2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-6512-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/EuroSP57164.2023.00034

EFϟCF: High Performance Smart Contract Fuzzing for Exploit Generation

Michael Rodler[1], David Paaßen[2], Wenting Li[3], Lukas Bernhard[4],
Thorsten Holz[5], Ghassan Karame[4], Lucas Davi[2]
[1] Amazon Web Services, [2] University of Duisburg-Essen, [3] NEC Laboratories Europe,
[4] Ruhr-University Bochum, [5] CISPA Helmholtz Center for Information Security
m@mrodler.eu, *{david.paassen, lucas.davi}@uni-due.de*, *wenting.li@neclab.eu*
{lukas.bernhard, ghassan.karame}@rub.de, *holz@cispa.de*

***Abstract*—Smart contracts are increasingly being used to manage large numbers of high-value cryptocurrency accounts. There is a strong demand for automated, efficient, and comprehensive methods to detect security vulnerabilities in a given contract. While the literature features a plethora of analysis methods for smart contracts, the existing proposals do not address the increasing complexity of contracts. Existing analysis tools suffer from false alarms and missed bugs in today's smart contracts that are increasingly defined by complexity and interdependencies. To scale accurate analysis to modern smart contracts, we introduce EFϟCF, a high-performance fuzzer for Ethereum smart contracts. In contrast to previous work, EFϟCF efficiently and accurately models complex smart contract interactions, such as reentrancy and cross-contract interactions, at a very high fuzzing throughput rate. To achieve this, EFϟCF transpiles smart contract bytecode into native C++ code, thereby enabling the reuse of existing, optimized fuzzing toolchains. Furthermore, EFϟCF increases fuzzing efficiency by employing a structure-aware mutation engine for smart contract transaction sequences and using a contract's ABI to generate valid transaction inputs. In a comprehensive evaluation, we show that EFϟCF scales better—without compromising accuracy—to complex contracts compared to state-of-the-art approaches, including other fuzzers, symbolic/concolic execution, and hybrid approaches. Moreover, we show that EFϟCF can automatically generate transaction sequences that exploit reentrancy bugs to steal Ether.**

1. Introduction

Ethereum is the most prominent blockchain platform that supports *smart contracts*: Programs that are stored and run as part of the blockchain protocol. Smart contracts are the backbone of the emerging decentralized finance (DeFi) industry. Due to its popularity, the security of Ethereum and its smart contracts layer has received considerable attention from the research community and industry. Since the first high profile attack against "the DAO" contract [28], the community has observed a continuous stream of attacks against smart contracts [54, 64].

Analyzing smart contract code is challenging due to its stateful nature and the large number of potential bug classes. Prior work on identifying vulnerabilities in smart contracts relied on various techniques, such as symbolic execution [33, 35, 40], model checking [23, 31], and static analysis [44, 56]. While these methods are promising, many tools primarily identify potential bugs with heuristics and do not give proof of exploitability, or they suffer from scalability issues (e.g., due to state explosion). Fuzz testing [10, 25, 27, 29, 39, 59] has emerged as a promising method for detecting smart contract bugs. Fuzzing-based approaches are able to generate the specific inputs that allow a developer to trigger and analyze the identified problem within a debugging environment. However, existing fuzzing approaches suffer from various drawbacks, most notably: (1) They do not scale to complex contracts that are used on the Ethereum blockchain today. We identify a clear trend that smart contracts are becoming more complex over time and therefore require more complex transaction sequences for comprehensive analysis (see Section 2). (2) They do not accurately model the complex interactions that are possible in Ethereum, such as reentrancy and cross-contract interactions. Especially reentrancy attacks have emerged as one of the most critical security issues, as demonstrated by multiple high-profile incidents over the last years [5, 21, 28, 54]. However, existing analysis tools over-approximate reentrancy attacks, leading to inaccurate analysis results (see Section 5.4).

In this paper, we tackle all of these challenges by introducing *Extremely Fast ϟ Contract Fuzzer* (EFϟCF), an optimized fuzzing framework and exploit generator. Comprehensively analyzing smart contracts is challenging because their behavior depends on their *internal state*, which changes depending on the order and parameters of called functions. Even for moderately complex contracts, it is not feasible for an analysis tool to simply execute all permutations of available functions. Using high-throughput fuzzing guided by code coverage, EFϟCF efficiently searches the space of possible transaction sequences to identify those that expose a vulnerability. To increase the fuzzing throughput, we propose to speed up Ethereum smart contracts by translating the Ethereum virtual machine (EVM) bytecode to equivalent C++ code and using a high-performance coverage-guided fuzzer.

EFϟCF utilizes a simple—yet powerful—bug oracle: Ether gains. This allows EFϟCF to effectively act as an exploit generator for Ether stealing attacks. However, Ether gains do not cover all current smart contract security issues. In order to extend EFϟCF to cover smart contract specific bugs, EFϟCF allows developers to define custom bug oracles in their Solidity code.

[1] *Work conducted at University of Duisburg-Essen before joining Amazon.*

Besides these bug classes, reentrancy issues remain one of the most challenging and critical vulnerabilities for Ethereum smart contracts [8, 54]. Many existing analysis tools attempt to detect potential reentrancy using over-approximate analyses [10, 19, 38, 43, 53, 56]. In contrast, EF⚡CF accurately detects reentrancy vulnerabilities. Instead of relying on heuristics, EF⚡CF simulates the behavior of multiple attacker-controlled smart contracts. Each test case generated by EF⚡CF not only specifies the transactions, but also the behavior of the simulated attacker contracts. This allows the coverage-guided fuzzing process to explore the behavior of the attacker-controlled contracts by attempting to execute reentrant calls and changing returned data of callbacks. EF⚡CF does not need an explicit bug oracle for reentrancy vulnerabilities and instead uses the Ether gains bug oracle. This allows EF⚡CF to automatically synthesize Solidity attack contracts that reproduce the reentrancy attack on a live blockchain. Moreover, EF⚡CF identifies compositional security issues [8] that arise only when *multiple* contracts are combined. Before deploying a certain combination of contracts, EF⚡CF can fuzz the combination to detect compositional security issues given a certain set of contracts.

We have performed a comprehensive evaluation of our fuzzing approach. To this end, we instantiated EF⚡CF with the AFL++ fuzzer [22]. We compare EF⚡CF to current state-of-the-art analysis tools, such as Manticore [37], Echidna [25], MAIAN [40], teEther [33], VeriSmart/SmartTest [47, 48], and Smartian [10] with respect to their *time-to-bug*. Overall, we spent more than 1300 CPU days to evaluate and compare existing analysis tools and EF⚡CF. We find that EF⚡CF is the only analysis tool capable of successfully analyzing all contracts in our benchmark and scales even to contracts requiring complex and long transaction sequences to trigger a bug. In addition, we compare the performance of EF⚡CF with the hybrid fuzzers ILF [27] and ConFuzzius [53] with respect to code coverage. Our experiments show that EF⚡CF outperforms existing approaches when analyzing complex smart contracts according to several code complexity metrics. We show that EF⚡CF identifies $99.9\,\%$ of the access control bugs that EthBMC [23] identified and that EF⚡CF finds vulnerabilities in contracts that EthBMC could not analyze. In addition, we compare EF⚡CF with the symbolic analyzer Sailfish [6] for reentrancy bugs, where we show that EF⚡CF accurately detects those reentrancy issues that are exploitable. Demonstrating the practicality of EF⚡CF, we find that only 5 out of the 26 verified reentrancy bugs of Sailfish are actually prone to reentrancy attacks that allow the attacker to steal Ether. Finally, we show that EF⚡CF is able to generate concrete transaction sequences for the compositional reentrancy bugs in the contracts used in the evaluation of the Serif [8] static analyzer tool. Furthermore, EF⚡CF is the first analysis tool that is capable of generating an exploit for the compositional reentrancy attack against the *Uniswap*/*IMBTC* contracts.

Contributions In summary, our main contributions are:

- We show how to leverage conventional software fuzzers to efficiently test smart contracts with coverage-guided fuzzing, allowing the fuzzer to scale to large and complex contracts. To achieve this, we propose a transpilation approach to accelerate the fuzzing of bytecode programs, such as the EVM. Our approach removes the interpreter by directly translating bytecode into native code, using C++ as an intermediate language (Section 3). We augment a conventional base fuzzer, such as *AFL++*, with a custom mutator to implement smart contract specific mutation operations.
- We efficiently model complex smart contract interactions during fuzzing, including multiple attacker-controlled smart contracts, reentrant calls, and cross-contract interactions in a fuzzer. Using this approach, EF⚡CF can automatically generate sophisticated reentrancy exploits (Section 4).
- We thoroughly evaluate the performance of EF⚡CF against a large number of state-of-the-art analysis tools (see Section 5): (a) EF⚡CF scales best to longer transaction sequences when compared using time-to-bug. (b) EF⚡CF achieves significantly better code coverage than prior fuzzers on existing real-world smart contracts, especially when considering various code complexity metrics. (c) EF⚡CF is capable of identifying access control and reentrancy bugs with high accuracy and fewer false alarms than current analysis tools.

To foster research on the security of smart contracts, we release EF⚡CF along with all benchmarks and experiments at https://github.com/uni-due-syssec/efcf-framework/.

2. Problem Statement

Ethereum Smart Contracts In Ethereum, each participant is identified by an address derived from cryptographic key material. Each Ethereum account is defined by its address and an associated balance of Ether, Ethereum's native cryptocurrency. Smart contract accounts are associated with a code and program state (called *storage*). An externally-owned account broadcasts a transaction to the Ethereum network to interact with a smart contract. Transactions are used to either transfer Ether to trigger the execution of a smart contract or both. A transaction consists of several fields, most importantly the destination address, the Ether value, and the input. If the destination address is a smart contract, Ethereum nodes execute the smart contract with the provided input, and once a new Ethereum block is generated, the state updates of the transaction are included in the block and committed to the blockchain.

Smart contracts are written in specialized programming languages (e.g., Solidity) and compiled into Ethereum Virtual Machine (EVM) bytecode. The EVM bytecode is a dedicated bytecode format optimized for small size, simplicity, and deterministic execution. Most production-grade EVM implementations use a bytecode interpreter to execute a smart contract. Ethereum smart contracts only have a single entry point. The executed high-level function is selected based on an identifier that is provided as part of the input. The function parameters are encoded according to the *Ethereum ABI* definition, which is a de-facto standard to encode parameters to function calls in the transaction input.

Challenges of Smart Contract Fuzzing When fuzzing smart contracts, the goal is to identify a sequence of

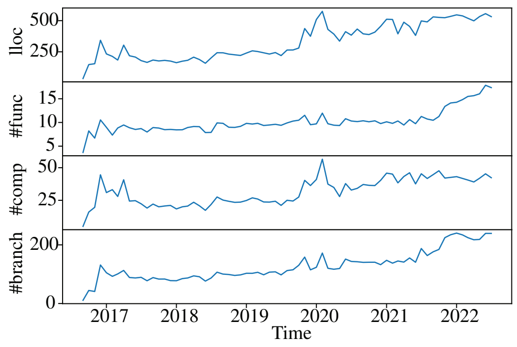


Figure 1: Increasing trend in smart contract complexity. We measure the complexity of all unique contracts with verified source code that appear in Ethereum until July 22, 2022.

transactions that exposes a software fault. The final transaction of the sequence triggers a software fault, while the preceding transactions set up the state of the contract such that the fault can be triggered. In this sense, testing Ethereum smart contracts is a variant of testing stateful software, a highly challenging problem [1, 11, 30, 52, 60]. In contrast to static analysis methods, generating complete transaction sequences by means of fuzzing has the advantage that it features a very low rate of false alarms, and the result is easy to analyze: A developer or security analyst can simply replay and debug the transaction sequence to determine the root cause and assess whether the bug can be triggered in practice. However, determining such a transaction sequence—or function call sequence—is challenging since the search space is extremely large. There are two dimensions that must be explored in parallel to reach high code coverage: (1) the input to individual transactions and (2) the ordering of the transactions. To efficiently cover this large search space, we can exploit the fact that many of the possible transaction sequences are redundant since they only exercise the same error-handling paths over and over again. *Coverage-guided fuzzing* can efficiently search the input space of a given contract for inputs that trigger distinct code coverage and was popularized by the success of the American Fuzzy Lop (AFL) fuzzer [63] and many follow-up works [22, 36]. In this context, test case throughput emerges as an important design aspect for an effective fuzzer. Intuitively, the greater the number of test cases generated and executed, the greater the likelihood that a fault will be triggered within a given time budget during fuzzing, an inherently probabilistic process. Most fuzz testing approaches for Ethereum smart contracts develop new fuzzers from scratch [25, 27, 39, 53], which are often not optimized for high throughput. For example, ILF performs at a rate of 148 transactions per second [27], while native code fuzzing with far more complex code regularly achieves 10,000 or more test case executions per second [61]. This low throughput effectively hampers a fuzzer's capability of analyzing complex smart contracts.

Support for Complex Smart Contracts Owing to their increasing prevalence and ability to encode complex business logic, smart contracts are becoming increasingly more complex. In Figure 1, we measure how the complexity of deployed contracts has evolved over time. We analyzed 120,556 unique contracts with verified source code that appeared in the Ethereum blockchain until July 22, 2022. For each contract, we measure the number of logical lines of code, the number of state-changing public functions, the number of comparison operators, and the number of branches in the control flow. Our analysis confirms that contracts are becoming increasingly complex in all aspects. For instance, among the unique smart contracts created in 2022, we find an average of roughly 530 lines of code and 16 state-changing public functions, while the same metric was at 182 lines of code and 8 functions in 2017.

To exercise all code paths while testing increasingly complex smart contracts, multiple consecutive transactions are required to explore the internal states of a smart contract. However, most prior studies are only limited to rather short transaction sequences of length 3 [23, 33, 40] or do not assess the ability to work with longer transaction sequences [25, 27, 53, 59]. Typically, more complex contracts also require longer transaction sequences to cover different states of the contract during testing. To assess the ability of existing analysis tools to cope with longer consecutive transaction sequences, we conducted an experiment with a set of benchmark contracts with artificial bugs. Our experiment, summarized in Table 1 and detailed in Section 5, shows that existing analysis tools are not sufficient to analyze more complex contracts. In particular, current fuzzing-based analysis tools [25, 53] were unable to identify bugs that require a specifically ordered sequence of six or more transactions. While symbolic execution tools [33, 37, 40] are capable of producing such sequences even up to ten transactions, they fail to identify faults that require accumulation of internal state over multiple transactions. Apart from this, none of the analysis tools we tested are able to identify all bugs within a generous time budget of 48 hours.

Support for Complex Smart Contract Interactions Another challenge that we need to tackle is the frequent interaction of smart contracts with each other. To precisely model such an interaction, whenever a smart contract calls another (potentially untrusted) smart contract, we must assume that the smart contract under test can be reentered at any function. So-called reentrancy attacks have had devastating consequences in the past [28]. To faithfully emulate the attacker's capabilities with respect to reentrancy, we must simulate the following scenarios: At each call to an untrusted and potentially attacker-controlled contract, the target smart contract can be (1) reentered at the same call depth multiple times, (2) reentered at multiple functions, and (3) reentered by a call originating from a different smart contract. Current analysis tools mostly refrain from modelling arbitrary reentrant transactions due to state explosion [31]; instead, most tools utilize over-approximative detectors for reentrancy bugs (i.e., state updates after calls [19, 53, 56]).

