Relaxed Effective Callback Freedom: A Parametric Correctness Condition for Sequential Modules With Callbacks

Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv

Abstract—Callbacks are an essential mechanism for event-driven programming. Unfortunately, callbacks make reasoning challenging because they introduce behaviors where calls to the module are interleaved. We present a parametric method that, from a particular invariant of the program, allows reducing the problem of verifying the invariant in the presence of callbacks, to the callback-free setting. Intuitively, we allow callbacks to introduce behaviors that cannot be produced by callback free executions, as long as they do not affect correctness. A chief insight is that the user is aware of the potential effect of the callbacks on the program state. To this end, we present a parametric verification technique which accepts this insight as a relation between callback and callback free executions. We implemented our approach and applied it successfully to a large set of real-world programs.

Index Terms—Smart contract verification, Event-driven programming, Unbounded re-entrancy, Callbacks

1 Introduction

Callbacks occur in sequential programs when a method of one module invokes a method of another module, and the latter, either directly or indirectly, invokes one or more of the former module methods before the original method invocation returns. Callbacks are useful: They provide an effective mechanism for implementing event-driven programming by allowing a callee module to delegate the execution back to the caller [19]. For example, in Ethereum, a module ("smart contract") transferring cryptocurrency to another module may be probed by the recipient to obtain more information about the on-going transaction. Though effective, callbacks can lead to subtle mistakes because they introduce behaviors where calls to the module are interleaved, which, as in concurrent programming [19], [22], can be very tricky to understand and reason about. This is particularly true in open environments, e.g., Ethereum, where callbacks can originate from added new code. Indeed, several high profile attacks on smart contracts exploit callbacks [4], [5], [6], [9], [20], [26].

The danger of callback attacks, also called *reentrancy attacks*, led to many suggestions for syntactical program restrictions, e.g., delaying external calls [8], [19]. However, these restrictions are overly severe and several realistic programs violate them. Seeking to tame the unruly behavior of callbacks without restricting programming led Grossman *et al.* [15] to introduce *Effective Callback Freedom* (ECF)—a correctness condition for sequential modules which guarantees that callbacks are well-behaved. Intuitively, a module is effectively callback free if for every execution trace of the module with callbacks, there exists "an equivalent" trace comprised of a sequence of invocations of the module methods without any interfering callbacks, dubbed a *callback free* execution, which starts and ends in the same states as the original execution.

Example 1. We consider the execution with callbacks ξ illustrated in Fig. 1to show the intuition of the ECF approach. In order to prove the safety of ξ, this approach considers callback-free executions—as ξ_1 and ξ_2— where the executed procedures have been reordered to remove the callbacks, and checks if they are final-state equivalent to the original execution (i.e., they start and finish at the same states as ξ). In this case, the execution ξ_1 is not final-state equivalent to ξ as they end in different states (e.g., if they start at the initial state $x = 0$, then ξ ends with $x = 6$ and ξ_1 with $x = 10$). However, the execution ξ_2 is final-state equivalent to ξ as switching the order between the instructions $x = x + 1$ and $x = x + 5$ does not modify the final result. Hence, the original execution ξ is safe as there exists a final-state equivalent callback-free execution.

The benefit of ECF is that it allows reasoning about *module invariants* (properties of the module's state at quiescent points,

- *Elvira Albert and Albert Rubio are with the Complutense University of Madrid, 28040 Madrid, Spain, and also with the Institute of Knowledge Technology, 28006 Madrid, Spain. E-mail: albert@cs.upc.edu, alberu04@ucm.es.*
- *Shelly Grossman, Noam Rinetzky, and Mooly Sagiv are with Tel-Aviv University, Ramat Aviv 69978, Israel. E-mail: {shelly, mooly}@certora.com, noam.rinetzky@gmail.com.*
- *Clara Rodríguez-Núñez is with the Complutense University of Madrid, 28040 Madrid, Spain. E-mail: clarrodr@ucm.es.*

Manuscript received 5 Aug. 2021; revised 27 Apr. 2022; accepted 4 May 2022. Date of publication 30 May 2022; date of current version 13 May 2023.
This work was supported in part by Spanish MCIU, AEI and FEDER (EU) projects under Grants RTI2018-094403-B-C31 and RTI2018-094403-B-C33, in part by CM project under Grant S2018/TCS-4314, in part by Israeli Science Foundation (ISF) under Grant 1810/18, in part by United States-Israel Binational Science Foundation (BSF) under Grant 2016260, in part by Blavatnik Interdisciplinary Cyber Research Center (Tel Aviv University), in part by Pazy Foundation under Grant 347853669, and in part by Israel Science Foundation (ISF) under Grant 1996/18.
(Corresponding author: Clara Rodríguez-Núñez.)
Digital Object Identifier no. 10.1109/TDSC.2022.3178836

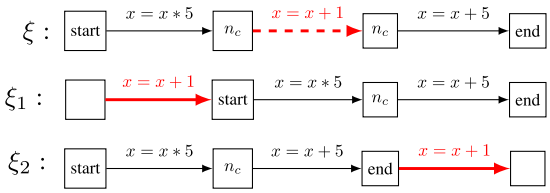


Fig. 1. Execution with a callback (ξ) and possible callback-free reorderings. We use dashed lines for the callback.

