Elysim: Context-Aware Bytecode-Level Patching to Automatically Heal Vulnerable Smart Contracts

Christof Ferreira Torres  
SnT, University of Luxembourg  
Luxembourg, Luxembourg  
christof.torres@uni.lu

Hugo Jonker  
Open University of the Netherlands  
Heerlen, Netherlands  
hugo.jonker@ou.nl

Radu State  
SnT, University of Luxembourg  
Luxembourg, Luxembourg  
radu.state@uni.lu

ABSTRACT

Fixing bugs is easiest by patching source code. However, source code is not always available: only 0.3% of the ∼49M smart contracts that are currently deployed on Ethereum have their source code publicly available. Moreover, since contracts may call functions from other contracts, security flaws in closed-source contracts may affect open-source contracts as well. However, current state-of-the-art approaches that operate on closed-source contracts (i.e., EVM bytecode), such as EVMPatch and SMARTSHIELD, make use of purely hard-coded templates that leverage fix patching patterns. As a result, they cannot dynamically adapt to the bytecode that is being patched, which severely limits their flexibility and scalability. For instance, when patching integer overflows using hard-coded templates, a particular patch template needs to be employed as the bounds to be checked are different for each integer size (i.e., one template for uint256, another template for uint64, etc.). In this paper, we propose ELYSIUM, a scalable approach towards automatic smart contract repair at the bytecode level. ELYSIUM combines template-based and semantic-based patching by inferring context information from bytecode. ELYSIUM is currently able to patch 7 different types of vulnerabilities in smart contracts automatically and can easily be extended with new templates and new bug-finding tools. We evaluate its effectiveness and correctness using 3 different datasets by replaying more than 500K transactions on patched contracts. We find that ELYSIUM outperforms existing tools by patching at least 30% more contracts correctly. Finally, we also compare the overhead of ELYSIUM in terms of deployment and transaction cost. In comparison to other tools, we find that generally ELYSIUM minimizes the runtime cost (i.e., transaction cost) up to a factor of 1.7, for only a marginally higher deployment cost, where deployment cost is a one-time cost as compared to the runtime cost.

KEYWORDS

Ethereum, smart contracts, bytecode, context-aware patching

1 INTRODUCTION

Ideally, bugs in programs should be repaired by simply patching the source code. However, in some cases, the original source code may not be available. A poignant example are smart contracts: their bytecode is publicly available via the blockchain, yet in March 2022, out of 49,183,523 smart contracts deployed on the Ethereum blockchain (via Google BigQuery [17]), only 152,996 (via the Smart Contract Sanctuary project [33]) have their source code publicly available. For the remaining ∼49M smart contracts, no source code is publicly available. Moreover, smart contracts may call functions from other smart contracts. This implies that insecurities in bytecode-only or “closed-source” contracts may even affect contracts for which source code is available.

There has been prolific research on smart contract security. This includes using “proxy” contracts (e.g., [32, 49]) to make smart contracts upgradeable, designing clients to automatically detect and block malicious transactions (e.g., [5, 10, 11, 18, 40]), as well as building tools to automatically catch bugs prior to deployment using techniques such as symbolic execution (e.g., [13, 14, 24, 26, 27, 30]), abstract interpretation and model checking (e.g., [3, 15, 23, 41, 43]), or even fuzzing (e.g., [12, 19, 22]). Despite all these efforts, even well-studied bugs with well-known countermeasures (e.g., reentrancy) still occur in high-value contracts. Prominent examples include the second Parity wallet hack in 2017, where despite the source code having been manually audited and fixed, an attacker was able to lock up $150M [36], or the 2020 reentrancy bugs in both the Uniswap and Lendf.me smart contracts [38], which resulted together in $25M worth of assets being stolen after being manually audited. These examples illustrate poignantly that automation of both bug finding and bug patching is sorely needed. Most effort has been spent on automating bug discovery and has not carried over to bug patching, which can arguably be seen as one of the main roadblocks to practical smart contract security.

While there has been research into automatically patching smart contracts [29, 39, 47, 48], existing works are still limited: (1) they only address a few types of vulnerabilities, (2) they use hard-coded templates that do not scale (i.e., templates are brittle and do not cover all cases, meaning that several templates need to be introduced to cover minor variants of existing cases, such as for example a new template for patching integer overflows with different sizes), and (3) they add a large overhead in terms of deployment and runtime costs.

We propose a new methodology to address these shortcomings by automatically generating context-aware patches which adapt to the contract that is being patched. For each contract, we first perform a number of analyses such as integer type inference and free storage space inference to understand the context of the smart contract sufficiently to be able to create tailored and efficient patches. Both analyses and patching are performed at the bytecode level. An added bonus is that bytecode level patching results in more efficient code in terms of size and gas usage than recompiling patched source code [39]. We follow a hybrid approach by combining the usability of template-based approaches with the flexibility and effectiveness of semantic-based approaches. Template-based patching simply inserts a fixed sequence of instructions irrespective of the semantics of the program, whereas semantic-based patching modifies the program while preserving its original semantics. Smart contract developers can either reuse existing patch templates or easily write...
new patch templates to fix new types of vulnerabilities without having to worry about the context of the smart contract (e.g., free state variables, integer types, etc.). Our templates contain place holders which are automatically replaced with contract-related (i.e., semantic) information during patch generation. Moreover, since our approach leverages already existing bug-finding tools, it can easily be extended to incorporate new bug-finding tools, giving it the flexibility to handle future vulnerabilities.

**Contributions.** We summarize our contributions as follows:

- We present a novel context-aware bytecode level patching approach that combines template-based with semantic-based patching to create flexible and tailored patches for smart contracts.
- We propose Elysium, a tool that implements our approach and that is able automatically patch the 7 different types of vulnerabilities in smart contracts.
- We compare our tool to existing works using 3 different datasets by replaying more than 500K transactions, and demonstrate that Elysium not only patches more bugs (at least 30% more), but also that it is more efficient than existing works in terms of runtime costs (up to 1.7 times less gas).

## 2 BACKGROUND

In this section, we provide background on smart contracts, Ethereum bytecode, and the Ethereum virtual machine.

### 2.1 Smart Contracts

Ethereum proposes two types of accounts: externally owned accounts (EOA) and contract accounts (i.e., smart contracts). Both account types, EOAs and smart contracts, are identifiable via a unique 160-bit address and contain a balance that keeps track of the amount of ether owned by the account. While EOAs are controlled via private keys and have no associated code, smart contracts are not controlled via private keys and have associated code. As a result, smart contracts operate as fully-fledged programs that are stored and executed across the Ethereum blockchain. They are different from traditional programs in many ways. Once deployed, smart contracts cannot be removed or updated, unless they have been explicitly designed to do so. Smart contracts are deployed by leaving a transaction’s receiver empty and adding the code of the contract to be deployed to a transaction’s data field. After deployment, smart contract functions can be invoked by encoding the function signature and arguments in a transaction’s data field. A so-called fallback function is executed whenever the provided function name is not implemented. A key-value store allows smart contracts to persist state across transactions. Smart contracts are usually developed using a high-level programming language. Despite a plethora of programming languages [6, 8, 25, 45], Solidity [16] remains the most prominent language for developing smart contracts in Ethereum. Independently of the chosen programming language, the high-level source code must be translated into a low-level representation, so-called Ethereum bytecode, in order to be executable by the Ethereum Virtual Machine (EVM).