To better illustrate the challenge, consider the example in Figure 2, which depicts a token-like contract with standard transfer and allowance mechanisms that is vulnerable to a reentrancy attack. If using the *checks-effects-interactions* code pattern [50] is not possible, the second best alternative to prevent reentrancy attacks is to

```
contract Bank {
mapping(address => uint256) balance;
mapping(address => bool) disableWithdraw;
mapping(address => mapping(address => uint256))
↪  allow;

modifier withdrawAllowed { // reentrancy locking
  require(disableWithdraw[msg.sender] == false); _; }

function addAllowance(address other,
                      uint256 amnt)
{ allow[msg.sender][other] += amnt; }

function transferFrom(address from,
                      uint256 amnt)
                      withdrawAllowed {
  require(balance[from] >= amnt);
  require(allow[from][msg.sender] >= amnt);
  balance[from] -= amnt;
  allow[from][msg.sender] -= amnt;
  balance[msg.sender] += amnt; }

function withdrawBalance() withdrawAllowed {
  // set lock
  disableWithdraw[msg.sender] = true;
  // reentrant calls possible here
  msg.sender.call{value: balance[msg.sender]}("");
  // release lock
  disableWithdraw[msg.sender] = false;
  balance[msg.sender] = 0; }
/* ... */ }
```

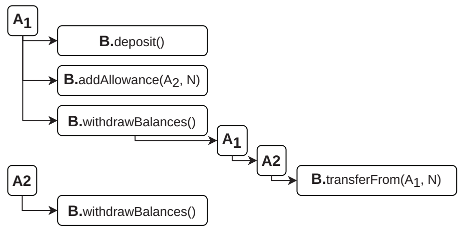


Figure 2: A contract (Bank, B) with bypassable reentrancy-locking. The attack, depicted below the source code, requires two colluding attacker-controlled smart contracts (A_1, A_2).

use locking mechanisms [49]. However, many analysis tools [19, 53, 56] do not handle locking mechanisms appropriately and simply report a potential reentrancy issue in the *withdrawBalance* function for the locking mechanism itself and the balance update. In the example in Figure 2, the modifier *withdrawAllowed* prevents an attacker from reentering the *withdrawBalance* function. This gives the developer a false sense of security, thinking that the *userBalances* variables are protected by the locking mechanism and thus the contract must be secured against reentrancy attacks. However, this assumes that an attack follows the call chain $A_1 \to B \to A_1 \to B$. Since the attacker A_1 has arbitrary control, they can transfer control to a different colluding smart contract A_2, which allows to execute the call chain $A_1 \to B \to A_1 \to A_2 \to B$. Reentrant calls from the second contract A_2 are not locked as it has not yet interacted with the target contract B. Therefore, the second attacker contract A_2 can call into the *transferFrom* function to move away the balance of A_1 before the call to *withdrawBalance* finishes and resets the balance. Using this attack, it is possible to bypass the reentrancy locking mechanism and withdraw twice the balance that was initially invested.

3. Design of EF/CF

The design of our Extremely Fast ⚡ Contract Fuzzer (EF⚡CF) is driven by two major features: optimizing test case throughput and accurately modelling complex interactions with smart contracts. To achieve the former, EF⚡CF uses two explicit phases, a compile and a run time phase (see Figure 3). At compile time, the EVM bytecode of the smart contract is translated to C++ code with our newly developed *evm2cpp* compiler and paired with a fuzzing-optimized EVM runtime, facilitating fast smart contract execution. To accurately model interactions with smart contracts, we devise an approach to allow the fuzzer to mutate the behavior of multiple simulated attacker-controlled smart contracts. Each generated test case specifies a sequence of transactions, which are executed by the fuzzing harness. However, in contrast to prior work [23, 25, 33, 40, 53], this transaction sequence also specifies the behavior of callbacks to attacker accounts, including return values and further reentrant transactions. To detect bugs, EF⚡CF features detectors that are directly built into the EVM runtime and the fuzzing harness. Here EF⚡CF supports a commonly featured Ether-based bug oracle that attempts to gain Ether, but also custom bug oracles that are specified by a developer in Solidity code. In what follows, we discuss and explain our design choices in more detail.

3.1. Modelling Blockchain Interaction

To faithfully model complex (possibly adversarial) interactions on the blockchain, we define an input format for Ethereum transaction sequences that supports (a) fuzzing the blockchain environment, (b) fuzzing return data, (c) reentrant transactions, and (d) targeting multiple contracts.

Blockchain Environment EF⚡CF runs the smart contract in a custom blockchain environment, which contains several attacker-controlled accounts. A user of EF⚡CF can also supply a custom initial blockchain state, e.g., to fuzz smart contracts that rely on other smart contracts or expect to be deployed at a certain address. EF⚡CF allows the fuzzer to choose and mutate several environmental values of the Ethereum environment. For example, the fuzzer chooses the block number and timestamp at the beginning of the transaction sequence and is allowed to advance both at every transaction. This enables us to handle smart contracts that expect a certain timespan to pass between two consecutive transactions. Furthermore, the fuzzer can increase the initial Ether balance of the target contract to simulate prior Ether investment into the contract.

The blockchain state is reset before every executed test case, which ensures that each generated transaction sequence can be deterministically executed. This is necessary to obtain reliable coverage measurements and eases root-cause analysis since the developer can reliably replay a transaction sequence. In many cases, the transaction sequence can be directly utilized as an end-to-end exploit against the deployed version of the contract.

Transaction Sequence Every test case in EF⚡CF consists of a header specifying the initial environment followed by a sequence of transactions. Similar to regular Ethereum

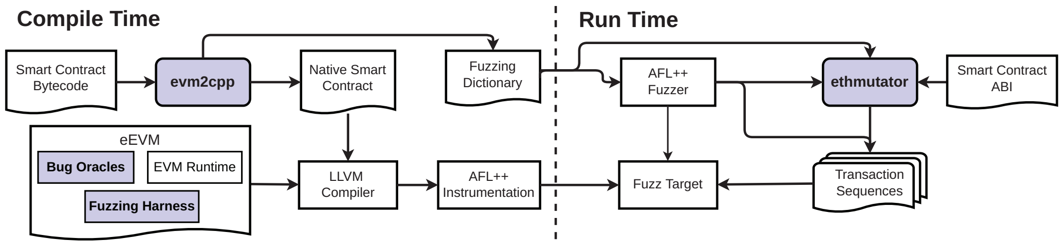


Figure 3: Architecture of EFϟCF.

transactions, each transaction consists of a sender, a receiver, a call value (i.e., transferred Ether), and associated input data. However, for performance reasons, we restrict the senders and receivers to a small set of accounts that are set when the fuzzer is launched. In a typical single-contract fuzzing setup, the set of receivers will include only the target smart contract.

EFϟCF simulates the behavior of arbitrary smart contracts at the attacker-controlled accounts. To achieve this, each transaction requires additional associated data beyond what a regular Ethereum transaction requires. This includes fields that specify what to do if the target smart contract calls back to an attacker-controlled address. Each transaction can have multiple associated *return-headers*, which specify (1) whether the call succeeded, (2) what data to return, (3) and how many reentrant calls can be performed. The fuzzer is then free to choose arbitrary values for any of these parameters. However, we bound the number of return headers per transaction to 255 and we also bound the number of reentrant transactions to 255 in our implementation of EFϟCF. If EFϟCF encounters a callback without an associated *return-header*, EFϟCF will simulate a failed call. This allows the fuzzing process to explore a large variety of behaviors of attacker-controlled smart contracts.

Reentrant Transactions Current dynamic analysis tools [29, 33, 53] focus on generating lists of top-level transactions that trigger an exploit. In contrast, EFϟCF simulates the behavior of a reentrancy-capable attacker. We model the transaction sequences as a tree of transactions. This enables EFϟCF to explore various shapes of the tree: reentering the same function repeatedly, reentering the same function only once, reentering the same contract in a different function, or reentering the same contract multiple times at the same call-depth.

However, in practice, not all shapes of the tree are possible for a certain contract. Only some of the functions of a contract allow for callbacks to the attacker and therefore further reentrant transactions. In general, it is not possible to compute the shape of the tree in advance for all contracts. Whether an external call to an attacker is performed by the target smart contract generally depends on the input and as such, cannot be determined before executing the transaction.

EFϟCF's fuzzing harness dynamically builds a transaction tree. All other components of EFϟCF still operate on a list of transactions. The harness uses the list of transaction as a queue: when an external call is encountered, the fuzzer can simulate a reentrancy with the next transaction in the queue. If the transaction queue is empty no reentrant call is performed. To mutate this ad-hoc tree structure, the fuzzer can mutate a single field in the return header that specifies how many reentrant transactions can be executed when an external call is encountered. The fuzzing harness ignores this field if the transaction does not trigger an external call.

We show an example of chain of mutations and how they affect the shape of a transaction tree in Figure 4. We start with a flat sequence of three transactions that target the functions *f1*, *f2*, and *f3*. All transactions have the *reenter* flag set to 0. The fuzzer then probabilistically mutates the *reenter* flag to modify the shape of the transaction tree. In the first transaction it is set to 1. If the call to *f1* triggers a callback to the attacker, EFϟCF will perform a reentrant call with the second transaction in the queue, i.e., a reentrant call to *f2*. The third transaction remains a top-level transaction. With this simple mutation, EFϟCF has now generated a cross-function reentrancy attack. Setting the *reenter* flag to 2 the produces a tree with two reentrant transactions at the same call depth. However, if the *reenter* flag on the second transaction is set to a positive value, the third transaction becomes a reentrant transaction at a deeper call depth. Since we do not have any more transactions in the queue, there is no second reentrant transaction at call depth 1.

This approach to defining the transaction tree in an ad-hoc manner can model arbitrary tree shapes. However, we introduce several limits in our implementation of EFϟCF because the shape of the transaction tree is bounded by several factors in practice. Ethereum's gas mechanism limits the number of possible callbacks per transaction. Therefore, in our implementation of EFϟCF, we limit the maximum number of callbacks to 255 and the number of reentrant transactions per callback to 255. Similarly, we stop execution when reaching the EVM defined maximum call depth of 1024.

Compositional Security Modern smart contracts are increasingly coupled with other smart contracts. For example, token contracts are often tied to exchange contracts, where the token can be traded for other tokens or Ether. Beyond the security of contracts in isolation, also the *composition of multiple contracts* must remain secure against attacks (compositional security). Recently, several attacks have been reported that were only possible due to composition of multiple smart contracts that were developed independently [5, 8, 21, 54]. For example, the *Uniswap* reentrancy attack was only possible because the Uniswap contract was combined with a new type of token contract that would perform a callback to the attacker. The Uniswap contract did not expect reentrancy to be possible on an external call and is indeed safe when paired with most token contracts.

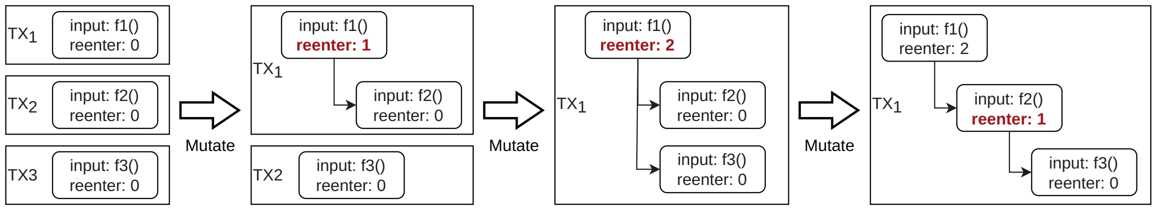
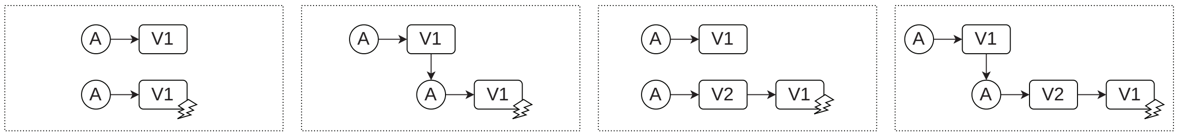


Figure 4: Mutating a transaction sequence to obtain reentrant transaction sequences with different shapes. The mutated *reenter* field is highlighted in bold red.



(a) Single Contract. (b) Single Contract Reentrancy. (c) Contract Composition. (d) Compositional Reentrancy.

Figure 5: Different settings illustrating the difference and similarities between reentrancy and compositional attacks.

The most prominent examples of compositional security issues were also reentrancy attacks [8]. However, compositional attacks are not necessarily reentrancy attacks. Figure 5 shows four different attack settings, where the last (sub-)transaction triggers a bug in contract *V1*. Figure 5a depicts a flat sequence of transactions: two subsequent calls to contract *V1*. Figure 5b shows a simple reentrancy attack, *V1* call back into the attacker *A*, which performs a reentrant call to *V1*. Figure 5c depicts a compositional attack, where a composition of two contracts is vulnerable. The DoS attack against the *Parity Multisig Wallet* is an example of such an attack [51]. The bug can be triggered by setting up a vulnerable state and then forcing *V2* to call *V1*. Figure 5d depicts a compositional reentrancy attack, where the call from *V2* to *V1* is a reentrant call. The *Uniswap* attack is an example of such a compositional reentrancy bug [8].

EFϟCF's design also allows for testing compositions of smart contracts. First, EFϟCF is designed such that it can import parts of the blockchain state to set up a realistic environment for the target smart contracts. For example, developers could test the security of their deployed contract compositions by setting up their initial state on a local test chain and then running EFϟCF to detect potential issues. Second, EFϟCF supports selecting the receiver of a transaction, including reentrant transactions. A developer or analyst just needs to configure the set of contracts to be analyzed by EFϟCF in composition.

3.2. Optimizing Test Case Throughput

Most state-of-the-art analysis tools [23, 25, 33, 37, 40, 53] develop or utilize custom EVM implementations that offer the introspection and extension possibilities necessary to perform dynamic smart contract analysis. Similarly, as part of our high throughput fuzzing framework, we develop an execution environment for smart contracts that is optimized for fuzzing. Here, test case throughput, i.e., both fast mutation/generation and fast execution, emerge as one of the most important properties of a fuzzer to achieve good results in practice.

Translating EVM to C++ Most widely used Ethereum clients implement an interpreter to execute EVM smart contracts. Typically, Ethereum nodes execute a large number of different smart contracts throughout their operation; in this case, it suffices to rely on an interpreter. However, in a fuzzing setting, the same smart contract is repeatedly executed. As a consequence, the overhead of the interpreter adds up over time and causes significant overhead over longer fuzzing periods. In EFϟCF, we remove this overhead by performing additional translation and optimization before starting the fuzzing campaign. To remove the interpreter overhead, we develop *evm2cpp*, a custom translation layer from EVM bytecode to C++, and pair this with a customized EVM runtime optimized for fuzzing. While this approach gives up the flexibility of an interpreter or a just-in-time compiler, an ahead-of-time compilation approach can utilize the full set of optimization techniques in modern C++ compilers, such as link-time optimization. Furthermore, *evm2cpp* performs optimizations to remove costly EVM stack operations. Note that ahead-of-time translation of bytecode targeting a virtual machine to C++ [46, 57] or directly to native code [2, 58] has been previously applied in various academic and industry settings to speed up execution time. In EFϟCF, we apply this technique to EVM bytecode for the first time.

We discuss correctness of our translation approach in Section 4.1. During all of our fuzzing campaigns and experiments, we have not identified any bug or crash that was caused by *evm2cpp* miscompilation. Furthermore, to quantify the performance improvements of translation we conduct an ablation study (see Section 5.2).