i.e., when its methods are not being executed), by considering *only* callback free executions: Any state the module might be in after executing a trace with callbacks is reproducible by a callback free trace. Hence, it is possible to prove module invariants assuming that callbacks do not occur, and these invariants hold even in the presence of callbacks. This use of the ECF correctness condition was suggested in [15] and made possible in [1] which describes an effective conservative static technique for verifying that a module is ECF.[1]

Neither the definition of ECF [15] nor the sound static analysis [1] which verifies it place syntactical restrictions on programmers. However, they do impose a rather severe semantical constraint: Callbacks are not allowed to introduce new behaviors, even when these may be considered benign or even desired by the programmer. While in general, and perhaps in most cases, the assurance ECF brings is that no surprising unintended states might arise due to callbacks is welcome, neither the property nor its verification method can be used when the module is allowed to have richer behaviors due to callbacks. Intuitively speaking, in these cases, we need a correctness condition (and a static analysis) that allow modules to exhibit unique behaviors in the presence of callbacks, yet still restrict these behaviors to a manageable, understandable set of behaviors. The price to pay for such an inclusive condition is that we will no longer have a "one-size fits all" correctness property, as ECF is, but rather a parametric property which encompasses the programmer's view of what constitutes benign callback-induced behaviors. These allowed deviations will naturally depend on the properties the module is designed to preserve.

In this article, we propose *Relaxed Effective Callback Freedom (R-ECF)*, a parametric property-guided correctness condition which allows modules to have non-ECF, yet benign, behaviors by integrating the programmer's feedback into the correctness condition. Technically, the programmer defines the deviant behaviors that callbacks are allowed to introduce using a reflexive and transitive relation (i.e., a preorder) $\sqsupseteq_R$ on module states. A module is R-ECF if every quiescent state σ_1 of the module which can be produced by an arbitrary execution starting at some state σ, has a "smaller" state σ_1^{cf} (i.e., $\sigma_1 \sqsupseteq_R \sigma_1^{cf}$) resulting from a callback-free execution starting at σ. The condition is a strict generalization of the ECF property which can be instantiated by using state equivalence $\equiv$ as the underlying preorder. The intended users of R-ECF are highly sophisticated programmers who develop systems where unique behaviors of callbacks are essential. The number of this kind of programmers is small, as typical solutions for reentrancy attacks are based on forbiding any deviant behaviour introduced by callbacks. This way, ECF accounts for most contracts in the wild, where programmers do not wish to burden themselves with reasoning about complicated behaviors introduced by callbacks. On the other hand, R-ECF is necessary to verify complex systems where callbacks are allowed if they introduce benign behaviours.

We accompany the new correctness condition with a verification methodology which allows establishing module invariants while considering only callback free executions. The technique expects the user to (i) prove that the desired invariant I holds in all *callback free* executions of the module, (ii) provide a preorder $\sqsupseteq_R$ on module states which *respects* I, i.e., $\sqsupseteq_R$ should ensure that if a state σ'' satisfies I then all states $\sigma' \sqsupseteq_R \sigma''$ "bigger" than σ'' satisfy it too, and (iii) establish that the module is R-ECF with respect to the given preorder. Once the user fulfills the aforementioned requirements, the methodology ensures that the desired property I is indeed a module invariant. Interestingly, we show that, in some cases, the required preorder $\sqsupseteq_R$ can be synthesized from the property I the module is designed to preserve (see Section 4.1).

To assist the user establish the third requirement, we provide a conservative static analysis algorithm that validates that a module is R-ECF with respect to the given preorder $\sqsupseteq_R$. The algorithm requires that the latter is $\sqsupseteq_R$-*monotonic* (see Section 3.1.) Intuitively, a module is $\sqsupseteq_R$-monotonic if the execution of code segments between invocations of methods preserves the order of states. Technically, the algorithm generalizes the *commutativity* and *projection*-based technique of [1] to consider R-ECF as the desired correctness condition instead of ECF. The generalization of the algorithm (and of the correctness property) is crucial: it allows proving safety of modules which have executions with callbacks that reach states that are not reachable without using callbacks but are considered harmless.

1.1 Motivating Example

We show the benefits of R-ECF using the standard example for callback safety, the DAO (Decentralized Autonomous Organization) contract [6]. The contract acts as a "bank" which allows client contracts to deposit funds and withdraw them later. Thus, it is critical that the contract would always be *solvent*, i.e., has enough cryptocurrency to pay back the deposited funds. The contract, as we show, is not ECF. Nevertheless, we describe how (i) we can prove the solvency property considering only callback free executions by using the R-ECF correctness condition parameterized with an appropriate preorder on states, and (ii) establish that the contract is R-ECF by considering only *simple executions* [15], i.e., executions comprised of a series of procedure invocations which can be interrupted whenever a method of another module is invoked by an arbitrary sequence of invocations of the module's procedures, however, these interrupting procedures themselves never get interrupted.