### 2.2 Ethereum Bytecode

Ethereum bytecode consists of a sequence of bytes that is interpreted by the EVM. Each byte either encodes an instruction or a byte of data. Figure 1 depicts the anatomy of Ethereum bytecode. Bytecode consists of two main parts: deployment bytecode and deployed bytecode.

![Figure 1: An illustrative example of the anatomy of Ethereum bytecode. Bytecode consists of two main parts: deployment bytecode and deployed bytecode.](image-url)

The EVM is a stack-based, register-less virtual machine that runs low-level bytecode and supports a Turing-complete set of instructions. Every instruction is represented by a one-byte opcode. The instruction set currently consists of 142 instructions and provides a variety of operations, ranging from basic operations, such as arithmetic operations or control-flow statements, to more specific ones, such as the modification of a contract’s storage or the querying of properties related to the executing transaction (e.g., sender) or the current blockchain state (e.g., block number). The EVM follows the Harvard architecture model by separating code and data into different address spaces. The EVM possesses four different address spaces: an immutable code address space, which contains the smart contract’s bytecode, a mutable but persistent storage address space that allows smart contracts to persist their data across executions, a mutable but volatile memory address space that acts as a temporary data storage for smart contracts during execution, and finally a stack address space that allows smart contracts to pass arguments to instructions at runtime. Moreover, the EVM employs a gas mechanism that assigns a cost to each instruction. This mechanism prevents denial-of-service attacks and ensures termination. When issuing a transaction, the sender has to specify a gas limit and a gas price. The gas limit is specified in gas units and must be large enough to cover the amount of gas consumed by the instructions...
Table 1: Decentralized Application Security Project Top 5

| Rank | Category                     | Associated Vulnerabilities                      |
|------|------------------------------|------------------------------------------------|
| 1    | Reentrancy                   | Same- and Cross-Function Reentrancy             |
| 2    | Access Control               | Transaction Origin, Suicidal, Leaking, Unsafe Delegatecall |
| 3    | Arithmetic                   | Integer Overflows and Underflows                |
| 4    | Unchecked Low Level Calls    | Unhandled Exceptions                             |
| 5    | Denial of Services           | Unhandled Exceptions, Transaction Origin, Suicidal, Leaking, Unsafe Delegatecall |

during a contract’s execution. Otherwise, the execution will terminate, and its effects will be rolled back. The gas price defines the amount of ether that the sender is willing to pay per unit of gas.

2.4 Smart Contract Vulnerabilities

In the last few years, a plethora of smart contract vulnerabilities have been identified and studied [1, 35]. The NCC Group initiated the Decentralized Application Security Project (DASP) with the goal of grouping the most common smart contract vulnerabilities into categories and ranking them based on their real-world impact [28]. Table 1, lists the top 5 categories and their associated vulnerabilities. Although more categories and vulnerabilities exist, our work primarily focuses on the top 5 categories. We leave it to future work to design patch templates for the missing categories.

3 METHODOLOGY

In this section, we describe the individual challenges as well as our approach towards patching vulnerabilities at the bytecode level for the vulnerabilities listed in Table 1.

3.1 Patching Reentrancy Bugs

The code snippet in Figure 2a provides an example of a function that is vulnerable to reentrancy at line 5. The function withdrawBalance transfers the balance of a user to the calling address. Note that a transfer is simply a call to an address. Hence, if msg.sender is a contract, then the transfer will trigger the code that is associated to msg.sender. This code can be malicious and call back the withdrawBalance function, and reenter function withdrawBalance while the first invocation has not finished yet. The issue here is that userBalances[msg.sender] has not been set to zero at that moment, and therefore an attacker can repeatedly withdraw its balance from the contract. This is clearly a concurrency issue that can be addressed in several ways. One solution, is to ensure that all state changes, such as the setting of userBalances[msg.sender] to zero, are performed before the call. However, this requires correctly identifying all state variable assignments that are affected by the call, and moving them before the call. Unfortunately, this process is rather tedious and error-prone, as it might break the semantics of a contract. A far more simple and less invasive approach, is to make use of mutual exclusion, a well studied paradigm from concurrent computing with the purpose of preventing race conditions [9]. The idea is to introduce a so-called mutex variable that locks the execution state and prevents concurrent access to a given resource.

Figure 2b depicts a patched version of the function withdrawBalance using mutual exclusion. A new state variable called locked has been introduced at line 2. The variable is used as a mutex variable and is initially set to false. The condition at line 6 first checks if locked is set to false before executing the call at line 8. Then, before executing the call, the variable locked is set to true and when the call has finished executing, the variable is set back to false. This mechanism ensures that the call at line 8 is not re-executed when the function withdrawBalance is reentered. Nevertheless, special care needs to be taken when working with mutexes. One has to make sure that there is no possibility for a lock to be claimed and never released, otherwise a so-called deadlock might occur and render the smart contract unusable. However, the greatest challenge of this approach is the introduction of a new state variable at the bytecode level. While this is straightforward when working at the source code level, it becomes more challenging when working at the bytecode level, where high level information such as state variable declarations are missing. Our idea is to use bytecode level taint analysis in order to learn about occupied storage space and infer which storage space is still available for inserting a new state variable (cf. Section 4 for more details on free storage space inference). It is crucial that we only introduce mutex variables at free storage space as otherwise we will overwrite already used storage space and break the semantics of the contract. Please note that the code presented in Figure 2a is an example of a so-called same-function reentrancy. However, Rodler et al. [40] presented other types of reentrancy such as cross-function reentrancy, delegated reentrancy, and create-based reentrancy. The idea is that an attacker can take advantage of a different function that shares the same state with the reentrancy vulnerable function. Thus, for a contract to be safe against any type of reentrancy, we have to apply the same locking mechanism to every function that shares state with the function that is vulnerable to reentrancy. We achieve this by searching the
Access control bugs includes: transaction origin, suicidal, leaking, and unsafe delegatecall. While the former requires its own approach, the latter can be patched using a common approach.

**Patching Transaction Origin.** The function withdraw in Figure 3a makes use of tx.origin to check if the calling address is equivalent to the owner. However, as tx.origin does not return the last calling address but the address that initiated the transaction, an attacker can try to forward a transaction initiated by the owner in order to impersonate itself as the owner and bypass the check at line 4. The process of patching a transaction origin vulnerability is rather simple. Figure 3b depicts a patched version of the function withdraw. The patch simply replaces tx.origin with msg.sender, which returns the latest calling address instead of the origin address, therefore not allowing an attacker anymore to impersonate itself as the owner.