Optimizing the EVM runtime The smart contract that is translated to C++ still requires an EVM runtime that implements the opcode handlers and interaction with the blockchain. We pair the C++ code generated by *evm2cpp* with an EVM runtime, which we adapted and optimized from the *eEVM* project [13]. We chose the *eEVM* project because of its relatively simple codebase that can be easily extended and adapted for fuzzing. This includes omitting or simplifying several features required by a full EVM implementation operating as part of an Ethereum node. For example, our EVM runtime does not feature instruction-accurate gas tracking. We do not need the gas mechanism to limit execution time, since it is limited per test case by the fuzzer. Detecting the majority of vulnerabilities, such as access control or reentrancy, does not require instruction-accurate gas tracking. However, checking the gas budget is necessary to accurately execute external calls.

Furthermore, we stop test case execution at the first failing transaction. Since the failing transaction would be rolled-back, this has no effect on subsequent transaction executions. Instead of performing roll-backs after every failing transaction, we stop execution of the current test case, reset the state of the Ethereum blockchain to its initial state, and let the fuzzer generate a new test case. This approach nudges the fuzzer to generate transaction sequences that only contain succeeding transactions. The only exception is the last transaction in the sequence, which is allowed to explore error handling paths. Furthermore, this approach increases the effectiveness of test case splicing: When EFϟCF combines two previously generated test cases into one, it will very likely generate a new test case containing only succeeding transactions.

Optimizing the Mutations At run time, the base fuzzer launches the target smart contract with seed inputs, i.e., a seed with a single transaction without any input. The fuzzing process is driven by a greybox fuzzing approach that involves mutating inputs, executing the fuzz target, and measuring the code coverage to find new interesting transaction sequences. Note that the base fuzzer is unaware of the structure that is inherent to the test cases, i.e., the structure of the transaction sequence and the structure of the individual transaction inputs. The base fuzzer's mutation strategies are not efficient in mutating the structure of the input. Hence, we augment the base fuzzer with a carefully engineered and optimized test case generator and mutator that performs structure-aware mutations and generations. Our custom mutator is called *ethmutator* and performs both (1) mutations on the transaction inputs according to the smart contract's ABI and (2) structural mutations on the transaction sequence.

3.3. Bug Oracles

Prior work on smart contract fuzzing attempted to address two orthogonal problems at the same time: triggering bugs and detecting bugs [10, 27, 29, 39, 53]. Bug oracles are dynamic analyses that signal the fuzzer that a fault was triggered. In this paper, we focus primarily on the aspect of triggering faults by developing a high throughput fuzzer to identify the right input to trigger a fault. We opt to primarily use a simple—yet powerful—bug oracle: Ether gains. This is in line with prior work on exploit generation for smart contracts [23, 33, 40]. However, we also support custom bug oracles defined in Solidity code by the smart contract developer to cover contract-specific bug classes.

EFϟCF defines an attack to satisfy one of the following conditions: (a) The attacker is able to trigger a *selfdestruct* to an arbitrary address, thereby allowing the attacker to drain the funds of the contract and create a *Denial-of-Service* scenario. (b) The attacker can redirect the control-flow to an arbitrary address (using the DELEGATECALL instruction), which would give the attacker control over the target's Ether. (c) The attacker is able to gain Ether by interacting with the contract, i.e., the sum of the balances of the attacker-controlled contracts must exceed the initial sum of balances of these contracts. In contrast to other bug oracles, this approach avoids a high number of false alarms by design. For example, accurately detecting integer overflows [20] or reentrancy [43] without high-level type information is challenging. However, it is comparably straightforward to detect if a fuzzer generates a transaction sequence exploiting an integer overflow or reentrancy to gain Ether. Interestingly, we found that this type of bug oracle is also *more accurate* than bug oracles implemented in other analysis tools. For example, the contract depicted in Figure 6 is not identified as vulnerable by either of two state-of-the-art hybrid fuzzers, Confuzzius [53] and Smartian [10]. EFϟCF's Ether-gains bug oracle turns out to cover more cases such as this example.

```
contract SimpleEtherDrainOther {
 function withdraw(address payable to) public {
   require(msg.sender != to);
   to.transfer(address(this).balance);
 }
 function deposit() public payable {} }
```

Figure 6: Contract that is not considered as vulnerable by the *leaking Ether* bug oracles of Confuzzius [53] and Smartian [10], but detected by EFϟCF's Ether gain bug oracle.

However, an Ether-based bug oracle often does not capture the semantics of some smart contract applications, such as token contracts. To tackle such contracts, we also implemented support for custom invariant checking and assertion checking. Smart contract developers specify invariants that the fuzzer tries to invalidate. We implement support for mechanisms that are also utilized in other analysis tools, such as Echidna [25] and Mythril [38] (see also Appendix A). Developers that already utilize one of these tools can directly reuse their custom invariants and assertions with EFϟCF.

4. EF/CF Implementation

We now present an overview of the implementation of EFϟCF. We release our implementation as open source at https://github.com/uni-due-syssec/efcf-framework/. We discuss further minor technical details in Appendix A.

4.1. EVM to C++ Translation

The *evm2cpp* component is a custom compiler that translates EVM bytecode to C++, implemented in roughly 2500 lines of Rust code. First, we perform a linear pass over the EVM bytecode to build the set of all basic blocks. In EVM bytecode, all jump destinations are marked with JUMPDEST instructions. We use these to identify the boundaries of basic blocks by looking for branching instructions and jump destination markers. We do not construct a full control-flow graph, avoiding costly and error-prone analysis. Instead, we perform local analysis and optimization at the EVM basic block level. We emulate basic blocks in isolation using abstract values as placeholders for non-constants to perform data-flow analysis and constant propagation with respect to the EVM stack. Contrary to full abstract interpretation, we stop

```
pc_5a : {
  /* JUMPDEST */
  /* CALLVALUE */
  const uint256_t v_1_0 = callvalue_v();
  /* DUP1 */
  /* ISZERO */
  const uint256_t v_3_0 = iszero_v(v_1_0);
  /* PUSH2 0x66 */
  /* JUMPI */
  if (v_3_0) {
    ctxt->s.push(v_1_0);
    goto pc_66;
  }
  ctxt->s.push(v_1_0);
  goto pc_62;
}
```

Figure 7: Example for basic block translated by *evm2cpp*. The comments show the original instructions; some stack-related opcodes have no direct counterpart in the emitted C++ code.

emulation at the end of a basic block and therefore do not need to handle control-flow instructions.

The code generation procedure translates each EVM basic block to a C++ lexical block. If we can infer the jump target at the end of a basic block with our constant propagation, we directly emit a *goto* statement to the target C++ lexical block. Otherwise, we have to fall back to using a large jump table via the *computed goto* feature. In both cases the C++ goto is translated into a jump instruction by the C++ compiler. The gotos between translated basic blocks are then instrumented by the coverage-instrumentation pass of the fuzzer.

Within a basic block, each opcode is translated to a call to the respective opcode handler function in the EVM runtime. We translate the stack-based EVM opcodes into a lightweight register-based form, where each register is translated to a C++ local variable and no register is reused (similar to single-static assignment form, albeit without the need for the ϕ instruction). At the beginning and the end of each translated basic block, we ensure that the stack effects of the register-based form and the original EVM opcodes are the same. Essentially, we use the EVM stack exclusively to pass parameters between translated basic blocks. This enables us to eliminate a number of costly EVM stack operations and emit C++ code that can be well optimized by recent clang versions (we tested *clang* ≥ 13).

Figure 7 shows an example of a translated basic block. Here, the `DUP1` opcode is eliminated, which duplicates a value on the EVM stack. Furthermore, we eliminate the `PUSH2` instruction, which pushes a constant to the stack. Owing to the constant propagation pass we perform, we can infer that this constant is used as a jump target later. Instead of dispatching the jump via the EVM stack, we emit a *goto* statement that directly targets the desired block. Before the jump, we fix up the EVM stack effects of the basic block by pushing the right values to the stack. With respect to the EVM stack, the original bytecode performs three pushes, two pops, and one replacement of the top element. In contrast, the generated C++ code uses only a single stack push.

Performance Improvements The translation approach allows for two major optimizations that result in execution speed-ups: (1) removal of the interpreter loop, and (2) removal of EVM stack operations. The former allows us to remove many repeated lookups of the opcode handlers and indirect dispatching. Because there is no indirect dispatch to opcode handlers, the C++ compiler is now able to inline opcode handlers in the emitted native code to avoid the overheads of calls. The EVM stack is typically implemented as a high-level data structure that resides on the native-code heap, e.g., a C++ *vector*. Accessing the EVM stack requires bounds and stack-overflow checking, resulting in significant overhead. The optimizations in *evm2cpp* remove many stack operations, resulting in faster execution. We present an ablation study in Section 5.2 to quantify the performance improvements due to *evm2cpp*.

Correctness Correctness of our compiler is paramount for EFϟCF. Our compilation approach is designed with minimum error potential. We utilize the EVM stack to pass data between translated basic blocks in the translated bytecode. Optimizations are performed within a basic block, allowing us to avoid error-prone and costly indirect jump target analysis. Nevertheless, our translation approach intentionally diverts from the EVM specification to enable optimizations. For example, due to the removal of EVM stack operations during compilation, a translated contract could temporarily exceed the maximum stack size imposed by the EVM specification. However, this does not raise any problems for EFϟCF as it must not be able to execute potentially malicious bytecode.

To detect inconsistencies early in development, we tested *evm2cpp* using a differential fuzzing approach. We execute both the original eEVM interpreter and *evm2cpp* code and check whether they behave the same. In all of our experiments (see Section 5), we did not encounter any issues that were caused by a miscompilation by *evm2cpp*.

EVM-level Auto-dictionary To increase fuzzing efficiency, many fuzzers scan the code for constants and create a dictionary of potentially interesting values (e.g., file format header magic values). However, they typically scan for up to 64 bit constants or null-terminated C strings and thus do not properly handle the 256 bit EVM words or 160 bit Ethereum addresses. Hence, we generate a dictionary of values based on the constants discovered during the constant propagation pass. This includes computed quasi-constants that are often found in EVM bytecode.

Dynamic Contract Creation EFϟCF executes contract constructors in the interpreter once before the start of the fuzzing run and fully supports common patterns such as proxy contracts. However, EFϟCF does not support fuzzing contracts that create other contracts at runtime, as EFϟCF would have to execute previously unknown EVM bytecode, which is not possible in the current ahead-of-time compilation model. When EFϟCF encounters an instruction that creates a new contract, EFϟCF will stop executing the current transaction sequence. However, we only encountered a single contract that creates a new contract at runtime during our evaluation. We believe that these cases are sufficiently rare to leave exploring this as future work.

4.2. Fuzzing Harness

We opted for a lightweight EVM implementation as the base for our fuzzing-optimized EVM runtime. To this end, we adapted the open-source *eEVM* project [13] such that it fits the code-generation of *evm2cpp* and added an implementation of several newer EVM opcodes, missing features, and various minor fixes.

Input Format Within the *eEVM* project, we created a *libfuzzer*-compatible fuzzing harness. The bug oracles are directly integrated into the fuzzing harness and runtime support code of the *eEVM* project. The fuzzing harness features a parser for a custom input format we developed. This format can be quickly parsed without ever failing, i.e., any unneeded data is discarded by the harness; for any missing data fields, default values are assumed. This robust parsing approach allows the use of standard bit-flipping mutations [22] that are unaware of the input structure. The input format consists of an initial header defining the initial environment, followed by a sequence of transactions. Each transaction is represented as a header and the transaction input. Mutating the header for a transaction allows the fuzzer to select transaction-specific parameters, such as the sender account, the call value, and the number of allowed reentrant transactions. For the input data, the parser simply consumes bytes from the fuzzer-provided data according to the input length specified in the header until the end of the fuzzer-provided data. Figure 16 in the Appendix shows an example for a test case generated by EF↯CF to exploit the contract depicted in Figure 2. We designed the input format such that it enables high throughput fuzzing, while being expressive enough to model complex smart contract interactions. Furthermore, we use the input format as a template to synthesize Solidity attack smart contracts that exploit the target. For each attacker-controlled account, we synthesize a Solidity contract that implements the behavior as specified by the generated test case.

Fuzzer and Harness Integration While the fuzzing harness itself is mostly oblivious to the used fuzzer, we opted to rely on AFL++ [22] as one of the most advanced general-purpose fuzzers. AFL++ supports various modern fuzzing techniques, such as collision-free coverage bitmaps, a Redqueen [4] implementation called *cmplog*, and support for custom mutators. Due to our transpilation approach, we are able to directly leverage the instrumentation of AFL++ for smart contract code. We built the fuzzing harness with *clang*, with the highest optimization setting and link-time optimization (LTO) enabled, instrumenting the harness with AFL++'s LTO-based collision-free code coverage instrumentation. Since we have translated the EVM bytecode to C++ code, we can utilize AFL++'s coverage instrumentation to instrument the combination of harness and transpiled smart contracts.

However, we noticed a problem with AFL++'s implementation of the Redqueen mutations [4]. By default, the optimized big integer library used by *eEVM* uses branchless code when comparing the four $64\,\mathrm{bit}$ words that make up a single $256\,\mathrm{bit}$ value. AFL++'s *cmplog* does not detect when only one of the four words matches, as no new code coverage can be observed. As a consequence, it fails to incrementally solve comparisons with large constants. However, this issue is only relevant for bypassing comparisons and not during other arithmetic operations. We opted to manually adapt the relevant functions in the EVM codebase to provide explicit coverage feedback to AFL++. This allows AFL++'s *cmplog* to solve a considerable number of fuzzing roadblocks caused by integer comparisons. To further increase fuzzing efficiency when applying structural mutations in the custom mutator, we added a lightweight tracing mode for certain opcodes (comparisons and returns) to the codebase. This enables us to identify quasi-constants and add them to the dictionary of our custom mutator at runtime.

When fuzzing for reentrancy attacks, we found it beneficial to notify the fuzzer about the call depth of the current execution. To this end we introduce a call-depth-sensitive coverage reporting in the fuzzing harness. Whenever a new basic block is executed, we record the current call depth in AFL++'s coverage map. This allows AFL++ to distinguish executions of the same contract at different call depths. Since, AFL++ receives a new coverage signal when a transaction is executed in a reentrant manner, the test case will be stored in the queue. In turn, this increases the probability of finding reentrancy attacks.

4.3. Custom Mutator

We implemented a mutator library, called *ethmutator*, in roughly $10\,\mathrm{kloc}$ of Rust to efficiently generate and mutate: (1) the structured transaction input expected by the smart contract code, and (2) the transaction sequence input format parsed by the fuzzing harness. The mutator library features a parser and emitter for the binary input format accepted by the harness code. Based on this library, we implement several related tools, such as a structured test case minimizer and an AFL++ custom mutator. The mutator is carefully engineered with high performance in mind. We reduce the number of required allocations and copy operations by applying copy-on-write semantics while performing mutations on a transaction sequence. We also use *mimalloc* [34] to increase the performance of the mutator by a factor of four.

In Ethereum, a transaction is associated with an input field, which is simply a variable-length byte string. Smart contracts use a de-facto standardized ABI format, which specifies how function calls with parameters of complex types are encoded. To enhance the efficiency, we use the ABI information in the *ethmutator* to perform mutations according to the ABI. Unlike existing general-purpose fuzzers, our custom mutator can handle the complexity of the ABI format by acting as a grammar fuzzer for the given ABI and generating structurally-valid inputs based on it. When choosing the values for primitive types, we rely on a fuzzing dictionary built into the custom mutator. Recall that this dictionary is seeded with the constants that are discovered during the analysis pass of *evm2cpp*. In addition, we extend the dictionary with "interesting" values that are likely to trigger bugs (e.g., the dictionary contains the maximum value for every integer type supported by Solidity to make it more likely to trigger integer overflows). When no ABI is available, we exploit the fact that ABI-encoded data is always similarly structured for efficient mutations. For example, when appending a new transaction, we first select a $4\,\mathrm{byte}$ constant from the dictionary as a prefix for the input. Since the basic unit of the ABI is a $256\,\mathrm{bit}$ EVM word, most of the input mutations operate on this word size if no ABI is available.