1.1.1 The DAO contract

Fig. 2 shows a simplified version of the DAO contract. The variable shares maps users to the sum of the individual

1. The original work [15] showed that verifying ECF is undecidable and provided a conservative *dynamic* technique to check that *a given trace* is ECF, i.e., has an equivalent callback-free trace.

```
1  contract DAO{
2   uint[] shares;
3   uint balance;
4   bool lock;
5
6   function deposit() payable{
7    // require(!lock);
8    shares[msg.sender] += msg.value;
9    balance += msg.value;
10  }
11
12  function withdraw(){
13   require(!lock);
14   if(shares[msg.sender] > 0){
15    lock = true;
16    balance -= shares[msg.sender];
17    msg.sender.transfer(shares[msg.sender]);
18    shares[msg.sender] = 0;
19    lock = false;
20 }}}
```

Fig. 2. Simplification of the DAO contract that uses a lock to forbid callbacks to withdraw. The contract is not ECF, but it is R-ECF.

funds they deposited in the contract. The variable balance is holding the total amount of funds owned by the contract.[2] The state of the contract can be manipulated using two procedures: deposit and withdraw. deposit stores funds in the contract by increasing the caller's shares by the value sent as a parameter. In Solidity, msg is a special variable that always exists, providing information about the current transaction. The field sender of msg stores the caller's 'address', which uniquely identifies it, and the field value stores the funds (in cryptocurrency) transferred in the transaction. The withdraw procedure inverts deposits: A user calling withdraw pulls out all their funds stored in the contract based on the value of shares, decreasing the current amount from the contract's own balance and transferring it to the caller's balance. The transfer is implemented using the operation transfer in Line 17. The semantics of the virtual machine allows the recipient of the money to execute their own code upon receiving funds. Thus, we refer to such an operation as a *call node*. The significance of the call node designation is that the recipient's code might execute a callback to the DAO code. If the transfer call succeeds, the caller's shares value is set to zero. If it fails, the contract's state implicitly reverts back to its state before withdraw was invoked.

In the original DAO contract, Line 13 was omitted, and hence the contract was vulnerable to an attack utilizing callbacks: Initially, an attacker would invoke withdraw, and in Line 17 control would be yielded to the attacker, thus, allowing the attacker to call back to the DAO contract by invoking the method withdraw again. Notably, the attacker already received the value stored in its entry at the shares map, but by calling withdraw again before the value of shares was updated, the attacker is able to receive the same amount a second time. This callback bug can be avoided by adding a lock variable to the contract that must be false to execute the body of withdraw, as shown in Line 13: A callback invocation of withdraw will find lock set to true and the execution will revert, without being able to maliciously reduce the balance. Such a fix has been introduced to many smart contracts to avoid similar problems.

Using lock to prevent invocations of withdraw as callbacks is sufficient to render the contract correct with respect to the solvency property, i.e., it ensures that the balance of the contract shown in Fig. 2 is always sufficient to allow all users to withdraw their funds. However, it does not make the contract ECF: A user calling deposit as a callback would send funds to the contract, but immediately after completing the execution of the callback and the return of control to withdraw, the shares entry of the user is nullified, thus the user would not be able to withdraw the sent funds. Such a sequence of actions would result in a state where the balance of the contract is higher than the sum of its shares, a situation which is impossible to recreate using callback free executions.

1.1.2 Verifying solvency using callback free executions

If we consider only callback free executions then it is rather easy to verify that the DAO contract is solvent. However, as the DAO contract is not ECF, we cannot apply the approach of [1] to use this fact to prove solvency.[3] However, if we carefully examine the effect of callbacks to deposit on the contract's state, we see that from the viewpoint of the solvency correctness property, they are rather harmless: executing deposit as a callback is not dangerous as it may only result in the contract gaining funds. We can utilize this observation to allow such "benevolent" callbacks as we discuss below.

Example 2. We consider the contract in Fig. 2, and the solvency property defined on the contract's state:

$$I(\sigma) = \sigma[balance \geq \sum shares].$$

If solvency is a module invariant of the contract, then we can ensure that the contract always has enough balance to give back to its clients the shares they have in their accounts. It can be easily proven that callback-free executions preserve property I, but as the contract is not ECF, we need to show it holds for executions with callbacks too. However, instead of employing a verification procedure which explicitly considers executions with callbacks, we lift the fact that the solvency property holds for callback free executions to the general case with the help of the R-ECF correctness condition parameterized with the following relation on states:

$$\begin{aligned}\sigma_1 \sqsupseteq_R \sigma_2 \Leftrightarrow &(\sigma_1[shares] = \sigma_2[shares]) \\ &\wedge (\sigma_1[balance] \geq \sigma_2[balance]) \\ &\wedge (\sigma_1[lock] = \sigma_2[lock])\,.\end{aligned}$$

The intuition behind the choice of $\sqsupseteq_R$ is that we allow executions with callbacks when their final state is "better" than a state that is reachable without using callbacks (in the sense that it stores the same shares in the accounts but may have a greater balance): We write $\sigma - t - \sigma_1$ to denote an execution that starts in a state σ and ends in a state σ_1 via the trace t. If we manage to prove

2. In Ethereum, the balance variable is maintained by the executing virtual machine. However, for clarity of the presentation, we avoid using the predefined variable and the special instructions used for money transfer and implement them by explicit updates to the state.

