Taming Callbacks for Smart Contract Modularity

ELVIRA ALBERT, Complutense University of Madrid, and Institute of Knowledge Technology, Spain
SHELLY GROSSMAN, Tel-Aviv University, Israel
NOAM RINETZKY, Tel-Aviv University, Israel
CLARA RODRÍGUEZ-NÚÑEZ, Complutense University of Madrid, Spain
ALBERT RUBIO, Complutense University of Madrid, and Institute of Knowledge Technology, Spain
MOOLY SAGIV, Tel-Aviv University, Israel

Callbacks are an effective programming discipline for implementing event-driven programming, especially in environments like Ethereum which forbid shared global state and concurrency. Callbacks allow a callee to delegate the execution back to the caller. Though effective, they can lead to subtle mistakes principally in open environments where callbacks can be added in a new code. Indeed, several high profile bugs in smart contracts exploit callbacks. We present the first static technique ensuring *modularity* in the presence of callbacks and apply it to verify prominent smart contracts. Modularity ensures that external calls to other contracts cannot affect the behavior of the contract. Importantly, modularity is guaranteed without restricting programming.

In general, checking modularity is undecidable—even for programs without loops. This paper describes an effective technique for soundly ensuring modularity harnessing SMT solvers. The main idea is to define a constructive version of modularity using *commutativity* and *projection* operations on program segments. We believe that this approach is also accessible to programmers, since counterexamples to modularity can be generated automatically by the SMT solvers, allowing programmers to understand and fix the error.

We implemented our approach in order to demonstrate the precision of the modularity analysis and applied it to real smart contracts, including a subset of the 150 most active contracts in Ethereum. Our implementation decompiles bytecode programs into an intermediate representation and then implements the modularity checking using SMT queries. Overall, we argue that our experimental results indicate that the method can be applied to many realistic contracts, and that it is able to prove modularity where other methods fail.

Additional Key Words and Phrases: program verification, program analysis, invariants, logic and verification, blockchain, smart contracts

1 INTRODUCTION

Modularity is a key principle in system design: Encapsulating code and data into different modules which communicate via clearly defined procedural interfaces allows separately designing, developing, understanding, testing, and reasoning about different parts of the system. For example, the

Authors' addresses: Elvira Albert, elvira@sip.ucm.es, Complutense University of Madrid, and Institute of Knowledge Technology, Spain; Shelly Grossman, shellygr@mail.tau.ac.il, Tel-Aviv University, Israel; Noam Rinetzky, maon@post.tau.ac.il, Tel-Aviv University, Israel; Clara Rodríguez-Núñez, clarrodr@ucm.es, Complutense University of Madrid, Spain; Albert Rubio, alberu04@ucm.es, Complutense University of Madrid, and Institute of Knowledge Technology, Spain; Mooly Sagiv, msagiv@acm.org, Tel-Aviv University, Israel.

2475-1421/2020/11-ART209
https://doi.org/10.1145/3428277

fully encapsulated programming model of the Ethereum blockchain allows for any object ("smart contract") to interact with other ones by invoking their methods, but prevents direct access to the other contracts' data. Modularity, however, is not a panacea as demonstrated by the infamous DAO bug [12]. The latter exploited the *callback mechanism* to temporarily steal money. Callbacks occur when a method of a module, say a smart contract, invokes a method of another module, say, another smart contract, and the latter, either directly or indirectly, invokes one or more methods of the former before the original method invocation returns.

Callbacks complicate program reasoning (see, e.g., [24]) because they require programmers to consider interleavings of calls to their own code, which, as in concurrent programming [36], can be very tricky. The danger of callback attacks, also called *reentrancy attacks*, led to many suggestions for syntactical program restrictions, e.g., delaying external calls (see, e.g., [14]). However, these restrictions are overly severe and several realistic programs violate them.

The goal of this paper is to develop a sound static analysis for proving immunity to reentrancy attacks while permitting benign use of callbacks, thus, allowing for flexible programming without placing syntactical restrictions. This problem is challenging since we need to prove *relational hyper-properties* properties of the code [5, 7, 19, 37]. Intuitively speaking, the static analysis will show that a program without a callback is semantically equivalent to a program with a callback (in such, modularity is ensured).

1.1 Effective Callback Freedom (ECF)

This paper is inspired by the work of [23] which defines the notion of **Effectively Callback Free** (ECF) modules. Intuitively, a module is effectively callback free if for every trace with a callback, there exists "an equivalent" callback free trace. [23] suggest two definitions of trace equivalence inspired by database theory [9]: (i) semantic equivalence based on final state inspired by final state serializability, and (ii) a syntactic notion of equivalence based on conflict, i.e., reordering based on reads and writes, inspired by conflict-serializability. The former asserts the existence of a sequence of invocations of the object methods without any interfering callbacks which transforms the object's state in the same way as the original trace. The latter provides a conservative mechanism to determine whether such a sequence exists by inspecting the read/write conflicts in the original trace.

The benefit of these definitions is that they allow for *dynamic* sound reasoning about properties of the module's state at quiescent points, i.e., when its methods are not being executed, by considering only callback-free executions. Therefore, our original motivation was to *statically* enforce these conditions using a static algorithm. It is an undecidable problem, and is more challenging than the dynamic case as it requires reasoning about an unbounded number of arbitrary unknown sets of program traces.

Unfortunately, we found out that neither one of the aforementioned definitions is useful for static analysis: final state ECF does not provide nor suggest an effective algorithm, and conflict ECF drastically limits programming, even disqualifying as non-ECF programs using simple "contract-locks" mechanisms to prevent reentrancy, as we will show in the examples through the paper.

1.2 Static verification of ECF

This paper develops a powerful method for statically verifying ECF using commutativity checks (which assure equivalence) while also allowing *projecting away* irrelevant pieces of code. Our starting point is the reduction of [23] for ECF: it shows that if there is a violation (using syntactic conflict equivalence) of the ECF property in a trace with arbitrary nested callback calls then there is one where callbacks are not nested. We generalize this reduction to semantic final state equivalence and develop our techniques for *simple* traces, i.e., ones where the execution of a single method can

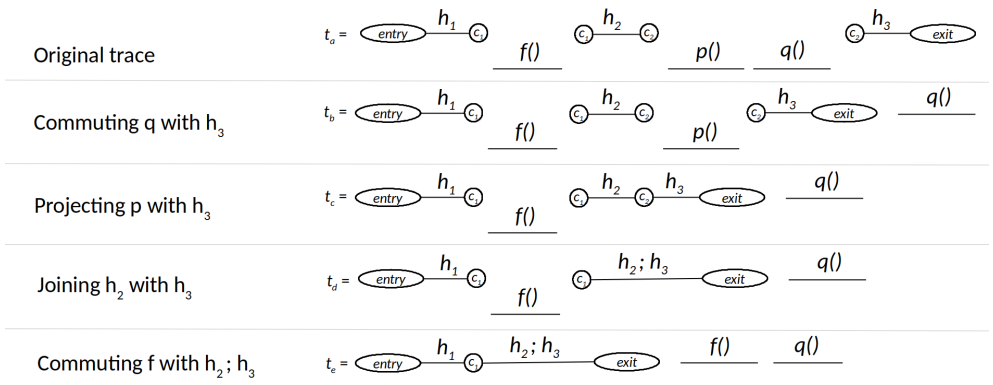


Fig. 1. Sequence of commutation and projection operations on an example trace.

be interrupted at call nodes by an arbitrary sequence of executions of other procedures, however these interrupting procedures themselves never get interrupted.

We prove that a simple trace is ECF by constructing an equivalent callback free trace via a sequence of swapping and removing of all possible different interrupting invocations that might arrive.

Example 1.1. Consider trace t_a shown in Figure 1. The trace depicts an execution of procedure $h()$ of a module m which is interrupted twice by different callbacks: $h()$ starts executing at its entry point and performs a sequence of primitive commands following its control flow graph (h_1) until it gets to a *call node* (c_1) where it relinquishes control to an external method. At that point, the external method invokes procedure $f()$ on m thus generating a callback. Control returns to $h()$ only after $f()$ exits and $h()$'s execution continues from c_1 by executing the next sequence of intra-procedural primitive commands (h_2) until another call node (c_2) is reached. At that point the external procedure generates two callbacks by invoking $p()$ and then $q()$. After control returns to $h()$ the sequence h_3 is executed and the execution ends. We turn t_a into the callback free trace t_e by either commuting the subtraces corresponding to the callback calls or *projecting* them away. (Note that callback $p()$ is not part of t_e). Trace t_b shows the result of (right) commuting q with h_3. Intuitively, such a transformation is possible if the composed effect of $q; h_3$ is preserved by $h_3; q$ (c.f. Section 4.3). Trace t_c shows a different way to transform the trace, namely by projecting p away. The elimination of p can be done by a (right) projection with h_3, provided the composed effect of $p; h_3$ is preserved by only executing h_3. Alternatively, we can achieve the same goal using (left) projection with h_2, provided the composed effect of $h_2; p$ is preserved by only executing h_2. At this point, we consider the call node c_2 "solved". Once we solved c_2, we can continue with the swapping and projecting operations to the other callbacks. However, we can do better. Note that once we solved c_2 the trace of $h_2; h_3$ is not interrupted. Thus, while we could, for example, try to swap f with h_2 and then with h_3 in order to solve call node c_1, we, instead, try to swap it with the "joined" trace $h_2; h_3$ (t_d). Note that if the separate swaps succeeds it is guaranteed that the swap over the *joint trace* $h_2; h_3$ succeeds too. This is not true, however, the other way around.

The aforementioned transformation ensures that the resulting trace is final-state equivalent to the original one, i.e., the effect of executing the original trace (t_a) on an initial state σ can be reproduced by executing it on the transformed (callback-free) trace (t_e). Importantly, since our method is static, the above transformations will be applied on *code segments* that represent sets of potentially infinite number of traces, rather than on a single trace.

```
1  pragma solidity ^0.4.24;
2
3  contract Bank {
4    mapping (address => uint) public shares;
5    function deposit() payable {
6      /* balance is an alias for
           address(this).balance */
7      balance += msg.value;
8      shares[msg.sender] += msg.value;
9    }
10 }
```

```
11 function withdraw() {
12   uint256 orig_balance = balance;
13   uint256 orig_shares = shares[msg.sender];
14   if (orig_shares > 0 && orig_balance >= orig_shares) {
15     balance = balance − orig_shares;
16     if (msg.sender.send_money(orig_shares)!=success) {
17       balance = orig_balance; // reverting
18       shares[msg.sender] = orig_shares;
19     }
20     else shares[msg.sender] = 0;
21   } }
```

Fig. 2. A Solidity contract illustrating the DAO bug. We write the balance update effects of payable functions and send operations explicitly using the balance variable. The send_money operation is the same as Solidity's send. success represents a success code as returned by send_money. Revert operations are also stated explicitly.

1.3 Summary of contributions

In summary, the paper makes the following technical contributions:

(1) Semantic commutation and projection, and segment-join operations. We present semantic notions of left/right/zero-projection, that together with the operations of commutation and segment-join (intuitively illustrated in the example in Figure 1), lay down our analysis.

(2) Static analysis. We introduce a novel static analysis (that will be intuitively outlined in Figure 8 and formalized through the rest of the paper) based on proving commutativity and projection between all the fragments of code (or code segments) in between call nodes and *all* other procedures of the module.

(3) Callback invariant. The framework is extended smoothly to work with invariants in order to increase accuracy. To this end, we introduce the new concept of callback invariant, which is an invariant that holds in a call node and, in addition to the properties of standard invariants, must be preserved by all procedures of the module, as they can be executed as callbacks in the call node.

(4) Implementation and evaluation. A prototype of our static analysis algorithm is implemented on top of the EVM bytecode [47] and evaluated on the most active Ethereum contracts and on a realistic decentralized finance application.

1.4 Outline of the rest of the Paper

This paper presents the first sound algorithm to attack ECF directly. The algorithm statically determines that all executions of the program are effectively callback free. Existing algorithms suffer from two limitations: (1) no full path coverage; and (2) checking a stronger property leading to practical false alarms. Our algorithm succeeds to prove ECF in sophisticated contracts in an open environment, in which any sequence of methods of the contract can be executed in a callback.

Section 2 provides an overview of the problem we want to solve and the intution behind our proposal. Sections 3 and 4 define the necessary notations and main definitions. Section 5 describes our static analysis technique for checking ECF. Section 6 extends the analysis to include the concept of *callback invariant*. Section 7 presents the implementation and its evaluation on Ethereum smart contracts. Section 8 discusses related work and concludes.