Another task of the custom mutator is to apply structured mutations to the transaction sequence. We implemented several mutations such as adding, duplicating, or dropping transactions. Furthermore, we implemented more involved mutators such as test case splicing or value propagation between transactions in a sequence. Whenever the base fuzzer adds a test case to its queue, the custom mutator parses this test case and keeps the transaction sequence in memory. The structured splicing mutation then replaces a randomly selected transaction sub-sequence with a sub-sequence obtained from a previous test case. The intuition here is that transaction sequences from prior test cases contain valid transaction combinations. Combining two valid transaction sequences is more likely to result in a new valid transaction sequence. We also propagate values from earlier transactions to later transactions. Hence, with some probability, values in the transaction input will be replaced with values that occurred as parameters in the input of earlier transactions. Similarly, we set the value of *address* types in the ABI to the address of attacker-controlled accounts that previously already sent a transaction. Similar to AFL [63], the custom mutator has multiple stages and a fixed set of mutations that is applied to every test case. Subsequently, random mutations are applied (similar to the *havoc* phase in AFL). The custom mutator also uses stacked mutations, where different random mutation operations are combined.

5. Evaluation

In this section, we evaluate various aspects of EFϟCF and compare them to the current state-of-the-art in fuzzing and symbolic execution. We evaluate EFϟCF with respect to scalability to longer transaction sequences, the test case throughput, and its bug detection capabilities.

5.1. Scalability Benchmarks

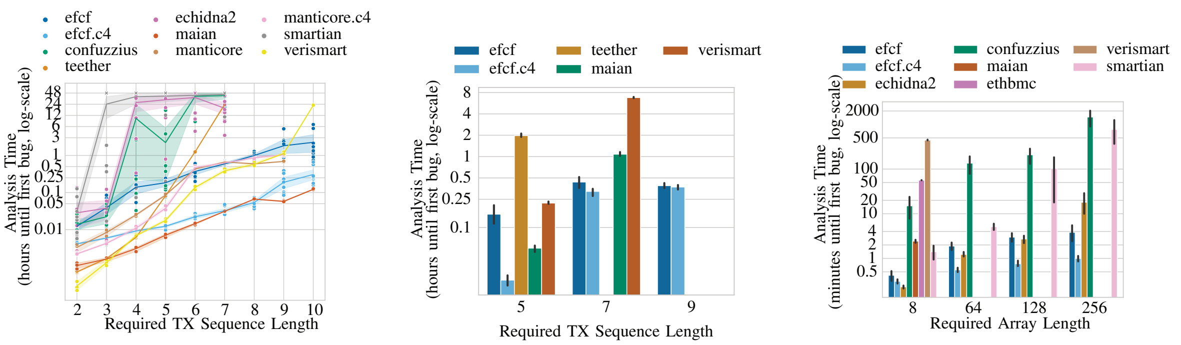
We start with an evaluation of the effectiveness of analysis tools in dealing with an increasing length of transaction sequences. To this end, we created a benchmark consisting of three types of contracts (*multi*, *complex*, and *justlen*), which model different code structures and roadblocks (that hinder analysis) typically found in smart contracts. For each type of contract, we devise several variants (9 for *multi*, 3 for *complex*, and 4 for *justlen*) that require an increasing number of transactions to reach an exploitable state plus another transaction to trigger a vulnerability (see Table 1 for a summary). Note that we cannot use real-world contracts here, as they do not allow us to scale the required number of transactions to trigger a bug. Each variant of *multi* and *complex* contracts is parameterized given the number of transactions that are needed to trigger the bug. For instance, contract $multi_{10}$ requires 10 transactions (or sequential function calls) to reach an exploitable state. The *justlen* example is adapted from Groce et al. [26] and is parameterized over the length of an array that must be reached using operations such as *push* and *pop*. We chose to insert a vulnerability in a function that simply triggers *selfdestruct* of the contract when the exploitable state is reached. This type of bug is widely supported by analysis tools and allows us to compare various tools according to the analysis time required to identify the bug. The benchmarks are constructed to exercise the capability of solving constraints on the inputs (*multi*, *complex*) and the capability of moving a target into a certain internal state (*justlen*). Both capabilities are necessary for analyzing current smart contracts. We provide more details on these benchmark contracts in Appendix C.

Table 1: Capability of analysis tools to identify bugs with increasing transaction sequence length. ✓ bug can be found, × bug never found within 48 h. Type: S symbolic execution, F fuzzer, H hybrid fuzzer.

Tool	Type	multi					complex			justlen			
		2	3-7	8	9	10	5	7	9	8	64	128	256
teEther [33]	S	✓	✓	×	×	×	✓	×	×	×	×	×	×
MAIAN [40]	S	✓	✓	✓	✓	✓	✓	✓	×	✓	×	×	×
EthBMC [23]	S	✓	×	×	×	×	×	×	×	✓	×	×	×
Manticore [37]	S	✓	✓	×	×	×	×	×	×	×	×	×	×
ConFuzzius [53]	H	✓	✓	×	×	×	×	×	×	✓	✓	✓	✓
Echidna [25]	F	✓	✓	✓	×	×	×	×	×	✓	✓	✓	✓
VeriSmart [47]	S	✓	✓	✓	✓	✓	✓	✓	×	✓	×	×	×
Smartian [10]	H	✓	✓	×	×	×	×	×	×	✓	✓	✓	✓
EFϟCF	F	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

We chose several state-of-the-art analysis tools that utilize different approaches to analysis, covering a large spectrum of analysis techniques (fuzzing, symbolic and concolic execution, and hybrids) and implementations. More concretely, we compare EFϟCF with tEther [33], MAIAN [40], EthBMC [23], Manticore [37], VeriSmart (SmarTest) [47, 48], Confuzzius [53], and Echidna [25, 26]. We analyze our benchmark contracts with all these tools and measure the time until the bug is discovered with a global timeout of 48 h. We run all analysis tools within Docker containers on an Intel Xeon Gold 6230 CPU clocked at 2.10 GHz with 188 Gbyte RAM. We run the tools in parallel, keeping all physical CPU cores fully occupied, but do not utilize hyperthreaded cores. All tools were executed on a single core, except those tools that support multi-core analysis, which we also run on 4 cores. To run this experiment, we had to patch the tools tEther [33], MAIAN [40], and ConFuzzius [53] such that they support longer transaction sequences. We excluded EthBMC [23] from most of our experiments since we were not able to patch the bug that causes it to not report any vulnerabilities with transaction sequences longer than 3. Moreover, we excluded ILF [27] because a machine learning-based fuzzer is unlikely to produce the 256 bit magic value constants used in the synthesized contracts. We bound the number of transactions to consider by 32 to ensure that all tools execute at a reasonable pace while leaving enough room for failing/duplicated transactions. For the Python-based tools, we utilize the PyPy JIT-compiler if we observe a speed-up during analysis (for MAIAN and teEther). We perform 3 trials for all symbolic execution tools (where randomness does not play a big role) and at least 10 trials for all fuzzers. In total, we spent approximately 1300 days of CPU time on this experiment.

Our results are summarized in Table 1 and in the plots in Figure 8. The plots show the log-scaled time required to solve the benchmark contracts with an increasing number of required transactions. We omit those tools/results from the plots where all runs fail to identify a bug. EFϟCF is the only analysis tool capable of solving all of the contracts in this benchmark dataset. By adapting state-of-the-art fuzzing techniques, the performance of EFϟCF is comparable to symbolic execution when it comes to



(a) *multi* (b) *complex* (c) *justlen*

Figure 8: Results of scalability experiments showing the analysis time required over the length of transaction sequences. Tools with the postfix *c4* were run in parallel on 4 cores.

overcoming integer constraints. EF⚡CF performs better in the case of path-explosion-inducing code structures, such as loops or array handling.

Symbolic and concolic tools perform very well on the *multi* benchmark because they explore the transaction ordering in a structured manner and can easily solve the integer-based path constraints using SMT solving. However, we observe that most symbolic or concolic tools have trouble coping with the input-dependent loops in the *complex* and *justlen* benchmarks.

Classic fuzzing, as is performed by Echidna, fails to overcome the complex integer path constraints in *multi* and *complex*, but can generate long transaction sequences easily (e.g., on the *justlen* benchmark). Confuzzius and Smartian adopt a hybrid symbolic/fuzzing approach, where the analysis is driven by fuzzing, but inputs are also generated with symbolic constraint solving. We believe that these tools perform poorly because they inherit the slow input generation from symbolic execution but perform probabilistic fuzzing of transaction sequences. We also run a version of the *multi*$_{10}$ contract without any inputs/constraints. It takes Confuzzius and Smartian, a mean of 2827 and 266 minutes, respectively, to find the correctly ordered transaction sequence. In this experiment, it takes EF⚡CF only about 2 minutes.

5.2. Performance Ablation Study

To evaluate the performance impact of various components of EF⚡CF, such as our *evm2cpp* compiler, we perform an ablation study concerning test case throughput and achieved code coverage. We leverage a set of contracts consisting of a mix of real-world contracts and one of our benchmark contracts: the contracts Crowdsale [27] and the synthetic multi$_{10}$ represent simpler contracts, while the IMBTC [18], SpankChain [16], CryptoBets, and PackSale

Table 2: Throughput measurements in average test case executions per second and mean code coverage. Best coverage is underlined.

		Interp		AFL		EM		Full	
Contract	LOC	exec/s	cov %	exec/s	cov %	exec/s	cov %	exec/s	cov %
Crowdsale	41	13,042	79.5	29,688	76.1	19,868	80.6	25,858	87.3
multi$_9$	150	12,814	43.7	43,053	40.8	20,641	75.7	25,817	52.5
IMBTC	664	6880	36.6	28,372	52.6	18,444	36.4	34,510	52.9
PackSale	730	7146	65.0	32,215	67.5	11,952	60.2	24,672	75.2
Spankchain	1048	6574	26.9	19,837	47.0	17,245	50.5	22,698	41.7
CryptoBets	1142	3174	30.8	25,122	35.0	10,256	45.0	12,246	40.5

contracts are more complex real-world contracts. We run each fuzzing configuration 20 times for 10 minutes and record the average test case executions per second along with the EVM basic block coverage. Our evaluation results are summarized in Table 2. We provide further experiments with the *multi*, *complex*, and *justlen* benchmark contracts in Appendix D.

When we disable *evm2cpp* and run the smart contract in the EVM interpreter provided by the *eEVM* project (labeled *Interp* in Table 2), we observe significantly lower test case throughput. This directly translates to achieving less code coverage with the fuzzer, highlighting the importance of *evm2cpp* in our design. For the *AFL* configuration, we disable the custom mutator and for the *EM* configuration we disable AFL's mutations. While the lightweight mutations performed by AFL++ result in the highest throughput, they are not aware of the input structure and often achieve worse code coverage. We noticed that AFL++ alone fails to create combinations of transactions (see Appendix D) because it lacks structural mutations. In this case, basic block coverage is not a good metric since it does not account for different combinations of transactions. The custom mutator alone sometimes cannot discover all interesting code paths due to worse throughput and the lack of AFL++'s advanced mutations. Since fuzzing campaigns utilize multiple cores in practice, we take inspiration from *ensemble fuzzing* [9] and automatically launch different fuzzer configurations in parallel if multiple cores are available to EF⚡CF (see Appendix A).

Comparison with other fuzzers We compare the test case throughput of other fuzzers with the throughput of EF⚡CF. Our measurement results show that the test case throughput of EF⚡CF is larger by an order of magnitude when compared with other fuzzers. Within our throughput benchmark set, EF⚡CF has a mean throughput of 24,301 exec/s ($\sigma = 7154$). Echidna [25] achieves a throughput of 189 ($\sigma = 183$) and a maximum throughput of 497 test cases per second. Confuzzius [53] has a mean throughput of 78 transactions per second ($\sigma = 25$). Notice that the transaction throughput is always higher or equal to the test case throughput since test cases typically consist of multiple transactions.

5.3. Code Coverage Comparison

We compare EFϟCF with the fuzzers ILF [27] and Confuzzius [53] on a set of real-world smart contracts. We cannot compare the fuzzers according to their time-to-bug, since (1) there are no datasets available with both ground truth and realistically complex contracts, and (2) the fuzzers' bug oracles differ too much to be comparable. Instead, we focus on the fuzzers' capability of achieving code coverage, which is a necessary but not a sufficient condition to discover bugs. We were not able to find a way to report basic block coverage in Echidna [25], which we exclude from the comparison. We utilize a set of real-world smart contracts, extracted from the *smartbugs-wild* dataset, which are supported by all fuzzers (see Appendix C for more details).

We configure all fuzzers such that they behave reasonably similarly and offer comparable results. Confuzzius achieves better code coverage when a smart contract contains hard-coded addresses. Only Confuzzius generates transactions originating from those hard-coded addresses. While this behavior does lead to good code coverage, the generated transaction sequences cannot actually be performed on the blockchain, as an arbitrary address cannot be impersonated. For this reason, we disabled marking the *caller*, i.e., the origin of a transaction, as an unconstrained symbolic value in Confuzzius. Additionally, we had to patch Confuzzius's coverage reporting to take bytecode as input instead of source code. We patched ILF to improve the reporting of code coverage and transaction inputs and replaced the existing threshold on the number of generated inputs with a time-based limit. For EFϟCF, we enabled an over-approximating mode, where the data returned by all external calls is fuzzed and allow EFϟCF to generate calls originating from the contract creator.

To compare EFϟCF with the other fuzzers, we follow state-of-the-art recommendations for fuzzer evaluation [3, 32]. We repeat every experiment 30 times to account for the randomness in the fuzzing process and limit the runtime to five minutes for each target. We use the SENF [41] framework to calculate the required statistical evaluation metrics. We find that EFϟCF performs better with statistical significance on 141 targets when compared with Confuzzius and 120 targets when we compare it with ILF. In contrast, Confuzzius and ILF perform better on 83 and 112 of the target contracts, respectively (see Appendix E for more details). We conduct an additional evaluation on the top 100 most complex targets with respect to the number of logical lines of code, functions, comparisons, and branches. As shown in Table 3, EFϟCF outperforms Confuzzius and ILF on the majority of contracts across all complexity properties. Thus, we conclude that EFϟCF can handle increasingly complex Ethereum smart contracts better than existing fuzzers.

Table 3: Number of targets, where EFϟCF statistically significantly outperforms ConFuzzius/ILF and vice versa on the top 100 targets in various complexity properties

	#LLOC	#funcs	#comp	#branch
EFϟCF : ConFuzzius	73 : 17	66 : 22	60 : 28	77 : 15
EFϟCF : ILF	55 : 37	46 : 42	66 : 26	54 : 37

5.4. Bug Detection Capabilities

To assess the bug detection capability of EFϟCF, we tested several real-world contracts obtained from prior studies [6, 8, 23, 64]. Note that after initial testing, we concluded that existing benchmark datasets with ground truth for Solidity/Ethereum analysis tools [12, 24] are not suitable for testing/comparing dynamic analysis tools such as fuzzers. Existing benchmark datasets consist mostly of rather simple and very similar contracts. For example, the curated reentrancy dataset of Durieux et al. [12] features mostly honeypot contracts designed to be easily analyzed [55]. Similarly, the reentrancy bugs injected by Ghaleb et al. [24] are too simplistic: Many of the injected bugs cannot be triggered by dynamic analysis tools (dead code) or are trivially exploitable (see Appendix B). For this reason, we rely on prior studies on real-world contracts for evaluating the bug detection capabilities of EFϟCF.

