Due to the nature of storage collision vulnerabilities, we focus exclusively on interactions that arise from the use of `DELEGATECALL` and ignore interactions that arise from the use of `CALL` and `STATICCALL`.

C. Contract Storage

All the state variables – or persistent variables – of a smart contract reside in the contract's storage. A contract's storage is a large array organized into slots that have a size of 32 bytes (256 bits), and each slot is identified by a unique index that ranges from 0 to $2^{256} - 1$ [59]. The storage slot index for each given state variable must be computed in a deterministic way to ensure consistent access to the storage data. In fact, the layout of the state variables in such storage slots is determined by the compiler, which takes into account the variable's order of declaration[2] in the source code. While the following paragraphs describe the storage layout choices of the Solidity [53] compiler, we observe that equivalent choices apply to other compilers (e.g., Vyper [54]).

Fixed-size variable types. Basic types such as `uint` (32 bytes), `address` (20 bytes), and `bool` (1 byte) are stored contiguously in 32-byte slots, starting from slot 0x0. It is important to note that multiple contiguous items that need less than 32 bytes are packed into a single storage slot for compactness, according to the following rules:

- The first item in a storage slot is always aligned toward the least significant bytes.
- Fixed-size types use only as many bytes as are necessary to store them.
- If a variable does not fit the remaining part of a storage slot, it is stored in the next storage slot.

Figure 1 shows an example of such a packed layout resulting from the compilation of the `Logic` contract in Figure 2 (the `Logic` contract starts at Line 16). Note how the variable `initialize` (`bool`, Line 17 in Figure 2) and `admin` (`address`, Line 18) are packed together in slot 0x0 Ⓐ as they require together less than 32 bytes of space.

[2]For contracts that use inheritance, their subclasses are first linearized [5] as it is typical in other object-oriented programming languages [29], and this determines the order of the variable's declarations.

Dynamic-size variable types. Dynamic types such as `arrays` and `mappings` are always stored in a new slot (the `BASE` slot), and their elements are stored according to the following rules:

- The `BASE` slot stores the length of the array, or, in the case of a mapping, it is left unused.
- Each array element (with index: `INDEX`) is stored at the slot `keccak256(BASE)+INDEX`.
- Each element of a mapping (with key: `KEY`) is stored at the slot calculated by concatenating `KEY` and `BASE` and using the resulting value as an input for the `keccak256` function (i.e., `keccak256(KEY.BASE)`).

Consider again Figure 1: the `BASE` slot for the array `artworkIDs` (Line 19) is slot 0x1 Ⓑ and contains the length of the array (0x1), while its first element (at `INDEX`=0) is 0x42, and can be found at address `keccak256(0x1)+0x0` Ⓓ. Finally, the `BASE` slot for the mapping variable `artworkHolders` (Line 20) is 0x2 Ⓒ and one of its elements (with `KEY=key`) can be found at `keccak256(key.0x2)` Ⓔ.

To read and manipulate the contract's storage, the EVM provides two instructions: SLOAD and SSTORE [59]. The SLOAD instruction retrieves the value stored at a specified storage slot, while the SSTORE instruction updates the value at a given storage slot with a new value. Note that both `SLOAD` and `SSTORE` *always* read/write a whole 32-byte storage slot.

When multiple variables are packed within the same storage slot, the relevant variable being accessed is extracted via a bit-masking procedure, which is transparently generated by the compiler. For instance, to extract the `initialized` variable from slot 0x0, the entire 32-byte value is first read and then masked with the value `0xff` to obtain `0x01`.

III. Motivation

Storage plays a vital role in a smart contract's life cycle, allowing data persistence across multiple transactions. In Ethereum, a contract and its storage are inextricably linked. That is, the storage associated with one contract cannot be simply transferred to another smart contract. Moreover, smart

```
1  contract Proxy {
2      uint visits;    // storage slot [0x0]
3      [..]
4      address LOGIC;  // storage slot [ERC-1967]
5
6      constructor() {
7          LOGIC = 0x[..];
8          LOGIC.initialize();
9          visits = 0;
10     }
11     fallback() external {
12         LOGIC.delegatecall(msg.data);
13     }
14 }
15
16 contract Logic {
17     bool initialized;        // storage slot [0x0]
18     address admin;           // storage slot [0x0]
19     array artworkIDs;        // storage slot [0x1]
20     mapping artworkHolders;  // storage slot [0x2]
21
22     function initialize() external {
23         require(!initialized);
24         initialized = 1;
25         admin = msg.sender;
26     }
27     function withdraw() external {
28         require(msg.sender == admin);
29         payable(admin).transfer(this.balance);
30     }
31 }
```

Fig. 2: Simplified Solidity code of the Proxy and Logic contracts. The interaction of such contracts results in a storage collision.

contracts are *immutable* and cannot be changed after deployment. Unfortunately, immutability makes the governance of a smart contract challenging. For example, when a bug is found in a contract, it is not possible to directly upgrade (patch) the contract's code. In addition, even if developers deployed a new, patched version of the contract, its state (storage) would need to somehow be migrated as well.

To address the need to upgrade contracts, even though they hold a significant amount of persistent state, developers devised a clever design approach called a *proxy pattern* [45]. This approach achieves upgradability by separating the application's logic from its persistent storage and keeping them in two separate contracts: the contract that contains the logic is typically referred to as the *logic contract*, while the persistent storage is typically held in what is called a *proxy contract*. A user interacts with the application by invoking a function in the proxy contract. The proxy, in turn, forwards this call to its logic contract. When the application logic needs to be changed, the developers can simply create a new logic contract, and then have the proxy forward calls to this new code. This can be done, for instance, by changing the storage variable in the proxy contract that holds the address of its active logic contract. In practice, a proxy contract might have multiple logic contracts that implement the desired functionality.

The key that makes the proxy pattern work is the `DELEGATECALL` instruction [37]. When the proxy uses `DELEGATECALL` to call a function in the logic contract, the code of the logic contract *operates directly* on the storage in the proxy contract. Hence, when the logic contract changes in case of an update or patch, the new logic contract has seamless access to the state of the application (in the proxy).

Intuitively, storage-logic decoupling allows for the contract's code to be updated without losing its storage state. This is especially important in complex smart contract systems and DeFi protocols with many interacting contracts that need continuous updates.

Storage Collision Vulnerability. Shared storage in general, and the proxy pattern in particular, can introduce *storage collision vulnerabilities*, which we define as follows. Consider two contracts A and B that, due to a `DELEGATECALL`, operate on the same underlying storage. A *storage collision* occurs when A and B do not agree about the type (or interpretation) of the variables in the shared storage. In particular, this collision occurs when A *writes* to a certain storage slot, and then B *reads* that slot with a different interpretation. We call such a slot a *conflicting storage slot*. This escalates to a vulnerability when a read that operates over a corrupted value can lead to denial of service, privilege escalation, theft of funds, or other unexpected behavior.

For ease of exposition, we will refer to a contract that is the source of a DELEGATECALL as the *proxy*, and the target of a DELEGATECALL as the *logic*. Note that, as a consequence of this "loose" definition, the terms *proxy* and *logic* will capture more situations than just the proxy pattern. For example, one contract (*proxy*) might use another contract (*logic*) simply as a library.

Example. To demonstrate a storage collision vulnerability, we present a simplified example equivalent to the real-world attack that was launched against the AUDIUS [17] protocol. Consider Alice, an artist who aims to establish an on-chain marketplace for selling her NFT artwork. Having learned about DELEGATECALL, and inspired by the standard proxy pattern [45], Alice decides to use them to build her own upgradeable marketplace. She achieves this by creating two distinct contracts, which we show in Figure 2. The first contract, `Proxy`, is responsible for storing the marketplace's state, which includes information such as the current owners of her NFTs. The second contract, `Logic`, contains the marketplace implementation. By using the proxy pattern, Alice can update the logic contract without modifying the state. For example, when Alice is ready to integrate new functionality (e.g., an auction system), she simply needs to adjust a pointer in her proxy contract to redirect it to a new logic contract. This approach allows the logic to be upgraded, ensuring that her NFT marketplace remains secure and relevant with the latest functionality.

The `Proxy` contract code is fairly simple. A `constructor` routine (Line 6) sets the value of the `LOGIC` variable (storage slot `ERC-1967` refers to the standard Proxy storage slot [26]), calls the logic's initialization routine, and sets the value of the `visits` variable. Any other interaction is handled by the `fallback` function, which delegates (forwards) function calls to the `Logic` contract (Line 12). As a result of this delegation call, both contracts share the same underlying storage.

The `Logic` contract has an `initialize` function (Line 22) and a `withdraw` function (Line 27). In the `initialize` function, the contract first checks the value of the variable `initialized`, stored in storage slot 0x0 (Line 23). If `initialized` is already 1, the contract execution fails.