**Patching Suicidal, Leaking, and Unsafe Delegatecall.** The contract in Figure 4a is considered suicidal. The function kill does not verify the calling address. As a result, anyone can destroy the contract. The vulnerabilities leaking and unsafe delegatecall are similar, although they relate to contracts that allow anyone to either withdraw ether or control the destination of a delegatecall. These three vulnerabilities share the same issue, namely the unprotected access to a critical operation. The idea is therefore to add the missing logic that limits the access to a critical operation to only a single entity, for example, the creator of the smart contract. Figure 4b depicts a patched version of the function kill. A new state variable owner has been added (line 2) as well as a constructor (lines 4-6) in order to initialize the variable owner during deployment with the address of the contract creator. Finally, a check has been added at line 9 to verify if msg.sender is equivalent to the address stored in the variable owner. Similar to reentrancy, this approach requires the identification of free storage space in order to introduce a new state variable owner. To initialize the variable owner at deployment (cf. Section 4 for more details on modifying the deployment bytecode), we are required to modify the deployment bytecode instead of the runtime bytecode. Please note that before creating a new owner variable, we first try to infer and reuse existing owner variables by employing certain heuristics (e.g., identify variables where msg.sender is written to). Also note that, deployment bytecode always contains a constructor at the bytecode level, therefore just append an assignment to the end of the existing constructor bytecode.

**3.3 Patching Arithmetic Bugs**

Arithmetic bugs such as integer overflows and underflows are a common issue in smart contracts. In 2018, several ERC-20 smart contracts have been victims to attacks due to integer overflows [34]. The code snippet in Figure 5a, provides an example of a function that is vulnerable to an integer overflow at line 5. The function buy is missing a check that verifies if the value contained in tokens[msg.sender] would overflow if amount would be added. A common way to ensure that unsigned integer operations do not wrap, is to use the SafeMath library provided by OpenZeppelin [31]. For example, in the case of addition, the library performs a post-condition check, where it first computes the result of \( a + b \) and then checks if the result is smaller than \( a \). If this is the case, then an overflow has happened and the library halts and reverts the execution. However, Solidity allows developers to make use of smaller types (e.g., uint32, uint16, etc.) in order to use less storage space and therefore reduce costs, despite the EVM being able to operate only on 256-bit values. As a result, the Solidity compiler artificially enforces the wrapping of integers on these smaller types to be consistent with the wrapping performed by the EVM on types of 256-bit. Unfortunately, the checks provided by the SafeMath library only work with values of type uint256 and do not
An unchecked low level call, also known as an unhandled exception, is vulnerable to integer overflows due to an added bounds check if an arithmetic operation is in bounds with respect to size. A call can fail due to several reasons: an out-of-gas exception, a revert that occurs whenever the return value of a call is not checked. A call that is set to false, while msg.sender has not received the prize. Fortunately, patching an unchecked low level call is rather trivial.

Figure 5: (a) Example of a function vulnerable to an integer overflow due to a missing bounds check guarding the update of tokens[msg.sender]. (b) Example of a function not vulnerable to integer overflows due to an added bounds check guarding the update of tokens[msg.sender].

Figure 6: (a) Example of a function vulnerable to an unhandled exception due to a missing return value check on send. (b) Example of a function not vulnerable to unhandled exceptions due to an added return value check on send.

4 DESIGN AND IMPLEMENTATION
In this section, we provide details on the overall design and implementation of Elysium.

4.1 Design and Implementation Overview
An overview of Elysium’s architecture is depicted in Figure 7. Elysium takes as input a smart contract as well as an optional bug report and outputs a patched smart contract together with a patching report. The input smart contract can be either bytecode or Solidity source code. The latter will be compiled into bytecode before performing any analysis or patching. The patched smart contract consists of the patched version of the bytecode of the original smart contract. The patching report contains information about execution time and the individual patches that have been applied. Elysium’s patching process follows four main steps: 1. bug localization, 2. context inference, 3. patch generation, and 4. bytecode rewriting. The bug localization step is responsible for detecting and localizing

---

```solidity
1 mapping (address => uint32) public tokens;
2 ...
3 function buy(uint32 amount) public 
4 require(msg.value == amount);
5 tokens[msg.sender] += amount;
6 }

(a) Before Patching

1 uint public prize;
2 address public winner;
3 bool public claimed = false;
4 ...
5 function claimPrize() public 
6 require(!claimed && msg.sender == winner);
7 msg.sender.send(prize);
8 claimed = true;
9 }

(b) After Patching
```

```
1 uint public prize;
2 address public winner;
3 bool public claimed = false;
4 ...
5 function claimPrize() public 
6 require(!claimed && msg.sender == winner);
7 * require(msg.sender.send(prize));
8 claimed = true;
9 }

(a) Before Patching

1 uint public prize;
2 address public winner;
3 bool public claimed = false;
4 ...
5 function claimPrize() public 
6 require(!claimed && msg.sender == winner);
7 msg.sender.send(prize);
8 claimed = true;
9 }

(b) After Patching
```
bugs in the bytecode. This step is skipped in case a bug report is provided. The context inference step is in charge of building the Control-Flow Graph (CFG) from the bytecode and inferring from the CFG context related information, such as integer types and free storage space, by leveraging taint analysis. The patch generation step is responsible for creating patches by inserting previously inferred context information within given patching templates. Finally, as a last step, the bytecode rewriting is in charge of injecting the generated patches into the original CFG and translating it back to bytecode. Elysium is written in Python and consists of roughly 1,600 lines of code\(^1\). In the following, we describe each of the four steps in more detail.

### 4.2 Bug Localization

In order to be able to patch bugs, Elysium first needs to know the exact location of a bug and its type. One option is to implement our own bug detection solution. However, this is time consuming and error-prone. Another option is to make use of already existing bug detection solutions for smart contracts and to simply incorporate them into Elysium. This approach has the advantage of adding modularity by decoupling the detection process from the patching process. This also makes it easy to extend Elysium with other future security analysis tools. Elysium leverages the following three well-known smart contract analysis tools to detect and localize bugs: Osiris\(^{[13]}\) to detect integer overflows, Oyente\(^{[26]}\) to detect reentrancy, and Mythril\(^{[27]}\) to detect unhandled exceptions, transaction origin, suicidal contracts, leaking ether, and unsafe delegatecalls. These tools are provided to Elysium as Docker images. Elysium spawns each tool as a separate Docker container, and once a tool has finished running, the output of the tool is parsed and bug information such as the opcode (e.g., CALL, ADD), exact bytecode location (i.e., program counter) and vulnerability type (e.g., reentrancy, integer overflow, etc.) is extracted. This information is then added to a bug report and used by the subsequent steps. Please note that, one can also directly provide a manually crafted bug report to Elysium. In such a case, Elysium will skip the bug localization step and will directly forward the bug report to the subsequent steps. A user only has to ensure that the bug report follows Elysium’s JSON format and that it contains the aforementioned information.

\(^{[1]}\)Elysium is publicly available on GitHub: https://github.com/christoftorres/Elysium.

### 4.3 Context Inference

To effectively patch vulnerabilities related to reentrancy, access control, and integer overflows, we require some context related information. We gather this information by traversing the CFG and leveraging taint analysis to infer information about integer types and free storage space. We build the CFG by using the EVM CFG Builder python library \(^{[20]}\).