2 CALLBACKS: THE PROBLEM AND THE PROPOSAL

2.1 The problem and the gap

We motivate our work using the infamous bug in the DAO (Decentralized Autonomous Organization) contract [12]. Figure 2 shows a simplified vulnerable *Solidity* contract. The purpose of the DAO contract is to facilitate voting on investment proposals by the owners of the DAO (referred to

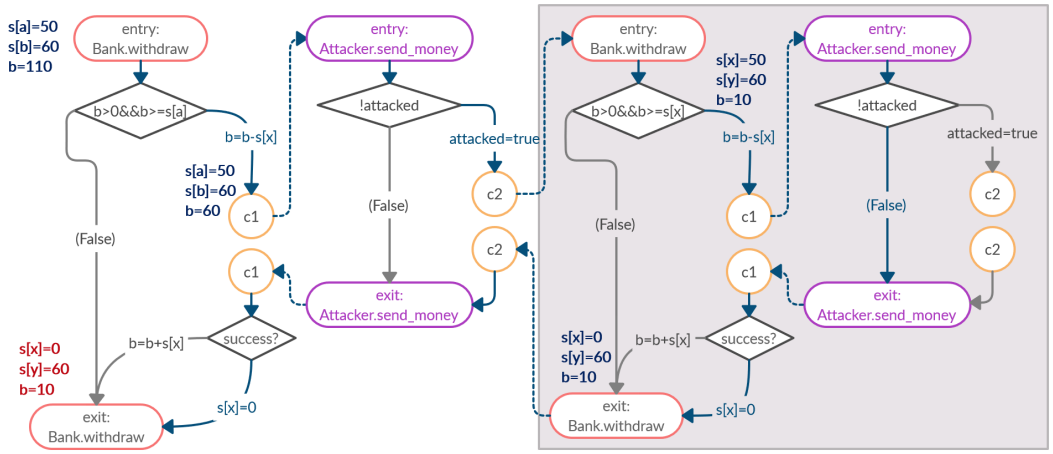


Fig. 4. The CFG of the withdraw method from the objects in Figure 2 and the malicious trace using the attacker object from Figure 3, marked with blue edges (b is balance, s is shares). The area under the grey rectangle pertains to the callback.

as objects in the following). The contract stores in the variable shares the individual investment for each object as well as the balance variable.

For clarity of the presentation, we avoid using predefined Solidity instructions for money transfer and state reversal, and implement them by explicit updates to the state. This includes the special reserved global variable balance representing the amount of money owned by the executing contract that is maintained by the runtime VM. The contract offers two functions that manipulate the state: deposit and withdraw. The purpose of deposit is to store money in the contract by increasing the object's shares by the value sent as parameter. In Solidity msg is a special variable that always exists in the global namespace, providing information about the blockchain. The field sender of msg stores the caller's object's address and the field value stores the "money" (Ether, the cryptocurrency of the Ethereum blockchain) transferred in the transaction.

```
22  bool attacked;
23  function send_money(uint value) {
24    if (!attacked) {
25      attacked = true;
26      balance += value;
27      Bank.withdraw();
28    } }
```

Fig. 3. Attacker object stealing money from DAO contract

The withdraw function allows pulling out all available shares of the object, which is implemented by decreasing the current shares amount from the contract's own balance and transferring it to the object by means of the send_money in line 16. This is a *call node* where control is relinquished to the callee object. At this point, the callee object might execute a callback. If the call does not fail (programmed as returning success), the object's shares is set to zero. Otherwise the state is reverted to the initial one (then branch). The DAO was attacked by a "callback loop-hole" in which the receiver object calls back the method withdraw to steal money, in particular, the code of the send_money function is designed to call withdraw again. Figure 3 shows a snippet of code that produces such a callback loop-hole and Figure 4 shows the exploit trace. Basically, when the attacker receives the control in send_money, it increases its balance and calls back withdraw again.[1] As the shares of the attacker are only updated in line 20 after the send_money has finished, the callback execution of withdraw will find the shares with the initial value and will make another

[1]In order to simplify the trace with the callbacks shown later, the attacker object is designed in a way that it can be invoked at most once (by using attacked as a lock), and generate only a single callback. Note that even without the lock, there is no infinite recursion here since eventually the condition for sending money will not hold.

```
29 contract Bank {
30     mapping (address => uint) public shares;
31     bool lock = false;
32     function deposit() payable {
33       balance += msg.value
34       require (!lock);
35       shares[msg.sender] += msg.value;
36     }
37     function withdraw() {
38       require (!lock);
39       uint256 orig_balance = balance;
40       uint256 orig_shares = shares[msg.sender];
41       if (orig_shares > 0 && orig_balance >=
             orig_shares) {
42         lock = true;
43         balance = balance − orig_shares;
44         if (msg.sender.call(orig_shares)!=success) {
45           balance = orig_balance;
46           shares[msg.sender]= orig_shares;
47           lock = false;
48         } else {
49           lock = false;
50           shares[msg.sender] = 0;
51         }
52       }
53   }
```

Fig. 5. Solidity contract avoiding the DAO bug. Not verifiable using previous approaches for ECF checking.

transfer to the attacker. Figure 4 depicts the bug in the malicious trace. The presence of the callback violates the invariant balance $\geq \sum$ shares.

Severity of Reentrancy Attacks. The DAO problem is also called 'Reentrancy Attack' since it exploits the non-reentrant nature of the stateful code. The attack is pervasive, e.g., [12, 15], and keeps occurring even after the DAO hack [10, 33, 43]. For example, [8] describe a bug in a test version of Synthetix [38], one of the three top-most valuable crypto assets according to [39]. As we will see in Section 7, this bug has been identified using the algorithm presented in this paper.

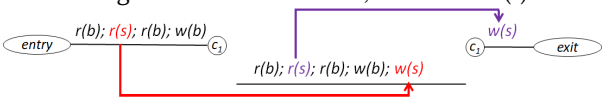
Pattern Based Tools. The standard way to identify problems like the DAO is by searching for a common pattern, of 'write-after-call', e.g., [11, 17, 18, 30, 40, 42]. The idea is that if there are no writes to the state after calls, then it is easy to see that the contract is ECF. Pattern-based solutions yield many false alarms on existing code, preventing the developers from using these tools.

Example 2.1. Consider the contract in Figure 5 that illustrates a "contract-locks" solution to avoid callbacks found in real contracts. It uses a boolean state variable lock to forbid callbacks such that a callback from a different object to execute withdraw will encounter lock set to true, and the require instruction will prevent the execution of the withdraw function.[2] Pattern-based tools flag this function as vulnerable to a reentrancy attack, which is not useful to smart contract developers and deterring them from deploying these tools.

Callback Free Objects. As observed in [23], a semantic way to guarantee immunity to reentrancy attacks is to show that every execution with a callback can also be simulated without callbacks, by making sure that an object is **E**ffectively **C**allback **F**ree (ECF). However, checking ECF is algorithmically challenging in both open and closed environments, since it is clearly undecidable when objects can have infinite state. By definition, the ECF property is a hyperproperty (e.g., see [5, 7, 19, 37]) relating two traces. It states that for every trace with a callback there exists a trace without callback yielding the same effect on the mutable state. Therefore, it is hard to check ECF both dynamically and statically. A constructive approach for creating the witness trace is reordering callbacks such that they are executed outside the context of a callback, and show that the resulting trace is in some sense equivalent to the original trace. This approach requires checking commutativity of potentially unbounded sequences of operations and each such check is potentially undecidable due to the infinite state nature of smart contracts.

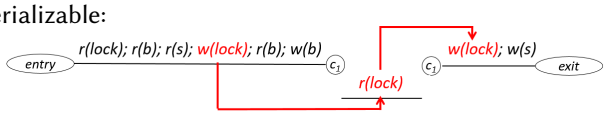
[2]In Solidity, if the condition within the require does not hold, the execution is reverted to the initial state.

Conflict Serializability. A simple conservative way to check for ECF already suggested in [23], is to check that an object is ECF by reordering operations without read/write conflicts. This method, called conflict serializability, is the basis for parallelization in modern database systems, e.g., [9]. Two executions are *conflict-equivalent* if every pair of read/write operations appears in the same order in both of these executions. An execution is therefore *conflict-serializable* if there exists another valid execution of the object that is conflict-equivalent to it. Thus, ECF can be ensured using the definition for conflict-serializability if we find a callback-free reordering of an execution which is conflict-equivalent to the execution with the callbacks. Consider the malicious execution trace from Figure 2 described above, where read (r) and write (w) operations appear on the edges:



this trace is not conflict-serializable because the read of s in the first call is conflicting with the write of s in the callback (marked red), and the write of s in the first call is conflicting with the read of s in the callback (marked purple). Thus, any attempt to reorder the callback before or after the first call will change the order of conflicts.

[23] suggested to check for conflict serializability dynamically, i.e., checking each execution at runtime. Even statically, conflict serializability is easier to check using some syntactic techniques, e.g., [41]. Therefore, the original motivation of this work was to implement a static algorithm for checking conflict serializability. Unfortunately, conflict serializability is over-conservative and prohibits valid solutions for reentrancy attacks. For example, the aforementioned Figure 5 contains a corrected version of the DAO. The main idea of this corrected code is to deploy a boolean lock preventing unintended callbacks. However, there are traces with callbacks which are not conflict serializable:



observe that the lock variable is written to before and after the callback, and thus the read of the lock variable in the callback cannot be reordered with respect to either write. Therefore, a tool based on conflict serialization would yield, as well, to a false alarm in these kind of solutions.

2.2 Simplified semantic solution based on commutativity checks

```
1  check_ECF_single_callnode(n, f):
2    prefix = extract_prefix(f,n)
3    suffix = extract_suffix(f,n)
4    L = get_left_movers(prefix)
5    R = get_right_movers(suffix)
6    if (L == F || R == F)
7        return ECF
8    else return MayNotBeECF
```

Fig. 6. Pseudocode of the algorithm for checking a function with a single call node.

Our Solution. We propose a constructive ECF analysis that can be realized using SMT solvers. For the overview, we present a simplified and intuitive version of the definition, that does not show all edge cases. The full definition appears in the main sections of the paper. For simplicity, we assume here that only one function f of all functions F in our object code contains a single location n in which callbacks may appear. (The method generalizes to any number of such nodes and functions.) We partition f in two parts *prefix* and *suffix*, representing respectively the code before and after the location n. Then, we consider sequences T of the form $prefix; A; suffix$ where $A \in F^*$, i.e., all possible sequences of function calls from the object, of unbounded length. Let $\alpha(\cdot)$ denote the multiset of letters in a sequence. The goal is to find subsequences G, H contained in A, i.e., $\alpha(G) \uplus \alpha(H) \subseteq \alpha(A)$, such that the callback-free sequence $G; prefix; suffix; H$ is final-state equivalent to T.

```
54 contract DeFi {
55     uint countTransfers;
56     bool transfersEnabled;
57     uint mintedTokens;
58     bool isMinting;
59     function init(uint amt) {
60       require (!transfersEnabled);
61       require (isMinting);
62       mintedTokens += amt;
63       transfersEnabled = true;
64       isMinting = false;
65       ext_call();
66       countTransfers += 1;
67       isMinting = true;
68     }
69     function transfer() {
70       require (transfersEnabled);
71       countTransfers += 1;
72     }
73     function mint() {
74       require (isMinting);
75       mintedTokens += 1;
76     }
```

Fig. 7. A challenging example that involves both left- and right-movement. The state consists of two boolean variables and two integer variables countTransfers and mintedTokens. It is ECF since state that can be reached with callbacks can also be reached without callbacks. Intuitively, after running init once, both mint and transfer can be run to account for any delta in the integer variables that could have been accrued in callbacks.

A pseudocode of the algorithm for checking a simplified version of our constructive ECF definition is given in Figure 6. It operates by extracting the code segments that pertain to the 'prefix' and 'suffix' parts of the code of f with respect to the *call node* N, which yields control to callbacks. The algorithm then computes the set of *left* and *right* movers (similar in spirit to [29]). The left and right movers determine how the subsequences G, H from the above definitions are chosen. The simplified version of the algorithm shown in Figure 6 assumes that *all* callbacks can be either moved to the left, or *all* can be moved to the right.

For the code in Figure 5, we note that all callbacks can be moved to the left thanks to *left-projection*: when lock is set to true, the requirements for both withdraw and deposit are not satisfied, thus they revert without changing the state. Therefore, they can be omitted. As a result, the check of left-movement for the segment of code of withdraw, from the start node until the call node N (the prefix), and the full code of either withdraw or deposit, succeeds. Thus, any execution of withdraw that has a callback is final-state equivalent to an execution of withdraw without a callback. Indeed, in the algorithm we will have $L = F$ and return ECF.