3. It is possible to make the contract ECF by using lock to prevent callbacks to deposit too. Indeed, it was shown in [1] that if we uncomment the command in Line 7, the contract is solvent.

that the contract is R-ECF with respect to $\sqsupseteq_R$ then for any execution $\sigma - t - \sigma_1$ there is a callback-free execution $\sigma - t^{cf} - \sigma_1^{cf}$ such that $\sigma_1 \sqsupseteq_R \sigma_1^{cf}$, i.e., for every execution of the contract there is a callback free execution from the same initial state that ends with a "worse" state. We can now prove that the contract is solvent: The execution $\sigma - t^{cf} - \sigma_1^{cf}$ does not contain callbacks, thus it preserves the property I. Hence, if $I(\sigma)$ holds then $I(\sigma_1^{cf}) = \sigma_1^{cf}[balance \geq \sum shares]$ also holds. The states σ_1 and σ_1^{cf} satisfy $\sigma_1 \sqsupseteq_R \sigma_1^{cf}$, therefore $\sigma_1[balance - \sum shares] \geq \sigma_1^{cf}[balance - \sum shares] \geq 0$. Hence, $I(\sigma_1)$ holds, and the execution $\sigma - t - \sigma_1$ preserves I. We conclude that I is a module invariant of the contract, and thus the contract is immune to reentrancy attacks.

We note that an invariant of this form:

$$I'(\sigma) = \sigma[balance = \sum shares]$$

will be also allowed in our framework. However, we achieve more flexibility in our provided invariant $I(\sigma)$ since we only require to have enough balance to cover all users. This allows for example that a user quits leaving her shares to the contract.

1.1.3 Verification of R-ECF Using Simple Executions

To complete our proof of solvency we need to verify that the contract in Fig. 2 is R-ECF with respect to the $\sqsupseteq_R$ preorder defined in Example 2. We do so by generalizing the static analysis of ECF given in [1] to our setting, as we explain below.

The static analysis of [1] is based on a reduction given in [15] which says that if a module is not ECF then there is a *simple* execution which exemplifies it, i.e., if a module is *not* ECF then there is a simple execution $\sigma - t - \sigma_1$ of the module which *cannot* be matched with a callback free execution of the form $\sigma - t^{cf} - \sigma_1^{cf}$ where $\sigma_1 \equiv \sigma_1^{cf}$ Thus, to prove that a module is ECF it suffices to show that no such problematic simple trace exists. The analysis performs this check conservatively by determining whether it is possible to transform every simple execution $\sigma_1 - t - \sigma_2$ to a callback free execution $\sigma_1 - t^{cf} - \sigma_2$ via a sequence of commutation and projection operations on t.

Our analysis is based on a generalization of the above reduction to R-ECF: We show that if the module is not R-ECF with respect to a given preorder on state $\sqsupseteq_R$ then there is a simple execution $\sigma - t - \sigma_1$ of the module which *cannot* be matched with a callback free execution $\sigma - t^{cf} - \sigma_1^{cf}$ such that $\sigma_1 \sqsupseteq_R \sigma_1^{cf}$. The reduction, however, does not apply to any preorder, it holds only if the analyzed module is *R-monotonic* with respect to $\sqsupseteq_R$ (see Definition 7). Intuitively, a module is *R-monotonic* if for every *intra*procedural execution $\sigma_s - t - \sigma_e$ of every procedure of the module, starting either at the procedure's entry or at a call node, ending at the procedure's exit or at a call node, and for every state σ'_s "smaller" than σ_s (i.e., $\sigma_s \sqsupseteq_R \sigma'_s$) there exists another execution $\sigma'_s - t' - \sigma'_e$ with the same starting and ending points such that $\sigma_e \sqsupseteq_R \sigma'_e$. With monotonic preorders, we find ourselves again at the fortunate situation where the proof of a module is reduced to reasoning about properties of callback free and simple executions. Our analysis consists of two parts. First, we verify that the analyzed module is R-monotonic. Then, we conservatively verify the absence of problematic simple traces by determining whether it is possible to transform every simple execution $\sigma_1 - t - \sigma_2$ to a callback free execution $\sigma_1 - t^{cf} - \sigma_2^{cf}$ such that $\sigma_2 \sqsupseteq_R \sigma_2^{cf}$ via a sequence of swap and remove operations of all possible different interrupting invocations.

Example 3. Any simple execution $\sigma - t - \sigma_1$ of the contract shown in Fig. 2 which contains an invocation of withdraw leads to a failed require command in line 13. This failure would cause the state of the module to revert back to its initial state (i.e., $\sigma = \sigma_1$), and thus allow us to match $\sigma - t - \sigma_1$ with the empty execution. (Recall that the $\sqsupseteq_R$ relation is reflexive.) Any simple execution $\sigma - t - \sigma_1$ in which only deposit is invoked as a callback can be replaced by a callback free execution $\sigma - t^{cf} - \sigma_1^{cf}$ in which the sequence of callback invocations occurs right before the invocation of the withdraw procedure they interrupt. As deposit increases the balance of the contract but the corresponding increase to the shares map get nullified by withdraw we get that $\sigma_1 \sqsupseteq_R \sigma'_1$.

1.2 Summary of Contributions

The main contributions of this paper can be summarized as follows.