Access Control Vulnerabilities Access control bugs such as an unprotected *selfdestruct* have been widely investigated [23, 33, 40]. We obtained a list of 2856 contracts that are vulnerable according to EthBMC [23]. We export the blockchain state of the contracts at block number $9{,}069{,}000$, import the state into EFϟCF, and fuzz all contracts until the bug is discovered (with a timeout of $20\,\mathrm{min}$). In total, EFϟCF detects a vulnerability in 2825 out of 2856 contracts. On average it takes EFϟCF $28.5\,\mathrm{s}$ ($\sigma = 111.4$) until the bug is discovered. For 18 contracts, EFϟCF detected no bug in our first run. The remaining contracts had issues due to errors during state export. Among the 18 contracts, we find that 5 are not vulnerable and have been mistakenly marked as vulnerable. We run with a slightly earlier blockchain state and find that EFϟCF identifies another 4 vulnerable contracts. The remaining 12 contracts contain bugs missed by EFϟCF caused by inefficiency when fuzzing without ABI information.

We also applied EFϟCF on $10{,}356$ contracts which EthBMC was unable to analyze because of timeouts after $30\,\mathrm{min}$, i.e., contracts that cannot be analyzed with bounded model checking. In contrast, EFϟCF successfully processed all these contracts and achieves an average code coverage of $72.4\,\%$ ($\sigma = 17.9$). Furthermore, EFϟCF detected 85 vulnerable contracts in this set, of which we manually checked 18 contracts with verified source code and found 7 true vulnerabilities. For the remaining 11 contracts, EFϟCF correctly identifies a transaction sequence to gain Ether. However, these contracts intentionally implement features that allow any user to obtain Ether. This is common for gambling contracts or contracts that pay out dividends (see Section 6).

Reentrancy Vulnerabilities We evaluated EFϟCF using a set of contracts vulnerable to reentrancy according to prior studies [6, 8, 43, 64]. Furthermore, to give an intuition about detection capabilities we also provide results for Confuzzius [53] and the static source code analyzer Slither [19] in Table 4. In contrast to EFϟCF, which generates reentrancy exploits, Confuzzius and Slither feature a heuristic detection of reentrancy issues. Slither defines any state update after an external call to be a potential reentrancy bug. Similarly, Confuzzius defines a reentrancy bug as an external call, where some state variable is read before the call and written after the call. Confuzzius does not actually generate transaction sequences that contain reentrant transactions. We filter out wrongly detected reentrancy bugs by manually analyzing the reported contracts from prior studies [43, 64]. Furthermore, many cases are trivial reentrancy bugs, summarized as *Trivial-RE* in Table 4, which includes reentrancy honeypot contracts [55] (see Appendix F).

EFϟCF is highly effective in discovering all known reentrancy issues. However, the prototype implementation of EFϟCF does not yet support contract creation at runtime (see Section 4.1). We noticed dynamic contract creation for “the DAO” contract, which is the reason EFϟCF does not detect the DAO reentrancy. The *HODLWallet* contract requires special attention: While this contract is vulnerable to a reentrancy bug, it cannot be exploited to gain Ether. According to our analysis, this contract allows users to invest Ether into the contract, but the contract never returns all the invested Ether. However, a reentrancy bug in the contract can be exploited to withdraw all previously invested Ether. EFϟCF’s bug oracle does not identify this as a reentrancy bug since no Ether can actually be gained. For the *InstaDice* contract, Confuzzius reports many additional reentrancy issues even when the contract calls into other trusted contracts instead. EFϟCF, on the other hand, executes trusted contracts exported from the Ethereum node and produces a fully working exploit for the reentrancy bug without reporting false alarms. Remarkably, EFϟCF is the *only* dynamic analysis tool that is able to accurately identify real-world reentrancy issues such as the reentrancy bugs in the SpankChain and DSEthToken contracts.

We also evaluated against the reentrancy bugs recently discovered by Bose et al. [6] with the SAILFISH tool. The study reports 26 contracts with true reentrancy bugs in the dataset. However, we were able to confirm only 5 of these contracts to be vulnerable to Ether stealing with EFϟCF. Our (manual) analysis on the remaining 21 contracts reveals that, while the contracts can be reentered in theory, all but one cannot be exploited (see Appendix G). We also identified one contract in this set that can be exploited because of an access control bug, not because of reentrancy. This shows that accurate bug oracles, such as those used by EFϟCF, are less likely to produce false alarms and therefore give better feedback to smart contract developers by providing concrete transaction sequences that trigger the reentrancy bug.

Compositional Security We follow the evaluation of the Serif static analyzer [8] to show the feasibility of detecting compositional security violations with EFϟCF.

Table 4: Results for reentrancy issues for various analysis tools: False Alarms (~), True Alarms (✓), not applicable/incompatible (N/A), or as Missed Bug (×).

Contract	EFϟCF	Confuzzius	Slither
Example Figure 2	✓	~	~
SpankChain [16]	✓	×	✓
DSEthToken [43]	✓	✓	N/A
TheDAO [43]	N/A	✓	N/A
HODLWallet [14, 64]	×	✓	✓
SysEscrow [17, 64]	✓	×	✓
InstaDice [15, 64]	✓	~	✓
Trivial-RE [55]	✓	✓	✓

We adapted the contracts from Serif’s evaluation set such that they are deployable and exploitable in a realistic setting. Where Serif relies on manual annotations to detect potential problems, we augment the contracts with assertions that are picked up by EFϟCF to detect vulnerabilities beyond Ether-stealing. EFϟCF accurately generates a reentrancy attack for the Uniswap and Multi-DAO contracts in this dataset. Furthermore, EFϟCF generates transactions that violate the assertions for the KV-Store and TownCrier contracts using reentrant transactions. To evaluate EFϟCF’s capabilities on a real-world example, we also fuzz the composition of Uniswap and IMBTC, which are exploitable due to a reentrancy bug [54]. We supply EFϟCF with the addresses of three contracts that should receive transactions (Uniswap, IMBTC and the ERC1820Registry) and export the state of these contracts at block number $9{,}600{,}000$, shortly before the first known attack. We then fuzz this composition on 40 cores for a maximum of $48\,\mathrm{h}$. We repeated the experiment 10 times and find that EFϟCF on average requires $1\,\mathrm{h}$ and $49\,\mathrm{min}$ ($\sigma \approx 14\,\mathrm{h}35\,\mathrm{min}$, geomean $6\,\mathrm{h}55\,\mathrm{min}$) to generate a reentrancy exploit that (1) registers an attacker contract in the *ERC1820Registry* to allow *ERC777* callbacks, (2) buys *IMBTC* tokens via the *Uniswap* contract, and (3) finally exploits the reentrancy to sell them again with a profit.

6. Discussion / Related Work

Fuzzing Structured Input On the protocol level, inputs for smart contracts are a byte string encoded according to the *ABI* definition. A fuzzer must provide inputs that are valid according to the ABI or the contract will stop execution early. Symbolic execution tools handle this by utilizing the SMT solver to identify valid input data. Most smart contract fuzzers [25, 27, 39] use an approach akin to grammar-fuzzing [7, 62] to generate inputs with the ABI. While EFϟCF also utilizes the ABI to efficiently mutate transaction inputs, it does not solely rely on the *ABI*. EFϟCF leverages the lightweight mutations of its base fuzzer to mutate raw input. Using the coverage feedback, EFϟCF discovers structurally valid inputs even without the ABI. This allows EFϟCF to fuzz contracts where no source code or only an incomplete ABI is available. For example, EFϟCF can fuzz inputs that contain byte strings that are further decoded elsewhere, e.g., because the data is forwarded to another smart contract. EFϟCF also fuzzes the return data of external calls, which is ABI-encoded but not included in a contract’s ABI. However, fuzzing without ABI information is currently less well-optimized in EFϟCF (see Appendix D).

Bug Oracles There is a wide spectrum of bug oracles in smart contracts. Analysis tools define their own bug oracles, sometimes with slightly different definitions of the same bug classes. Many analysis tools [6, 20, 27, 35, 53] feature bug oracles that indicate issues, but not necessarily a security vulnerability. For example, detecting a data dependency on the block timestamp might be a sign of a bad attempt at using randomness, or it might be a legitimate use to implement a time-limited sale. While such oracles also result in a larger number of alarms, they have the advantage that they can uncover a larger set of issues. Developers can use these findings to improve code quality. Other analysis tools implement Ether-based bug oracles for exploit generation [23, 33, 40] with few false alarms. EFϟCF also utilizes such a bug oracle but also covers complex interactions (e.g., reentrancy). The downside is that token-related vulnerabilities cannot be directly detected using this bug oracle. However, similar to Echidna [25], EFϟCF supports using developer annotations as bug oracles, allowing EFϟCF to identify token-related bugs and other logic bugs.

Simulating Benign Interactions An important property of a fuzzer is whether it simulates the behavior of benign users, i.e., whether the fuzzer can generate transaction sequences of the form $(t_u, t_a, t'_u, t'_a, \ldots)$, where t_u and t'_u are from a benign user, and t_a and t'_a are from an attacker. For example, many smart contracts implement the *Owned* pattern: There is one Ethereum account that has special privileges for the contract. If the fuzzer aims for optimal code coverage, then simulating arbitrary addresses is beneficial to reach code paths guarded by access control checks (e.g., implemented in Confuzzius [53] and ILF [27]). While this approach leads to good code coverage, it also entails more false alarms. For example, in contracts with transferable ownership, the fuzzer will make the simulated owner transfer the ownership to an attacker account. In turn, this would allow the attacker to drain the funds of the contract. This is a false alarm since the real owner would never transfer ownership to an attacker. However, this *Owned* pattern is prevalent in smart contracts, making existing fuzzers that adopt this behavior produce false alarms. We therefore opted to disable the simulation of any non-attacker-controlled accounts in EFϟCF.

EFϟCF only fuzzes return data that is attacker-controlled, i.e., data returned by callbacks. By default, EFϟCF stops execution if an unknown contract is called. EFϟCF requires all contract dependencies to be set up using a custom blockchain state. Optionally, EFϟCF supports a mode that mutates data returned by any external call, similar to e.g., Confuzzius [53]. However, this mode is disabled by default to minimize false alarms. For example, we observed that when testing a contract that relies on a token contract to manage user balances, mutating all returned data would result in impossible paths being executed, e.g., two subsequent calls to *getBalance* returning different values.

False Alarms and Missed Bugs Like any fuzzer, EFϟCF provides neither sound nor complete analysis. However, we designed EFϟCF as an exploit generator with few false alarms at the cost of missing some bugs. Other analysis tools [6, 53] focus on detecting reentrancy patterns, but do not generate an exploit. As such, it is not clear whether their reported findings are true vulnerabilities, and often they are not (e.g., see our comparison with Sailfish in Section 5.4 and Appendix G). In contrast, EFϟCF only reports reentrancy attacks that are exploitable, leading to fewer false alarms compared to prior work. The downside of EFϟCF's approach to reentrancy is that it misses bugs that are not covered by the Ether-gains oracle or custom assertions.

The Ether-gains bug oracle does report false alarms in some edge cases. In our evaluation, we identified several types of contracts that repeatedly lead to false alarms. (1) Gambling contracts, where EFϟCF identifies the right blockchain state and input such that the attacker always wins. (2) Token airdrops, where EFϟCF deterministically triggers the airdrop, which often results in Ether gains by selling the airdropped tokens again. (3) Interest pay-outs, where EFϟCF flags interest or dividends that are paid out over time. In all these scenarios, there is a way to gain Ether from those contracts, which EF/CF correctly reports. However, these are not considered vulnerabilities, as the contracts are intended to give out Ether.

We identified two major reasons for EFϟCF missing bugs: (1) The bug cannot be uncovered by the Ether-gains bug oracle. (2) The vulnerable state is not reachable within EFϟCF's EVM environment because some prerequisite is missing. For example, the target depends on a second contract not available to EFϟCF, or the contract requires additional state setup by the contract's owner (e.g., the *Pausable* pattern).

Testing Multi-Contract Setups Smart contracts increasingly feature dependencies on third-party contracts. Various recent incidents [5, 21, 54] show that multi-contract analysis is required for automatic analysis of complex DeFI applications that consist of compositions of smart contracts that have been independently developed. Here, *compositional security* is an important security property [8]. We show that EFϟCF can handle complex inputs and dependencies between transactions. Similar to analysis tools [23, 25], EFϟCF fully supports calling other contracts that act as dependencies. Recently, Echidna [25] introduced a *multi-abi* mode, where the fuzzer is allowed to call functions on multiple smart contracts. Similarly, EFϟCF supports generating transactions targeting multiple different smart contracts to identify issues due to unsuspected state changes in a contract's dependencies. In contrast to Echidna, EFϟCF also supports reentrant transactions to different contracts, allowing it to detect compositional reentrancy such as the Uniswap/IMBTC issue.

7. Conclusion

There is a demand for developing efficient and scalable techniques for security testing of smart contracts, which are being increasingly used to encode complex business logic on blockchain platforms. We show that high-throughput fuzzing, as implemented in EFϟCF, improves on prior analysis tools, including increased code coverage, reduced time to discover bugs, the capability to model complex interactions (such as reentrancy), and the capability to analyze even complex contracts and compositions of contracts. We show that several optimizations facilitate the high fuzzing throughput: translating contract bytecode to native code and employing efficient structural mutations on transaction sequences and the associated transaction inputs. EFϟCF has comparable capabilities in solving complex input constraints as symbolic execution tools, while gracefully handling contracts that induce path explosion. We release EFϟCF along with all of our benchmarks as open-source software. We hope that this allows efficient automated testing of contracts and fosters additional research in smart contract security.

Acknowledgment

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, the German Federal Ministry of Education and Research (BMBF, project iBlockchain – 16KIS0901K), and the European Union (ERC, CONSEC, No. 101042266). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. The authors would like to thank Alexander Meyring, Justin Nobles, and Simon Janzon for their contributions to the implementation of EF↯CF as part of their project group at University of Duisburg-Essen.

References

[1] Dave Aitel. *The advantages of block-based protocol analysis for security testing*. Tech. rep. 2002. URL: http://www.immunityinc.com/downloads/advantages_of_block_based_analysis.pdf (cit. on p. 3).

[2] *Android Runtime (ART) and Dalvik*. URL: https://source.android.com/docs/core/runtime (visited on 10/24/2022) (cit. on p. 6).

[3] Andrea Arcuri and Lionel Briand. "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering". In: *Software Testing, Verification and Reliability* (2014). DOI: 10.1002/stvr.1486 (cit. on p. 12).

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. "REDQUEEN: Fuzzing with Input-to-State Correspondence". In: *Proceedings 2019 Network and Distributed System Security Symposium*. NDSS. Internet Society, 2019. DOI: 10.14722/ndss.2019.23371 (cit. on pp. 9, 17, 20).

[5] BlockSec. *Revest Finance Vulnerabilities: More than Re-entrancy*. Mar. 2022. URL: https://blocksecteam.medium.com/revest-finance-vulnerabilities-more-than-re-entrancy-1609957b742f (visited on 06/07/2022) (cit. on pp. 1, 5, 14).

[6] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. "SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds". In: *43rd IEEE Symposium on Security and Privacy*. S&P. IEEE, 2022. DOI: 10.1109/SP46214.2022.9833721 (cit. on pp. 2, 12–14, 22).