**Integer Type Inference.** Integer type information is composed of a size (e.g., 32-bit for type `uint32`) and a signedness (e.g., signed for type `int` and unsigned for type `uint`). Both are essential in order to correctly check whether the result of an arithmetic operation is either in-bound or out-of-bound. However, type information is usually lost during compilation and it is therefore only available at the source code level. Fortunately, we can leverage some behavioral patterns of the Solidity compiler in order to infer the size as well as the signedness of integers. For example, for unsigned integers, we know that the compiler introduces an AND bitmask in order to “mask off” bits that are not in-bounds with the integer’s size (i.e., a zero will mask off the bit, whereas a one will leave the bit set). Thus, a variable of type `uint32` will result in the compiler adding to the bytecode a PUSH instruction that pushes a bitmask with the value `0xffffffff` onto the stack followed by an AND instruction. Hence, from the AND instruction we infer that it is an unsigned integer and from the bitmask we infer that its size is 32-bit, since `0xffffffff = 2^{32} - 1`. For signed integers, the compiler will introduce a sign extension via the `SIGNEDEXT` instruction. A sign extension is the operation of increasing the number of bits of a binary number while preserving the number’s sign and value. The EVM uses two’s complement to represent signed integers. In two’s complement, a sign extension is achieved by appending ones to the most significant side of the number. The number of ones is computed using `\lfloor \log_2(x+1) \rfloor`, where `x` is the first value passed to `SIGNEDEXT`. For example, a variable of type `int32` will result in the compiler adding to the bytecode a PUSH instruction that pushes the value 3 onto the stack followed by a `SIGNEDEXT`. Hence, from the `SIGNEDEXT` instruction we infer that it is a signed integer and from the value 3 we infer that its size is 32-bit, by solving the following equation: `y = 8(x + 1)`, where in this case `x = 3`. Knowing these patterns, we can use taint analysis to infer integer type information at the bytecode level. First, we iterate in a Breadth First Search (BFS) manner through the CFG until we find the basic block that contains the instruction that is labeled as

---

**Figure 7: Architecture of Elysium. The shaded boxes represent the four main steps of Elysium.**
when we arrive at the instruction of the bug location, we check we come across a visited instruction to a list. Each list represents one execution path into a contained in the CFG. An execution path is terminated whenever through it in a Depth First Search (DFS) manner and adding each visited instruction to a list. Each list represents one execution path contained in the CFG. An execution path is terminated whenever we come across a STOP, RETURN, SUICIDE, SELFDESTRUCT, REVERT, ASSERTFAIL, or INVALID instruction. Moreover, whenever we run into a JUMPI instruction we split the execution by creating a copy of the list of instructions visited so far and continue iterating first on one branch and then on the other branch. The EVM provides two different instructions to interact with storage: SLOAD and SSTORE. The former takes as input a storage key from the stack and pushes onto the stack the value stored at that key. The latter takes as input a storage key and a value, and stores the value at the given key. Storage keys are usually pushed onto the stack as constants. Thus, whenever a storage instruction is executed (i.e., SLOAD or SSTORE), a PUSH instruction will be executed before at some point in the execution with the goal of pushing the storage key onto the stack for the storage instruction to use. Our idea is therefore to run our taint analysis on all the collected execution paths and to introduce taint whenever we execute a PUSH instruction. The taint includes the PUSH instruction and will be propagated across stack as well as memory. Eventually, we will reach a storage instruction, where we then simply check the taint and infer the used storage key from the propagated PUSH instruction. Afterwards, we add the inferred key to the list of identified storage keys \( sk \). Finally, after having analyzed all execution paths, we can compute the next available free storage key as \( k = \text{max}(sk) + 1 \). This approach ensures that we do not collude with existing storage keys and it preserves the contiguous layout of state variables in Ethereum smart contract.


effective

Free Storage Space Inference. Patching reentrancy and access control bugs requires the introduction of an additional state variables at the bytecode level. State variables are associated with EVM storage, a key-value store, where both keys and values are of size 256-bit. In Solidity, statically-sized variables (e.g., everything except mappings and dynamically-sized array types) are laid out contiguously in storage starting from key zero, whereas the storage location for dynamically-sized variables is computed using a hash function. Moreover, the Solidity compiler tries to pack multiple, contiguous items that need less than 256-bit into a single storage slot. To not collide with existing statically-sized state variables, we need to find which storage keys are already in use. To do this, we first extract all the possible execution paths from the CFG by iterating through it in a Depth First Search (DFS) manner and adding each visited instruction to a list. Each list represents one execution path contained in the CFG. An execution path is terminated whenever we come across a STOP, RETURN, SUICIDE, SELFDESTRUCT, REVERT, ASSERTFAIL, or INVALID instruction. Moreover, whenever we run into a JUMPI instruction we split the execution by creating a copy of the list of instructions visited so far and continue iterating first on one branch and then on the other branch. The EVM provides two different instructions to interact with storage: SLOAD and SSTORE. The former takes as input a storage key from the stack and pushes onto the stack the value stored at that key. The latter takes as input a storage key and a value, and stores the value at the given key. Storage keys are usually pushed onto the stack as constants. Thus, whenever a storage instruction is executed (i.e., SLOAD or SSTORE), a PUSH instruction will be executed before at some point in the execution with the goal of pushing the storage key onto the stack for the storage instruction to use. Our idea is therefore to run our taint analysis on all the collected execution paths and to introduce taint whenever we execute a PUSH instruction. The taint includes the PUSH instruction and will be propagated across stack as well as memory. Eventually, we will reach a storage instruction, where we then simply check the taint and infer the used storage key from the propagated PUSH instruction. Afterwards, we add the inferred key to the list of identified storage keys \( sk \). Finally, after having analyzed all execution paths, we can compute the next available free storage key as \( k = \text{max}(sk) + 1 \). This approach ensures that we do not collude with existing storage keys and it preserves the contiguous layout of state variables in Ethereum smart contract.

4.4 Patch Generation

To generate patches, we use a combination of template-based and semantic patching. Table 2 provides an overview of all patching templates that are currently provided out-of-the-box by Elysium. A patch template is selected according to the vulnerability type that is to be patched. Elysium includes templates for seven vulnerability types. Moreover, existing templates can be modified or new ones can be added by users in order to patch vulnerabilities that are not supported yet by Elysium. We developed our own domain-specific language (DSL) that enables users to easily write and integrate their own context-aware patch templates into Elysium. The structure of a patch template consists of a sequence of instructions to be deleted, a sequence of instructions to be inserted, and an insert mode and constructor flag. The insert mode determines whether the instruction sequence to be inserted should be inserted before or after the bug location. The constructor flag
Table 2: Patch templates provided by Elysium.