2.3 Strengthening the technique for more challenging examples

The algorithm in Figure 6 is already capable of handling two popular schemes for safe handling of callbacks, i.e., locks for ensuring left-movement and putting callback in the end for ensuring right-movement. Furthermore, by checking final-state equivalence, it has precision that surpasses the existing state of the art. However, we show a simple example of a safe code whose callback safety cannot be explained with only left- or only right-movements, but a mix of both.

The code given in Figure 7 contains a single call node. Two boolean variables, transfersEnabled and isMinting, are used as locks to the functions transfer and mint, respectively. Each of these increment an integer variable, countTransfers and mintedTokens, respectively. The init function increases by amt the value of mintedTokens assuming minting is enabled and transfers are not. It then flips the value of both locks and initiates an external call from which callbacks can be triggered. Upon returning from the external call, countTransfers is incremented and minting is re-enabled. This code is safe since any calls to mint in the callback will fail and have no effect on the final state, while calls to transfer can be postponed after init. The algorithm in Figure 6 fails to show that the code in Figure 7 is safe. We note that a key property of the callbacks is that they commute with each other: transfer and mint effects are not interfering with each other and thus not blocking movements in opposite direction: mint projected left and transfer moved right. To

simplify the presentation, we ignore the case that init is called as the callback. Importantly though, init does not violate our property, and our technique is able to prove that.

```
1  check_ECF_single_callnode(n, f):
2    prefix = extract_prefix(f,n)
3    suffix = extract_suffix(f,n)
4    L = get_left_movers(prefix)
5    R = get_right_movers(suffix)
6    if (L + R == F)
7      return check_no_move_collisions(L, R)
8    else return MayNotBeECF
9
10 check_no_move_collisions(L, R):
11   mleft = F − R
12   mright = F − L
13   while mleft changes:
14     mleft = mleft + not_move_right_for(mleft)
15   while mright changes:
16     mright = mright + not_move_left_for(mright)
17   if mleft & mright = empty:
18     return ECF
19   else return MayNotBeECF
```

Fig. 8. Pseudocode of an algorithm for checking a function with a single call node, that allows bidirectional movement of callbacks.

A more precise algorithm for ECF checking is given in Figure 8. For simplicity of the presentation, it is still not handling more than a single call node, but we describe the generalized technique in Section 5. The algorithm of Figure 8 does not require all callbacks to move to one determined side. Instead, as long as all callbacks can be moved to either side, and callbacks cannot block the movement of another, we can prove a single call node is not adding new behaviors if callbacks are invoked in it. The key property that guarantees the soundness of the algorithm is to show a callback cannot block another callback, in any sequence of callbacks that occurs. We consider initially the sets F-L and F-R. They represent callbacks that must move to the right and to the left, respectively. Since L+R==F this means F-L $\subset R$ and F-R $\subset L$. To illustrate the problem of collisions, we assume there is a function $f \in$ F-R, that must move to the left of the call node, and that there is a function g such that g cannot move to the right of f and that $g \notin$ L. Then, for a sequence of callbacks $g; f$ we cannot prove the existence of an equivalent execution using the reordering technique. A concrete example will be given later in Figure 13.

The function check_no_move_collisions generalizes this check for any potential sequence of callbacks by computing sets mleft and mright that represent the set of callbacks that must move to the left or to the right, respectively. These sets are updated iteratively until a fixed point is reached, starting from the left and right sets computed for commutativity over prefix and suffix segments as before, and updated in each round relative to the set of callbacks F. It is easy to see that the example from Figure 7 can be proven ECF by Figure 8 as transfer and mint commute.

In the following sections, we explain how the definition and algorithm are generalized to handle the case of multiple call nodes in a function. The method applied by Figure 8 can be generalized for any number of call nodes using induction, see Section 5.

2.4 Checking mechanics

The lifting of the dynamic trace-based case to the static case uses the notion of *segments*. For a program Pr we define a finite set of segments which conservatively cover all traces in Pr. We show that if there is a trace violating ECF, then the segments also violate the commutativity and projection properties. This is realized using SMT solvers for checking commutativity and projection.

SMT solvers can be used to soundly reason about commutativity properties, e.g., [1, 4, 44], and we use those in the implementation. Given the known limitations of such solvers in large scale, our chief insight is that for ECF, it is possible to minimize the number of commutativity checks discharged with the SMT solver. This is described in further detail in Section 7.

To intuitively illustrate how our algorithm operates, and how counterexamples are given, we go back to the buggy code from Figure 2. This code contains two functions, one of them containing a single call node (withdraw). Therefore, the algorithm analyzes whether both functions, withdraw

Call node at function withdraw(): line 16		
	withdraw()	deposit()
Move before	X	not checked
Move after	X	not checked
ECF check of withdraw() failed due to the following callback trace at line 16: withdraw();		

(a) Simple counterexample to ECF produced by the analysis

Call node at function withdraw(): line 49		
	withdraw()	deposit()
Move before	✓	✓
Move after	not checked	not checked
ECF check of withdraw() succeeded.		

(b) Proof of ECF produced by the analysis

and deposit, can commute with the code segments before and after the call node, which we denote as withdraw_prefix and withdraw_suffix, resp. It can be seen that withdraw does not commute with either withdraw_prefix nor with withdraw_suffix. Thus, the SMT solver shows us traces for violating the commutativity for both, and the conclusion overall would be that withdraw cannot be moved out if it runs as a callback in this call node. An example summarized output of the analysis is given in Figure 9a.

For the corrected code from Figure 5, assuming the algorithm starts by trying to move both functions to the left, then clearly the callbacks can be projected away with respect to the prefix of the call node–the lock is set to true, and the callbacks have no effect and can be omitted. An example summary is given in Figure 9b.

2.5 Implementation

Implementing our analysis in the realistic setting of Ethereum smart contracts introduces even more challenges: The Solidity language is Turing-complete and implements unique features such as reverting when reaching an exceptional state, and the concept of 'gas' to bound the length of executions, [3] among others. Furthermore, in Solidity, the distinction between a function call that invokes the same contract, and a function that invokes an external contract, is not always clear. While this problem is easier when operating on the EVM bytecode instead (to which Solidity is compiled), the EVM bytecode introduces additional challenges, e.g., hiding local variables and fields using hash functions. Our implementation operates on the EVM bytecode.

3 PRELIMINARIES

Programming language. We formalize our results using a simple imperative programming language in which a program Pr is a (finite) collection of procedures $p_1, \ldots p_k$. Each procedure has its own (finite) set of *local variables* which only it can access, and all the procedures share access to a (finite) set of *global variables*. Procedures are represented using control-flow graphs (CFGs). Every edge e of the CFG is annotated with a *precondition* c and a set of variable assignments a. We refer to the nodes of the CFG as *program locations* and to its annotated edges as *transitions*. We usually range over program locations and transitions using n and ρ, resp. As our results are not tied to a particular syntax of conditions or assignments, we leave those unspecified.

Every procedure has a unique *entry node*, to which no edge leads, and a unique *exit point*, from which no edge leaves. In addition, some of the program locations of a procedure may be *call nodes*. We sometimes refer to call nodes as callback points. Every time a procedure reaches a call node it

[3] In this paper, we will ignore the subject of gas for the sake of a cleaner presentation. Contracts that are verified as ECF will stay safe regardless of actual gas allocation and prices for instructions. This is justified since gas consumption per instruction can be changed [25], exposing existing contracts to new reentrancy exploits.

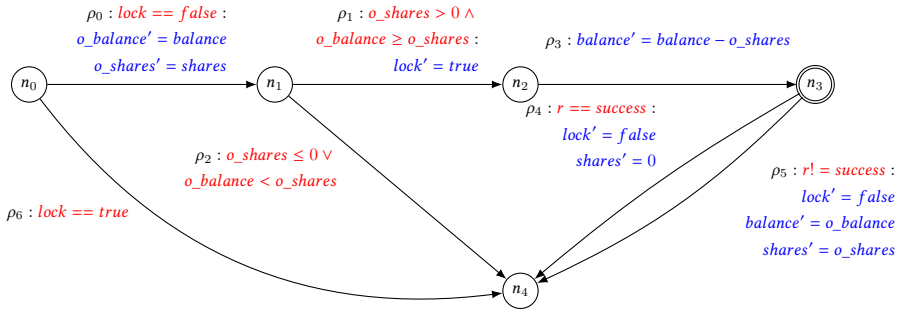


Fig. 10. TS for withdraw procedure from Fig. 5 written in our programming language. Conditions appear in red and assignments in blue.

may invoke arbitrary procedures an arbitrary number of times and then finally *havoc* the value of a specially designated *return variable* r by setting it to an arbitrary value.

Program states $\sigma \in \Sigma$ record the program counter (which is the current location), the values of the program's global variables and the local variables of the currently executing procedure. The state also maintains a stack of the program locations and values of the local variables of pending calls. Note that in our approach we will many times compare states at call nodes, before the execution of the procedure is completed, and hence local variables should also be taken into account. We assume to have at our disposal a semantic function $[\![\cdot]\!]$ which assigns meaning to transitions $[\![\rho]\!] \subseteq \Sigma \times \Sigma$ as a binary relation over program states. Our programs are deterministic in the sense that at most one output state can be produced by applying a transition (with the exception of the aforementioned havoc transitions) to any input state. The intention is that the program can proceed from the program location n at the *source* of a transition $\rho = \langle n, c : a, n' \rangle$ to the *target* program location n' of ρ only when the program is in an *input state* σ which satisfies c and it then produces an *output state* σ' according to the assignments a annotating ρ. Thus, $[\![\rho]\!]$ is comprised of all such pairs of states $\rho = \langle \sigma, \sigma' \rangle$ that define a transition relation. Hence, from now on, we will refer to our CFGs as (a symbolic denotation of) *Transition Systems* (abbreviated as TS). Figure 10 depicts the TS of the withdraw procedure from Figure 5 where n_0 and n_4 are the entry and exit nodes, resp. We write the assignments annotating edges using two-vocabularies in the standard way: The primed variables v' represent the value of a variable v after the transition and the unprimed version v represents its value before the transition executes. We mark its sole call node (n_3) using a double circle.

In our programming language we can describe encapsulated objects as programs defined as the set of TSs for their procedures, and the non-deterministic call mechanism used to represent callbacks. The programming model considered is general enough to define the relevant part of our analysis for most programming languages, and its simplicity helps clarify our presentation.

Traces. A *trace* is a (finite) sequence of transitions $t = \rho_1; \ldots; \rho_n$. We say that a trace *starts* resp. *ends* at program location n if n is the source resp. target program location of its first resp. last transition. We denote the starting resp. ending program location of a trace t by $start(t)$ resp. $end(t)$. We denote the length of a trace t by $|t|$, the *empty* trace by ε, and the *trace composition* operator which concatenates two traces by ; . We say that a trace t_1 is a *subtrace* of a trace t if $t = t_0; t_1; t_2$ for some traces t_0 and t_2. A trace is a *trace of procedure* p if all its transitions come from p's transition system. A trace of procedure p is *well-formed* if the target program location of every transition in it is the source program location of the next transition. A well-formed trace t of p is *complete* if $start(t)$ is p's entry node and $end(t)$ is p's exit node. We refer to complete well-formed traces of procedures as *function traces*. We denote the *set of well-formed procedure traces of a program* Pr by

$TR(Pr)$ and the set of all well-formed traces of procedures in Pr starting at program location n and ending at n' by $TR_{Pr}(n, n') = \{t \in TR(Pr) \mid start(t) = n \wedge end(t) = n'\}$. (We omit the Pr subscript in what follows as we assume to be working with an arbitrary fixed program Pr.).

Example 3.1. In the program shown in Figure 10, we have, for instance, that $TR(n_0, n_3) = \{\rho_0; \rho_1; \rho_3\}$, $TR(n_0, n_4) = \{\rho_0; \rho_1; \rho_3; \rho_4,\ \rho_0; \rho_1; \rho_3; \rho_5, \rho_0; \rho_2, \rho_6\}$, and $TR(n_3, n_4) = \{\rho_4, \rho_5\}$.

A trace t is a *complete callback-free* trace of a program Pr if $t = t_1; \ldots; t_n$, for some $0 \leq n$ such that every t_i, for $i = 1..n$, is a function trace. Thus, the execution of the procedures is not split due to an incoming call. A trace is *callback-free* if it is a subtrace of a complete callback-free trace.