[7] W H Burkhardt. "Generating test programs from syntax". In: (Mar. 1967). DOI: 10.1007/BF02235512 (cit. on p. 13).

[8] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. "Compositional Security for Reentrant Applications". In: *42nd IEEE Symposium on Security and Privacy*. S&P. IEEE, 2021. DOI: 10.1109/SP40001.2021.00084 (cit. on pp. 2, 5, 6, 12–14).

[9] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. "EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers". In: *28th USENIX Security Symposium*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang (cit. on p. 11).

[10] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. "SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses". In: *36th IEEE/ACM International Conference on Automated Software Engineering*. ASE. IEEE, 2021. DOI: 10.1109/ASE51524.2021.9678888 (cit. on pp. 1, 2, 7, 10, 18).

[11] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP. 2000. DOI: 10.1145/351240.351266 (cit. on p. 3).

[12] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE. 2020. DOI: 10.1145/3377811.3380364 (cit. on pp. 12, 19, 20, 22).

[13] Microsoft. *Enclave EVM*. URL: https://github.com/microsoft/eEVM (cit. on pp. 6, 9, 17).

[14] *etherscan.io: HODLWallet*. URL: https://etherscan.io/address/0x4a8d3a662e0fd6a8bd39ed0f91e4c1b729c81a38 (visited on 12/12/2021) (cit. on p. 13).

[15] *etherscan.io: InstaDice*. URL: https://etherscan.io/address/0xfe1b613f17f984e27239b0b2dccfb1778888dfae (visited on 12/12/2021) (cit. on p. 13).

[16] *etherscan.io: LedgerChannel (SpankChain)*. URL: https://etherscan.io/address/0xf91546835f756da0c10cfa0cda95b15577b84aa7 (visited on 12/12/2021) (cit. on pp. 11, 13).

[17] *etherscan.io: SysEscrow*. URL: https://etherscan.io/address/0x903643251af408a3c5269c836b9a2a4a1f04d1cf (visited on 12/12/2021) (cit. on p. 13).

[18] *etherscan.io: The Tokenized Bitcoin (imBTC)*. URL: https://etherscan.io/address/0x3212b29E33587A00FB1C83346f5dBFA69A458923 (visited on 12/12/2021) (cit. on p. 11).

[19] Josselin Feist, Gustavo Grieco, and Alex Groce. "Slither: a static analysis framework for smart contracts". In: *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. WETSEB@ICSE. IEEE, 2019. DOI: 10.1109/WETSEB.2019.00008 (cit. on pp. 2–4, 12).

[20] Christof Ferreira-Torres, Julian Schütte, and Radu State. "Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC. 2018. DOI: 10.1145/3274694.3274737 (cit. on pp. 7, 13).

[21] C. R. E. A. M Finance. *C.R.E.A.M. Finance Post Mortem: AMP Exploit*. Sept. 2021. URL: https://medium.com/cream-finance/c-r-e-a-m-finance-post-mortem-amp-exploit-6ceb20a630c5 (visited on 06/07/2022) (cit. on pp. 1, 5, 14).

[22] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++: Combining incremental steps of fuzzing research". In: *14th USENIX Workshop on Offensive Technologies*. WOOT. 2020 (cit. on pp. 2, 3, 9, 17).

[23] Joel Frank, Cornelius Aschermann, and Thorsten Holz. "ETHBMC: A Bounded Model Checker for Smart Contracts". In: *29th USENIX Security Symposium*. USENIX Association, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/frank (cit. on pp. 1–4, 6, 7, 10, 12, 14).

[24] Asem Ghaleb and Karthik Pattabiraman. "How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA. 2020 (cit. on pp. 12, 19).

[25] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. "Echidna: effective, usable, and fast fuzzing for smart contracts". In: *29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA. ACM, 2020. DOI: 10.1145/3395363.3404366 (cit. on pp. 1–4, 6, 7, 10–14, 18).

[26] Alex Groce and Gustavo Grieco. "echidna-parade: a tool for diverse multicore smart contract fuzzing". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA. 2021. DOI: 10.1145/3460319.3469076 (cit. on pp. 10, 21).

[27] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. "Learning to Fuzz from Symbolic Execution with Application to Smart Contracts". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2019. DOI: 10.1145/3319535.3363230 (cit. on pp. 1–3, 7, 10–14, 18).

[28] Christoph Jentzsch. *The History of the DAO and Lessons Learned*. URL: https://blog.slock.it/the-history-of-the-dao-and-lessons-learned-d06740f8cfa5 (cit. on pp. 1, 3).

[29] Bo Jiang, Ye Liu, and W. K. Chan. "ContractFuzzer: fuzzing smart contracts for vulnerability detection". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ASE. ACM, 2018. DOI: 10.1145/3238147.3238177 (cit. on pp. 1, 5, 7).

[30] Rauli Kaksonen, Marko Laakso, and Ari Takanen. "Software Security Assessment through Specification Mutations and Fault Injection". In: *Communications and Multimedia Security Issues of the New Century: IFIP TC6 / TC11 Fifth Joint Working Con-*

ference on Communications and Multimedia Security (CMS'01). 2001. DOI: 10.1007/978-0-387-35413-2_16 (cit. on p. 3).

[31] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. "ZEUS: Analyzing Safety of Smart Contracts". In: *25th Annual Network and Distributed System Security Symposium*. NDSS. The Internet Society, 2018 (cit. on pp. 1, 3).

[32] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. "Evaluating Fuzz Testing". In: *ACM Conference on Computer and Communications Security (CCS)*. 2018. DOI: 10.1145/3243734.3243804 (cit. on p. 12).

[33] Johannes Krupp and Christian Rossow. "teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts". In: *27th USENIX Security Symposium*. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp (cit. on pp. 1–7, 10, 12, 14).

[34] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. "Mimalloc: Free List Sharding in Action". In: *Programming Languages and Systems - 17th Asian Symposium*. APLAS. 2019. DOI: 10.1007/978-3-030-34175-6_13 (cit. on pp. 9, 17).

[35] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. "Making Smart Contracts Smarter". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2016 (cit. on pp. 1, 13, 18).

[36] Valentin J M Manes, Hyungseok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. "The art, science, and engineering of fuzzing: A survey". In: *IEEE Trans. Software Eng.* 47.11 (Nov. 2021), pp. 2312–2331. ISSN: 0098-5589, 1939-3520. DOI: 10.1109/tse.2019.2946563 (cit. on p. 3).

[37] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts". In: *34th IEEE/ACM International Conference on Automated Software Engineering*. ASE. IEEE, 2019. DOI: 10.1109/ASE.2019.00133 (cit. on pp. 2, 3, 6, 10).

[38] ConsenSys. *Mythril v0.22.1*. URL: https://github.com/ConsenSys/mythril (cit. on pp. 2, 7, 18).

[39] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. "sFuzz: an efficient adaptive fuzzer for solidity smart contracts". In: *ICSE '20: 42nd International Conference on Software Engineering*. Ed. by Gregg Rothermel and Doo-Hwan Bae. ACM, 2020. DOI: 10.1145/3377811.3380334 (cit. on pp. 1, 3, 7, 13).

[40] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC. 2018. DOI: 10.1145/3274694.3274743 (cit. on pp. 1–4, 6, 7, 10, 12, 14, 18, 19).

[41] David Paaßen, Sebastian Surminski, Michael Rodler, and Lucas Davi. "My Fuzzer Beats Them All! Developing a Framework for Fair Evaluation and Comparison of Fuzzers". In: *Proc. of European Symposium on Research in Computer Security*. ESORICS. Springer International Publishing, 2021. DOI: 10.1007/978-3-030-88418-5_9 (cit. on p. 12).

[42] Gregory Popovitch. *The Parallel Hashmap*. Mar. 2019. URL: https://greg7mdp.github.io/parallel-hashmap/ (visited on 04/22/2022) (cit. on p. 17).

[43] Michael Rodler, Wenting Li, Ghassan Karame, and Lucas Davi. "Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks". In: *Proceedings of the Network and Distributed System Security Symposium*. NDSS. 2019. URL: https://www.ndss-symposium.org/ndss-paper/sereum-protecting-existing-smart-contracts-against-re-entrancy-attacks/ (cit. on pp. 2, 7, 12, 13, 19).

[44] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. "eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts". In: *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2020. DOI: 10.1145/3372297.3417250 (cit. on p. 1).

[45] Clara Schneidewind, Markus Scherer, and Matteo Maffei. "The Good, The Bad and The Ugly: Pitfalls and Best Practices in Automated Sound Static Analysis of Ethereum Smart Contracts". In: *Leveraging Applications of Formal Methods, Verification and Validation: Applications*. Springer International Publishing, 2020. DOI: 10.1007/978-3-030-61467-6_14 (cit. on p. 22).

[46] Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. "On fast large-scale program analysis in Datalog". In: *Proceedings of the 25th International Conference on Compiler Construction*. CC. 2016. DOI: 10.1145/2892208.2892226 (cit. on p. 6).

[47] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. "SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution". In: *30th USENIX Security Symposium*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/so (cit. on pp. 2, 10).

[48] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. "VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts". In: *2020 IEEE Symposium on Security and Privacy*. SP. IEEE, 2020. DOI: 10.1109/SP40000.2020.00032 (cit. on pp. 2, 10).

[49] *Solidity by Example: Re-Entrancy*. URL: https://solidity-by-example.org/hacks/re-entrancy/ (visited on 12/12/2021) (cit. on p. 4).

[50] *Solidity: Security Considerations - Use the Checks-Effects-Interactions Pattern*. URL: https://docs.soliditylang.org/en/v0.8.7/security-considerations.html#use-the-checks-effects-interactions-pattern (visited on 09/09/2021) (cit. on p. 3).

[51] Parity Technologies. *A Postmortem on the Parity Multi-Sig Library Self-Destruct*. Nov. 2017. URL: http://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct (visited on 06/03/2020) (cit. on p. 6).

[52] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. "Synthesizing method sequences for high-coverage testing". In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA. 2011. DOI: 10.1145/2048066.2048083 (cit. on p. 3).

[53] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. "ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts". In: *IEEE European Symposium on Security and Privacy*. EuroS&P. IEEE, 2021. DOI: 10.1109/EuroSP51992.2021.00018 (cit. on pp. 2–7, 10–14, 18).

[54] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. "The Eye of Horus: Spotting and Analyzing Attacks on Ethereum Smart Contracts". In: *Financial Cryptography and Data Security - 25th International Conference*. FC. Springer, 2021. DOI: 10.1007/978-3-662-64322-8_2 (cit. on pp. 1, 2, 5, 13, 14, 22).

[55] Christof Ferreira Torres, Mathis Steichen, and Radu State. "The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts". In: *28th USENIX Security Symposium*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira (cit. on pp. 12, 13, 22).

[56] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. "Securify: Practical security analysis of smart contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2018 (cit. on pp. 1–4).

[57] *Unity Documentation: IL2CPP Overview*. URL: https://docs.unity3d.com/Manual/IL2CPP.html (visited on 09/14/2021) (cit. on p. 6).

[58] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. "Initialize once, start fast: application initialization at build time". In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019). DOI: 10.1145/3360610 (cit. on p. 6).

[59] Valentin Wüstholz and Maria Christakis. "Harvey: a greybox fuzzer for smart contracts". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. ACM, Nov. 2020. DOI: 10.1145/3368089.3417064 (cit. on pp. 1, 3, 18).

[60] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. "Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution". In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference*. TACAS. 2005. DOI: 10.1007/978-3-540-31980-1_24 (cit. on p. 3).

[61] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. "Designing New Operating Primitives to Improve Fuzzing Performance". In: *Proceedings of the 2017 ACM SIGSAC Confer-*

ence on Computer and Communications Security. CCS. 2017. DOI: 10.1145/3133956.3134046 (cit. on p. 3).

[62] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, June 2011. DOI: 10.1145/1993498.1993532 (cit. on p. 13).

[63] Michal Zalewski. *American Fuzzy Lop*. URL: https://lcamtuf.coredump.cx/afl/ (visited on 04/28/2022) (cit. on pp. 3, 10).

[64] Shunfan Zhou, Zhemin Yang, Jie Xiang, Yinzhi Cao, Min Yang, and Yuan Zhang. "An Ever-evolving Game: Evaluation of Real-world Attacks and Defenses in Ethereum Ecosystem". In: *29th USENIX Security Symposium*. USENIX Association, 2020 (cit. on pp. 1, 12, 13).

A. Implementation Details

EVM Changes Similar to many dynamic analysis tools we use a custom Ethereum virtual machine (EVM) environment. This allows us to optimize the EVM for fuzzing by adding instrumentation and bug oracles. We used the open-source *eEVM* project [13] as the base for our EVM environment. We added an implementation of several newer EVM opcodes, missing features, and various minor fixes. Furthermore, we replaced the usage of C++ exceptions with return values in hot code paths. This results in considerably better performance since fuzzing tends to frequently exercise the error handling paths of the smart contract code. We also switched to a more optimized hashmap implementation [42] and use *mimalloc* [34] as the default allocator.

Test Case Format Figure 16 depicts an example of the test case format used by EF↯CF. During fuzzing EF↯CF utilizes a custom binary format that can be mutated by the base fuzzer using normal bit-flipping mutations without inducing parsing failures. However, for introspection we convert the test cases to a yaml-based textual representation, which is shown in Figure 16. Here the transactions are depicted as a sequence of transactions. However, the second transaction specifies a return header with the *reenter* flag set to two. This means that the following two transactions are executed as reentrant calls.

In EF↯CF, every transaction has up to 255 associated *return-headers*. Each return header specifies what the simulated attacker contracts do when they are called by the target contract, i.e., a *callback*. The return header contains the return value, the return data, and the reenter field. Mutating the reenter field allows the fuzzer to explore different shapes of call trees that represent different kinds of reentrancy attacks. Figure 4 depicts several examples of call-trees that EF↯CF is capable of generating.

AFL++ Integration We patched AFL++ for optimal integration with our custom mutator. Our patches make AFL++ report an internal performance score to our custom mutator. The custom mutator can then select the number of fuzzing rounds for a given test case based on this score. Depending on the size and complexity of the test case, we apply different types of mutations and a different number of fuzzing rounds in the custom mutator.

We ensure that AFL++ extensively uses the structured trimming provided by our custom mutator. We found that structured trimming is beneficial to the fuzzing process in EF↯CF. Additionally, we utilize the trimming step of AFL++ to update the internal test case queue of our custom mutator. This allows us to only add trimmed test cases to the internal queue. In contrast to AFL++'s queue, we keep the complete internal queue in memory and use it for efficient structural splicing operations.

We also extend AFL++ with an additional manual feedback API with a function that allows to reserve a larger part of AFL++'s coverage map for direct feedback. This is used by our modifications to the *eEVM* runtime to provide explicit feedback on two properties of the execution. First, we provide explicit feedback on the progress of solving comparison operators. For example, for the `EQ` opcode we provide explicit coverage feedback to AFL++ for every 64 bit part of the 256 bit EVM-native integer that is equal. This allows AFL++'s *cmplog* mode, which implements input-to-state correspondence on the 64 bit comparison level, to effectively solve fuzzing roadblocks on EVM-native 256 bit level. Similarly, we provide explicit coverage feedback to AFL++ whenever a contract executes in a reentrant call. This allows the fuzzer to distinguish a reentrant execution from a normal execution, i.e., a simple form of context-sensitive coverage. We found this beneficial for AFL++ to keep both reentrant and non-reentrant variants of the same transaction sequence in the queue.