| Vulnerability                  | Patch Template                                                                 | Source Code Representation                                                                 |
|-------------------------------|-------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| Reentrancy                    | ("delete": "", "insert": "free_storage_location SLOAD
PUSH1_0x1 EQ ISZERO PUSH1_0x1 JUMP1 PUSH1_0x1 JUMPI PUSH1_0x1 free_storage_location
SSTORE", "insert_mode": "before", "constructor": false) | + require(!locked);
+ locked = true;                                                                |
|                               | ("delete": ", insert": "free_storage_location SLOAD
PUSH1_0x0 free_storage_location
SSTORE", "insert_mode": "after", "constructor": false)                  | + locked = false;                                                                            |
| Transaction Origin            | ("delete": "ORIGIN", "insert": "CALLER", "insert_mode": "before", "constructor": false) | + require(tx.origin == ...);
+ require(msg.sender == ...);                                                      |
| Suicidal, Leaking & Unsafe    | ("delete": "", "insert": "CALLER free_storage_location
SSTORE", "insert_mode": "after", "constructor": true)                    | + constructor() {
+ owner = msg.sender;
+ }                                                                 |
| Delegatecall                  | ("delete": "", "insert": "free_storage_location SLOAD
PUSH12_0xffffffffffffffffffffffffffffffffffffffff AND
CALLER EQ PUSH1_0x1 JUMPI PUSH1_0x1 DUP1
REVERT JUMPDEST_jump_loc_1", "insert_mode": "before", "constructor": false) | + require(msg.sender == owner);
+ require(tx.origin == ...);                                                      |
| Integer Overflow (Addition)   | ("delete": "", "insert": "DUP2 DUP2 integer_bounds
SUB LT ISZERO PUSH_jump_loc_1 JUMPI PUSH1_0x1 DUP1
REVERT JUMPDEST_jump_loc_1", "insert_mode": "before", "constructor": false) | + require(MAX_VALUE - a >= b);
| Integer Overflow (Multiplication) | ("delete": "", "insert": "DUP2 DUP2 MUL integer_bounds AND
DUP3 ISZERO DUP1 PUSH_jump_loc_1 JUMPI POP DUP3 SWAP1 DIV
DUP2 EQ DUP1 JUMPDEST_jump_loc_1 SWAP1 POP PUSH_jump_loc_2
JUMPI PUSH1_0x1 DUP1 REVERT JUMPDEST_jump_loc_2", "insert_mode": "before", "constructor": false) | + require(b != 0 && a * b / b == a);
| Integer Underflow             | ("delete": "", "insert": "DUP2 DUP2 LT ISZERO
PUSH_jump_loc_1 JUMPI PUSH1_0x1 DUP1
REVERT JUMPDEST_jump_loc_1", "insert_mode": "before", "constructor": false) | + require(a == b);
| Unhandled Exception           | ("delete": "", "insert": "DUP1 ISZERO ISZERO PUSH_jump_loc_1
JUMPI PUSH1_0x1 DUP1
REVERT JUMPDEST_jump_loc_1", "insert_mode": "after", "constructor": false) | + require(...);                                                                 |

specifies if the deletion and insertion should occur at the deployment bytecode. Our DSL is a combination of the mnemonic representation of EVM instructions and custom keywords that act as place holders for context dependent information. We leverage the pyevmasm library [21] to translate the mnemonic representation of EVM instructions into EVM bytecode. The following four keywords exist: free_storage_location, integer_bounds, PUSH_jump_loc_{x}, and JUMPDEST_jump_loc_{x}. The free_storage_location keyword is used to get the current free storage location and it is automatically replaced with a PUSH instruction that pushes the current free storage location onto the stack when generating the patch. The integer_bounds keyword is used to get the integer bounds on the instruction at the bug location and it is automatically replaced with a PUSH instruction that pushes the inferred integer bounds onto the stack when generating the patch. The PUSH_jump_loc_{x} and JUMPDEST_jump_loc_{x} keywords work in conjunction. They are used to mark jumps across instructions within a template. The PUSH_jump_loc_{x} keyword is replaced in the bytecode rewriting step with a PUSH instruction that pushes the jump address of the JUMPDEST_jump_loc_{x} keyword. The JUMPDEST_jump_loc_{x} keyword simply acts as a marker and is afterwards replaced with a normal JUMPDEST instruction.

4.5 Bytecode Rewriting

Ethereum smart contracts are always statically linked, meaning that the bytecode already includes all the necessary library code that is needed at runtime. This makes EVM bytecode rewriting easier than compared to traditional programs. Nonetheless, rewriting EVM bytecode still poses some challenges. Similar to traditional programs, EVM bytecode uses addresses to reference code and data in the bytecode. Thus, when modifying the bytecode, one must ensure that the addresses that reference code and data are either adjusted or preserved. There are two popular ways to deal with this...
want to minimize costs, we decided to not employ a trampoline-
at address 

0x0000009c PUSH2 0xff
0x0000009b DUP2
0x00000099 SWAP2
0x00000097 PUSH2 0xff
0x00000096 SUB
0x00000095 SWAP4
0x00000094 ADD
0x00000093 SWAP1
0x00000092 DUP3
0x00000091 AND
0x00000090 DUP4
0x0000008f ADD
0x0000008e AND
0x0000008d SWAP4
0x0000008c DUP2
0x0000008b DUP1
0x0000008a SUB
0x00000089 LT
0x00000088 ISZERO
0x00000087 PUSH1 0xb4
0x00000086 JUMPI
0x00000085 ADD
0x00000084 SUB
0x00000083 LT
0x00000082 DUP2
0x00000081 AND
0x00000080 SUB
0x0000007f ADD
0x0000007e SUB
0x0000007d AND
0x0000007c ADD
0x0000007b SUB
0x0000007a AND
0x00000079 ADD
0x00000078 SUB
0x00000077 AND
0x00000076 ADD
0x00000075 SUB
0x00000074 AND
0x00000073 ADD
0x00000072 SUB
0x00000071 AND
0x00000070 ADD
0x0000006f SUB
0x0000006e AND
0x0000006d ADD
0x0000006c SUB
0x0000006b AND
0x0000006a ADD
0x00000069 SUB
0x00000068 AND
0x00000067 ADD
0x00000066 SUB
0x00000065 AND
0x00000064 ADD
0x00000063 SUB
0x00000062 AND
0x00000061 ADD
0x00000060 SUB
0x0000005f AND
0x0000005e ADD
0x0000005d SUB
0x0000005c AND
0x0000005b ADD
0x0000005a SUB
0x00000059 AND
0x00000058 ADD
0x00000057 SUB
0x00000056 AND
0x00000055 ADD
0x00000054 SUB
0x00000053 AND
0x00000052 ADD
0x00000051 SUB
0x00000050 AND
0x0000004f ADD
0x0000004e SUB
0x0000004d AND
0x0000004c ADD
0x0000004b SUB
0x0000004a AND
0x00000049 ADD
0x00000048 SUB
0x00000047 AND
0x00000046 ADD
0x00000045 SUB
0x00000044 AND
0x00000043 ADD
0x00000042 SUB
0x00000041 AND
0x00000040 ADD
0x0000003f SUB
0x0000003e AND
0x0000003d ADD
0x0000003c SUB
0x0000003b AND
0x0000003a ADD
0x00000039 SUB
0x00000038 AND
0x00000037 ADD
0x00000036 SUB
0x00000035 AND
0x00000034 ADD
0x00000033 SUB
0x00000032 AND
0x00000031 ADD
0x00000030 SUB
0x0000002f AND
0x0000002e ADD
0x0000002d SUB
0x0000002c AND
0x0000002b ADD
0x0000002a SUB
0x00000029 AND
0x00000028 ADD
0x00000027 SUB
0x00000026 AND
0x00000025 ADD
0x00000024 SUB
0x00000023 AND
0x00000022 ADD
0x00000021 SUB
0x00000020 AND
0x0000001f ADD
0x0000001e SUB
0x0000001d AND
0x0000001c ADD
0x0000001b SUB
0x0000001a AND
0x00000019 ADD
0x00000018 SUB
0x00000017 AND
0x00000016 ADD
0x00000015 SUB
0x00000014 AND
0x00000013 ADD
0x00000012 SUB
0x00000011 AND
0x00000010 ADD
0x0000000f SUB
0x0000000e AND
0x0000000d ADD
0x0000000c SUB
0x0000000b AND
0x0000000a ADD
0x00000009 SUB
0x00000008 AND
0x00000007 ADD
0x00000006 SUB
0x00000005 AND
0x00000004 ADD
0x00000003 SUB
0x00000002 AND
0x00000001 ADD
0x00000000 SUB
0x00000000 AND
0x00000000 ADD