1) We present *Relaxed Effectively Callback Freedom (R-ECF)*, a novel parametric correctness condition for sequential modules with callbacks which allows to conveniently specify the behavior of modules with *meaningful* callbacks–ones which introduce benign behaviors leading to states that cannot be reached in callback-free executions.
2) We provide an effective conservative static analysis for verifying the module is R-ECF with respect to a given preorder $\sqsupseteq_R$.
3) We introduce a verification framework which allows to establish module invariants by considering only callback-free executions for R-ECF modules.
4) We present a methodology for automatically synthesizing a preorder relation from the desired invariant.
5) We implemented a prototype verifier based on our technique and apply it successfully to a large set of real-world smart contracts.

This article extends our OOPSLA'21 paper [1] by generalizing both the definition and the analysis of ECF to R-ECF. The relevance of R-ECF is that it enables more relaxed relations that are necessary to verify modules in which meaningful callbacks are allowed. Our extended experiments confirm that we are able to ensure the correctness of contracts that could not be verified using previous approaches as ECF.

1.3 Organization of the Article

The rest of the article is structured as follows: Section 2 defines the necessary notations and basic definitions and formalizes the notion of ECF. Section 3 defines R-ECF and describes our static analysis technique for checking R-ECF. Section 4 discusses our technique for verifying module invariants and presents our approach for synthesizing the required preorder from the desired invariant. Section 5 presents the implementation and its evaluation on Ethereum smart contracts. Section 6 discusses related work and concludes.

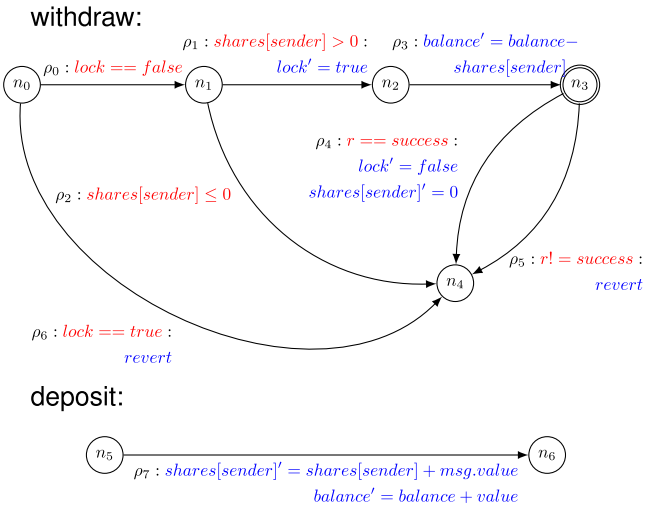


Fig. 3. CFGs for withdraw and deposit procedures from Fig. 2 written in our programming language.

2 Background

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). Fig. 3 depicts the CFGs of the withdraw and deposit procedures from Fig. 2. Every edge e of the CFG is annotated with a *preconditionc* and a set of variable assignments a. 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*. Every time a procedure reaches a call node it 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. In Fig. 3, n_0 and n_4 are the entry and exit nodes of the withdraw procedure, resp. We mark its sole call node (n_3) using a double circle.

A trace is a (finite) sequence of transitions $t = \rho_1; \ldots; \rho_n$. A trace is a trace of procedure p if all its transitions come from p's transition system, and it is complete if it starts at p's entry node and ends at p's exit node. We refer to complete traces of procedures as function traces. We denote the set of traces of procedures in Pr by $TR(Pr)$, and set of traces in Pr starting at program location n and ending at n' by $TR(n, n') = \{t \in TR(Pr) \mid start(t) = n \wedge end(t) = n'\}$ (with $start(t)$ and $end(t)$ representing the starting and ending location of the trace t resp.). For example, if we consider the program in Fig. 3, then $TR(n_0, n_4) = \{\rho_0; \rho_1; \rho_3; \rho_4\ ,\ \rho_0; \rho_1; \rho_3; \rho_5\ ,\ \ \rho_0; \rho_2\ ,\ \rho_6\}$. A trace t is a complete callback-free trace of a program Pr if $t = t_1; \ldots; t_n$ with every t_i a function trace. Thus, the execution of the procedures of a complete callback-free trace is not split due to an incoming call. For example, the trace $\rho_0; \rho_1; \rho_3; \rho_5$ is callback-free, but the trace $\rho_0; \rho_1; \rho_3; \rho_7; \rho_4$ is not as the call to deposit (ρ_7) is intercalated.

A state is an assignment to all local and global variables as well as the current node. We write $\sigma - t - \sigma_1$ to denote an execution that starts in a state σ, ends in a state σ_1 via the trace t. We say that 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 $Feasible(t)$. For example, if we consider again the program in Fig. 3, then the feasible states for the trace $\rho_0; \rho_1$ are those in which $lock = false$ and $shares[msg.sender] > 0$ since only in such states we can execute both transitions.

2.1 Segments and Segment-Sequences

We use the notion of *segment* to characterize all traces that can arise from executing a fragment of code. Segments represent potentially unbounded number of traces, going between start, exit, and call nodes. For the ease of the presentation, in the definition of segment we refer to the start and exit nodes of a procedure as call nodes too.