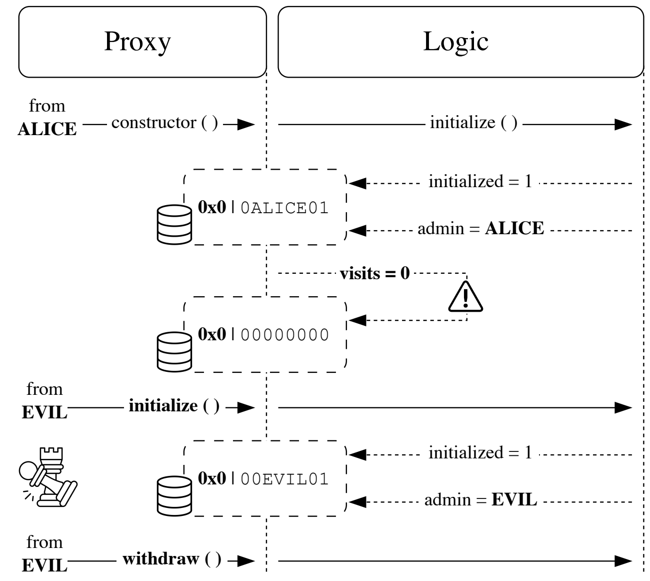


Fig. 3: Sequence of actions taken to exploit the storage collision vulnerability.

Otherwise, `initialized` is set to 1 and the variable `admin`, also stored in storage slot 0x0 (but packed at a different offset as explained in Section II) is set to the caller address, i.e., the address of the user that initiated the transaction. The `withdraw` function transfers all funds to the marketplace `admin`, and can only be called from the `admin` address.

When Alice's proxy contract performs a `DELEGATECALL` to her logic contract, the EVM runs the code of the logic contract using the storage of the proxy contract. Consequently, since both contracts work on the same storage, it is crucial to pay close attention to their storage layout, including the variables' types and positions.

In fact, a careful observer immediately notices that `visits`, `initialized`, and `admin` are all stored in the same storage slot, which is slot 0x0. The `Logic` and `Proxy` contracts interpret storage slot zero in two different ways, making the contract vulnerable to a storage collision. The constructor itself first writes `initialized` and `admin` (in the initialization routine of the `Logic` contract) and then immediately overwrites them with `visits=0` (Line 9). As a result, an attacker can now call `initialize` again, which reads the corrupted value and makes the attacker the marketplace `admin`. The attacker may then call `withdraw` and steal all the contract's funds. Figure 3 shows the sequence of actions carried out by the attacker to exploit this storage collision vulnerability.

In Appendix B, we present a real-world version of this attack, which is automatically identified by our system.

IV. Detection Approach

In the following section, we describe our approach to identifying contracts that suffer from storage collision vulnerabilities. As shown in Figure 4, our approach proceeds in three main stages: *component discovery*, *collision discovery*, and *vulnerability discovery*.

In the first stage, we analyze on-chain transactions to extract the source (proxy) and the target (logic) of each `DELEGATECALL`. Over time, a proxy contract may make use of several different logic contracts – perhaps even several simultaneously. Thus, we also study the activity window (*lifespan*) of each proxy and logic contract.

In the second stage, we infer the storage layout of each proxy and logic contract and detect conflicts between their layouts. That is, we first perform type analysis to determine the correspondence between variables – in the form of access masks – and storage slots. Then, we determine whether there are any collisions where two distinct contracts access the same storage slot but with different types.

Finally, in the third stage, we automatically assess the security impact of each detected collision and synthesize an exploit.

All the analyses presented in this section are built on top of the register-based Intermediate Representation (IR) provided by the Gigahorse framework [33], [35]. In particular, Gigahorse provides disassembling and lifting capabilities over a smart contract's bytecode, without the need for its source code. Moreover, Gigahorse provides out-of-the-box analysis results, such as the contract's control-flow-graph (CFG), which we leverage to implement our own analyses.

A. Component Discovery

We first analyze on-chain transactions and extract those that contain a `DELEGATECALL`. This allows us to discover proxy and logic contracts that live on the Ethereum blockchain. The main benefit of using on-chain transactions as the basis for finding `DELEGATECALL`s is that it greatly reduces the scope of our analysis and focuses it on contracts that actually make use of delegations. In a second step, we use lightweight static analysis techniques to determine the *lifespan* of each proxy and logic contract.

❶ **Proxy Detection.** For every observed on-chain `DELEGATECALL`, we extract the (source, destination) pair of interacting contracts. We represent all observed interactions in a graph, where nodes represent distinct contracts and a directed edge corresponds to a `DELEGATECALL` from the contract initiating the call to the contract being called.

We then extract from this graph all groups of contracts that share the same root node. We refer to the root contract as the proxy contract, and to all the contracts that share the same root node as its logic contracts. We refer to each group of proxy and corresponding logic contracts as a COMPONENT.

For example, consider a group of interacting contracts A, B, C, D, and E. Assume that the root contract A (the proxy):

1) delegates to B, then to C (in the same transaction).
2) delegates to B, then to D (in a different transaction).
3) delegates to B, which in turn delegates to E.

The graph for this COMPONENT would be:

$$\mathcal{G}=\{(A\rightarrow B),\ (A\rightarrow C),\ (A\rightarrow D),\ (B\rightarrow E)\}.$$

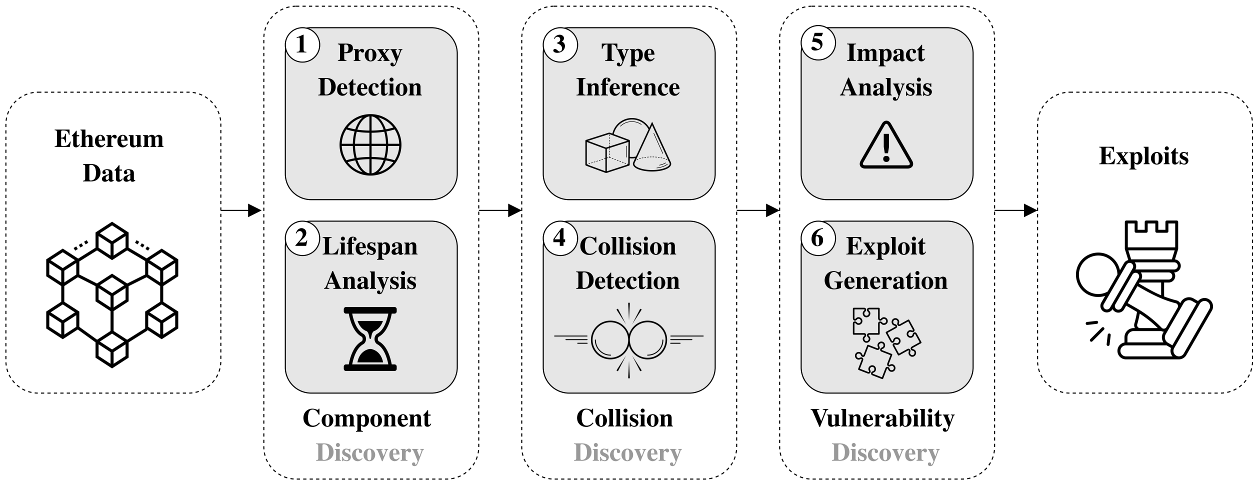


Fig. 4: Overview of our approach, implemented in CRUSH. The analysis pipeline follows the order of the circled numbers.

❷ **Lifespan Analysis.** We refer to the blocks between the creation of a proxy contract and a reference block [3] as the *lifespan* of the proxy. If the proxy contract was deleted at a block that precedes our reference block, we consider such block to be the end of the proxy lifespan.

Determining the lifespan of a logic contract is more complicated and requires the analysis of the corresponding `DELEGATECALL` in the proxy contract. This is necessary to determine if an upgrade has changed which logic contract is active. If an upgrade did occur, we adjust the lifespan of the logic contract accordingly. More specifically, we use a lightweight static analysis (backward slicing [57]) to extract all instructions that affect the target of the `DELEGATECALL`, and distinguish between three possible cases:

- When the target is **constant**, it cannot change over time. As a result, we consider a constant logic contract to be "active" for the entire lifespan of the proxy.
- When the target is read from a **storage slot**, we analyze how the value of that storage slot changes over time to infer the lifespan of the referenced contract. For instance, if the slot containing the target logic stores the address A at block X, and changes to the value B at block X+100, the lifespan of A is from X to X+100 – i.e., 100 blocks.
- When the target comes from an **external source**, for example, the transaction's calldata, we conservatively consider the corresponding logic "active" for the entire lifespan of the proxy.

In Figure 5, we exemplify a result of the Lifespan Analysis in the form of a timeline. In particular, we show the lifespan of a proxy contract A and its four corresponding logic contracts: B, C, D, and E.

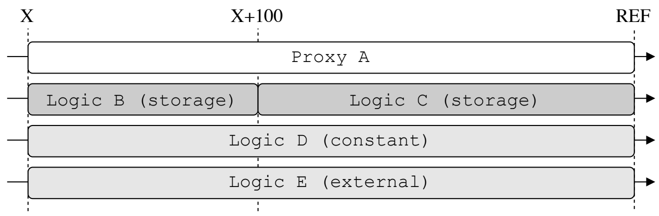


Fig. 5: Example of proxy and the corresponding logic contracts' timelines. The lifespan of the proxy is from block X to the reference block REF. In this example, the addresses of logic B and logic C are saved in a storage slot. In particular, the storage slot is updated at block X+100 from B to C. Logic contracts D and E are active for the entire lifespan of the proxy: The reason is that the address used to call D is a constant hardcoded in the proxy's code, while the address for E is obtained through the function's calldata.