Figure 9: An example on bytecode rewriting, where a guard is added to an unguarded ADD instruction using the integer overflow (addition) patch template.
Table 3: A comparison of the individual patching tools evaluated in this work.

| Toolname     | Bug Localization | Patching Level | Approach               | Availability | IO   | RE   | UE   | TO   | SU   | LE   | UD   |
|--------------|------------------|----------------|------------------------|--------------|------|------|------|------|------|------|------|
| EVMPatch [39] | Outsourced       | Bytecode       | Template               | Not Available | Yes  | Yes  | Yes  | Yes  | Yes  | Yes  | Yes  |
| SMARTSHIELD [48] | Outsourced       | Bytecode       | Template/ Semantics   | On Request   | No   | Yes  | Yes  | Yes  | Yes  | Yes  | Yes  |
| SCRepair [47]  | Outsourced       | Source Code    | Mutation               | Open Source1 | No   | Yes  | Yes  | Yes  | Yes  | Yes  | Yes  |
| sGuard [29]    | Insourced        | Source Code    | Template/ Semantics   | Open Source  | Yes  | Yes  | Yes  | Yes  | Yes  | Yes  | Yes  |
| Elysium        | Outsourced       | Bytecode       | Template/ Semantics   | Open Source  | Yes  | Yes  | Yes  | Yes  | Yes  | Yes  | Yes  |

† Publicly available source code does not compile. ● Not supported. ○ Patching partially supported. ◆ Patch template must be specified manually. ● Fully automatic patching supported. IO: integer overflow, RE: reentrancy, UE: unhandled exception, SU: suicidal, LE: leaking, UD: unsafe delegatecall.

Table 4: CVE dataset overview.

| Contract | CVE     | Bugs | Total | Benign | Attacks |
|----------|---------|------|-------|--------|---------|
| BEC      | 2018-10299 | 1 | 409,837 | 409,836 | 1       |
| SMT      | 2018-10376  | 1 | 34,164  | 34,163  | 1       |
| UET      | 2018-10468  | 1 | 23,725  | 23,670  | 55      |
| SCA      | 2018-10706  | 9 | 281     | 280     | 1       |
| HXG      | 2018-11239  | 4 | 1,284   | 1,274   | 10      |

Table 5: SmartBugs dataset overview.

| Vulnerability | Contracts | Annotated | Detected | Overlap |
|---------------|-----------|-----------|----------|---------|
| Reentrancy    | 31        | 32        | 29       | 28      |
| Access Control| 18        | 19        | 12       | 12      |
| Integer Overflow| 15    | 23        | 20       | 16      |
| Unhandled Exception| 52 | 75        | 23       | 23      |
| Total         | 116       | 149       | 84       | 79      |

Table 6: Horus dataset overview.

| Vulnerability  | Contracts | Bugs | Total | Benign | Attacks |
|----------------|-----------|------|-------|--------|---------|
| Reentrancy     | 44        | 47   | 4,593 | 2,656  | 1,937   |
| Access Control | 589       | 823  | 2,116 | 264    | 1,852   |
| – Parity Wallet Hack 1 | 585 | 585  | 1,877 | 263    | 1,614   |
| – Parity Wallet Hack 2 | 238 | 238  | 358   | 120    | 238     |
| Integer Overflow| 125       | 235   | 42,768 | 42,327 | 441     |
| Unhandled Exception| 901 | 993  | 80,997 | 78,144 | 2,853   |
| Total Unique   | 1,655     | 2,098 | 129,863 | 122,830 | 7,041   |

5.2 Experimental Results

RQ1: Effectiveness. We first measure the effectiveness of Elysium and the other tools on the SmartBugs dataset. The dataset only consists of annotated contracts and does not contain attacking nor benign transactions. We therefore first run the bug-finding tools (i.e., Osiris, Oyente, and Mythril) on the contracts and match the reported bugs with the annotated bugs. The overlap marks the validated ground truth (see overlap column in Table 5). From the 149 annotated bugs, only 79 overlap with the bugs detected by the bug-finding tools. Moreover, 5 false positives have been reported by the bug-finding tools. Next, we patch the contracts by running the patching tools and rerunning the bug-finding tools on the patched version returned by each patching tool, and mark a bug as successfully patched if the bug-finding tool does not report the bug anymore. Table 7 shows that Elysium is able to patch all

provide partial patching capabilities for a given type of vulnerability. For instance, all tools, except Elysium, only support the patching of 256-bit unsigned integers and do not support integers of smaller size. Another example is reentrancy, where tools such as SMARTSHIELD and SCRepair only provide support for patching same-function reentrancy. Furthermore, some tools such as EVMPatch require developers to write contract specific patches for access control related bugs and therefore do not provide generic fully automatic patching. Elysium on the other hand, provides complete support and fully automatic patching for all vulnerabilities.

Datasets. We run our experiments on three different datasets. The first dataset is the CVE dataset [44] used by Rodler et al. We chose this dataset to be able to compare our tool with EVMPatch. It consists of real-world ERC-20 token contracts that were victims of integer overflow attacks. Moreover, the dataset also provides a list of attacking and benign transactions (see Table 4). However, the dataset is limited to integer overflows and only contains 5 contracts. The second dataset is the SmartBugs dataset [42]. This dataset consists of 116 manually crafted contracts with 149 annotated vulnerabilities across 4 different vulnerability types (see Table 5). While the dataset brings in a large diversity of vulnerabilities, it does not contain a list of benign or attacking transactions. The third dataset is the Horus dataset [4]. The dataset consists of 1,655 unique real-world contracts vulnerable to one of 4 different vulnerabilities, with 129,863 annotated transactions, where 122,830 transactions are benign and 7,041 transactions are attacks (see Table 6).
Table 7: Number of bugs patched by each tool on contracts from the SMARTBugs dataset.

| Vulnerability       | Bugs | SMARTSHIELD | sGUARD | ELYSIUM |
|---------------------|------|-------------|--------|---------|
| Reentrancy          | 28   | 7           | 28     | 28      |
| Access Control      | 12   | -           | 2      | 12      |
| Integer Overflow    | 16   | 16          | 3      | 16      |
| Unhandled Exception | 23   | 22          | -      | 23      |
| Total               | 79   | 45          | 33     | 79      |

Table 8: Number of non-blocked benign transactions and blocked attacking transactions from the HORUS dataset.