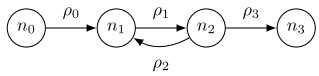
Definition 1 (Segments). *Given two call nodes n and n', the* segment *between n and n' is the set of traces $TR(n, n')$. If n is the start node of a procedure and n' is its exit node, the segment represents the set of all* function traces *of the procedure.*

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

Example 4. The segment . for the program shown in Fig. 3 for n_0 and n_3 is $\tau_0 = \{\rho_0; \rho_1; \rho_3\}$, for n_3 and n_4 is $\tau_1 = \{\rho_4\ ,\ \rho_5\}$ and for n_0 and n_4 is $\tau_2 = \{\rho_0; \rho_1; \rho_3; \rho_4\ ,\ \rho_0; \rho_1; \rho_3; \rho_5\ ,\ \rho_0; \rho_2\ ,\ \rho_6\}$.

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

```
21 function loop(int val){
22   int aux = 0;
23   do { aux += val; }
24   while (aux < 10);
25 }
```



As there are no call nodes, the procedure 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. We can manage these unbounded segments representing loops by considering compact abstract representations of the sets using standard techniques for loops abstraction, or unrolling them a finite number of times. As usual, we might gain precision by the unrolling, but it only ensures the correctness of the analysis up to a number of iterations.

We use sequences of segments (segment-sequences) in order to represent execution traces and prove properties about them; the notation τ is used for segments and π for segment-sequences.

Definition 2 (Segment-sequence). *A* segment-sequence *is a non-empty sequence of segments of the program. 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 $t_i \in \tau_i$.*

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 $\sigma - t - \sigma'$.

Following Example 4, 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 that interleaves its execution.

2.2 (Static) Effective Callback Freedom

The notion of Effectively Callback Free (ECF) programs was introduced by [15] as a way of proving *modularity* in the presence of callbacks. As usual, modularity ensures that external calls to other programs cannot affect the behavior of a program, hence its verification can be done independently. As mentioned earlier, a program is ECF if for every trace with callbacks, there exists a final state-equivalent callback-free trace. ECF ensures that the use of callbacks cannot generate unexpected behaviours as any execution can be simulated without using callbacks.

Definition 3 (dECF). *An execution $\sigma - t - \sigma_1$ is dECF if there is a callback-free execution $\sigma - t^{cf} - \sigma_2$ such that $\sigma_1 = \sigma_2$.*

The static definition of ECF ensures the property for a given program by relying on dynamic ECF (dECF) defined for executions above.

Definition 4 (ECF). *A program Pr is ECF if every execution of Pr is dECF.*

3 Static Analysis of Relaxed-ECF

In this article, we develop a generalization of ECF that is not restricted to checking final-state equivalence but allows considering other (more relaxed) relations and that, besides, can be statically checked. We are interested in preorder relations on states R (i.e., binary relations that are reflexive and transitive), hence instead of requiring that the final states of two executions σ_1 and σ_2 are equal ($\sigma_1 = \sigma_2$ in Definition 3), we can check if they satisfy a more relaxed (weaker) relation $\sigma_1 \sqsupseteq_R \sigma_2$.

Let R be a preorder relation on states for the entirety of this article. We generalize the notions of an ECF execution and ECF program to the (relaxed) relation R.

Definition 5 (R-dECF). *An execution $\sigma - t - \sigma_1$ is R-dECF if there is a callback-free execution $\sigma - t^{cf} - \sigma_1^{cf}$ such that $\sigma_1 \sqsupseteq_R \sigma_1^{cf}$.*

Definition 6 (R-ECF). *A program Pr is R-ECF if every execution of Pr is R-dECF.*

The basis for our static analysis of R-ECF is the following reduction: if there is a violation of the R-ECF property in a trace with arbitrary nested callback calls, then there is one where callbacks are not nested. Traces of executions without nested callbacks are called *simple traces*: the execution of a simple trace can be interrupted at call nodes by an arbitrary sequence of executions of other procedures, however these interrupting procedures themselves never get interrupted.

This reduction holds if we consider the final-state equivalence relation, thus we can check if a program is ECF by only studying the execution of its simple traces.

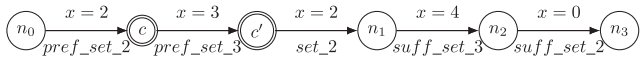
Lemma 1. *If all executions of simple traces of a program Pr are dECF then Pr is ECF.*

However, for a weaker relation, this reduction from violations in traces with nested callbacks to violations in simple traces is unsound as the following example shows:

Example 5. We consider the following program Pr and the reflexive and transitive relation on states $\sigma_1 \sqsupseteq_R \sigma_2 = (\sigma_1[x] \geq \sigma_2[x])$:

```
27 contract No_R-ECF{
28   uint x;
29   function set_2(){
30     x = 2;
31     call();
32     if( x == 4)
33       x = 0;
34   }
35   function set_3(){
36     x = 3;
37     call();
38     if(x != 3)
39       x = 4;
40   }
41 }
```

All executions of simple traces of Pr are R-ECF: the possible final states σ_1 are $[x = 2]$, $[x = 3]$ and $[x = 4]$, thus we can always find a callback-free execution whose final state σ_2 verifies $\sigma_1 \sqsupseteq_R \sigma_2$ (it is enough to consider the execution of set_2 that leads to the state $[x = 2]$). However, the program is not R-ECF, as there are executions with nested callbacks that lead to states σ_1 that do not satisfy $\sigma_1 \sqsupseteq_R \sigma_2$ for any state σ_2 reachable without using callbacks. For example, consider the following execution of a trace with a nested callback to set_2 inside a callback to set_3.