A trace t is a *complete well-formed* trace if it is a complete callback-free trace of Pr or there exist traces t_1, t_2, and t_3 such that (i) t_2 is a complete well-formed trace of Pr, (ii) $end(t_1)$ is a call node, and (iii) the trace $t_1; t_3$ is a *complete well-formed* trace of Pr. Note that conditions (ii) and (iii) ensure that $start(t_3) = end(t_1)$. When t_1 and t_3 are not complete traces and $end(t_1) = start(t_3)$ is a call node, then t_2 is a sequence of complete subtraces which we refer to as the *callbacks*. Thus, a trace t_c is a *callback* in trace t if it is a function trace and there are non-empty traces t_0, t_1 such that $t = t_0; t_c; t_1$. A trace is *well-formed* if it is a subtrace of a *complete well-formed* trace. In the following, unless stated otherwise, we use the term trace to mean a well-formed trace.

Example 3.2. Examples of traces without callbacks from n_0 to n_4 are shown in Ex. 3.1 in $TR(n_0, n_4)$. Examples with callbacks would be (the callback trace is underlined): $\rho_0; \rho_1; \rho_3; \underline{\rho'_6}; \rho_4$ where ρ'_6 is a callback trace, or $\rho_0; \rho_1; \rho_3; \underline{\rho'_0; \rho'_2}; \rho_4$. However, the latter would be pruned out by the execution since it is not feasible to execute ρ'_0 at this point as ρ_1 sets lock to true and hence the condition in ρ'_0 does not hold.

Executions. We denote the set of executions of a trace t by $[\![t]\!]$. An execution $\xi = \sigma_0\, \rho_0 \sigma_1 \ldots \sigma_{n-1} \sigma_n$ is an alternating sequence of states and transitions which start and end with a state and for every $i = 0..n-1$, $\langle \sigma_i, \sigma_{i+1} \rangle \in [\![t_i]\!]$. We say that ξ is an *execution of trace* t if t is the subsequence of transitions in ξ. We denote the *first* and *last* states of ξ by $start(\xi)$ and $end(\xi)$, respectively. We write $\sigma - t - \sigma'$ to denote an execution $\xi \in [\![t]\!]$ of t such that $start(\xi) = \sigma$ and $end(\xi) = \sigma'$. All notions for traces, like being complete, well-formed or callback-free are extended to executions in the natural way.

Definition 3.3 ($\simeq_{FS}$). Executions ξ_1 and ξ_2 are final state equivalent, written $\xi_1 \simeq_{FS} \xi_2$, if $start(\xi_1) = start(\xi_2)$ and $end(\xi_1) = end(\xi_2)$.

It is now possible to use the above notations to define ECF for both executions (dynamic) and programs (static), similarly to [23].

Definition 3.4 ($dECF_{FS}$). A complete well-formed execution ξ is *effectively callback-free*, written $\xi \models dECF_{FS}$, if it is final state equivalent to a complete callback-free execution.

Definition 3.5 ($sECF_{FS}$). A program Pr is *effectively callback-free* (denoted $P \models sECF_{FS}$) if every complete well-formed execution of Pr is effectively callback free.

The notion of *feasible states* will be useful in the following sections:

Feasible states. A state σ is *feasible* for a trace t if t can be fully executed starting at σ, i.e., there exists a state σ' such that $\sigma - t - \sigma'$ is an execution. We denote the set of feasible states for t by $Feasible(t)$ and the set of all feasible states of a set of traces P by $Feasible(P) = \bigcup_{t \in P} Feasible(P)$.

When a state is feasible for a trace, we also say that the trace is feasible for the state. For example, if the trace contains two transitions $(n_1, x \leq 0 : x' = x + 1, n_2); (n_2, x \geq 0 : x' = x * 2, n_3)$ (and x is an integer variable) then the feasible states for this trace are those where x is either 0 or -1 since only in such states we can execute both transitions (as we need both $x \leq 0$ and $x + 1 \geq 0$).

4 SEGMENTS, PROJECTION AND COMMUTATION

This section introduces auxiliary definitions that the static analyses in Sec. 5 relies on, namely segments of code, and the projection and commutation operations on segments. As usual, the static analysis handles many traces at once: the concept of *segment* will allow us to characterize all traces that can arise from using the fragment of code that forms the segment. In order to explain the intuition of our operations, we consider a simple complete well-formed trace which is not callback-free $t_1; t_f; t_2$, where t_f is a function trace and $t_1; t_2$ is a function trace as well. (Note that $end(t_1) = start(t_2)$ is a call node.) We say that t_1 is the left subtrace, and t_2 is the right subtrace, and denote by τ_1, τ_f and τ_2 the segments to which t_1, t_f and t_2, resp., belong. Our technique aims at guaranteeing ECF by proving that the final state of an execution of $t_1; t_f; t_2$ is the same as the final state of an execution of either $\tau_1; \tau_2; \tau_f$ or $\tau_f; \tau_1; \tau_2$ or $\tau_1; \tau_2$ (when starting from the same initial state). In order to prove the equivalence, we define *projection* and *commutation* of pairs of segments. Applying these operations guarantees that the resulting state is the same *and* that in all *feasible states* from which the original segment sequence can start and fully execute, so can the new one. Informally, the projection operation applied on τ_1 and τ_f ensures that an execution of $\tau_1; \tau_f$ leads to the same state as an execution of τ_1 alone. If it holds, we have proven ECF for the considered sequence. Commutation ensures that an execution of $\tau_1; \tau_f$ results in the same state as an execution of $\tau_f; \tau_1$.

4.1 Basic definitions on segments

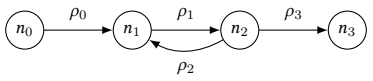
Segments represent potentially unbounded number of traces, going between start, exit, and call nodes. In the definition for segments, we refer to the start and exit nodes of a procedure as call nodes too. In the rest of the paper, we assume to be working with an arbitrary fixed program Pr and that the locations of all its functions are uniquely identified.

Definition 4.1 (Segment). Given two call nodes n and n', the *segment* between n and n' is the set of traces $TR(n, n')$. A segment $TR(n, n')$ is a *function* if n is the start node of a procedure and n' is its exit node, in this case the segment represents the set of all function traces of the procedure. The set of function segments of a program Pr is denoted by $F(Pr)$. A segment belongs to a procedure p if its start and exit nodes belong to p.

Example 4.2. The segment for the program shown in Figure 10 for n_0 and n_3 is $\tau_0 = \{\rho_0; \rho_1; \rho_3\}$, for n_3 and n_5 is $\tau_1 = \{\rho_4\ ,\ \rho_5\}$ and for n_0 and n_4 is $\tau_2 = \{\rho_0; \rho_1; \rho_3; \rho_4\ ,\ \rho_0; \rho_1; \rho_3; \rho_5\ ,\ \rho_0; \rho_2\ ,\ \rho_6\}$, where τ_2 is a function segment, since its traces go from the start node n_0 to the end node n_4.

Importantly, the notion of segments applies to programs with loops, as the next example illustrates. Consider the following function (whose TS is shown to the right):

```
77  function loop(int val) {
78    int aux = 0;
79    do { aux += val; } while (aux < 10);
80  }
```



The function loop has only one segment that goes from the start to the end node, although this segment might contain an infinite number of traces (as val can be negative). In particular, the segment $TR(n_0, n_3)$ contains the traces that start in the node n_0 and end in n_3, but there might be an unbounded number of these traces since we can take the path $\rho_1; \rho_2$ as many times as we like before taking the transition ρ_3 and end at n_3.

Definition 4.3. Given a segment τ, we say that $\sigma - \tau - \sigma'$ if and only if there exist a trace $t \in \tau$ such that $\sigma - t - \sigma'$.

4.2 Segment-sequences

We use sequences of segments (*segment-sequences*), in order to prove that an execution is *ECF*. We use the notation τ for segments and π for segment-sequences.

Definition 4.4 (Segment-sequence). A *segment-sequence* is a non-empty sequence of segments of the program. A segment-sequence is *well-formed* if the end node of each segment is the initial node of the next one.

Following the example shown in Example 4.2, the segment-sequence for the execution trace $\rho_0; \rho_1; \rho_3; \rho_6'; \rho_4$ would be $\tau_0; \tau_2'; \tau_1$, where we have primed the segment τ_2' of the callback procedure.

We need to distinguish when a segment-sequence includes a particular trace of the program.

Definition 4.5. We say that a trace t is represented by a segment-sequence $\pi = \tau_1; \tau_2; \ldots; \tau_n$ if and only if $t = t_1; t_2; \ldots; t_n$ for some traces $t_1, t_2, \ldots, t_n$ such that for every $i = 1, \ldots, n$ we have that $t_i \in \tau_i$.

Definition 4.6. Given a segment-sequence π, we say that $\sigma - \pi - \sigma'$ if and only if there exists a trace t represented by π such that $\sigma - t - \sigma'$.

4.3 Commutation and projection

We define the following concepts about commutativity and projection.

Definition 4.7 (Commutation). Given two segments τ_1 and τ_2, we say that τ_1 commutes with τ_2 for the state $\sigma \in \mathit{Feasible}(\tau_1; \tau_2)$ if and only if $\sigma \in \mathit{Feasible}(\tau_2; \tau_1)$ and if $\sigma - \tau_1; \tau_2 - \sigma'$ and $\sigma - \tau_2; \tau_1 - \sigma''$ then $\sigma' = \sigma''$.

Here the condition $\sigma \in \mathit{Feasible}(\tau_2; \tau_1)$ means that if τ_1 commutes with τ_2 for a state $\sigma \in \mathit{Feasible}(\tau_1; \tau_2)$ then we can execute $\tau_2; \tau_1$ from σ as well. Therefore, commutation for the state σ implies both (i) we can execute $\tau_2; \tau_1$ from the state σ and (ii) it produces the same state. In order to clarify requirement (i), let τ_a and τ_b be the segments containing only the trace with a single transition $\langle n,\ y \geq 0 : x' = 0\ ,\ y' = y - 1,\ n' \rangle$ and $\langle m,\ y \leq 1 : x' = y, y' = y - 1,\ m' \rangle$, respectively. They do not commute for any state σ such that $\sigma[y] = 0$ since $\tau_a; \tau_b$ can be executed, but $\tau_b; \tau_a$ cannot: The first transition in τ_b decrements y to -1, thus the condition $y \geq 0$ in τ_a does not hold. Hence, although when both can be executed they end in the same state, we cannot directly replace $\tau_a; \tau_b$ by $\tau_b; \tau_a$ since when $\sigma[y] = 0$ the second execution would not be feasible and therefore we cannot guarantee that we have an alternative execution.

Definition 4.8 (Left-projection). Given two segments τ_1 and τ_2, we say that τ_1 left-projects with τ_2 for the state $\sigma \in \mathit{Feasible}(\tau_1; \tau_2)$ if and only if if $\sigma - \tau_1; \tau_2 - \sigma'$ and $\sigma - \tau_1 - \sigma''$ then $\sigma' = \sigma''$.

Definition 4.9 (Right-projection). Given two segments τ_1 and τ_2, we say that τ_1 right-projects with τ_2 for the state $\sigma \in \mathit{Feasible}(\tau_1; \tau_2)$ if and only if $\sigma \in \mathit{Feasible}(\tau_2)$ and if $\sigma - \tau_1; \tau_2 - \sigma'$ and $\sigma - \tau_2 - \sigma''$ then $\sigma' = \sigma''$.

Consider the segments τ_0 and τ_2 defined in Example 4.2. τ_0 represents the traces of withdraw until the call node point. τ_2 is representing the withdraw function. We study whether they commute or project in order to prove ECF for traces of withdraw that have withdraw called as a callback. τ_0 does not commute over τ_2 since there is an initial state where the final values of the balance variable could be different: $\tau_0; \tau_2$ does not decrement balance a second time in the callback τ_2 due to the lock being set in τ_0, while $\tau_2; \tau_0$ may fully execute the first withdraw, decrementing balance, after which the trace in τ_0 decrements balance again. However, we have left-projection as $\tau_0; \tau_2$

leads to the same state as τ_0 (because the lock is taken when τ_2 executes and there is only one decrement of balance).

We now define *movement* as a combination of commutativity and projection properties. *Left-movement* expresses that for all feasible states we can either commute or left-project, *right-movement* expresses that we can either commute or right-project.

Definition 4.10 (Left-movement). Given two segments τ_1 and τ_2, we say that $\tau_1;\tau_2$ left-moves if and only if for all $\sigma \in \mathit{Feasible}(\tau_1;\tau_2)$ we have that either τ_1 commutes or left-projects with τ_2 for the state σ.

Definition 4.11 (Right-movement). Given two segments τ_1 and τ_2, we say that $\tau_1;\tau_2$ right-moves if and only if for all $\sigma \in \mathit{Feasible}(\tau_1;\tau_2)$ we have that either τ_1 commutes or right-projects with τ_2 for the state σ.