| Vulnerability       | Non-Blocked Benign | BLOCKED ATTACKS |
|---------------------|--------------------|-----------------|
|                     | SMARTSHIELD | ELYSIUM | SMARTSHIELD | ELYSIUM |
| Reentrancy          | 653 (34%) | 2,608 (98%) | 33 (2%) | 1,937 (100%) |
| Access Control      | -        | 264 (100%) | - | 1,852 (100%) |
| - Parity Wallet Hack 1 | -      | 263 (100%) | - | 1,614 (100%) |
| - Parity Wallet Hack 2 | -      | 120 (100%) | - | 238 (100%) |
| Integer Overflow    | 40,996 (97%) | 41,012 (97%) | 432 (98%) | 441 (100%) |
| Unhandled Exception | 63,199 (81%) | 74,379 (95%) | 2,650 (93%) | 2,853 (100%) |
| Total Unique        | 104,292 (85%) | 117,713 (96%) | 3,073 (44%) | 7,041 (100%) |

79 bugs, whereas SMARTSHIELD and sGUARD can only patch 45 and 33, respectively. We see that SMARTSHIELD has issues in patching reentrancy. This is because SMARTSHIELD patches reentrancy by moving storage instruction before the call instruction. However, this process is often very imprecise. sGUARD has issues in patching integer overflows due to its in-house bug detection not always being able to identify arithmetic bugs correctly. If we only consider the bug types that all three tools have in common (i.e., reentrancy and integer overflows), then we count a total of 23, 31, and 44 bugs patched, for SMARTSHIELD, sGUARD, and ELYSIUM, respectively. This means that ELYSIUM patches at least 30% more bugs than the other tools. To measure the effectiveness of ELYSIUM and the other tools on the CVE and HORUS datasets, we re-execute the attack transactions of each dataset, once on the original bytecode and once one the patched bytecode returned by each tool. We mark an attack as successfully blocked if the patched bytecode resulted in the transaction being reverted. EVMPATCH, SMARTSHIELD, and ELYSIUM were able to successfully block all attacks for all the contracts within the CVE dataset. For the HORUS dataset, Table 8 shows that ELYSIUM is able to successfully block 100% of all attacks, while SMARTSHIELD is able to block only 44% of all attacks.

**RQ2: Correctness.** ELYSIUM’s correctness depends heavily on the accurate recovery of the CFG and the accurate inference of free storage locations. We downloaded from Etherscan the bytecode and source code for the top 100 smart contracts according to their ether balance. Their lines of source code range from 19 to 3,299 and their number of functions range from 1 to 291. The EVM CFG Builder library [20] is able to recover 100% of the CFG for 85 contracts. Overall, the library achieves an average of 96% recovery with an average time of 6.7 seconds. We improved the library by adding the techniques proposed in [7]. The improved version is able to fully recover the CFG for 88 contracts and achieves an average of 98% recovery with an average time of 7.5 seconds. For the 12 non-fully recovered contracts, our improved version of the EVM CFG Builder library is able to recover on average 16% more of the CFG than the original version. To measure the accuracy of the free storage location inference employed by ELYSIUM, we leveraged the ability of the Solidity compiler to generate the storage layout of a smart contract and compared the storage layout generated by the Solidity compiler with the storage layout inferred by ELYSIUM. ELYSIUM is able to correctly infer the storage layout and thus next available free storage location for all 100 contracts. Besides measuring CFG recovery and free storage location inference, we also measured the correctness of ELYSIUM by replaying benign transactions on the patched contracts of the EVMPATCH and HORUS datasets. A benign transaction is considering successful if the result is identical to the result of the original unpatched transaction. On the EVMPATCH dataset, EVMPATCH, SMARTSHIELD, and ELYSIUM correctly executed the same number of benign transactions. For the HORUS dataset, ELYSIUM is able to correctly execute 96% of the benign transactions whereas SMARTSHIELD is able to execute only 85% of the benign transactions (see Table 8). We found that the reason for certain benign transactions not being executed correctly is either due to not enough gas being provided for the transaction to be executed on the newly patched smart contract or the CFG simply not being recovered to 100% and therefore introducing invalid jump destinations that lead to exceptions at runtime.

**RQ3: Costs.** We differentiate between deployment cost and transaction cost. Deployment cost is associated to the cost when deploying a contract on the blockchain. It is computed based on the size of the bytecode. The larger the bytecode, the higher the cost. Transaction cost (also known as runtime cost) is associated to the cost when executing a contract on the blockchain. It is computed based on the gas consumed by the executed instructions. The more expensive instructions are executed, the higher the cost. While deployment cost is a one-time cost, transaction cost is a repeating cost. Our goal is therefore to primarily minimize transaction cost when introducing patches. Figure 11 highlights the deployment cost increase for all datasets. The deployment cost is measured by computing the difference in terms of size between the patched and the original bytecode. We state that the patches introduced by EVMPATCH and sGUARD add the largest deployment cost. This is because those tools use templates that have been generated from source code. In contrast, ELYSIUM and SMARTSHIELD, use manually crafted and optimized bytecode level templates that use less instructions.
is in most cases the cheapest in terms of deployment cost. This is because \texttt{SmartShield} injects its patching template only once into the code and then simply jumps to it. In addition, for reentrancy it does not introduce new code but rather tries to move the existing code around (e.g., move writes to storage before a call). For example, \texttt{SmartShield} only adds on average 4 bytes of overhead for reentrancy on the \texttt{Horus} dataset, whereas \texttt{Elysium} adds 270 bytes. However, we have shown that \texttt{SmartShield}’s patching technique for reentrancy is often inaccurate. Moreover, for unhandled exceptions, \texttt{Elysium} is more efficient than \texttt{SmartShield}, because of its optimized patch template. Figure 12, highlights the transaction cost increase measured for each dataset. The transaction cost is measured by computing the gas usage difference between the patched and original contract for all successfully executed benign transactions. We state that \texttt{Elysium} adds in most all the cases the smallest overhead in terms of transaction cost. For instance, in the \texttt{Horus} dataset, \texttt{Elysium} only adds on average 58 gas units of overhead when patching integer overflows, whereas \texttt{SmartShield} adds 80. However, we also see that \texttt{Elysium} adds 25,238 gas units on average when patching reentrancy, whereas \texttt{SmartShield} adds none. This is because \texttt{SmartShield} does not add new code, while \texttt{Elysium} adds two storage instructions which consume together already 25,000 gas units. Figure 10 shows the execution time of \texttt{Elysium} in proportion to the contract size for the \texttt{Horus} dataset. The contracts are sorted according to their execution time/contract size. We observe that 90% of the contracts are patched in less than 9 seconds. The median is 0.07 seconds and the maximum is around 24 minutes. \texttt{Elysium} spends roughly 70% of the execution time on the recovery of the CFG. We also state that the execution time grows linear to the size of a smart contract.