This execution is not R-ECF as the states σ_2 reachable without using callbacks ($[x = 2]$ and $[x = 3]$) do not satisfy $\sigma_1 \sqsupseteq_R \sigma_2$ wrt. the final state of the execution $\sigma_1 = [x = 0]$. Therefore, Pr is not R-ECF although all executions of simple traces of Pr are R-ECF.

The rest of the section is organized as follows. In Section 3.1, we introduce some properties for the relations and the programs that make it possible to develop a R-ECF analysis based on simple traces: our goal is to find conditions for a program Pr such that if they hold for a relation R then we can reduce the analysis of verifying that Pr is R-ECF to verifying that all executions of simple traces of Pr are R-ECF. In Section 3.2, we define the basic operations of the analysis: namely, commutation and projection operations on program segments. Finally, Section 3.3 outlines the analysis algorithm used to prove R-ECF and hence ensure callback safety, and Sections 3.4 and 3.5 introduce generalizations of the algorithm for handling procedures with multiple call nodes with less and more accuracy.

3.1 R-Monotonicity

The main condition that will allow us to soundly use a relation R in our analysis of simple traces is *monotonicity*. Monotonicity relates between a relation R and a segment of the analyzed program.

Definition 7 (R-monotonocity). *Given a segment τ, we say that τ is R-monotone if for all states σ_1, σ_1' and σ_2, such that $\sigma_1 \sqsupseteq_R \sigma_1'$ and $\sigma_1 - \tau - \sigma_2$, there exists a state σ_2' such that $\sigma_1' - \tau - \sigma_2'$ and $\sigma_2 \sqsupseteq_R \sigma_2'$.*

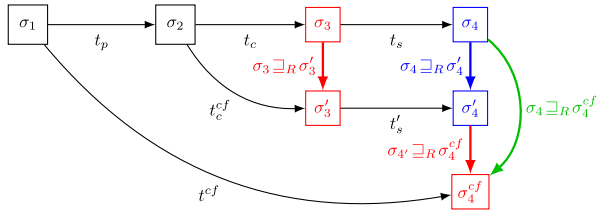


Fig. 4. Scheme of the relations between the states of the different executions that we consider during the reasoning.

First, we show the intuition behind how R-monotonicity enables the reduction to simple traces. We consider a program Pr such that all executions of simple traces of Pr are R-ECF. Let $t = t_p; t_c; t_s$ be a trace of Pr where t_c is a simple trace (that is executed as a callback), and t_p and t_s are traces that belong to the "prefix" and the "suffix" segments of a procedure f, respectively. We consider the following execution: $\sigma_1 - t_p - \sigma_2 - t_c - \sigma_3 - t_s - \sigma_4$. Fig. 4 illustrates this execution and the relations satisfied by the states of the different executions that we are going to consider during the reasoning. First, we observe that the execution $\sigma_2 - t_c - \sigma_3$ is R-ECF as t_c is a simple trace. Hence, there is a callback-free trace t^{cf}_c such that $\sigma_2 - t^{cf}_c - \sigma'_3$ and $\sigma_3 \sqsupseteq_R \sigma'_3$. However, this does not imply that the execution of t_s from the state σ'_3 leads to a state σ'_4 that satisfies $\sigma_4 \sqsupseteq_R \sigma'_4$, as we observed in Example 5. But, *if we require the segment representing the suffix τ_s to be R-monotone*, then we can ensure that there is a trace $t'_s \in \tau_s$ such that $\sigma'_3 - t'_s - \sigma'_4$ and $\sigma_4 \sqsupseteq_R \sigma'_4$. Finally, we consider the execution: $\sigma_1 - t_p - \sigma_2 - t^{cf}_c - \sigma'_3 - t'_s - \sigma'_4$. The executed trace is simple, thus the execution is R-ECF. Hence, there is a callback-free trace t^{cf} such that $\sigma_1 - t^{cf} - \sigma^{cf}_4$ and $\sigma'_4 \sqsupseteq_R \sigma^{cf}_4$. The relation R is reflexive and transitive, thus $\sigma_4 \sqsupseteq_R \sigma^{cf}_4$ is also satisfied.

In our analysis of a program Pr and a desired relation R, we require that:

- All procedures of the program are R-monotone.
- All suffixes of call nodes of the program are R-monotone.

We define R-monotonicity for procedures and call nodes:

Definition 8. *A procedure f of a program Pr is R-monotone if and only if the segment that represents the procedure (the one that contains all traces of the procedure) is R-monotone.*

Definition 9. *A call node c of a program Pr is R-monotone if and only if all its right segments (segments $TR(c, c')$ that go from the call node c to the exit node or to another reachable call node of the procedure) are R-monotone.*

Finally, we extend this notion to the entire program:

Definition 10. *A program Pr is R-monotone if and only if all its procedures and call nodes are R-monotone.*

If a program Pr is R-monotone, our following result ensures that in order to show that Pr is R-ECF, it is sufficient to check that all executions of simple traces of Pr are R-ECF.