We distinguish between left and right movements to ensure that the resulting segment sequence represents a trace of the procedure. For example, for the segment-sequence $\pi = \tau_1; f; \tau_2$, if $\tau_1; f$ left-moves we build an equivalent callback-free segment-sequence: for all feasible states either the execution of $\tau_1;\tau_2$ or $f;\tau_1;\tau_2$ is final-state equivalent to π. Both contain real traces of the program. However, we could not use that $\tau_1; f$ right-moves: in case it right-projects we would get the sequence $f;\tau_2$ that does not represent any complete trace.

On the other hand, any movement between different functions preserves the ability to generate a real program trace. This is the reason why we consider a more general kind of movement that includes left-projection, right-projection, commutation and a new kind of projection that eliminates both functions: the zero-projection.

Definition 4.12 (Zero-projection). Given two segments τ_1 and τ_2, we say that τ_1 zero-projects with τ_2 for the state $\sigma \in \mathit{Feasible}(\tau_1;\tau_2)$ if and only if, $\sigma - \tau_1;\tau_2 - \sigma'$, implies $\sigma = \sigma'$.

Zero-projection expresses that the executions of the two segments from a state, do not change that state. For example, if x is an integer variable, assuming $0 \leq x \leq 1000$, the segments $\tau_1 : x' = x * 2$ and $\tau_2 : x' = x/2$ zero-project, but they do not left or right-project or commute.

We define the notion of movement, expressing that for all feasible states we can either commute or left, right or zero-project.

Definition 4.13 (Movement). Given two segments τ_1 and τ_2, we say that $\tau_1;\tau_2$ moves if and only if for all $\sigma \in \mathit{Feasible}(\tau_1;\tau_2)$ we have that either τ_1 commutes, right-projects, left-projects or zero-projects with τ_2 for the state σ.

We use the terminology left-movement to express that if $\tau_1;\tau_2$ left-moves, then the equivalent sequence we obtain keeps the left segment τ_1 (the equivalent sequence is τ_1 or $\tau_2;\tau_1$). The same happens for the right-movements: if $\tau_1;\tau_2$ right-moves, then τ_2 remains. Movements may not preserve any segment: for $\tau_1;\tau_2$, the resulting sequence may be either ϵ, τ_1, τ_2, or $\tau_2;\tau_1$.

Finally, the final state equivalence check used in the definitions of this section can be effectively implemented using SMT encodings for simple fragments of code containing no loops and no use of data structures (like arrays or maps). In presence of these elements, the problem becomes harder. In our system, we have overcome these difficulties by means of abstractions using uninterpreted functions, as described e.g. in the commutativity checks of [1]. Developing more accurate *movement* checkers is an independent problem that can be the focus of future research. Furthermore, our overall analysis can also be parametrized with efficient movement checkers based on syntactic overapproximations relying on read/write operations.

5 THE STATIC ANALYSIS

This section presents our static analysis to prove that a given program satisfies the $sECF_{FS}$ property. We first introduce in Section 5.1 the basic approach to prove that one call node is *solvable* in isolation, i.e., it does not break the ECF property. In order to handle all call nodes in the program, we extend in Section 5.2 our approach with an operation that, once a call node has been solved, we allow joining its left and right segments to gain further accuracy.

Our techniques have to ensure that given a trace we can always find an alternative callback-free one. To this end, we first prove that if we can *solve* (i.e. find a final state equivalent callback-free trace) all traces with callbacks only at *depth* one (i.e. no callbacks inside another callback), then we can solve all traces. Moreover, we only have to show that we can solve traces where all callbacks occur inside a single function, considering all its call nodes. This result generalizes to final state equivalence the reduction to simple traces of [23] that was based on conflict-equivalence.

Definition 5.1 (simple trace). Given a trace $t_1; \ldots; t_n$ with $t_i \in TR$, the depth of t_i in $t_1; \ldots; t_n$ is the number of entry nodes visited minus the number of exit nodes visited in $t_1; \ldots; t_{i-1}$. The depth of the trace is the highest depth of all its t_i. A trace is *simple* if: (1) it is of depth one, and (2) after removing all t_i that are callbacks we obtain a trace $t_{i_1}; \ldots; t_{i_m}$ that is a trace of a procedure p of the program, and we say that it is a simple trace of p.

Lemma 5.2. *If all executions of simple traces of a program Pr are $dECF_{FS}$ then Pr is $sECF_{FS}$.*

The proofs of all our results are provided within the supplementary material.

Therefore, from now on, we will focus on ensuring that all executions of simple traces can be solved. Every simple trace of a procedure p can be represented by a segment-sequence of the form

$$\tau_0; f_0^1; \ldots; f_{k_1}^1; \tau_1; \ldots; f_0^m; \ldots; f_{k_m}^m; \tau_m$$

where all f_j^i are function segments and the start node of τ_0 is the start node of p, the end node of τ_m is the end node of p, and for all $i \in 0 \ldots m-1$, the end node of τ_i and the start node of τ_{i+1} are the same call node. Note that, every pair τ_i and τ_{i+1} captures, resp., the code before and after a call node where any number of callbacks $f_0^{i+1}; \ldots; f_{k_{i+1}}^{i+1}$ can enter. The rest of this section will provide sufficient conditions to ensure that all callbacks can either be removed by projections or sent before τ_0 or after τ_m.

5.1 Solvable call nodes

We first apply commutation and projection operations over a single call node to ensure that, for this call node, we can convert all executions with callbacks in this call node into executions without callbacks in this call node. When defining the segments on which the operations are applied, for the soundness of the analysis, we need to take the *minimal* segments, i.e., segments that do not include any other call node apart from the start and end node.

In this definition we consider that the initial and end nodes of a procedure are call nodes too, as we did before introducing the definition of segment in Def. 4.1

Definition 5.3 (Minimal left/right segments). Given a call node c of a procedure p of Pr with a set of call nodes C, we define the set of minimal left/right segments resp. as follows:

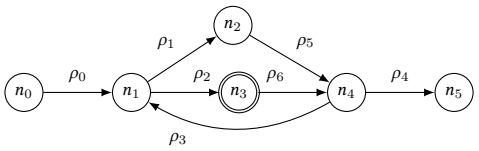
- $SLeft(c) = \cup_{c'}\{ TR(c', c) | \forall t = \rho_1; \ldots; \rho_n \in TR(c', c), \forall j \in \{2 \ldots n\}.\ source(\rho_j) \notin C\}$
- $SRight(c) = \cup_{c'}\{ TR(c, c') | \forall t = \rho_1; \ldots; \rho_n \in TR(c, c'), \forall j \in \{1 \ldots n-1\}.\ target(\rho_j) \notin C\}$

Intuitively, the left (resp. right) segments are those segments τ of p whose end (resp. initial) node is c' for some $c' \in C$, and there are no more call nodes occurring in τ.

Example 5.4. Let us illustrate these sets on the examples of the paper. First, we consider the example in Fig. 5, which is the fixed DAO, and whose TS is given in Fig. 10. Here, in addition to the initial node n_0 and the final node n_4, there is a single call node n_3. Then, $SLeft(n_3) = \{\{\rho_0; \rho_1; \rho_3\}\}$ and $SRight(n_3) = \{\{\rho_4; \rho_5\}\}$. For the original DAO problem in Fig. 2 (where there is no use of the lock variable), we have the same $SLeft(n_3)$ and $SRight(n_3)$ since its TS is like Fig. 10, but omitting transition ρ_6 and all conditions or assignments involving the lock variable.

Example 5.5. We can apply these notions to call nodes that appear in loops. Consider a function with one call node in the loop:

```
82 function loop1(int val) {
83   int aux = 0;
84   do {
85     if (val != 0){
86       aux += val;
87       val++;
88     }
89     else{
90       aux = call();
91     }
92   }
93   while (aux < 10);
94 }
```



The only call nodes are n_3 and the initial and final nodes n_0 and n_5. The set $SLeft(n_3)$ contains segments that represent traces from a call node (the initial n_0 or n_3) to n_3 and $SRight(n_3)$ from n_3 to a call node (the final n_5 or n_3). We first consider the segment that goes from n_0 to n_3: it contains all the traces between these two nodes that do not include any other call node apart from themselves. There might be an unbounded number of such traces since we can take the path $\rho_1; \rho_5; \rho_3$ as many times as we like before taking the transition ρ_2 to end at n_3. The same happens for the traces from n_3 to n_3 and the ones from n_3 to n_5. Then, using the notation $t = \rho_1; \rho_5; \rho_3$,

$$\begin{aligned} SLeft(n_3) &= \{\{\rho_0; \rho_2\ ,\ \rho_0; t; \rho_2\ ,\ \rho_0; t; t; \rho_2\ ,\ \ldots\}, \\ &\quad\ \{\rho_6; \rho_3; \rho_2,\ \rho_6; \rho_3; t; \rho_2\ ,\ \rho_6; \rho_3; t; t; \rho_2\ ,\ \ldots\}\} \\ SRight(n_3) &= \{\{\rho_6; \rho_4\ ,\ \rho_6; \rho_3; t; \rho_4\ ,\ \rho_6; \rho_3; t; t; \rho_4\ ,\ \ldots\}, \\ &\quad\ \{\rho_6; \rho_3; \rho_2\ ,\ \rho_6; \rho_3; t; \rho_2\ ,\ \rho_6; \rho_3; t; t; \rho_2\ ,\ \ldots\}\} \end{aligned}$$

The static analysis needs to consider sequences of n callbacks, e.g., of the form $\tau_1; f_1; \ldots; f_n; \tau_2$, where the f_i (for $i = 1, \ldots, n$) are function segments for the callbacks to all n different procedures in the program. As we do not know which call(s) might arrive at runtime, all permutations of the f_i must be considered. Thus, we cannot just apply the operations for movements in Sec. 4 to each of the functions since it could be the case that, for instance, $f_1; \tau_2$ right-moves (but $\tau_1; f_1$ does not left-move) and $\tau_1; f_n$ left-moves (but $f_n; \tau_2$ does not right-move). A necessary condition in this case is that $f_1; f_n$ must move as well, since f_1 may appear before f_n. However, it is insufficient since there are additional calls in the middle ($f_2, \ldots, f_{n-1}$) whose own ability to move with τ_1 and τ_2 must be preserved independently of f_1 and f_n. Therefore, this imposes additional movement properties of f_1 over all of $f_2, \ldots, f_n$ and of f_n over $f_1, \ldots, f_{n-1}$. The example in Fig. 13 illustrates this situation for only two calls. There, we have a single call node n_1, τ_1 is the segment that contains only the trace with ρ_0 and τ_2 is the segment that contains only the trace with ρ_1. Thus, although f_1 commutes with τ_2 (but not with τ_1) and f_2 commutes with τ_1 (but not with τ_2), because $f_1; f_2$ does not move, any trace represented by the segment-sequence $\tau_1; f_1; f_2; \tau_2$, does not have a final state equivalent callback-free trace, and hence the program is not ECF. This is the reason why we must require $f_1; f_2$ to move.

The aforementioned situation requires leveraging the projection and commutation operations to handle multiple callbacks at a call node. Basically, we classify in Def. 5.6 the calls at this node as either *left-solvable* (commute or project with the minimal left segment) and/or *right-solvable* (commute or project with the minimal right segment), and then Def. 5.7 requires movement properties for those that are exclusively left- or right-solvable.

```
95  contract Example_no_ECF {
96    uint c;
97    uint s;
98    function inc() {
99      c = c+1;
100     call();
101     s = s+1;
102   }
```

```
104   function f_1() {
105     s = s+1;
106     c = 0;
107   }
108   function f_2() {
109     s = 0;
110     c = c+1;
111   }
```

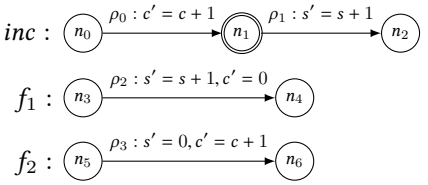


Fig. 13. Example of functions f_1 and f_2 that do not commute. The contract is not *ECF* (trace $\rho_0; \rho_2; \rho_3; \rho_1$)

Definition 5.6. Given a call node c of a procedure p, we define sets of function segments $Left(c)$ and $Right(c)$ as follows:

(1) for every function g in Pr we have that $g \in Left(c)$ iff $\tau; g$ left-moves for all $\tau \in SLeft(c)$.
(2) for every function g in Pr we have that $g \in Right(c)$ iff $g; \tau$ right-moves for all $\tau \in SRight(c)$.