5.3 Limitations

\texttt{Elysium} highly depends on the ability to fully recover the CFG and correctly infer the context regarding integer bounds and free storage locations to be able to correctly patch the bytecode of a smart contract. Our preliminary experiments on CFG recovery show that we are able to recover 100% of the CFG for 88% of the cases and that we are able to correctly infer storage locations for all tested contracts. Moreover, during our evaluation we were able to block all attacks related to integer overflows, which means that we are able to correctly infer the integer bounds. However, to prevent breaking semantics or introducing new bugs when patching, \texttt{Elysium} first checks if the CFG has been fully recovered by checking if the CFG contains any basic blocks that are unreachable (i.e., basic blocks with no jump instructions pointing to them as well as no jump instructions pointing from them to existing basic blocks). If there are any unreachable basic blocks, then \texttt{Elysium} outputs a warning regarding the possibility of breaking the semantics of the smart contract and requests for the user’s consent to continue.

Another limitation is \texttt{Elysium}’s evaluation. There exist multiple techniques to validate the correctness of patches such that they do not only fix the bug but also do not introduce new bugs. Our evaluation follows the same strategy as previous works (e.g., \cite{39, 48}) by re-executing previous transactions of real-world contracts on the patched bytecode. We split previous transactions in benign and attacking transactions. If an attacking transaction is blocked, then we assume that the patch is working correctly. If the result of a benign transaction is the same for the original bytecode and the patched bytecode, then we assume that the semantics have been preserved. However, this does guarantee soundness since our evaluation depends on previous inputs generated by users where it can still be the case that new bugs have been introduced or that the
actual semantics have been changed while those previous inputs simply do not cover those new cases. An alternative could be to apply differential fuzzing on the original and patched version of the bytecode to detect discrepancies. The input generation could be even driven by symbolic execution that leverages a constraint solver to synthesize inputs for the fuzzer. However, this approach would also heavily depend on correctly inferring the CFG.

6 RELATED WORK

Framing code patching as a search and optimization problem has led several authors [37, 46] to leverage well-established heuristics and search algorithms to patch smart contracts. SCRepair [47] uses a genetic algorithm to find a patch. There are inherent limits in terms of quality and depth of the results. For instance, complex reentrancy patterns, such as cross-function reentrancy or faulty access control, cannot be trivially patched and contrary to claims made by SCRepair, patches linked to transaction order dependency are not addressed. Moreover, genetic algorithms are notoriously slow since a population of solutions needs to be evolved and this process is entirely random. Several techniques from automated program repair research have been applied to smart contracts. Nguyen et al. [29] present a tool called $\text{TGUARD}$, that patches smart contract vulnerabilities at the source code level. The disadvantage of this approach is that the compiler often adds unnecessary/unoptimized code, increasing bytecode size and thus causing increased deployment and transaction costs. The main difference with our work is that we patch directly at the bytecode level and can highly optimize our patches. Moreover, our tool is language independent, while $\text{TGUARD}$ only works for Solidity. Recently, the academic community has shifted its interest to automated patching of EVM bytecode. For instance, Ayoade et al. [2] patch integer overflows via bytecode rewriting and verify the equivalence of orginal and patched contract via the Coq theorem prover. However, their verification does not scale and their approach is not context sensitive and therefore can not be used to patch reentrancy or access control. Rodler et al. propose EVMPatch [39], a methodology that can patch integer overflow and access control bugs at bytecode level. Integer overflows are patched through hard-coded patches restricted to type uint256 overflows and underflows. In order to patch access control patterns, the developer is required to use a custom domain-specific language for specifying a contract specific patch. Thus, patching is not fully automated and the developer is required to understand and fix the bug manually. Claims that unhandled exceptions can be patched are not backed by experiments and patching access control bugs (such as suicidal contracts and leaking contracts), requires manual effort for every contract. Our approach is fully automated, covers more classes of bugs, and does not require the kind of manual preparation reported in [39]. Targeting more complex bugs, Zhang et al. [48] presented SMARTSHIELD, which automatically patches integer overflows, reentrancy bugs, and unchecked exceptions at the bytecode level. The tool is limited to only use hard-coded patches for integer overflows of type uint256. We observed in our experiments that SMARTSHIELD has issues in patching reentrancy bugs due the complexity of identifying data and control dependencies across bytecode. Our approach addresses these challenges by leveraging taint analysis at the bytecode level to infer contract related information (e.g., integer bounds and free storage space) and uses it to generate automatically contract specific patches.

7 CONCLUSION

In this work, we presented ELYSIUM, a tool to automatically patch smart contracts using context-related information that is inferred at the bytecode level. ELYSIUM is currently able to patch 7 types of vulnerabilities. It can easily be extended by adding further vulnerability detectors and by writing new patch templates using our custom DSL. We compared ELYSIUM to existing tools by patching almost 2,000 smart contracts and replaying more than 500K transactions. Our results show that ELYSIUM is able to effectively and correctly patch at least 30% more contracts than existing tools. Moreover, when compared to existing tools, the resulting transaction overhead is reduced by up to a factor of 1.7. We leave it to future work, to further optimize the overhead in terms of deployment costs.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers and our shepherd Kevin Roundy for their valuable comments and feedback. We also gratefully acknowledge the support from the RIPPLE University Blockchain Research Initiative (UBRI) and the Luxembourg National Research Fund (FNR) under the grant 13192291.

REFERENCES

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In International conference on principles of security and trust. Springer, 164–188.
[2] Gbadebo Ayoade, Erick Bauman, Lutfur Khan, and Kevin W. Hamlen. 2019. Smart Contract Defense through Bytecode Rewriting. In IEEE International Conference on Blockchain, Blockchain 2019, Atlanta, GA, USA, July 14–17, 2019. IEEE, 384–389.
[3] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Françoise Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.09891 (2018).
[4] C. Torres. 2021. Horus: A framework to detect attacks and trace stolen assets across Ethereum (FC 2021). Retrieved August 6, 2021 from https://github.com/christofertor/Horus.
[5] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, GuoFei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. 2020. SODA: A Generic Online Detection Framework for Smart Contracts. In Proceedings of the Network and Distributed System Security Symposium (NDSS’20).
[6] Michael Coblenz. 2017. Obsidian: a safer blockchain programming language. In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). IEEE, 97–99.
[7] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mìa Dalla Preda. 2021. EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode. arXiv preprint arXiv:2103.09113 (2021).
[8] Cornell Blockchain. 2018. Bamboo: a language for morphing smart contracts. https://github.com/CornellBlockchain/bamboo.
[9] Edsger W Dijkstra. 2001. Solution of a problem in concurrent programming control. In Pioneers and Their Contributions to Software Engineering. Springer, 289–294.
[10] Christof Ferreira Torres, Mathis Baden, Robert Norvill, Beltran Borja Fiz Pintoferos, Hugo Jonker, and Sjoake Mauw. 2020. æGIS: Shielding vulnerable smart contracts against attacks. In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security. 584–597.
[11] Christof Ferreira Torres, Antonio Ken Iannullo, Arthur Gervais, and Radu State. 2021. The Eye of Horus: Spotting and Analyzing Attacks on Ethereum Smart Contracts. In International Conference on Financial Cryptography and Data Security.
[12] Christof Ferreira Torres, Antonio Ken Iannullo, Arthur Gervais, and Radu State. 2021. Confuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In 2021 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE.
[13] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Orisir: Hunting for integer bugs in ethereum smart contracts. In Proceedings of the 36th Annual Computer Security Applications Conference. 664–676.
[14] Christof Ferreira Torres, Mathis Steichen, and Radu State. 2019. The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. In 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, CA.