B. Collision Discovery

We now proceed to identify storage collisions among the contracts within a COMPONENT. In particular, we first infer the storage layout of each of the contracts involved, and then search for type inconsistencies. In the following, we discuss our approach to type inference and collision detection.

Similar to other solutions that aim to detect smart contract vulnerabilities [30], [40], we base our analysis on symbolic execution of EVM bytecode. Specifically, we leverage a symbolic execution engine equivalent to those proposed in [40] and [30]. However, we choose to build our symbolic execution engine on top of Gigahorse [33]. This choice provides us with the flexibility of adjusting the engine's internals to the needs of our approach. Nonetheless, it is possible to implement our approach on top of any existing symbolic execution framework [15], [30].

❸ **Type Inference.** The goal of our type inference analysis is to understand the types (Section II-C) associated with each storage slot used by the contract's code. This analysis targets all storage access instructions (i.e., `SLOAD` and `SSTORE`) in a target contract. At a high level, this includes two steps: (1) we identify the target storage slot accessed by a given `SLOAD`/`SSTORE`, and (2) we identify the types of variables within the target slot.

[3]We choose as our reference block the latest block available at the time of writing (16976770).

Intuitively, we observe that while variable types are lost after compilation, such types can still be inferred from the contract's bytecode. To do that, we examine the compiler-generated access masks that are used to extract the separate variables packed within the same slot. More precisely, an access mask determines the exact subset of bytes that refer to each variable.

Our type inference analysis first needs to understand exactly which slots are used by a contract. To do this, we examine each `SLOAD` and `SSTORE` instruction and determine which target slot it accesses. Since the target slot value must be computed *before* a storage access operation can access it, we use backward slicing [57] to retrieve all the instructions involved in its computation. Then, we symbolically execute these instructions and examine the resulting symbolic variable. In particular, the target slot can be:

Constant. In this case, the solution for the constraints over the target slot variable is a constant. Furthermore, since this type of access is typical for fixed-size types, we learn that the slot contains one or more fixed-size type(s). While a simpler technique – for example, constant folding – would also allow us to infer a constant slot value, our symbolic-execution-based approach allows us to also handle the two following cases, which are more complex.

In the form "keccak256(slot)+index". In this case, the constraints over the target slot variable define it as the sum of a `keccak256` hash computation and an arbitrary `index`. Since this matches the access pattern of an array type as discussed in Section II-C, we conclude that the slot contains a dynamic array type.

In the form "keccak256(key.slot)". Similar to the previous case, the constraints over the target slot variable capture the computation of a `keccak256` over the concatenation of a generic key and the target slot. Since this matches the access pattern of a mapping type, we conclude that the slot contains a mapping.

Unknown. If the access type cannot be determined, the slot type remains unknown. We discuss the impact of type inference imprecisions in Section VI.

In the case of dynamic-size types (dynamic mapping and array), we do not need to examine the access mask since such types occupy one whole storage slot (32 bytes). On the other hand, for fixed-size types, multiple variables can end up in the same slot, and we must determine their *access masks*. For the sake of this discussion, in the following paragraphs, we describe our analysis for read accesses (`SLOAD`). An equivalent analysis is performed for write accesses (`SSTORE`).

To determine the access mask of an `SLOAD` instruction, we first compute a forward slice [57] and retrieve all the instructions that operate on the result of the `SLOAD`. We then symbolize all 32 individual bytes in the target storage slot and execute the instructions in the forward slice.

Consider the simplified (4-byte) example in Figure 6. The forward slice computed for the `SLOAD` instruction includes three opcodes: `SLOAD`, `RSHIFT`, and `AND`. First, we start by symbolizing the 4-byte value fetched by `SLOAD` – for the sake of this example, with the value `0xAABBCCDD` (Line 1). Then, we execute the instructions in the forward slice that operate over such symbolized value, and study their effect. For instance, in Line 2, when `0xAABBCCDD` is manipulated by the `RSHIFT` instruction – producing `0x00AABBCC` – we infer that the associated mask is `0xffffff00`. Similarly, in Line 3, the value `0x00AABBCC` is manipulated by the `AND` instruction, producing `0x0000BBCC`. As a result, we infer that the final mask for this storage access is `0x00ffff00`.

```
0  [INS]           [OUTPUT]              [MASK]
1  SLOAD   0x1     result:0xAABBCCDD     mask:0xffffffff
2  RSHIFT  0x1     result:0x00AABBCC     mask:0xffffff00
3  AND   0xffff    result:0x0000BBCC     mask:0x00ffff00
```

Fig. 6: Simplified example of masking in EVM bytecode. The three instructions represent the forward slice that manipulates the return value of the `SLOAD`. Every instruction that operates on the symbolized result of `SLOAD` (i.e., `0xAABBCCDD`) provides information regarding the mask used to extract the variable from slot 0x1.

We express the resulting access mask as $bytes_{nbytes}^{offset}$. The values of `nbytes` and `offset` indicate the number of masked bytes and their position in the mask. For example, we express the mask `0x00ffff00` as $bytes_{2}^{1}$.

❹ **Collision Detection.** In the previous step, we determined the storage layouts and access masks for all contracts. In this step, we perform a pairwise comparison between all contracts that are part of a COMPONENT, with the goal of detecting type inconsistencies. Note that a storage collision can happen between *any* two contracts that share the same underlying storage. Thus, a collision can happen not only between the logic contract and the proxy contract, but also between two logic contracts. For example, given a COMPONENT that comprises contracts (A, B, C), we analyze all possible pairs, e.g., (A, B), (A, C), and (B, C).

In particular, for each byte in each storage slot, we check whether the two contracts use different access masks. For example, if the least-significant byte in a storage slot is written as $bytes_{1}^{0}$ in one contract, and read as $bytes_{20}^{0}$ in the second contract, our analysis reports a potential storage collision. We refer to such potential storage collisions as *storage collision candidates*. We record these as a 3-tuple $(\mathrm{A}, \mathrm{B}, \mathrm{S})$ where A and B are the colliding contracts, and S is the colliding slot.

Data types and semantics. The previous paragraphs show how CRUSH can infer data types from a smart contract's bytecode and detect storage collisions. It is worth noting that for a collision to happen, it is sufficient – but *not necessary* – that the two variables have different data types. In fact, it is possible for the two variables to share the same data type, but have different semantics (e.g., an unsigned integer value might represent the total balance in one contract and the number of users in another one). While it is extremely challenging (and out of the scope of this paper) to infer the semantics of contract variables, we observe that one of the primary causes of semantic-type collisions is the "cascading" collision that arises from the introduction of a new state variable in a smart contract's source code. In our approach, we detect and leverage this misalignment to reveal semantic-type collisions.

For example, Figure 7 depicts the storage layout of the contract `Logic` before and after an upgrade.

```
contract Logic {
    bool initialized;        // storage slot [0x0]
    address admin;           // storage slot [0x0]
    array artworkIDs;        // storage slot [0x1]
    mapping artworkHolders; // storage slot [0x2]
    [..]
}

contract LogicUpgraded {
    uint visits;             // storage slot [0x0] ←
    bool initialized;        // storage slot [0x1] ↓
    address admin;           // storage slot [0x1] ↓
    array artworkIDs;        // storage slot [0x2] ↓
    mapping artworkHolders; // storage slot [0x3] ↓
    [..]
}
```

Fig. 7: Introducing a new state variable in a smart contract can result in multiple "cascading" collisions.

A developer might think that adding the state variable `visits` aligns the storage layout of `LogicUpgraded` with the storage layout of the `Proxy` contract (as shown in Figure 2). However, introducing the state variable `visits` has unintended consequences, and impacts the storage layout of all following state variables in `LogicUpgraded`. This introduces several collisions. For example, in the new contract `LogicUpgraded`, the variable `visits` collides with the variables `initialized` and `admin` of the original `Logic` contract. As a result, when `LogicUpgraded` attempts to read the variable `admin`, it is effectively reading the old variable `Logic.artworkIDs`, which was previously stored in Slot `0x1`. Additionally, the variables `initialized`, `admin`, `artworkIDs`, and `artworkHolders` are forced into a new slot, creating several other collisions between `Logic` and `LogicUpgraded`.

C. Vulnerability Discovery

Consider a tuple (A, B, S) where the contracts A and B are part of the same COMPONENT and have conflicting interpretations of storage slot S. We say that the smart contract B is in a vulnerable state if there exists a WRITE operation in contract A that can be followed by a USE operation in contract B, and both operations manipulate the same slot S. Intuitively, such a sequence of WRITE and USE operations results in the two contracts (A and B) accessing the same storage slot S with conflicting types.

To determine the impact of a given storage collision, we develop an automated technique that ❺ analyzes the impact of the colliding storage slot to discard collisions that are (likely) not security-relevant, and that ❻ analyzes whether the storage collision candidates are already in a vulnerable state, or can be brought into a vulnerable state with a WRITE-USE pair of operations.