The idea is that the sets $Left(c)$ and $Right(c)$ include the functions that, individually and independently of other functions, can move over the left and right segments of the call at call node c. But as the functions may appear in the callback at any order, we have to take into account the movements between the possible functions. For example, consider functions f_1, f_2 such that $f_1 \notin Right(c)$ and $f_2; f_1$ does not move, then if we consider the sequence of callbacks $f_2; f_1$, then the only possibility for f_2 is to move to the left, although it may belong to $Right(c)$. This happens because the movement to the right of f_1 is impossible. To make sure we are able to handle all potential permutations of functions appearing as callbacks in a call node c, we introduce the sets $MLeft(c)$ (must-left) and $MRight(c)$ (must-right). Informally, these sets include the functions that cannot move over the right and left segments resp.; either because they are not members of $Right(c)$ or $Left(c)$, or because they are blocked by a function, or sequence of functions, that must move left or right.

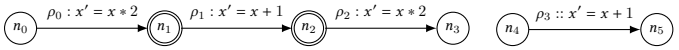
Definition 5.7. Given a call node c of a procedure p, and denoting the set of functions of Pr by $F(Pr)$, we define sets of function segments $MLeft(c)$ and $MRight(c)$ using the least fixed point operator as follows:

(1) $MLeft(c) = LFP_X\left(X \cup \left\{f | f \in F(Pr) \wedge \exists x \in X. f; x \text{ not moving}\right\}\right)$ with $X_0 = F(Pr) \setminus Right(c)$
(2) $MRight(c) = LFP_X\left(X \cup \left\{f | f \in F(Pr) \wedge \exists x \in X. x; f \text{ not moving}\right\}\right)$ with $X_0 = F(Pr) \setminus Left(c)$

Intuitively, we can now define when a call node is solvable by ensuring that we can always take the callbacks at that node and either remove them or send them before its minimal left segment or after its minimal right-segment.

Definition 5.8 (Solvable call node). Given a program Pr, we say that a call node c of Pr is solvable if $MLeft(c) \cap MRight(c) = \emptyset$.

If all procedures in our program have a single call node then, if they are all solvable, it is easy to show that the program is $sECF_{FS}$. However, if a procedure has several consecutive call nodes, we cannot handle each one of them in isolation, as the following example illustrates. Consider a procedure p with two call nodes (left) and a procedure f (right).



There, f is only in $Right(n_1)$ as it only commutes with its minimal right segment, and it is only in $Left(n_2)$ as it only commutes with its minimal left segment. This shows a circularity that implies that we cannot move a callback to f in n_1 out of the trace since it will be moved to n_2 (by commutation) and then back to n_1 (by commutation) again.

We can only ensure ECF if we also impose that, for every function, we will always be able to move it to the right or to the left of all call nodes as the following theorem states:

Definition 5.9 (sECF$_{SS}$). Given a program Pr, it is *sECF$_{SS}$* if and only if for all procedures p in Pr with call nodes C we have that, for every c, c' in C such that c' is reachable from c or $c' = c$, it holds that $MRight(c) \cap MLeft(c') = \emptyset$.

Example 5.10. Consider again the example in Figs. 5 and 10 which is sECF. In Example 5.4, we have seen that $SLeft(n_3) = \{\{\rho_0; \rho_1; \rho_3\}\}$ and $SRight(n_3) = \{\{\rho_4; \rho_5\}\}$. Now let τ_d be the function segment of deposit and τ_w be the function segment of withdraw. We have that $Left(n_3) = \{\tau_d, \tau_w\}$ as for both $\{\rho_0; \rho_1; \rho_3\}; \tau_d$ and $\{\rho_0; \rho_1; \rho_3\}; \tau_w$ left project to $\{\rho_0; \rho_1; \rho_3\}$, since ρ_1 sets lock to true (which is not changed in ρ_3), and in such state both deposit and withdraw do nothing. Then all functions are in $Left(n_3)$ and hence the program is *sECF$_{SS}$*.

Now, we show why the example in Fig. 2 (which is not ECF) is not *sECF$_{SS}$*. As seen in Example 5.4 we have that $SLeft(n_3) = \{\{\rho_0; \rho_1; \rho_3\}\}$ and $SRight(n_3) = \{\{\rho_4; \rho_5\}\}$, and recall that we do not use the lock variable and we do not have transition ρ_6. Here, we have that τ_w neither belong to $Left(n_3)$ nor to $Right(n_3)$, since without using lock, we cannot project or commute.

Theorem 5.11. *If a program is sECF$_{SS}$ then it is sECF$_{FS}$*

5.2 Segments Join

The technique we have considered in the previous section is powerful, but it can be more accurate if, once a call node has been solved, we allow joining its left and right segments. For instance, consider a general segment-sequence representing simple traces of some procedure of our program $\tau_0; f_0^1; \ldots; f_{k_1}^1; \tau_1; \ldots; f_0^m; \ldots; f_{k_m}^m; \tau_m$. Then if we solve the call node between τ_0 and τ_1, i.e., if we take all functions $f_0^1; \ldots; f_{k_1}^1$ out of this call node, by projecting or commuting with τ_0 or τ_1, we will have τ_0 and τ_1 together without any callback in the middle. Hence, we can consider them together as a single segment $\tau_{0;1}$ after joining them. The reason for joining them is that having larger segments leads to strictly more accurate results. The following example shows a situation where we can gain accuracy by joining segments:

```
112 contract Example_no_ECF {
113   uint c;
114   function discount2() {
115     c = c − 1;
116     call();
117     c = c − 1;
118     call();
119     c = 0;
120   }

122
123   function
          mult(){
124     c = c ∗ 2;
125   }
126 }
```

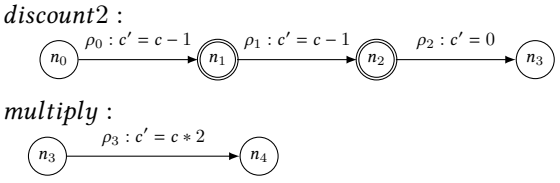
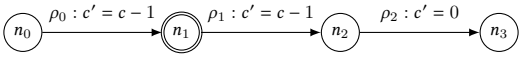


Fig. 14. ECF contract that requires call node removal and cannot be proven using minimal segments)

Example 5.12. Consider the example in Fig. 14 whose procedure discount2 has three transitions and two call nodes, namely n_1 and n_2 (where callbacks can enter), while the function *multiply* has a single transition and no call nodes. Assume that our trace has a callback (to multiply) at each call node: $\rho_0; \rho_3; \rho_1; \rho_3'; \rho_2$ (we have primed the second use of multiply). The minimal segments of discount2 are (i) the set of traces from n_0 to n_1, i.e. $\tau_0 = \{\rho_0\}$, (ii) the set of traces from n_1 to n_2, i.e. $\tau_1 = \{\rho_1\}$, and (iii) the set of traces from n_2 to n_3, i.e. $\tau_2 = \{\rho_2\}$. We use f for the function segment $\{\rho_3\}$ of multiply. Now, the segment-sequence representing our trace is $\tau_0; f; \tau_1; f; \tau_2$. We start by handling the second call node, n_2, first. We can do either commutation of f over τ_2 or we can do right-projection of $f; \tau_2$ to τ_2, e.g., in the latter we have solved the call node n_2, and the new segment-sequence (representing final state equivalent traces to our trace) is $\tau_0; f; \tau_1; \tau_2$. But

now we cannot go further and solve n_1 since we cannot apply any projection or commutation on $\tau_0; f$ or $f; \tau_1$. However, if we use the fact that n_2 has already been solved, we can consider that n_2 is no longer a call node, since it does not have callbacks in it, then our transition system would be:



and hence if we compute the right segment of n_1 we obtain the segment $\tau_{1;2} = \{\rho_1; \rho_2\}$, which is the join of segments τ_1 and τ_2, and hence the sequence we have to consider now is $\tau_0; f; \tau_{1;2}$. Then, we can right-project $f; \tau_{1;2}$ to $\tau_{1;2}$, and the result $\tau_0; \tau_{1;2}$ is a callback-free sequence (which implies that we have a callback-free execution). The following table compares the different options to try to solve the call nodes, with and without joins ($\oslash$ means no operation can be applied, and $\square$ means that callbacks were successfully removed):

$\tau_0; f; \tau_1; f; \tau_2$		
start with n_1	start with n_2	start with n_2 with joins
$\oslash$	RightProj(f, τ_2)	RightProj(f, τ_2)
	$\oslash$	remove n_2 as call node
		RightProj$(f, \tau_{1;2})$
		$\square$

Note that the reason we can right-project $f; \tau_{1;2}$ to $\tau_{1;2}$ is that after setting c to zero, we have that $2 * 0 = 0$, thus f is not changing c.

We will thus consider that we can apply an operation to *remove call nodes* that enables a more accurate static analysis for procedures with multiple call nodes. However, once we introduce this operation, the order in which call nodes are solved might affect the accuracy of the analysis results. Assume we have a segment-sequence π with k callbacks ($n_1, \ldots, n_k$ ordered by their position at the execution). We establish a new order in which they are solved, by means of a permutation $i_1, \ldots, i_k$ of $1, \ldots, k$ which indicates that we will solve the callback nodes in the order $n_{i_1}, \ldots, n_{i_k}$. For instance, the order $2, 1$ leads to a solution in Example 5.12. The general concept we have is an order $<_O$ that indicates when a call node is solved before another, i.e. if $c' <_O c$ then we know that c' has been solved when we solve c. This means that when checking if c is solvable we have to first remove as call nodes from the transition systems all those call nodes c' such that $c' <_O c$. Now, we present a generalization of the $sECF_{SS}$ property to the case where we solve the call nodes in a given order. First we define the notion of solvable call node for a given order $<_O$.

Definition 5.13 (Orderly solvable call node). Given a program Pr and an order $<_O$ on the call nodes of Pr. We say that a call node c of Pr is solvable wrt. $<_O$ if c is solvable after removing as call nodes from Pr all $c' <_O c$.

Our main result is that if there exists an order for which all call nodes in our program are solvable, then the program is ECF:

Definition 5.14 ($sECF_{OS}$). We say that a program Pr is $sECF_{OS}$ if there exists a total order $<_O$ for the call nodes C of Pr such that all $c \in C$ are solvable with respect to $<_O$.

Theorem 5.15. *If a program is $sECF_{OS}$ then it is $sECF_{FS}$.*

Example 5.16. Consider the example in Fig. 14 for the function discount2 whose TS is in Ex. 5.12, taking O as $n_2 <_O n_1$, we have that $SLeft(n_2) = \{\{\rho_1\}\}$ and $SRight(n_2) = \{\{\rho_2\}\}$, and $SLeft(n_1) = \{\{\rho_0\}\}$ and $SRight(n_1) = \{\{\rho_1; \rho_2\}\}$. Now, we can prove that both discount2 and multiply belong to $Right(n_2)$ and to $Right(n_1)$.

6 CALLBACK INVARIANT

Motivated by challenging contracts found in the Ethereum environment (similar to the one in Example 6.2 to follow), we introduce the notion of *callback invariant* as a way to increase the accuracy of the $sECF_{SS}$ and $sECF_{OS}$ approaches. As a standard invariant, a callback invariant is a property that holds whenever we reach the call node but, in addition, it must also hold after executing any possible sequence of callbacks. The notion of callback invariant can be extended to several call nodes, having an invariant per call node. Note that we can always take *true* as invariant in a call node if we do not need it. Then, taking *true* as a (fictitious) invariant for the initial node, we have that the invariants must be preserved by all transitions between two call nodes (or the initial node) and they need to be preserved when executing any of the functions in the contract. Being precise:

Definition 6.1. Given a procedure p with call nodes C and initial node n_0, we say that $I(C)$, from nodes to properties, is callback invariant of C, if, taking $I(n_0) = true$, we have that

- For every $c \in C$ and every segment τ in $SLeft(c)$ starting at node $n \in C \cup \{n_0\}$, we have that if σ satisfies $I(n)$ and $\sigma - \tau - \sigma'$, then σ' satisfies $I(c)$.
- For all $c \in C$ and $g \in F(Pr)$ if σ satisfies $I(c)$ and $\sigma - g - \sigma'$, then σ' satisfies $I(c)$.

Example 6.2 (Monotone lock). The contract appearing in Figure 15 is a simplification with no loops of the Synthetix case study. It uses a counter to prevent callbacks that can lead to harmful results. This contract only has two call nodes: n_2 and n_3. The node n_2 is solvable according to the $sECF_{OS}$ approach, but n_3 is not. The minimal segments of the node n_3 are $SLeft(n_3)$, which only contains the segment $\tau_l = \{\rho_0; \rho_3\}$, and $SRight(n_3)$, which only contains $\tau_r = \{\rho_5; \rho_6,\ \rho_5; \rho_7\}$. This node is not solvable: the function exchange does not left-move nor right-move with the segments τ_l and τ_r, resp. The states that are problematic for the right-movements are only the ones where $\sigma[count] = \sigma[lc] - 1$. For any other state, after executing exchange we will obtain a state σ' such that $\sigma'[count] \neq \sigma'[lc]$, thus the execution will *revert*. Hence, if we could prove that no execution gets to the call node n_3 in the problematic state described above, we would be able to prove that the contract is $sECF_{FS}$.