When launching AFL++, we disable the byte-level *auto-dict* feature since it is superseded by our replacement acting on the EVM bytecode level. Re-implementing the *auto-dict* feature in *evm2cpp* results in significantly smaller and more useful dictionaries than relying on AFL++'s auto-dict mode, which scans the final binary for constants and also picks up irrelevant data, such as strings only relevant for the *eEVM* runtime.

Multi-Core Fuzzing While AFL++ has a single-threaded design, it is capable of synching with other instances of AFL++ (and even other fuzzers) via the filesystem. EF↯CF inherits the same technique, and the wrapper scripts we provide as part of EF↯CF can automatically launch multiple AFL++ instances. Generally, it is recommended to launch AFL++ using multiple different configurations when using multiple cores [22]. We adapt these recommendations to EF↯CF. When running on 4 or more cores, we launch the following configurations:

1) A main instance with AFL++'s deterministic mutation stages enabled.
2) A compare solver instance, with input-to-state [4] and EVM-level compare tracing enabled.
3) One instance fuzzes only with the custom mutator.
4) The remaining cores utilize AFL++'s lightweight havoc mutations and our custom mutator.

Other Bug Oracles The standard bug oracles are implemented inside of the *eEVM* runtime code. To implement a new bug oracle, one has to modify the C++ implementation of the runtime code. For performance reasons, EF↯CF does not rely on heavyweight program analysis techniques such as taint tracking to implement bug oracles. Instead, the bug oracles in EF↯CF are limited to detecting bugs based on the state of the simulated Ethereum blockchain. However, we believe that the existing bug oracles supported by EF↯CF already cover a large set of use cases.

```
contract CrowdFund {
  uint goal;
  uint closetime;
  bool is_closed;

// invariant: if the crowdfunding is closed then
// either the time ran out or the funding goal
// was met.
function property_check() public view returns(bool)
{
  return (!is_closed) ||
         (address(this).balance > goal
          || block.timestamp > closetime);
}

  // [...]
}
```

Figure 9: EF⚡CF supports property-based fuzzing. The developer specifies a custom function that checks a property of a smart contract that must always hold. EF⚡CF repeatedly calls this function after every executed transaction to check whether the property still holds and reports a bug if not.

Furthermore, developers can use custom properties or the event mechanism to implement custom bug oracles directly in Solidity code.

Currently, EF⚡CF supports optional fuzzing modes, which are also used by industry fuzzers such as Echidna [25] or Mythril [38, 59]. For example, EF⚡CF supports property-based fuzzing with an interface that is fully compatible with the Echidna fuzzer. The developer specifies a property of the contract that must always hold, i.e., an invariant of the contract code. Such properties are specified as a Solidity function that returns whether the property is currently true or false. After every executed transaction, EF⚡CF calls the configured property functions and checks whether the return value signals a violated property. Similarly, EF⚡CF can utilize the EVM event logging and error propagation mechanisms to detect bugs. The smart contract developer emits a certain event whenever a bug is triggered. Whenever this event is logged during fuzzing, EF⚡CF will consider the execution to trigger a bug and report it. Similarly, Solidity version 0.8 or above report special error messages to the caller, whenever an assertion is violated or an integer overflow happens. If configured, EF⚡CF picks up these special error message return codes as a bug and reports it. This way EF⚡CF can be utilized to fuzz for more than Ether-based bugs and also uncover contract-specific logic bugs. Figure 9 shows an example for an invariant specified as a solidity function. EF⚡CF checks whether this function returns true after every executed transaction.

Comparison of Bug Oracles Previous analysis tools often implement a wide variety of bug oracles [10, 35, 40, 53] to detect security vulnerabilities, code smells, and other potentially interesting properties of the code. However, the definition of the bug oracles and what oracles should be considered as a security vulnerability differ across the literature. We identify the *unprotected selfdestruct* bug oracles as one of the few oracles that are recognized in almost all analysis tools. We utilize this bug oracle in our benchmarks (see Section 5 and Appendix C). In EF⚡CF, we focus on Ether gains as our primary bug oracle, as it features the least number of false alarms in practice. However, this single bug oracle in EF⚡CF actually maps to multiple bug oracles in other tools. Furthermore, we implement several additional optional bug oracles that can be used in EF⚡CF. We show a comparison of supported bug oracles in Table 5. In the following, we discuss some of the bug oracles in more detail.

Locking Ether is fundamentally a liveness property. In general, liveness properties are hard to prove with a fuzzer. A standard fuzzing approach can only show that a certain code path *can* be reached. However, to accurately report locked Ether, the fuzzer would have to show that a certain code path cannot be reached. For this reason, many fuzzers actually implement a static analysis approach to detecting locked Ether. For example, ILF [27] and Confuzzius [53] simply scan the contract for any instruction that can, in theory, send Ether. However, they do not verify that the instruction can actually be executed. For this reason, this approach will only detect simple cases of locked Ether.

Leaking Ether and *Ether Gains* are two very related bug oracles. Both attempt to identify bugs where the contract can be used to send Ether to some unrelated address. EF⚡CF supports detecting leaking Ether but disables the bug oracle by default. The idea is that the attacker can trick the contract into sending Ether to some contract that has no previous relationship with the contract. However, many contracts support transferring Ether indirectly to another address as a feature. For example, all token contracts must support transferring tokens to arbitrary addresses as a feature. In contrast, EF⚡CF uses Ether gains as a bug oracle that covers more realistic cases. EF⚡CF will report an Ether gain bug whenever the sum of the Ether balances of all attacker-controlled accounts exceeds the initial sum of balances. This allows EF⚡CF a wider and more realistic set of issues. For example, the vulnerability depicted in Figure 6 is detected neither by Confuzzius [53] nor by Smartian [10]. However, since EF⚡CF simulates multiple attacker-controlled accounts, it will quickly generate a transaction originating from the first account and leaking the Ether to a second attacker-controlled address.

Most analysis tools feature explicit *Reentrancy* bug oracles. In contrast, EF⚡CF does not feature an explicit detector for reentrancy but simply generates reentrant

Table 5: Comparison of bug oracles in various fuzzing-based analysis tools with the bug oracles available in EF⚡CF. ✓ fully supported. × not supported. ✓* supported but not enabled by default. ✓† only supported for contract compiled with Solidity version > 0.8. ×‡ only if it leads to triggering another bug oracle.

Bug Name	EF⚡CF	Confuzzius [53]	Smartian [10]	Echidna [25]
Assertion Failure	✓†	✓	✓	✓*
Arbitrary Write	×‡	×	✓	×‡
Block State Dependency	×‡	✓	✓	×‡
Control-flow Hijack (`JUMP`)	×‡	×	✓	×‡
Custom Event Oracle	✓*	×	×	✓*
Custom Property Checking	✓*	×	×	✓*
Ether Gains	✓	×	×	×
Integer Overflow	×‡/ ✓†	✓	✓	×‡
Leaking Ether	✓*	✓	✓	×
Locking Ether	×	✓	✓	×
Multiple Send	×	×	✓	×
Reentrancy	×‡	✓	✓	×
Require Violation	×	×	✓*	×
Transaction Origin Use	×‡	×	✓	×‡
Transaction Order Dependency	×	✓	×	×
Unsafe Delegatecall	✓	✓	✓	×
Unprotected Selfdestruct	✓*	✓	✓	✓*
Un/Mishandled Exception	×‡	✓	✓	×

transaction sequences that trigger other bug oracles, such as Ether gains. In this way, EFϟCF detects reentrancy bugs, but only if they are actually exploitable. In contrast to other analysis tools, this leads to fewer false alarms, e.g., when encountering manual reentrancy locking [43].

By default, EFϟCF reports *unprotected selfdestruct* only if the selfdestruct will transfer the remaining Ether of the target contract to the attacker, i.e., the address parameter of the selfdestruct is controlled by the attacker. Optionally, EFϟCF can also report *DoS*-style unprotected selfdestructs, i.e., if the selfdestruct can be triggered by anyone, but always targets a trusted address such as the owner. This style of detection is featured in most other analysis tools.

With Solidity versions > 0.8, contracts feature automatic integer overflow checking and proper assertion violation reporting. EFϟCF supports this new Solidity exception mechanism to signal errors to the fuzzer. Previously, *assert* statements were implemented with the *INVALID* opcode that also triggers a transaction revert. However, earlier contracts (pre 0.4) also used this to implement failure of input sanitization, which makes it hard to reliably distinguish between a regularly failing transaction and a assertion violation across Solidity versions. We expect developers to use newer Solidity versions for newly deployed contracts. We opted to support only the new Solidity exception mechanism, which allows us to reliably detect internal errors in a smart contract. This includes memory allocation failure, integer overflows, and internal assertions. Developers can utilize EFϟCF for general robustness testing of their newly deployed smart contracts.

En/Decoding the ABI We use the *ethabi* Rust library to parse the JSON-based ABI definition, which allows us to en- and decode the ABI format expected by the smart contract. Sometimes the base fuzzer will break the ABI encoding, which results in the custom mutator attempting to decode an extremely large input data. In fact, during development of EFϟCF, we uncovered and fixed two bugs in the *ethabi* library such that it would not result in an irrecoverable error when attempting to decode broken ABI-encoded data. However, we still set an $8\,\mathrm{kbyte}$ limit to the number of bytes we attempt to decode. This guards against spending too much time on attempting to decode an unusually large input byte-string of a transaction. Since it is unlikely a valid or useful input for the smart contract under test, it is preferable to avoid the lengthy decoding process altogether. In these pathological cases, we fall back on the random mutations provided by the base fuzzer.

Additional Tooling Based on our custom mutator code, we additionally implemented several tools that proved to be useful for smart contract fuzzing. For instance, EFϟCF also features a test case minimizer that performs structural minimization on a test case, allowing an analyst to reduce the size of a test case. Furthermore, EFϟCF integrates a translator between our binary test case format and a human-readable *yaml*-based format, allowing for easy manual modification of test cases. EFϟCF can also convert a test case into a Solidity attack contract that can be deployed within a blockchain environment to study the generated attack. These tools help an analyst performing root cause analysis given a test case that triggers a bug.

```
1  bool not_called_re_ent27 = true;
2  function bug_re_ent27() public {
3    require(not_called_re_ent27);
4    if( ! (msg.sender.send(1 ether) ) ){
5      revert();
6    }
7    not_called_re_ent27 = false;
8  }
```

Figure 10: An injected reentrancy bug, which is also detected by bug oracles that only identify leaking Ether without considering reentrancy.

```
1  // initialized with 0
2  mapping(address => uint) redeemableEther_re_ent25;
3  function claimReward_re_ent25() public {
4    /// since the balances mapping is never written
   ↪  anywhere else,
5    /// this require cannot be bypassed by a dynamic
   ↪  analysis tool.
6    require(redeemableEther_re_ent25[msg.sender] > 0);
7    /// unreachable code
8    uint transferValue_re_ent25 =
   ↪  redeemableEther_re_ent25[msg.sender];
9    msg.sender.transfer(transferValue_re_ent25);
   ↪  //bug
10   redeemableEther_re_ent25[msg.sender] = 0;
11 }
```

Figure 11: An injected reentrancy bug that cannot be triggered because of a guarding condition that cannot be satisfied.

B. Problems with Existing Datasets

By now, there are multiple attempts to create standardized datasets to evaluate smart contract analysis tools [12, 24]. However, they lack diversity of bugs, especially with respect to reentrancy bugs. They often only contain the simple same-function reentrancy pattern and contain many honeypot contracts (see Appendix F).

Ghaleb et al. [24] attempt to synthesize a benchmark dataset using artificial bug injection. Unfortunately, we find that these synthesized datasets are not suitable to test reentrancy detection based on fuzzing or symbolic execution. The injected reentrancy bugs focus on same-function reentrancy and do not cover more complex reentrancy patterns, such as cross-function reentrancy [43]. Furthermore, the injected patterns themselves are suboptimal. For example, many of the injected reentrancy bugs do not *require* a reentrant call to be exploitable. Figure 10 shows an injected bug that is prone to reentrancy. However, the function can be identified as vulnerable without resorting to a reentrancy attack. For example, MAIAN [40] detects an Ether leaking vulnerability in this function because the injected function will unconditionally send Ether (line 4) the first time a caller calls the function. Reentrancy is only necessary to repeatedly leak Ether. A second type of injected reentrancy bug is shown in Figure 11. This reentrancy bug can never be triggered during fuzzing or symbolic execution. The injected vulnerable function contains a guard, the *require* statement in line 6, that cannot be satisfied, because it accesses a storage variable that is never modified in the contract's code. This makes the reentrant call practically dead code. We therefore conclude that the reentrancy bugs injected as part of the *SolidiFI* project are not suitable to test dynamic analysis tools, i.e., fuzzers or symbolic execution-based analyzers.

C. Details on Benchmarks

Scalability Benchmark Dataset We create a set of contracts that can be used to evaluate tools with respect to their scalability to generating long transaction sequences. Here we utilize a time-to-bug approach to benchmarking. We therefore require contracts with ground truth and bugs that are widely supported by analysis tools. We opted to create our own set of benchmark contracts. Furthermore, we devise multiple variants of the same contracts, allowing us to measure scalability to longer transaction sequences in a more fine-grained manner.

To this end, we automatically synthesize the *multi* contracts as follows: For each function, several equality or i-equality constraints are enforced on up to six integer arguments before setting a boolean internal state variable. These benchmark contracts test the capability of solving input constraints across multiple transactions. The functions of the contract must be called in the right order and with the right inputs to trigger a *selfdestruct* of the contract. The synthesized contracts favor symbolic execution tools because there is no potential for path explosion within the functions. This is because they do not contain any control-flow statements except for error handling.

The three *complex* contracts consist of a manual adaptation of the *multi* contracts with more varying constraints on the input and the internal state of the smart contract (e.g., requiring an array input of a certain length or requiring the *sha3* hash of a fixed value as input). The idea is to add more complex and diverse input requirements that are more akin to real-world smart contracts. We only use three different variants of these contracts because they are created manually.

Code Coverage Dataset For our code coverage experiments in Section 5.3, we utilize a set of real world smart contracts. To create a realistic set of smart contracts, we first analyzed the *smartbugs-wild* dataset [12] and ranked the contracts according to their peak Ether balance (as reported by etherscan.io). Ranking according to peak Ether balance naturally excludes toy or test contracts and creates a dataset that is focused on contracts that are actually in use.

We then processed the top 1000 contracts according to this ranking and created a minset of those contracts that are supported by all fuzzers. For example, the contract must not require constructor parameters such that the contracts can easily be deployed in every fuzzer. The resulting dataset consists of a set of 253 contracts from the top 1000 contracts of the *smartbugs-wild* dataset that offer a certain level of code diversity.

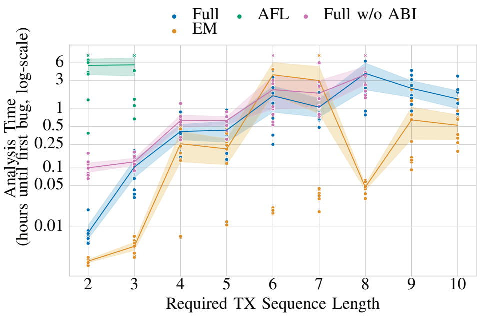
D. Ablation Study: Scalability

As part of our ablation study, we also compare the various configurations in terms of scalability to longer and complex transaction sequences. We utilize the same scalability experiment as in Section 5.1. However, we bound the execution time to a maximum of $8\,\mathrm{h}$. We run EF↯CF in four configurations: Full EF↯CF with ABI, full EF↯CF without knowledge of the ABI, fuzzing with the custom mutator only (*EM*), and fuzzing with AFL++'s mutations only (*AFL*). The results are shown for the *multi*, *complex*, and *justlen* benchmarks in Figure 12a, Figure 12b, and Figure 12c, respectively. While the *AFL* configuration generally provides the highest throughput (see Table 2), it lacks structured mutation operations, such as splicing at the transaction level. As a result, the fuzzing process becomes ineffective and fails to reliably generate even short meaningful transaction sequences. Furthermore, the *AFL* configuration fails to identify a bug in any of the *complex* contract variants. This shows that the custom mutator in EF↯CF is essential for good fuzzing performance. Furthermore, we can see that in this benchmark, the *EM* performs best since the magic value comparisons are best solved using the dictionary-based sampling employed by the custom mutator for integer types. Since the custom mutator utilizes the dictionary probabilistically, we also observe some extreme outliers in these experiments. Since the custom mutator performs mostly structural mutations on the transaction sequences, it also performs best on the *justlen* experiment.