❺ **Impact Analysis.** Among all the storage collision candidates detected in ❹, we identify those that have a likely impact on the contracts' security. In particular, we leverage the analysis proposed by Brent et al. [9] to identify storage slots that participate in access control decisions, i.e., that restrict specific functionality of the contract's code. We use the results of this analysis to label all such slots as *sensitive*. Then, we label in the same way *any* other storage slot that can be written *only* when executing a restricted section of the contract's code. This makes intuitive sense: if a slot can be written only in a portion of code that is reachable by specific users, this suggests that it contains important variables, hence, we consider it a *sensitive* slot. Furthermore, we label as sensitive (1) all the read-only slots, for example, the contract creator's address, and (2) the slots that store the size of a dynamic array.

The corruption of any sensitive slot can critically compromise the functionality of a smart contract. For example, overwriting the variable that identifies a contract's administrator compromises its access control policies, and corrupting the size of a dynamic array can result in an out-of-bound write. Finally, we study the colliding slots to understand whether any of them guards an explicit WRITE to a sensitive slot (for example, the `initialized` variable in Figure 2). We label a storage slot as a *guarding* slot when there exists a sensitive slot that can only be written *after* a comparison against the value of such guarding slot.

At the end of this stage, we discard all collision candidates that – based on the analysis outlined above – are unlikely to have a security impact.

❻ **Exploit Generation.** In this final analysis stage, CRUSH confirms whether it is possible to craft (or observe) a WRITE-USE pair of operations that brings the storage collision candidates into a vulnerable state. Given a collision candidate (A, B, S), CRUSH attempts to generate a transaction that WRITES the sensitive or guarding slot S with the type of contract A, and a subsequent transaction that USES S with the type of contract B. If successful, the target contracts (A and B) are reported as vulnerable.

Attack Preparation. The interactions with all contracts always happen through the proxy. Thus, if contract A (or B) is the proxy contract itself, it is fairly simple to craft a transaction that directly calls (and executes) the proxy's code. However, if contract A (or B) is a logic contract, we need to generate a transaction that reaches the relevant code in the logic contract *through* the proxy. As discussed in ❷, we know which `DELEGATECALL` instruction in the proxy allows us to call which particular logic contract. As a result, to interact with a logic contract, we simply execute the code in the proxy that reaches the corresponding `DELEGATECALL`, and then, we continue executing the code of the target logic contract.

WRITE. First, CRUSH attempts to WRITE the slot S in the context of contract A. We leverage symbolic execution to reach any `SSTORE` instruction in contract A that allows us to WRITE slot S. In fact, symbolic execution allows us to both verify the feasibility of such path and to automatically craft a concrete transaction that can reach such instruction. If this is unsuccessful, CRUSH inspects the historical on-chain WRITES to the target storage slot S and checks whether the most recent WRITE to the slot S happened in the context of contract A, i.e., the desired write already happened.

Sensitive USE. If slot S is a sensitive slot, CRUSH attempts to USE the slot in the context of contract B. We use symbolic execution to reach any `SLOAD` instruction in contract B that allows us to USE slot S.

Guarding USE. On the other hand, if slot S is a guarding slot, CRUSH attempts to WRITE any sensitive slot which is guarded by slot S (as described previously in ❺).

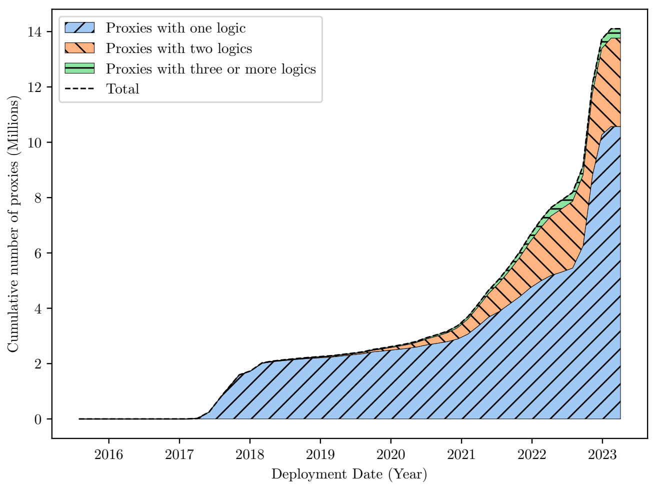


Fig. 8: Cumulative number of proxies deployed over time.

Exploit Synthesis & Verification. Finally, CRUSH combines the WRITE and USE operations discovered in the previous steps to generate a proof-of-concept exploit for the collision candidate. In particular, we solve the path constraints obtained from the symbolic execution of contracts A (WRITE) and B (USE) to obtain the concrete inputs (i.e., the calldata) that bring the contracts into a vulnerable state. Finally, we use a concrete implementation of the EVM [28] to verify that the generated exploits are reproducible on-chain in the lifespan of contract B. Replaying the generated attacks provides us with high confidence that our exploits could have successfully compromised the vulnerable contracts.

If all the above steps succeed, CRUSH reports a verified storage collision vulnerability.

V. EVALUATION

For all our experiments, we use one server equipped with 512GB of RAM and dual Intel(R) Xeon(R) Gold 6330 CPUs. We use GNU Parallel [52] to parallelize our tasks, and always limit each task to 60 minutes of CPU time and 20GB of RAM.

Dataset. We consider all Ethereum blocks (and contracts) from its genesis block (July 2015) up to and including block 16976770 (April 2023). Since there are almost 2 billion external transactions in the Ethereum network at the time of writing, inspecting and indexing them requires a substantial amount of time – approximately 60 days. We consider this as a necessary one-off preparation step for any work that aims to study the Ethereum blockchain. We rely on a local deployment of the `go-Ethereum` client [32] to access and inspect transactions in an efficient way.

A. Component Discovery

We run our on-chain Proxy Detection ❶ on top of `go-Ethereum`. This step takes approximately one hour. Similarly, we run the Lifespan Analysis ❷ on the discovered proxy contracts and observe a median execution time of 2.7 seconds per proxy. We observe a memory usage below 500MB.

❶ **Proxy Detection.** In the studied time window, we observe 53,580,899 deployed smart contracts. Out of those, we extract all the contracts that ever participate either as the source (proxy) or as the target (logic) of a `DELEGATECALL` instruction, 14,237,696 in total. In the following sections, we will always refer to this subset of contracts. As per our definition of proxy contract, we label 14,134,133 proxy contracts that are the source of a delegation. Similarly, we label 103,833 logic contracts that are the target of a delegation.

In Figure 8, we present the cumulative number of proxy contracts deployed over time, highlighting the number of associated logic contracts. The majority of proxy contracts interact with 3 or less logic contracts, while only very few contracts interact with more than 3 different logic contracts.

A careful reader will notice how the number of proxy and logic contracts does not sum up to exactly the number of relevant contracts. In fact, we observe that some of the contracts are used both as a proxy and as a logic contract. Although counter-intuitive, we believe it demonstrates that blockchain developers leverage composability, re-using the existing on-chain contracts as much as possible.

Results Discussion. Out of the 14,237,696 proxy and logic contracts identified by our analysis, 7,744,861 (54%) have source code available on Etherscan [23]. Interestingly, among the proxy contracts *with* available source, more than 90% call at least one logic contract *without* available source. This leaves us with only 84,283 (0.6%) of the interacting components that are fully open-sourced, justifying our choice of developing a bytecode-based analysis.

We further observe that only 93,879 proxy contracts and 51,667 logic contracts have distinct bytecode. This result suggests that many of these contracts are aggressively re-deployed. However, while many contracts might not have distinct bytecode, it is important to note that each separate deployment has a completely independent storage. Since the storage contents are critical for our analysis, we always consider the total number of identified proxy and logic contracts, including the ones with identical bytecode.

Finally, we measure the size of the proxy and logic contracts. As a fair unit of measure, we choose to use the number of basic blocks in the contract (as reported by Gigahorse). We present the contracts' sizes at the 25th, 50th, and 75th percentile in Table I. The low median size of proxy contracts (3 basic blocks) and the larger median size of logic contracts (147 basic blocks) confirm the basic assumption about proxy design pattern: proxy contracts are small because they only hold the storage, and the functionality is implemented in the logic contracts.

Contract Type	Size (25th)	Size (Median)	Size (75th)
Proxy	3	3	8
Logic	45	147	440

TABLE I: Contract size in number of basic blocks.

❷ **Lifespan Analysis.** As shown if Figure 9, we find that a majority of the deployed proxy contracts (12.5M, or 88%) use at least one **constant** target logic – which cannot change over time. Furthermore, 5.6M (39%) of the proxy contracts use a target logic address that comes from an **external source** – for

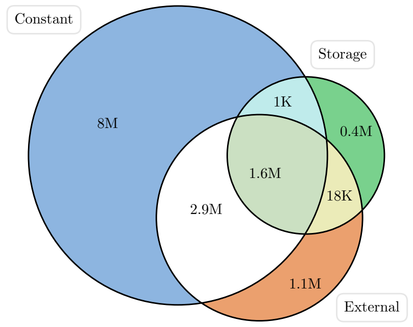


Fig. 9: Usage of target logic types among the proxy contracts.