We can check that I, with $I(n_3) = \{lc \leq count\}$ and $I(n_2) = true$, is a callback-invariant. First, it is clear that the only trace that goes from n_0 (the initial node) to n_3 is $t = \rho_0; \rho_3$. Then, for any initial state σ if $\sigma - t - \sigma'$ then $\sigma'[count] = \sigma[count] + 1$ and $\sigma'[lc] = \sigma[count] + 1$, thus σ' satisfies $I(n_3)$. On the other hand, if we execute any function of the program from a state that satisfies $I(n_3)$, then it ends at a state that satisfies $I(n_3)$: the value of the local variable lc does not change and $count$ can only increment. Note that the property is invariant provided there are no overflows, however since we start in 0 and can only increment by 1 in each call, the assumption that we will not reach 2^{256} is reasonable. There is a more complex invariant which does not need this assumption but for readability reasons we have decided not to present it.

We want to use the information that a callback invariant gives us to check the commutation and projection of the callbacks. We first adapt the definition of movements to take into account the invariants: in the previous version we included all feasible states, now we are going to restrict it to the ones that satisfy the invariant.

Definition 6.3 (Left-movement with precondition). Given two segments τ_1 and τ_2, and a property P, we say that $\tau_1; \tau_2$ left-moves assuming the precondition P if and only if for all $\sigma \in Feasible(\tau_1; \tau_2)$ such that σ satisfies P we have that either τ_1 commutes or left-projects with τ_2 for the state σ.

The definitions of *right-movement with precondition*, *zero-projection with precondition* and *movement with precondition* are modified analogously.

```
pragma solidity ^0.4.24;
contract Bank {
  mapping (uint => uint) public deposits;
  uint initIndex;
  uint count = 0;
  function exchange(uint remaining) {
    count += 1;
    uint lc = count;
    deposit = deposits[initIndex];
    if(deposit == 0){
      initIndex++;
    }
    else if(deposit > remaining){
      uint newAmount= deposit − remaining;
      deposits[initIndex] = newAmount;
      user.send(remaining);
    }
    else{
      deposits[initIndex] = 0;
      user.send(deposit);
      initIndex++;
    }
    require(lc == count);
  }
}
```

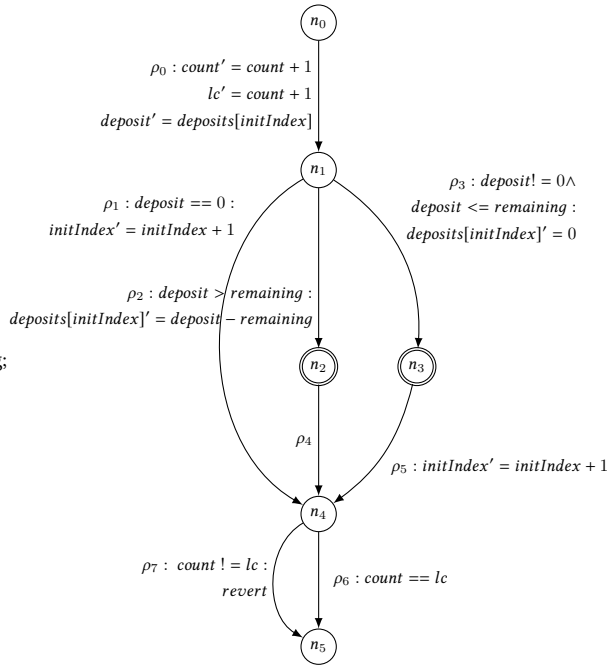


Fig. 15. Simplified Synthetix contract that requires a callback invariant

Consider the segment $\tau_r = \{\rho_5; \rho_6,\ \rho_5; \rho_7\}$ and τ_{exc} representing the function exchange. Using the previous definition, we can check that $\tau_{exc}; \tau_r$ does not right-move: it reverts for any state σ such that $\sigma[count] \neq \sigma[lc] - 1$, but for any state σ such that $\sigma[count] = \sigma[lc] - 1$ they do not commute or right project. Nevertheless, $\tau_{exc}; \tau_r$ right-moves assuming the precondition $I(n_3)$, because the problematic states do not satisfy $I(n_3)$.

Then, we just have to adapt the definitions of $Left(c)$, $Right(c)$ to use these new movements using a precondition map I and modify the sets $MLeft(c)$ and $MRight(c)$ according to them.

Definition 6.4. Given a procedure p with call nodes C and initial node n_0 and a map I from nodes to properties. We extend function $Left(c, I)$ and $MLeft(c, I)$ for some $c \in C$ as follows:

(1) for every function g in Pr we have that $g \in Left(c, I)$ iff for all $\tau \in SLeft(c)$ starting at node $n \in C \cup \{n_0\}$ we have that $\tau; g$ left-moves assuming the precondition $I(n)$.
(2) $MLeft(c, I)) = LFP_X \left(X \cup \{f | f \in F(Pr) \wedge \exists x \in X . f; x \text{ not moving assuming precondition } I(c)\}\right)$ with $X_0 = F(Pr) \setminus Right(c, I)$

The definition $MRight(c, I)$ is modified analogously and the definition of $Right(c, I)$ only varies in that it assumes $I(c)$ as precondition.

For the call node n_3 of the previous example, according to the original definition $Left(n_3) = \emptyset$ and $Right(n_3) = \emptyset$, but if we consider the invariant I with $I(n_3) = \{lc \leq count\}$ then $Right(n_3, I) = \{\tau_{exc}\}$ so $MLeft(n_3, I) = \emptyset$.

Finally, we define the notion of $sECF_{IOS}$ program that takes callback invariants into account.

Definition 6.5 ($sECF_{IOS}$). Given a program Pr, it is $sECF_{IOS}$ if and only if there exist an order $<_O$ for the call nodes C of Pr and a callback invariant I of C such that all $c \in C$ are solvable assuming I with respect to $<_O$.

Theorem 6.6. *If a program is $sECF_{IOS}$ then it is $sECF_{FS}$*

Finally, we can prove that the above contract is *ECF*. The map I, with $I(n_3) = \{lc \leq count\}$ and $I(n_2) = true$, is a callback invariant and $MLeft(n_3, I) = \emptyset$, which implies that n_3 is solvable. Since n_2 is also solvable, we conclude that the contract is $sECF_{IOS}$.

7 IMPLEMENTATION AND EXPERIMENTAL EVALUATION

Our implementation decompiles [27] smart contracts given as EVM bytecode and produces code in an intermediate representation amenable to static analysis and the generation and discharge of verification conditions using SMT solvers, such as Z3 [16]. Furthermore, since the EVM bytecode does not contain a notion of procedures or functions, and the Solidity compiler generates generic 'dispatch' code to jump to the appropriate function code, we split out the function implementations from the large EVM bytecode. Currently, we have bounded support for loops using finite unrolling, we are working on the general extension.

Motivated by the real smart contracts analyzed, the actual algorithm implemented is based on $sECF_{OS}$, but with a predetermined call node ordering: going linearly from latest (in program-order) call nodes to earlier call nodes. The considerations for choosing that particular approach are:

- The $sECF_{OS}$ is strictly more precise than $sECF_{SS}$ approach, thanks to join operations.
- Nevertheless, trying all possible call node orders, given that there are functions that have over 10 call nodes, may be impractical due to the number of required SMT queries.
- The later-to-early call node order is a good fit for well-written contracts that make sure to place call nodes after all updates to the global state were performed. For these contracts, the approach would lead to faster proofs of ECF.

We have run our benchmarks on an Amazon AWS c5n.2xlarge machine. The SMT solver used is Z3, with a timeout of 60 seconds per query. To each call node we set a timeout of 5 minutes for analyzing it, requiring all needed SMT queries to run within the time span.

Choice of call nodes. Call nodes are detected in a conservative manner—any instance of a call instruction, except for STATICCALL, is considered a call node. The STATICCALL instruction is not considered a call node because it enforces the VM to avoid any writes to the global state in all calls until the STATICCALL returns, and therefore trivially projects. Our method assumes a completely open environment, in which only the contract checked is fixed and known. As we show later, many contracts use other contracts as libraries and thus establish properties that should hold when the contract calls the library. In such cases, it is possible to ignore certain call nodes, because the callee contract is guaranteed not to trigger a callback. In result, this would lead to a greater number of verified contracts (those marked * in Table 1).

Delegate calls. Two special instructions in the EVM bytecode are DELEGATECALL and CALLCODE. These instructions allow executing an external code, that is not necessarily known at compile-time, and execute it in the context of the caller's state. We are treating these instructions as regular call nodes in order to prove ECF, but it should be noted that if a contract contains such delegating instructions, then ECF does not guarantee sound modular reasoning.

Realistic setting. To validate the usefulness of our approach in a realistic setting, we picked as benchmark set the most used and invoked smart contracts. To that end, we extracted the top-150 contracts based on volume of usage, as of December 31st, 2019[4]. A total of 132 contracts were successfully decompiled, but 38 contracts did not contain call nodes and are excluded. Since the ECF property that we check is based on the results for all functions, we give in Table 1 the summarized results for all functions extracted out of all contracts.

[4]up to Ethereum blockchain block number 9193265 until 2019-12-31 23:59:45 UTC

	# fs	% all fs	% fs w. CN	Avg. T (sec.)
ECF Verified (>0 CNs)	242	8.9	62.7	30
ECF Violated	133	4.9	34.5	132
Timeout	11	0.4	2.8	1240

Analysis of violations (# fs)	
Confirmed violations	18
No source code	18
FPs due to call node choice*	56
FPs due to the implementation	30
FPs to $sECF_{OS}$	11

Table 1. Summarized ECF results. 'CN' stands for 'call node', and 'f' for 'function'.

Out of the total 2733 functions extracted, 386 contained call nodes, and thus are candidates to ECF verification. Out of these 386 functions, 242 are verified to be ECF (62.7%), 133 are reported as violating ECF (34.5%), and 11 time out (2.8%) before a definite answer is returned.

Manual assesment of the violations. We manually analyzed 115 of the violations (18 did not have source code). 18 functions are confirmed to be true violations.[5] The majority of the violations (56) are due to the over-conservative choice of call nodes. After a careful inspection, we believe those call nodes can be omitted, because they are calling into contracts that cannot generate callbacks. As our analysis considers just the contract inspected for ECF, it cannot infer properties of the callees. We therefore conclude that by extending the analysis tool to allow the user fine-grained control over the choice of call nodes, the precision of the analysis increases significantly. 30 of the violations are a result of overapproximations in the tool, mainly due to the intricacies of analyzing low-level EVM such as pointer arithmetic based on hashing and compiler-generated copy loops. The remaining 11 violations are true false-positives, since we found that the functions have ECF behavior that cannot be proven using $sECF_{OS}$—namely, it is possible to construct an equivalent execution using a different function from the one being checked.

Challenging real case study. The vast majority of the contracts analyzed in Table 1 are rather simple. Therefore, the reader may conclude that all smart contracts are simple, which is not our experience. Some of the valuable smart contracts actually implement complex logic, which makes checking ECF and other properties quite hard. One such example is the reentrancy bug [2] in *Synthetix* [38]—a high-volume De-Fi[6] application.[7] Our technique can mechanically verify both: one of them as-it-is, the other using callback invariants. To the best of our knowledge, none of the techniques available are able to show that immunity to reentrancy attacks is true for the fixed contract.

Comparison to other tools. We compared our implementation to other existing tools whose premise is to handle 'reentrancy bugs': *Securify2* [42] and *Slither* [17]. Notably the properties checked by these tools are more restrictive than ECF: *Securify* and *Slither* check that there are no global state updates following a call instruction. When we ran this case study (as well as our lock-based example of Figure 5), *Securify* and *Slither* both failed to show that it is actually safe (*Securify* times out after hours of running on the Amazon machine). The same holds for the simplified version of our case study as appears in Figure 15. In addition, neither *Securify* nor *Slither* were able to prove the correctness of the lock-based example of Figure 5.

We compared *Securify* and *Slither* against our tool on a compatible subset of 110 contracts from the benchmark. We could not compare all contracts from the benchmark because the other tools accept Solidity source code and sometimes even specific Solidity versions, rather than EVM

[5]We have contacted the code owners and are waiting for their responses.
[6]Decentralized Finance
[7]according to [39], rated 2nd in locked USD value, with $116.7M locked as of May 5th, 2020.

bytecode. Because of that we could only compare *Securify* to 10 contracts, and the results were aligned with ours in 9 contracts. *Securify* crashed on the last contract.