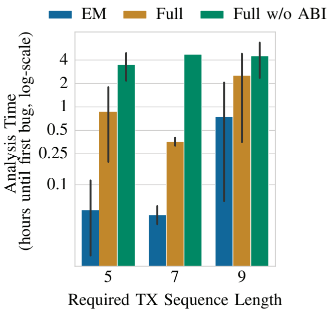
While the *EM* configuration performs best on the benchmarks presented here, we found that without AFL's mutations (especially the input-to-state correspondence [4] mutations) there are several fuzzing roadblocks in practice that cannot be solved by the *EM* configuration. Furthermore, the benchmarks focusing on real-world smart contracts in Section 5.1 show that the standard EF↯CF configuration performs best.

Fuzzing without ABI When fuzzing without ABI information, EF↯CF fully relies on the coverage feedback to discover useful transaction inputs. Generally, EF↯CF without ABI information expectedly performs worse compared to fuzzing with ABI information. On the *multi* and *complex* contracts, fuzzing without ABI information identifies the bug in a mean of $223\,\mathrm{min}$ ($\sigma = 205$). In comparison, fuzzing with ABI information requires a mean of $87\,\mathrm{min}$ ($\sigma = 127$) to identify the bug. On the *justlen* benchmark, the difference is much smaller: $3.9\,\mathrm{min}$ with ABI and $6.6\,\mathrm{min}$ without ABI. To identify a bug in the *justlen* benchmark, the fuzzer does not need to identify correct input parameters, as most exposed functions simply do not require parameters. Performing structural mutations on the transaction sequence does not require ABI information. For this reason, the performance difference is much smaller.

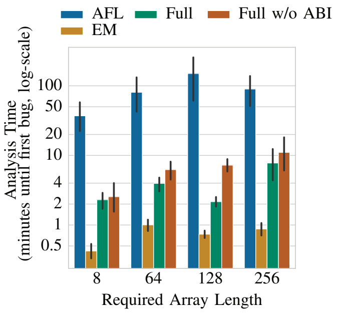
The largest difference can be observed in the *multi* benchmark. This contract primarily tests the analysis tool's capability of finding solutions to multiple input constraints while creating long transaction sequences. This benchmark heavily favors symbolic execution tools, as there is nearly no potential for state explosion within functions. Figure 13 shows a comparison of EF↯CF with and without ABI with the best performing symbolic execution tool and the fuzzers we evaluated. While EF↯CF with ABI features comparable performance to symbolic execution tools, EF↯CF without ABI performs significantly worse. Because of the lack of knowledge about the input structure, EF↯CF samples the dictionary less often than with ABI. This leads to a decreased chance of placing the correct values into the input. However, remarkably, EF↯CF without ABI information still performs significantly better than state-of-the-art fuzzing tools that utilize the ABI.



(a) *multi*



(b) *complex*



(c) *justlen*

Figure 12: Results of scalability experiments showing the analysis time required over the length of transaction sequences with various configurations of EF⚡CF.

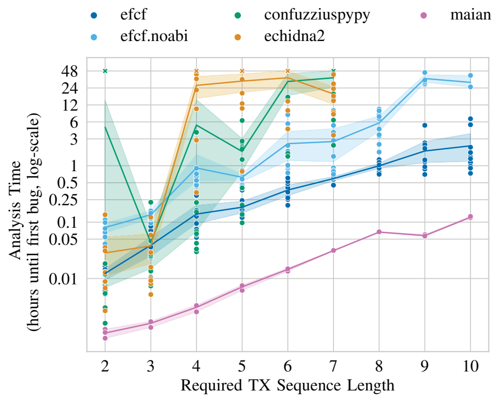


Figure 13: Comparison of EF⚡CF with and without ABI and other analysis tools. MAIAN is the best analysis tool in this benchmark and never uses ABI information. Echidna and Confuzzius always utilize ABI information. We can see that EF/CF without ABI performs significantly worse but still outperforms the other fuzzers that utilize ABI information.

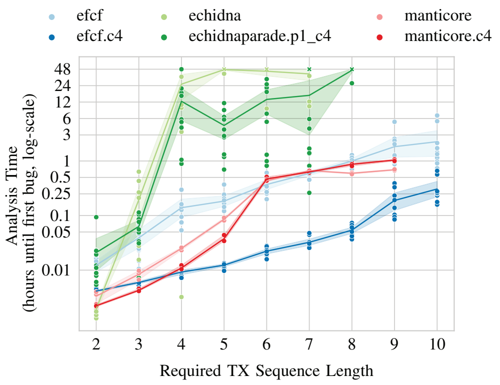


Figure 14: Results of running multiple analysis tools on a single core vs. running on 4 cores (marked with *c4*) in parallel on the *multi* dataset.

E. Additional Benchmarks and Evaluation Details

Multi-Core Performance We also evaluate multi-core performance of those analysis tools that support it: EF⚡CF, Manticore, and Echidna. To parallelize Echidna, we utilize the *echidna-parade* [26] tool to run multiple instances of Echidna in parallel. In contrast to the normal mode of operation in *echidna-parade*, we always fuzz the full set of functions by not excluding any function from the fuzzing runs. In our benchmarks, the default mode of operation is detrimental to performance in terms of time-to-bug. Manticore natively supports multi-threaded analysis to leverage multiple cores. For EF⚡CF, we leverage the multi-core fuzzing approach of AFL++.

Figure 14 shows the multi-core performance of several analysis tools on the *multi* contracts. We can see that with EF⚡CF, the performance significantly increases between the single and multi-core versions. This is primarily because EF⚡CF utilizes an ensemble fuzzing-like approach that spawns EF⚡CF's core fuzzer in multiple different configurations. Similarly, parallelizing Echidna with the *echidna-parade* tool shows significant improvements over the single-core Echidna. In a multi-core setting, Echidna can find the transaction sequences with up to 8 transactions and features a significant speed-up for the transaction sequences with lengths 4 to 7. For both fuzzers, we observe that some single-core runs are as fast or faster than other multi-core runs. Overall, however, the multi-core runs reduce the variance between the runs, allowing the fuzzer to identify the bug in a given time span more consistently. Interestingly, Manticore, the only other tool with built-in multi-core support, does not gain a significant speedup in this experimental setup. We suspect that the symbolic execution approach taken by manticore cannot fully leverage multiple cores.

Code Coverage In Section 5.3, we describe an experiment to assess the ability of current fuzzers to reach code coverage on a set of real-world smart contracts. We describe the dataset in more detail in Appendix C. The overall results are shown in Table 6. Here, we show the number of targets where one fuzzer outperforms another fuzzer with statistical significance.

F. Fuzzing Reentrancy Honeypots

Torres et al. [55] discussed the phenomenon of honeypot contracts. These contracts are deployed with source code often available on etherscan.io that appears to be vulnerable to, e.g., reentrancy attacks. However, these contracts are, in fact, a form of scam. They target malicious actors that search for easily exploitable contracts on the blockchain. They require the attacker to first invest a number of Ether to later exploit the seemingly vulnerable contract. However, the code hides a mechanism that prevents exploitation, locking the previously invested Ether of the attacker. Most of the known reentrancy honeypot contracts use a call to an external library-like contract to revert attack transactions. The deception works by suggesting that the source code on etherscan also provides the source code for the external contract when, in fact, a different contract is used.

Many of the known honeypot contracts feature very obvious reentrancy vulnerabilities, as these contracts are designed to be easily analyzable (i.e., to lure more people into attempting to attack the honeypot). Many of those reentrancy honeypot contracts ended up in various datasets of prior studies [6, 12]. In the curated version of the smartbugs dataset, the majority of the contracts identified as vulnerable to reentrancy are, in fact, honeypot contracts. This dataset contains 19 reentrancy honeypots and 12 other contracts vulnerable to reentrancy.

Honeypot contracts introduce significant bias into datasets. For example, a tool that detects all honeypot contracts in the curated smartbugs dataset already seems to detect the majority of reentrancy bugs. However, in reality, all these reentrancy bugs follow the exact same code pattern. For this reason, we chose to summarize all these cases as *trivial reentrancy* in Section 5.4.

The reentrancy honeypots can be analyzed in two ways: by relying on the source code only and by importing code and state data directly from the blockchain. It is important to distinguish both cases since, in the former, the contract is exploitable, while in the latter, it is not. We verified that EF↯CF correctly identifies the reentrancy attacks in the first case (see Section 5.4). Here we deploy a fresh instance of the contract, and the mechanism to prevent exploitation does not work. In this case, EF↯CF can correctly identify the reentrancy vulnerability. However, if we export the contract's state from the blockchain, including the external contract that is called, then the mechanism to prevent exploitation is working. In this case, EF↯CF also executes the second external contract, reverting the transaction before the reentrancy takes place. Thus, EF↯CF correctly does not report any false alarm.

Table 6: Comparison of all fuzzers on the test set: the number of times fuzzer A outperformed fuzzer B.

Fuzzer A \ Fuzzer B	EF↯CF	Confuzzius	ILF
EF↯CF	-	141	120
ConFuzzius	83	-	105
ILF	112	136	-

G. Sailfish Reentrancy Findings

As part of the evaluation of the SAILFISH tool, Bose et al. released a list of contracts where reentrancy causes inconsistent state according to SAILFISH. This list contains 1904 contracts, of which the Bose et al. verified 26 to be true positives[1]. Among the list of 1904 contracts, EF↯CF identifies vulnerabilities in only 67 contracts. However, in 8 of these 67 contracts, EF↯CF discovers a vulnerability unrelated to reentrancy, e.g., a controlled *delegatecall* vulnerability.

Furthermore, we analyzed the list of verified true positives in more detail. Among the vulnerable contracts reported by the SAILFISH tool, EF↯CF correctly identifies 5 contracts that can be exploited with reentrancy to steal Ether. Among these five contracts, one is a test contract, one a known honeypot, and the remaining 3 contracts are duplicates. Furthermore, EF↯CF identifies one contract that can be exploited due to an access control bug, not a reentrancy. Note that EF↯CF only identifies the honeypot as vulnerable when deploying from source code (see Appendix F). We manually identified one contract that seems to be vulnerable to reentrancy, but no Ether is at stake. The remaining contracts exhibit reentrancy patterns but are probably not exploitable.

For example, the *CommonWallet* contract, depicted in Figure 15, is affected by a similar token-related reentrancy as the *Uniswap-V2* contract [54]. Here, the attacker must supply an *ERC777* token where an *ERC20* token is expected. Most (legitimate) *ERC20* token contracts do not perform callbacks to the attacker and therefore the attacker cannot trigger a reentrancy situation. However, this is different for *ERC777* contracts that feature callbacks by design. They allow the attacker to reenter the *CommonWallet* contract and trigger a reentrancy. Currently, EF↯CF does not detect token-related bugs since EF↯CF has no concept of tokens and therefore does not regard token gains as a bug.

However, in reality, this reentrancy bug cannot be used to cause damage. The reason for this is that the attacker would have to cause an integer underflow, which is prevented due to the integer checking leveraged by the contract. EF↯CF would not identify a possible reentrancy attack, even if there were a bug oracle for tokens, because there is no way to exploit the reentrancy attack. We verified this by testing EF↯CF with a contract that exhibits the same vulnerability but with Ether instead of tokens. We believe the reason for this false alarm is that Sailfish does not accurately model the transaction-based execution model of the EVM. It will eagerly report a bug without considering that the transaction will revert later, a flaw common to many analysis tools [45].

Interestingly, Sailfish uses a rather broad definition of state inconsistency caused by reentrancy. Any reachable state write that could cause inconsistency is considered a true alarm. We found several contracts where reentrancy is possible but where the reentrancy will never actually cause inconsistent state. Here, a state variable is written but never changed, e.g., the reentrant code path will perform a subtraction with 0. However, this state variable update is identified by SAILFISH to cause state inconsistency. In

1. https://github.com/ucsb-seclab/sailfish

```
function safeSub(uint256 _x, uint256 _y)
↪  internal pure returns (uint256) {
  assert(_x >= _y);
  return _x - _y;
}
function sendTokenTo(address tokenAddr,
↪  address to_, uint256 amount) {
  require(tokenBalance[tokenAddr][msg.sender]
↪  >= amount);
  /* external call - might cause reentrancy
↪  */
  if(ERC20Token(tokenAddr).transfer(to_,
↪  amount)) {
    /* state update with underflow check in
↪  safeSub */
    tokenBalance[tokenAddr][msg.sender] =

↪  safeSub(tokenBalance[tokenAddr][msg.sender],
↪  amount);
  }
}
```

Figure 15: *CommonWallet* reentrancy, which is not exploitable due to the integer overflow check.

contrast, EFϟCF will only report bugs that actually cause damage. Furthermore, as a static analysis tool, Sailfish must over-approximate and consider all external calls as being able to cause reentrancy. However, in practice, many contracts issue calls to trusted contracts that do not allow the attacker to perform reentrancy. In contrast, EFϟCF will not perform reentrancy on calls to unknown contracts by default, avoiding false reentrancy alarms. For highest precision, it can also import and execute trusted contract dependencies of the target.

```
number: 0
difficulty: 0
gas_limit: 0
timestamp: 0
initial_ether: 14000000000000000000
txs:
  - length: 4
    return_count: 0
    receiver_select: 0
    sender_select: 2
    block_advance: 0
    call_value: 9227875636482146304
    input: "0xd0e30db0" # deposit()
    returns: []
  - length: 4
    return_count: 1
    receiver_select: 0
    sender_select: 2
    block_advance: 0
    call_value: 0
    input: "0x5fd8c710" # withdrawBalance()
    returns:
      - value: 1
        reenter: 2
        data_length: 0
        data: "0x"
  - length: 68
    return_count: 0
    receiver_select: 0
    sender_select: 2
    block_advance: 0
    call_value: 0
# addAllowance(
# 0xc3cf2af7ea37d6d9d0a23bdf84c71e8c099d03c2,
# 1117873197643827594651545771110674982630890210242
# )
    input: "0xf3c40c4b0000000...."
    returns: []
  - length: 68
    return_count: 0
    receiver_select: 0
    sender_select: 3
    block_advance: 0
    call_value: 0
# transferFrom(
#  0xc2018c3f08417e77b94fb541fed2bf1e09093edd,
#  2951479051793528258S6
# )
    input: "0x01c6adc30000000...."
    returns: []
  - length: 4
    return_count: 1
    receiver_select: 0
    sender_select: 3
    block_advance: 0
    call_value: 0
    input: "0x5fd8c710" # withdrawBalance()
    returns:
      - value: 1
        reenter: 0
        data_length: 0
        data: "0x"
```

Figure 16: Textual representation of the transaction sequence generated by EFϟCF to exploit the contract from Figure 2.