Contract Type	Lifespan (25th)	Lifespan (Median)	Lifespan (75th)
Proxy	3	5	10
Proxy (upgraded)	10	21	28
Logic	2	4	9
Logic (upgraded)	0	1	7

TABLE II: Lifespan in months of all proxy and logic contracts.

example, the transaction's calldata, or an external contract – and 2M (14%) of the proxy contracts use a target logic that is read from a **storage slot**. Notably, a large number of proxy contracts employs two or more different types of target logics. Our analysis is robust against the type of target logics used by the proxy contract, and CRUSH can identify and exploit vulnerabilities across different target logic types.

We measure the lifespan of all proxy and logic contracts, and present our measurements in Table II. While the median lifespan of the proxy contracts is approximately 5 months, we find that the median lifespan of the associated target logics is slightly shorter – approximately 4 months. We also measure the lifespan of proxy and logic contracts that are involved in at least one upgrade. In this case, we find that the median lifespan of proxy and logic contracts is significantly different – 21 months and 1 month, respectively.

B. Collision Discovery

We run our Type Inference ❸ analysis on all the discovered proxy and logic contracts. We observe a median execution time of 1.8 seconds per contract. We measure the analysis time spent in the Collision Detection ❹ stage to be in the order of a fraction of a second per contract – or a few hours of cumulative execution time to analyze all the COMPONENTS. In fact, in this stage we leverage the results from ❸, and compare the inferred types to detect inconsistencies, with very little computational overhead.

❸ **Type Inference.** Although the aim of this work is not to provide a precise type analysis, we provide a comparison of our type inference with the state-of-the-art analysis from the Gigahorse framework [33]. To the best of our knowledge, the latest version of Gigahorse that is at our disposal (commit `59599ec`) does not explicitly show any type information. Nonetheless, as our best attempt to provide a fair comparison, we modify Gigahorse to expose additional internal analyses, including storage variable types.

As mentioned above, only a subset of the discovered proxy and logic contracts are open-source. As our source of

Tool	Correct Types
Gigahorse	425,740 (76.9%)
CRUSH	484,465 (87.3%)

TABLE III: Number (and percentage) of correct types reported by Gigahorse and CRUSH.

ground truth, we attempt to reproduce the compilation of all source-available contracts, and rely on the compiler's output to understand their storage layouts. In this way, we build a ground truth of more than 50 thousand contracts and more than 500 thousand storage slot types. We then analyze all such contracts both with Gigahorse and with our analysis, and compare their results in Table III. Gigahorse recognizes 425,740 (76.9%) of the types correctly. Notably, CRUSH shows a significant improvement and correctly identifies 484,465 (87.3%) types.

Results Discussion. We find that the key reason for the difference in the performance of CRUSH and Gigahorse is that Gigahorse's type analysis is rule-based and only recognizes access patterns that the developers explicitly encode. For example, incomplete rules cause Gigahorse to mistype packed variables that are updated simultaneously. Instead, CRUSH uses symbolic execution to track the casting operations and does not need to introduce ad-hoc rules.

❹ **Collision Detection.** After inferring the types of all the proxy and logic contracts, we run our collision detection analysis. CRUSH uncovers 15,092 different contracts with a potential storage collision. We refer to such collisions as "candidates" since they do not yet account for either the security impact of the vulnerability (❺) or the blockchain state (❻). More specifically, CRUSH uncovers 46,403 collisions candidates over a total of 15,092 contracts. We further (automatically) dissect these results, and find that 4,877 (11%) of the detected collision candidates arise from a cascading collision.

To investigate false positives among the collision candidates detected by CRUSH, we perform the following manual analysis. We randomly sample 50 proxy addresses with at least one storage collision and arbitrarily select one such storage collision per proxy for manual inspection. We observe that in 30 cases, the inferred types were correct and the storage collision was a true positive. In the remaining 20, we observe an incorrect type inference that caused a false positive in our collision detection. We elaborate on the causes – and impact – of these type inference imprecisions in Section VI.

Results Discussion. As discussed in Section IV, a collision between a fixed-sized type and the slot `BASE` of a mapping type does not have any security impact – since any value written in the slot BASE does not fundamentally change the behavior of the mapping. As a result, we ignored 79,748 simple collisions on mapping types. On the other hand, the security impact of a displacement of the BASE slot – e.g., resulting from a cascading collision – is more subtle and dependent on the contract's logic. We observe 428 such collisions on mapping types. We elaborate on these scenarios in Section VI.

C. Vulnerability Discovery

We run our Impact Analysis ❺ and Exploit Generation ❻ on all the discovered collisions candidates. We observe a median execution time of 25 seconds per collision, and

Code Hash (Anonymized)	Financial Impact (USD)	Previously Reported	Currently Exploitable
b29f †	6M	Yes	No
cfa7	4.6M	No	No
295b	1.1M	No	No
Others	242k	No	Yes

TABLE IV: Overview of several attacks automatically generated by CRUSH, highlighting those previously unreported. † is the attack against the AUDIUS protocol [3].

memory use below 1GB. Out of the 15,092 contracts with collision candidates, we could not analyze 660. In particular, 588 of them timed out (60 minutes), and 72 exceeded our memory limit (20GB of RAM).

❺ **Impact Analysis.** Among the 46,403 collisions discovered in ❹, CRUSH discards 6,508 (14%) because either the analysis fails or the collision does not impact any sensitive slot. Out of the remaining ones, we observe that 39,042 (98%) affect a sensitive slot, while 853 (2%) affect a guarding slot. This leaves us with a total of 39,895 collisions over 14,891 contracts.

❻ **Exploit Generation.** CRUSH analyzes the exploitability of all collisions that are possibly security relevant and attempts to automatically synthesize an attack. We present the results of our automatic exploit generation in Table V.

Impact	Exploitable Collisions	Exploitable Contracts
Sensitive USE	7,759	878
Guarding USE	424	143

TABLE V: Number of exploitable collisions and contracts. We separately report the exploits based on their impact.

In total, CRUSH successfully generates an end-to-end exploit for 956 contracts. More specifically, CRUSH successfully exploits 7,759 sensitive collisions (WRITE → Sensitive USE) in 878 contracts, and 424 guarding collisions (WRITE → Guarding USE) in 143 contracts. Note that the same contract may be exploited both by affecting a sensitive slot and a guarding slot.

We show a few examples of such automatically synthesized attacks in Appendix A and Appendix B. These examples are anonymized to prevent precise contract identification since the contracts are still vulnerable at the time of writing.

Results Discussion. In Table IV, we show a breakdown of the most impactful, manually-verified [4], exploits generated by CRUSH, sorted by financial impact. Most of these attacks are new and previously unreported, others (i.e., the AUDIUS attack [3]) were instead previously reported. To calculate the financial damage of an exploit, we determine all assets that can be stolen or lost – i.e., Ether and (non)-fungible tokens – at the latest exploitable block, and then convert their value to USD, considering the historical conversion rates.

CRUSH uncovers at least \$6 million of *novel, previously unreported* financial damage that results from the exploitation of a storage collision vulnerability. We present the distribution of exploitable contracts over time in Figure 10. Since 2018, we observe a linear growth in the number of vulnerable contracts. Coincidentally, OpenZeppelin's proxy pattern [45] was indeed introduced in 2018, prompting a growth in the adoption of such a design pattern. We report that at least 132 out of the 956 contracts with a working exploit are *still exploitable* at the time of writing, while the remaining 768 have been exploitable in the past. [5]

To investigate false positives among the exploits generated by CRUSH, we study a sample of 165 of the exploits that have been automatically generated. From this analysis, we were able to confirm at least 60 exploits with a clear security impact – e.g., denial of service or theft of funds – and at least 70 exploits with a possible security impact – i.e., that allow the modification of a sensitive storage slot. Finally, we identified 35 false positives – of which 20 due to imprecisions in our Type Inference ❸ (we expand on this in Section VI) and 15 due to imprecisions in our Impact Detection ❺.

D. Comparison with Existing Systems

We compare CRUSH's collision detection module (❹) with the state-of-the-art tool for collision detection, USCHUNT [7], and report the results in Table VI.

Tool	TP	FP	FN
USCHUNT	21	22	81
CRUSH	98	40	4

TABLE VI: True Positives (TP), False Positives (FP), and False Negatives (FN) resulting from the comparison of CRUSH with USCHUNT.

USCHUNT does not support an end-to-end detection and validation of storage collision vulnerabilities, and instead leverages a source-code-only analysis implemented in Slither [24] that checks if storage variables 1) appear in the same order and 2) have the same names. For a fair comparison, we run our analysis on USCHUNT's dataset – comprising 5,335 proxy contracts with source code available – and show that CRUSH implements a more precise analysis that detects more storage collisions.

On this dataset, CRUSH identifies 138 collisions and USCHUNT detects 43 collisions. We manually investigate these collisions to establish the ground truth, and find that 102 of them are true collisions, out of which CRUSH identified 98 and USCHUNT identified 21.

Thus, we report 40 false positives and 4 false negatives for CRUSH. All the false positives observed are the result of an incorrect type inference – in the majority of the cases caused by

[4]We estimate the manual effort involved in the verification of the exploits to be in the order of a few minutes per exploit. Since the exploit generation is fully automated, the human analyst must verify that the exploit is security relevant.