For *Slither*, there were 15 examples where the results did not agree. In two of them *Slither* reported a bug, but our tool was able to prove the contracts correct. In the other two *Slither* missed real bugs, and our tool detected them. In the remaining 11, our tool detected false bugs while *Slither* proved them correct. These bugs were caused by our conservative choice of call nodes and overapproximations in the static analysis.

8 CONCLUSIONS AND RELATED WORK

We have presented a novel static analysis that proves modularity of the contract for any execution and can be applied to ensure effective-callback freedom prior to deployment. Reentrancy attacks have led to the most severe exploits in the blockchain and, as we have shown in the paper, general techniques for ensuring modularity of programming languages can be used to detect ECF violations and avoid these malicious attacks. This kind of reentrancy problems were pinpointed as a possible source of correctness bugs [3, 30]. As discussed in Sections 1 and 2, our work is inspired by that of [23] who pioneered the idea of ECF as means to immune modules (contracts) from reentrancy attacks and enable modular reasoning. However, the analysis of [23] is dynamic hence it cannot be used to verify ECF. In the rest of this section, we review other closely related work.

[31] present a framework, called FSolidM [9], that allows preventing reentrancy via a built-in locking mechanism. In contrast, we present a technique for verifying ECF, and thus the absence of reentrancy bugs, which is language-agnostic while allowing judicious use of callbacks. [21] survey on recent theories and tools for formal verification of Ethereum smart contracts focusing on the F*-formalized small-step semantics presented by [22] and its Horn clauses-based abstraction. Most relevant to our work is over-approximation of the single-reentrancy property [22, 35] which, intuitively, states a contract is single-entrant if it cannot perform any more calls once it has been reentered. This restriction, however does not mean that callbacks may not have unique behaviors which cannot be exposed in callback-free executions. [42] report of a parametric static verification tool which can detect whether a contract violates a given security property encoded as a bad pattern in the contract's data-flow graph. To detect reentrancy-related bugs, they use a pattern which forbids writes after calls. Thus, their restrictions are more severe even than the ones imposed by conflict-based ECF. Similar patterns are used by [17, 40].

[26] identified a family of bugs in blockchain-based smart contracts, dubbed event-ordering (or EO) bugs, which are related to the dynamic ordering of contract events, i.e. calls of its functions. However, in contrast to our work, the ordering they investigate is between different transactions while our focus is on errors which occur within one transaction. Thus, the class of bugs we are after does not overlap with theirs. Also, our tool is static while theirs is based on dynamic (symbolic) testing. In MAIAN [32] the authors present a symbolic execution tool for detecting contracts vulnerabilities such as ether leaking. Such vulnerabilities may intersect with reentrancy vulnerabilities (for example, the DAO's reentrancy attack leads to leaked ether).

[11] checks information-flow properties to identify vulnerabilities that occur in a multi-transaction setting, including callbacks.

[34] employ taint analysis on Ethereum traces to detect reentrancy vulnerabilities. The dynamic check implemented there is more precise than the as-of-then static analysis tools and its performance is similar to [23] for non CREATE-generated callnodes. (the latter did not include CREATE as a callnode candidate). A work by [18] define a language for patterns in Ethereum transactions representing malicious behaviors, and an instrumented Ethereum client that can detect such patterns in-vivo. Patterns can be added and removed based on voting in a smart contract. 4 out of 6 patterns presented in [18] are related to reentrancy vulnerabilities. Of most relevance to our work

is the comparison between pattern-based detection of malicious attacks and semantic equivalence checking. In both the dynamic and static settings, the pattern-based approach can easily lead to over-approximation and false positives, while on the other hand not giving full clarity about the actual immunity of the code to malicious callbacks. In contrast, our approach, while more expensive computationally, gives strong guarantees about callbacks not being able to influence the execution in unexpected ways, while also being more resistant to false positives.

As [36] note when discussing the similarity of smart contracts to concurrent objects, enabling modular verification is one of the highlighted challenges. A key benefit of our semantic equivalence based approach, when compared to pattern-based techniques, is that it enables to modularly check properties of ECF contracts. For example, [46] present VeriSol, a tool for static verification of smart contracts against a state machine model specification and an access-control policy. The analysis is capable of inferring *contract invariants*—properties of the state of the contract which are true when none of its procedures is pending. However, the analysis is not modular. We believe that our approaches can be combined so that once the contract is verified as ECF, VeriSol can infer its class invariants in a sound modular way. A different approach to modularity is given in [13], where reentrancy is defined as an information-flow property, and reentrancy security as a property that guarantees invariants inductiveness even in the presence of callbacks. Their approach has the benefit of finer-grained policies, enabling supporting systems that consist of multiple contracts, but also requires the user to annotate 'critical sections' in the code. Complementary approaches to modularity check an invariant of the program, e.g., [6, 28].

As regards the state equivalence check, we have implemented an SMT-based technique similar to the ones proposed to check commutativity in the context of model checking of concurrent programs (see, e.g., [1, 44]). However, our method is generic wrt. the particular check used and we will benefit for future improvements in this domain. For example, [4] present a refinement-based technique for synthesizing commutativity conditions for operations on representations (implementations) of abstract data types (ADTs). The algorithm is generalized to handle left/right-movers [29]. We utilize commutativity checks as a "black box" in our algorithms. Thus, in that respect our works are complementary. Nevertheless, the projection checks and the gradual simplification of the commutativity checks done in the treatise algorithm are novel.

Finally, our problem is also related to the atomicity analysis [20, 45] studied in the concurrency setting. An atomicity analysis infers that code blocks are atomic, i.e., that every execution of the program is equivalent to one in which those code blocks execute without interruption by other threads. An important difference of our work with the atomicity analysis in [45] and earlier work in [20] is that our "must-left" and "must-right" sets provide strictly more accurate analysis than their left-movers and right-movers. This is because we do not require that an action commutes in the same direction (left or right) with all the actions of the other threads. Here is a simple example considering atomicity at the instruction-level, composed of 3 functions (that in the concurrent setting would be 3 threads and an interleaving point in the call):

```
function f(){
  x = x + 1;
  call();
  y = y + 1;
}
```

```
function f1(){
  x = 1;
}
```

```
function f2(){
  y = 1;
}
```

We can verify atomicity/ECF because f1 right-commutes with the segment "y=y+1" and f2 left-commutes with the segment "x=x+1" and additionally the action in f1 and the action in f2 commute (this commutation is not considered by previous approaches and hence they would fail to verify atomicity). While our technique makes the analysis potentially more costly, as there are

more commutations to be checked, in our setting the number of actions (functions in our context) is typically small, compared to the number of actions in multi-threaded programs.

ACKNOWLEDGMENTS

We would like to thank the anonymous referees for their comments that have helped improve the presentation and contents of this paper. This work was funded partially by the Spanish MCIU, AEI and FEDER (EU) projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33, and by the CM project S2018/TCS-4314. This research was partially supported by the Israeli Science Foundation (ISF) grant No. 1810/18. This material is based upon work supported by the United States-Israel Binational Science Foundation (BSF) grant No. 2016260. The research was supported in part by the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University, and Pazy Foundation grant No. 347853669; The Israel Science Foundation (ISF) grant No. 1996/18.

REFERENCES

[1] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. 2018. Constrained Dynamic Partial Order Reduction. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. 392–410.

[2] Anonymized for the submission. 2020. Anonymized for the submission. .

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts SoK. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Springer-Verlag New York, Inc., New York, NY, USA, 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

[4] Kshitij Bansal, Eric Koskinen, and Omer Tripp. 2018. Automatic Generation of Precise and Useful Commutativity Conditions. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 115–132.

[5] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. 2019. Verifying Relational Properties using Trace Logic. In *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*. 170–178.

[6] Sidi Mohamed Beillahi, Gabriela Ciocarlie, Michael Emmi, and Constantin Enea. 2020. Behavioral Simulation for Smart Contracts. (2020), To appear.

[7] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. *ACM SIGPLAN Notices* 39, 1 (2004), 14–25.

[8] Thomas Bernardi, Nurit Dor, Anastasia Fedotov, Shelly Grossman, Alexander Nutz, Lior Oppenheim, Or Pistiner, Mooly Sagiv, John Toman, and James Wilcox. 2020. Preventing Reentrancy Bugs - Another Use Case for Formal Verification. https://www.certora.com/blog/reentrancy.html.

[9] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[10] Alina Bizga. 2020. A hackers' dream payday: Ledf.Me and Uniswap lose $25 million worth of cryptocurrency. https://securityboulevard.com/2020/04/a-hackers-dream-payday-ledf-me-and-uniswap-lose-25-million-worth-of-cryptocurrency/. [Online; accessed 11-May-2020].

[11] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. (2020), To appear.

[12] Vitalik Buterin. 2016. CRITICAL UPDATE Re: DAO Vulnerability. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/. [Online; accessed 2-July-2017].

[13] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew Myers. 2020. Securing Smart Contracts with Information Flow. In *Third International Symposium on Foundations and Applications of Blockchain 2020*.

[14] Consensys. 2019. Ethereum Smart Contract Best Practices. https://consensys.github.io/smart-contract-best-practices/known_attacks/. [Online; accessed 14-May-2020].

[15] Phil Daian. 2016. (2016). http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[16] Leonardo De Moura and Nikolaj Bjørner. [n.d.]. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[17] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[18] Christof Ferreira Torres, Mathis Baden, Robert Norvill, and Hugo Jonker. 2019. ÆGIS: Smart Shielding of Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS 19)*. Association for Computing Machinery, New York, NY, USA, 2589–2591.

[19] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2019. Monitoring hyperproperties. *Formal Methods Syst. Des.* 54, 3 (2019), 336–363.

[20] Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*. ACM, 338–349.

[21] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Foundations and Tools for the Static Analysis of Ethereum Smart Contracts. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 51–78.

[22] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 243–269.

[23] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL* 2, POPL (2018), 48:1–48:28.

[24] Fernando Hernandez. 2019. Understanding Callbacks and Promises. https://dev.to/_ferh97/understanding-callbacks-and-promises-3fd5. [Online; accessed 14-May-2020].

[25] Hudson Jameson. 2019. Security Alert: Ethereum Constantinople Postponement. https://blog.ethereum.org/2019/01/15/security-alert-ethereum-constantinople-postponement/. [Online; accessed 11-May-2020].

[26] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2019. Exploiting the Laws of Order in Smart Contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 363–373. https://doi.org/10.1145/3293882.3330560

[27] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective.

[28] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing Smart Contract with Runtime Validation. (2020), To appear.

[29] Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (Dec. 1975), 717–721.

[30] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 254–269.

[31] Anastasia Mavridou and Aron Laszka. 2018. Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 270–277.

[32] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 653–663.

[33] Daniel Palmer. 2018. SpankChain Loses $40K in Hack Due to Smart Contract Bug. https://www.coindesk.com/spankchain-loses-40k-in-hack-due-to-smart-contract-bug. [Online; accessed 11-May-2020].

[34] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/sereum-protecting-existing-smart-contracts-against-re-entrancy-attacks/

[35] Clara Schneidewind, Markus Scherer, Ilya Grishchenko, and Matteo Maffei. 2020. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. (2020), To appear.

[36] Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. In *Financial Cryptography and Data Security*, Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (Eds.). Springer International Publishing, Cham, 478–493.

[37] Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 57–69.

[38] Synthetix. 2020. Synthetix - Decentralised synthetic assets. www.synthetix.io.

[39] The Concourse Open Community. 2019. DeFi Pulse. https://defipulse.com/. [Online; accessed 11-May-2020].

[40] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 9–16.

[41] Omer Tripp, Roman Manevich, John Field, and Mooly Sagiv. 2012. JANUS: exploiting parallelism via hindsight. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 145–156.

[42] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

[43] Cooper Turley. 2020. imBTC Uniswap Pool Drained for $300k in ETH. https://defirate.com/imbtc-uniswap-hack/. [Online; accessed 11-May-2020].

[44] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. 2008. Peephole Partial Order Reduction. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 382–396.

[45] Liqiang Wang and Scott D. Stoller. 2005. Static analysis of atomicity for programs with non-blocking synchronization. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*. ACM, 61–71. https://doi.org/10.1145/1065944.1065953

[46] Yuepeng Want, Shuvendu Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Bprb, and Immad Naseer. 2019. Formal Specification and Verification of Smart Contracts for Azure Blockchain. , 13 pages. arXiv:1812.08829v2.

[47] Gavin Wood. 2016. Ethereum: A Secure Decentralised Generalised Transaction Ledger. http://gavwood.com/paper.pdf. [Online; accessed 5-July-2017].