[5]Our exploit generation establishes whether a contract is currently exploitable (at the reference block) or has been exploitable before (but not any longer). A contract is no longer exploitable when the logic has been upgraded, the contract is corrupted and inoperable, or the new logic has overwritten the corrupted slot with the correct value.

non-standard access patterns such as inline assembly. One of the false negatives is a semantic-type collision easily detectable only in the contracts' source code via a corresponding variable name's change. Each of the remaining false negatives is a consequence of our online proxy detection strategy, which does not consider proxy contracts without any interaction.

Among the collisions detected by USCHUNT, we report 22 false positives and 81 false negatives. We identify four core reasons for the false positives in USCHUNT. First, 5 of the false positives result from its wrong understanding of variable packing and storage layout – USCHUNT detects any change in variables' alignment in the source code as a storage collision, ignoring the fact that the new variable could be safely packed into an existing slot, or never even allocated if left unused. Second, 7 of the false positives result from the variable names being intentionally different (e.g., `logic` versus `_logic`) but preserving type and semantics. Third, 6 of the false positives result from an intentionally inserted storage gap (e.g., 50 unused storage slots). Finally, in 4 cases, the contract is incorrectly identified as a proxy and does not implement any delegation functionality.

Finally, the three core reasons for the higher number of false negatives in USCHUNT are:

- To identify proxy and logic contracts, USCHUNT relies on Etherscan labels [22]. However, Etherscan labels are not always reliable, and we find that 40 of the 81 false negatives in USCHUNT are caused by a failure in identifying one or more of the logic contracts.
- While CRUSH detects collisions based on type inference and variable layout reconstruction (Section IV-B), USCHUNT approximates the storage layout based on variable names and leverages an existing solution, Slither [24], to perform static analysis on the contracts' source code. This design choice creates many false negatives. In fact, we find that 41 of the 81 false negatives in USCHUNT are caused by a failure in Slither's static analysis.
- Since USCHUNT relies on the availability of source code, it cannot detect a storage collision when one or more of the contracts involved are closed-source. While this is not the direct cause of any of the 81 observed false negatives, we find that in many cases even fixing the original problem would not help, since one or more of the logic contracts are closed-source.

E. Ethical Concerns

To prevent any damage to the contracts identified as vulnerable, we anonymize the reported results, and we do not disclose any identifying information in this submission. Furthermore, all the discovered exploits have been verified exclusively on our private blockchain fork, and never on the public Ethereum mainnet. We strongly believe that the role of responsible disclosure is key to fostering a safer blockchain ecosystem. Therefore, we made an effort to report our findings to the developers of all the affected smart contracts, and are currently awaiting acknowledgment. However, since this was not always possible, we also reported the confirmed vulnerabilities to the Cybersecurity and Infrastructure Security Agency [12].

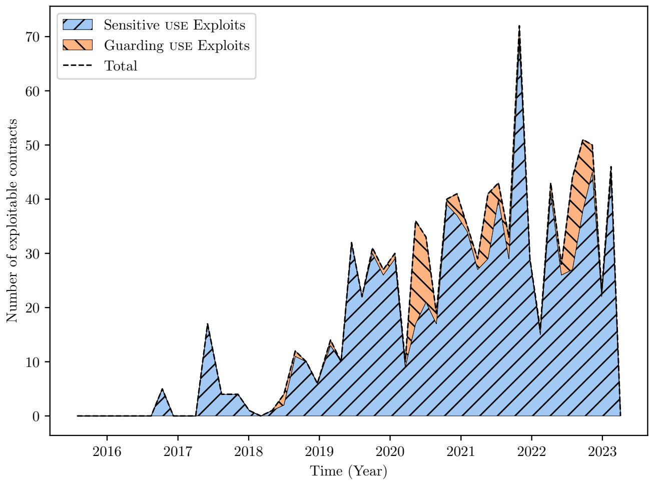


Fig. 10: Number of reported exploitable contracts over time.

VI. DISCUSSION

External Target. For our Component Discovery Analysis (❶/❷) presented in Section IV-A, we need to understand the target of each `DELEGATECALL`. Currently, when the target address is not defined in the code of the contract, or saved in one of its storage slots, we process it as an *external target*. An external target means that the address is provided for example within the calldata of a user's transaction or within the return value of a call to another contract (i.e., an oracle contract). For simplicity, in this work, we fully model only the former case. While this means that we could potentially miss further elements in a group of interacting contracts (i.e., COMPONENT), and thus, potential attacks, our results (Section V) shows how our approach is already effective in finding hundreds of attacks.

Type Inference Imprecision. Our Type Inference Analysis ❸ compares favourably to the state-of-the-art in our evaluation (Figure III), but is not always precise. This imprecision in understanding the high-level types from their low-level representation can produce both false positives and false negatives for the following analyses, e.g., the Collision Detection Analysis ❹ discussed in Section IV-B. We investigate these imprecisions and observe that they are caused by either handwritten low-level assembly code, compiler optimizations, or the use of non-standard access patterns. In all such cases, it is possible to design ad-hoc heuristics to improve the precision of our type inference.

Cascading Collisions & Mappings. As explained in Section II-C, the content of the `BASE` slot of a mapping variable is left uninitialized (i.e., STORAGE[`BASE`]=`0x0`) and the value `BASE` is used *only* to calculate the storage offset of the elements in the mapping (i.e., an element with key: `KEY` is stored at the offset `keccak256(KEY.BASE)`). Any value written in the slot `BASE` does not fundamentally change the behavior of the mapping. Thus, any collision that involves the `BASE` slot of a mapping does not usually have security implications. That being said, a separate case should be made when a mapping variable's `BASE` slot is displaced in a cascading collision (Figure 7). In this case, the `BASE` slot changes its value, afflicting the computation of the storage offset for *every* mapping element. In most cases, all the elements in the map would be now pointing to uninitialized values (i.e., `0x0`), however, it is theoretically possible to instead achieve arbitrary memory write/read over other critical variables in the storage. Triaging this logical vulnerability requires a significant manual effort, therefore, we left it out of scope in our Vulnerability Discovery (❺/❻) and consider it as a future work.

VII. Related Work

Proxy Discovery and Storage Collision. A work by Bodell et al. [7] identifies smart contract proxies leveraging the source code uploaded on platforms like Etherscan [23]. In particular, a contract is identified as a proxy if it has a fallback function in its source containing a `DELEGATECALL`. Our Logic Discovery analysis (❶/❷) is also able to recognize proxies on-chain, but, rather than leveraging insights from the contracts' code, we can identify a proxy-like behavior by observing the type of contracts' interactions. As our definition is by design more generic, our system is able to identify a larger number of proxies than [7]. We also observe that only half of the proxies that we identify are open-source. Moreover, as opposed to [7], we do not only detect proxy contracts but also uncover their corresponding logic contracts. Surprisingly, the aforementioned work estimates the impact of storage collision as negligible for real on-chain contracts, which does not seem to be the case according to our evaluation.

Another work [31], mainly focused on studying the impact of the `CREATE2` [59] instruction on the Ethereum smart contracts ecosystem, includes a discussion about proxies and smart contracts upgradability issues. Finally, Salehi et al. [49] investigated the upgradability patterns used on-chain and performed a large-scale analysis of the access-control mechanisms used to manage the upgrade of smart contracts. We consider the two latter works as orthogonal to ours as we are instead investigating storage collision vulnerabilities, which are not covered by any of the aforementioned studies.

Type Inference. SigRec [11] presents an automatic method for function signature recovery, which includes the identification of the function arguments' types via analyzing how the `CALLDATA` is processed by the contract's code. However, as the primary scope of SigRec is to extract the function signatures of smart contracts with unknown ABIs, we consider it orthogonal to the work presented in this paper.

Our Type Inference analysis (❸) operates on the register-based intermediate representation of the originally stack-based bytecode provided by the Gigahorse framework [33], [35], which drives several efficient vulnerability discovery efforts [9], [34], [41], [51]. While Gigahorse can be employed out-of-the-box to recover variables' types in a contract binary, we demonstrated in Section V (Table III) how our symbolic analysis is able to provide more precise results.

Storage Layout. The work by Ayub et al. [4] leverages a source code analysis to automatically recover a smart contract's storage layout. That is, given a high-level type, its low-level layout is identified with the help of the Solidity compiler's rules. Our approach also encompasses an understanding of storage layout, however, the layout is recovered from the smart contract's bytecode rather than its source. Rodler et al. [48] also discusses the criticality of compatible storage layouts when performing code upgrades. However, their work is focused on automatically hardening smart contract functionality against common errors such as integer overflows and access control bugs, which is different from the goal of Crush.

"Classic" Smart Contract Attacks. In the realm of analyzing smart contracts, several approaches have been proposed by researchers for "classic" smart contract attacks. For example, Tsankov et al. developed Securify [56], which utilizes a smart contract's dependency graph to identify vulnerability patterns. Another approach, proposed by Ma et al. [43], performs symbolic analysis to explore inter-contract control-flow graphs and detect classic bugs like re-entrancy and arithmetic issues. Similarly, Liao et al. presented SmartDagger [42], a static analysis framework that combines data-flow analyses and optimization techniques to uncover inter-contract vulnerabilities. Ye et al. introduced Clairvoyance [60], a static analysis tool that constructs a cross-contract control-flow graph based on the contracts' source code to identify candidate critical paths for re-entrancy bug exploitation. Xue et al. proposed xFuzz [61], employing machine learning to filter benign cross-contract execution paths and enhance fuzzing efficiency. Unlike these works, our approach, Crush, is not centered on the analysis of classic vulnerabilities, like re-entrancy. Instead, our system targets storage collisions: a cross-contract vulnerability that arises when specific circumstances are met. To the best of our knowledge, storage collision vulnerabilities are not detected by any of these state-of-the-art systems.

Exploit Generation. Other works [25], [30], [38]–[40], [62] focus on automatic exploit generation, i.e., synthesizing the input (transactions) needed to exploit a vulnerable smart contract's code. However, differently from our work, all the aforementioned papers focus on generating attacks for "classic" vulnerabilities such as integer overflow, re-entrancy, controllable calls, and batch-overflow [46].

VIII. Conclusions

In this paper, we study a widely under-investigated cross-contract vulnerability known as a *storage collision*, which occurs when a group of smart contracts operates over the same underlying storage data with a different interpretation of its types and/or semantics. When successfully exploited, this vulnerability can result in unexpected behaviors such as denial of service, privilege escalation, and direct theft of financial assets in a smart contract. To identify storage collision vulnerabilities at scale, we proposed Crush, a novel system that we use to analyze 14,237,696 contracts deployed since the beginning of the Ethereum network operations. Crush identified a total of 14,891 potentially vulnerable contracts and automatically synthesized 956 exploits. We estimated at least $6 million of *novel*, *previously unreported* potential financial damage uncovered by our system. Our work highlights the importance of thorough security analysis of smart contract interactions to improve the resilience of the DeFi ecosystem.

Acknowledgments

We want to thank the anonymous reviewers for their valuable feedback. This material is based upon work supported by the Google Security, Privacy, and Anti-Abuse Award. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of Google.

References

[1] 1inch. 1inch. https://1inch.io, 2023.

[2] 1inch. 1inch introduces Chi Gastoken. https://blog.1inch.io/1inch-introduces-chi-gastoken, 2023.

[3] Inc. Audius. Audius. https://audius.co, 2023.

[4] Maha Ayub, Tania Saleem, Muhammad Umar Janjua, and Talha Ahmad. Storage state analysis and extraction of ethereum blockchain smart contracts. *ACM Transactions on Software Engineering and Methodology*, 2022.

[5] Kim Barrett, Bob Cassels, Paul Haahr, David A Moon, Keith Playford, and P Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 69–82, 1996.

[6] blockworks. Ethereum market cap. https://blockworks.co/price/eth, 2023.

[7] William E Bodell III, Sajad Meisami, and Yue Duan. Proxy hunting: understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1829–1846, 2023.

[8] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 161–178. IEEE, 2022.

[9] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 454–469, 2020.

[10] Chainlink. Chain oracle. https://chain.link/education/blockchain-oracles, 2023.

[11] Ting Chen, Zihao Li, Xiapu Luo, Xiaofeng Wang, Ting Wang, Zheyuan He, Kezhao Fang, Yufei Zhang, Hang Zhu, Hongwei Li, et al. Sigrec: Automatic recovery of function signatures in smart contracts. *IEEE Transactions on Software Engineering*, 48(8):3066–3086, 2021.

[12] CISA. Cybersecurity and infrastructure security agency. https://www.cisa.gov, 2023.

[13] Coinmarketcap. Btc. https://coinmarketcap.com/currencies/bitcoin/, 2023.

[14] Cointelegraph. Decentralized exchange. https://cointelegraph.com/learn/what-are-decentralized-exchanges-and-how-do-dexs-work, 2023.

[15] Consensys. Consensys mythx. https://mythx.io, 2023.

[16] Consensys. Fallback functions. https://consensys.github.io/smart-contract-best-practices/development-recommendations/solidity-specific/fallback-functions/, 2023.

[17] Decrypt.com. 'audits are not bulletproof': How audius was hacked for 6m usd in ethereum tokens. https://decrypt.co/105913/how-audius-was-hacked-6m-ethereum-tokens, 2022.

[18] Ethereum. Eip-7: Delegatecall. https://eips.ethereum.org/EIPS/eip-7, 2015.

[19] Ethereum. Decentralized finance (defi). https://ethereum.org/en/defi/, 2022.

[20] Ethereum. Ethereum. https://ethereum.org/en/, 2022.

[21] Ethereum. What is eth? https://ethereum.org/en/eth/, 2022.

[22] Etherscan. ContractChecker. https://etherscan.io/proxyContractChecker, 2023.

[23] Etherscan. The ethereum blockchain explorer. https://etherscan.io/, 2023.

[24] Josselin Feist, Gustavo Grieco, and Alex 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)*, pages 8–15. IEEE, 2019.

[25] Yu Feng, Emina Torlak, and Rastislav Bodik. Precise attack synthesis for smart contracts. *arXiv preprint arXiv:1902.06067*, 2019.

[26] Ethereum Foundation. Erc-1967: Proxy storage slots. https://eips.ethereum.org/EIPS/eip-1967, 2019.

[27] Ethereum Foundation. Erc-3156: Flash loans. https://eips.ethereum.org/EIPS/eip-3156, 2020.

[28] Ethereum Foundation. Py-evm. https://github.com/ethereum/py-evm, 2023.

[29] Python Software Foundation. Python language reference. https://python.org, 2023.

[30] Joel Frank, Cornelius Aschermann, and Thorsten Holz. Ethbmc: A bounded model checker for smart contracts. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 2757–2774, 2020.

[31] Michael Fröwis and Rainer Böhme. Not all code are create2 equal. In *6th Workshop on Trusted Smart Contracts (WTSC'22)*, 2022.

[32] geth.ethereum. go-ethereum. https://geth.ethereum.org/, 2023.

[33] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186. IEEE, 2019.

[34] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Analyzing the out-of-gas world of smart contracts. *Communications of the ACM*, 63(10):87–95, 2020.

[35] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. Elipmoc: advanced decompilation of ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27, 2022.

[36] Fabio Gritti, Nicola Ruaro, Robert McLaughlin, Priyanka Bose, Dipanjan Das, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. Confusum contractum: confused deputy vulnerabilities in ethereum smart contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1793–1810, 2023.

[37] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.

[38] Bo Jiang, Ye Liu, and Wing Kwong Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269, 2018.

[39] Ling Jin, Yinzhi Cao, Yan Chen, Di Zhang, and Simone Campanoni. Exgen: Cross-platform, automated exploit generation for smart contract vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[40] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, 2018.

[41] Sifis Lagouvardos, Neville Grech, Ilias Tsatiris, and Yannis Smaragdakis. Precise static modeling of ethereum "memory". *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–26, 2020.

[42] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. Smartdagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 752–764, 2022.

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

[44] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, page 21260, 2008.

[45] OpenZeppelin. Proxy pattern. https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies, 2023.

[46] Peckshield. New batchoverflow bug in multiple erc20 smart contracts (cve-2018–10299). https://peckshield.medium.com/alert-new-batchoverflow-bug-in-multiple-erc20-smart-contracts-cve-2018-10299-511067db6536, 2018.

[47] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*, pages 1661–1677. IEEE, 2020.

[48] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Evmpatch: Timely and automated patching of ethereum smart contracts. In *USENIX Security Symposium*, pages 1289–1306, 2021.

[49] Mehdi Salehi, Jeremy Clark, and Mohammad Mannan. Not so immutable: Upgradeability of smart contracts on ethereum. *arXiv preprint arXiv:2206.00716*, 2022.

[50] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 621–640, 2020.

[51] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. Symbolic value-flow static analysis: deep, precise, complete modeling of ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[52] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[53] Solidity Team. Solidity. https://soliditylang.org, 2022.

[54] Vyper Team. Vyper. https://vyperlang.org, 2022.

[55] Truffle. Truffle suite. https://trufflesuite.com, 2023.

[56] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.

[57] Wikipedia. Program slicing. https://en.wikipedia.org/wiki/Program_slicing, 2023.

[58] Wikipedia. Sha-3. https://en.wikipedia.org/wiki/SHA-3, 2023.

[59] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[60] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1029–1040, 2020.

[61] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[62] Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. Ethploit: From fuzzing to efficient exploit generation against smart contracts. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 116–126. IEEE, 2020.

[63] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. Sok: Decentralized finance (defi) attacks. *Cryptology ePrint Archive*, 2022.

Appendix A
Denial of Service

In the following section, we describe one of the attacks automatically synthesized by Crush. The attack involves two smart contracts and leads to the critical compromise – and potential denial of service – of such contracts.

The `ChiProxy` contract (anonymized code hash: `e961`), presented in Figure 11, allows one to forward user interactions to the respective implementation (through the `fallback` function at Line 22) and uses a `wallet` address to provide and keep track of the consumed gas tokens. The proxy also allows the owner to withdraw funds from the contract through the functions `withdrawETH` (Line 12) and `withdrawToken` (Line 15) – restricted to the owner with the `onlyOwner` modifier.

The CHI gas token [2] contract (or `ChiToken`, with anonymized code hash: `e9da`) is a mechanism designed by 1inch [1] to save on gas costs. For the sake of this example, it is not necessary to understand the functionality of the gas token, thus, we refer the interested reader to the official documentation [2] and only show the relevant part of its source code in Figure 11. In particular, it is critical to note that both `ChiProxy` and `ChiToken` contracts store some of their variables sequentially starting from the storage slot zero (Lines 4-6 and Lines 33-35 respectively).

The developers deploy the proxy contract `ChiProxy` and point its implementation `logic` to the `ChiToken` contract. Similarly to the previous example discussed in Appendix B, the implementation's code (executed by the `DELEGATECALL` on Line 24) operates on the same storage as the proxy, creating a storage collision between the proxy variables (`logic`, `wallet`, and `owner`) and the implementation variables (`_-`

```
1  import "StorageSlot.sol";
2
3  contract ChiProxy {
4      address logic;
5      address wallet;
6      address owner;
7      constructor(address _logic, address _wallet)
           payable {
8          logic = _logic;
9          wallet = _wallet;
10         owner = msg.sender;
11     }
12     function withdrawETH() public onlyOwner {
13         msg.sender.call().value(this.balance);
14     }
15     function withdrawToken(address token_contract)
           public onlyOwner {
16         uint256 balance_proxy =
               token_contract.balanceOf(this);
17         address(token_contract).transfer(msg.sender,
               balance_proxy);
18     }
19     function setLogicContract(address _logic) public
           onlyOwner {
20         logic = _logic;
21     }
22     fallback() external payable virtual {
23         calldatacopy(0, 0, calldatasize());
24         delegatecall(gas(), logic, 0, calldatasize(),
               0, 0);
25         balance_wallet = CHI_TOKEN.balanceOf(wallet);
26         if (balance_wallet > 0) {
27             CHI_TOKEN.freeFromUpTo(wallet, <CHI_GAS>);
28         }
29     }
30 }
31
32 contract ChiProxy2 {
33     address owner;
34     address logic;
35     address wallet;
36     [..]
37 }
38
39 contract ChiToken {
40     mapping(address => uint256) private _balances;
41     mapping(address => mapping(address => uint256))
           private _allowances;
42     uint256 public totalMinted;
43     uint256 public totalBurned;
44     [..]
45     function mint(uint256 value) public {
46         [..]
47         totalMinted = totalMinted + value;
48     }
49 }
```

Fig. 11: Simplified Solidity code for the `ChiProxy`, `ChiProxy2`, and `ChiToken` contracts.

balances, _allowances, and totalMinted). Since – as discussed in Section VI – the BASE slot of a mapping variable remains unused, the collisions with the variables _balances and _allowances do not result in a vulnerability. However, any statement in the implementation that updates the totalMinted variable effectively overwrites the proxy's owner address.

In our – automatically discovered – attack scenario, an attacker can then call the mint function (Line 45) in the implementation to increment the value of totalMinted by a specified amount – therefore incrementing the proxy's owner address by such a specified amount. In the general case, this interaction will overwrite the proxy's owner address and brick the proxy contract, breaking any functionality that is restricted by the onlyOwner modifier. However, it is theoretically possible for a sufficiently motivated attacker to manipulate such value and point the owner address to an attacker-controlled address.

To fix this vulnerability, the developers must modify the implementation contract to use a storage padding and prevent the storage variables from overlapping, and then use the setLogicContract (Line 19) functionality in the proxy to upgrade the implementation. Unfortunately, after the vulnerability is exploited, there is no way to upgrade the implementation since the attack corrupts the value of the proxy's owner address, therefore breaking any onlyOwner-restricted functionality.

We observe a variation of the ChiProxy contract – ChiProxy2 (with anonymized code hash: ec3f) – also presented in Figure 11 (Line 32). In this case, the variables of the contract are reordered so that the storage collision happens between the proxy's wallet address and the implementation's totalMinted variable. As a result, any statement in the implementation that updates the totalMinted variable effectively overwrites the proxy's wallet address. As opposed to the previous example, in this case, an attacker can manipulate (i.e., increment) the value of the wallet address through the mint function in the implementation. However, while manipulating the owner and logic variables has clear security implications (i.e., loss of funds or denial of service), the manipulation of the wallet address is not as critical and might result in a temporary disruption of the contract functionality.

Appendix B
Theft of Funds

The following section describes another attack automatically synthesized by Crush. This attack involves two NFT-related contracts and leads to the critical compromise – and potential theft of funds – of such contracts. Figure 12 shows the simplified Solidity code for the two contracts NFTParentProxy (anonymized code hash: a2fc) and NFTParentImpl (anonymized code hash: 6392). The former contract implements a simple proxy to the latter.

The code of the proxy allows to forward any user interaction to the implementation (see the fallback function on Line 22) and to upgrade such implementation if needed (see the upgradeTo function on Line 42). The upgrade functionality is restricted to the owner of the contract (via the onlyOwner modifier). The implementation address is properly stored at

```
1  import "Address.sol";
2  import "StorageSlot.sol";
3
4  abstract contract Ownable {
5      address private _owner;
6      modifier onlyOwner() {
7          require(_owner == msg.sender, "Ownable: caller
              is not the owner");
8          _;
9      } [..]
10 }
11
12 abstract contract Proxy {
13     constructor(address _logic) payable {
14         _upgradeTo(_logic);
15     }
16     function _upgradeTo(address _logic) internal {
17         uint256 SLOT = uint256(keccak256("eip1967[..]"));
18         bytes _data =
               abi.encodeWithSignature("initialize()")
19         StorageSlot.getAddressSlot(SLOT).value = _logic;
20         Address.functionDelegateCall(_logic, _data);
21     }
22     fallback() external payable virtual {
23         calldatacopy(0, 0, calldatasize());
24         uint256 SLOT = uint256(keccak256("eip1967[..]"));
25         address impl =
               StorageSlot.getAddressSlot(SLOT).value;
26         delegatecall(gas(), impl, 0, calldatasize(), 0,
               0);
27         [..]
28     } [..]
29 }
30
31 abstract contract Initializable {
32     bool private _initialized;
33     bool private _initializing;
34     [..]
35 }
36
37 contract NFTParentProxy is Proxy, Ownable {
38     constructor(address _logic) payable {
39         Proxy(_logic);
40         owner = msg.sender;
41     }
42     function upgradeTo(address _logic) public onlyOwner
            {
43         _upgradeTo(_logic);
44     }
45 }
46
47 contract NFTParentImpl is Initializable {
48     uint256[50] private __gap;
49     [..]
50     address private admin;
51     modifier onlyAdmin() {
52         require(admin == msg.sender, "Caller is not the
               admin");
53         _;
54     }
55     function initialize() external {
56         [..]
57         admin = msg.sender;
58     }
59     function mintNFT([..]) public onlyAdmin {
60         [..]
61     }
62 }
```

Fig. 12: Simplified Solidity code for the NFTParentProxy and NFTParentImpl contracts.

a high offset to avoid collisions. The proxy also stores the variable owner at storage offset zero (Line 5 in the contract Ownable, inherited by the proxy).

The code of the implementation allows minting a new NFT, and provides some additional functionality (see the initialize function on Line 55) to initialize the contract and set the administrator address. The minting functionality `mintNFT` (Line 59) is restricted to the administrator of the contract (via the `onlyAdmin` modifier).

The developers were careful in designing the implementation storage and used a padding of 50 slots (Line 48) to avoid collisions with the proxy contract. As a result, the implementation stores the variable `admin` at storage offset 50 – which is not used in proxy, and thus, collision-free. However, the developers did not consider that the contract `Initializable` (Line 31) – inherited by the implementation – also has two storage variables: `initialized` and `initializing`. For the sake of this example, we will not explain how these variables are used, but it is important to note that such variables are stored at storage offset zero.

When the contracts are deployed, `NFTParentProxy` in its constructor performs a `DELEGATECALL` to `NFTParentImpl.initialize()` (the constructor call at Line 39 ultimately triggers the execution of the code at Line 20) and then sets the proxy's owner address at Line 40. Importantly, the delegate call to `NFTParentImpl.initialize()` operates on the same storage as the proxy, creating a storage collision between the two boolean variables (`initialized`, `initializing`) and the proxy's `owner` address. This causes the statement at Line 40 (which sets the proxy's `owner` address) to effectively overwrite the boolean values that were previously set in `NFTParentImpl.initialize()`, causing the implementation to think that the contract was not yet initialized.

An attacker can then freely call `NFTParentImpl.initialize()` – through the proxy contract – and become the administrator of the implementation contract. This allows the attacker to access any restricted methods, such as minting any arbitrary NFTs using the mint function `mintNFT` (Line 59).

To fix this vulnerability, the developers must modify the implementation contract to move the two boolean variables after the storage padding and then use the upgrade functionality in the proxy to upgrade the implementation. However, as in the previous example, after the vulnerability is exploited, there is no way to upgrade the implementation.