Lemma 2. *Let Pr be a R-monotone program. If all executions of simple traces of Pr are R-dECF then Pr is R-ECF.*

Example 6. The program in Fig. 2 is R-monotone with respect to the relation $\sqsupseteq_R$ introduced in Example 2. Hence, it is only necessary to analyze the executions of simple traces of the program to prove that it is R-ECF.

3.2 Commutativity/Projection Checks

The static analysis for checking R-ECF is based on commutativity/projection operations between segments that can be realized using SMT solvers. The intuition is the following: the analysis uses commutation and projection checks on program segments to remove the callbacks and construct an equivalent callback-free execution, but it considers a generalization of these operations that is not restricted to checking final-state equivalence.

Definition 11 (R-Commutation, and left and right R-projection). *Given two segments τ_1 and τ_2, and a preorder relation on states R,*

- *We say that τ_1 R-*commutes *with τ_2 for the state $\sigma \in Feasible(\tau_1; \tau_2)$ if and only if, $\sigma \in Feasible(\tau_2; \tau_1)$ and, if $\sigma - \tau_1; \tau_2 - \sigma'$ and $\sigma - \tau_2; \tau_1 - \sigma''$, then $\sigma' \sqsupseteq_R \sigma''$.*
- *We say that τ_1 R-*left-projects *with τ_2 for the state $\sigma \in Feasible(\tau_1; \tau_2)$ if and only if, if $\sigma - \tau_1; \tau_2 - \sigma'$ and $\sigma - \tau_1 - \sigma''$, then $\sigma' \sqsupseteq_R \sigma''$.*
- *We say that τ_1 R-*right-projects *with τ_2 for the state $\sigma \in Feasible(\tau_1; \tau_2)$ if and only if, $\sigma \in Feasible(\tau_2)$ and, if $\sigma - \tau_1; \tau_2 - \sigma'$ and $\sigma - \tau_2 - \sigma''$, then $\sigma' \sqsupseteq_R \sigma''$.*

Basically, the above definition generalizes the well-known commutation and projection operations that check final-state equivalence to satisfy a reflexive and transitive relation R. The standard operations are a particular case of these R-operations using the equivalence relation $\sigma_1 \sqsupseteq_{R_=} \sigma_2 \Leftrightarrow \sigma_1 = \sigma_2$. Indeed, $R_=$ is the strongest relation that we can consider: in case two segments $R_=$-commute for a state σ, then they R-commute for any reflexive relation.

Example 7. Consider the program in Fig. 2. The segments τ_d, τ_w represent the procedures deposit and withdraw, respectively. The segment τ_0, defined in Example 4, represents the traces of withdraw until its call node. We study the R-commutation and R-projection of these segments according to different relations. First, we consider the relation $R_=$. If the procedure deposit includes the require instruction, then both $\tau_0; \tau_d$ and $\tau_0; \tau_w$ $R_=$-left-project for any feasible state: their execution leads to the same final state as executing τ_0 alone, since τ_0 sets lock to true and in such state both deposit and withdraw revert. But if deposit does not include the require, then $\tau_0; \tau_d$ does not $R_=$-commute or $R_=$-left-project, e.g., the execution of $\tau_0; \tau_d$ from the initial state $\sigma_0 = [shares = [msg.sender \rightarrow 10], balance = 10, msg.value = 5]$ leads to $\sigma_1 = [shares = [msg.sender \rightarrow 15], balance = 5]$, but $\tau_d; \tau_0$ and τ_0 lead to different states ($\sigma_2 = [shares = [msg.sender \rightarrow 15], balance = 0]$ and $\sigma_3 = [shares = [msg.sender \rightarrow 10], balance = 0]$, respectively). However, if we consider the relation R introduced in Example 2, then τ_d R-commutes with τ_0 for any state in $Feasible(\tau_0; \tau_d)$. For example, if we consider again the initial state σ_0, the states σ_1, σ_2 satisfy $\sigma_1 \sqsupseteq_R \sigma_2$ thus the segments R-commute.

The analysis works by applying left and right movements of callbacks constructing a callback free execution, where callbacks no longer appear at call nodes, and which

lead to a "smaller" state compared to the original one. For this purpose, we define the notion of movement for a relation R, referred to as *R-movement*, as a combination of R-commutativity and R-projection properties. In particular, an *R-Left-movement* expresses that for all feasible states we can either R-commute or R-left-project, *R-right-movement* expresses that we can either R-commute or R-right-project.

Example 8. Consider the segments τ_d, τ_w, τ_0 and τ_1 defined in previous examples. If we take the relation $R_=$, then $\tau_0;\tau_w$ $R_=$-left-moves as τ_0 $R_=$-left-projects with τ_w for all feasible states, but $\tau_0;\tau_d$ only $R_=$-left-moves if the procedure deposit includes the require instruction. However, if we consider the relation R introduced in Example 2, then both procedures deposit and withdraw R-left-move with τ_0, as $\tau_0;\tau_d$ R-commutes and $\tau_0;\tau_w$ R-left-projects for any feasible state for executing these segments.

Given an execution that leads to a state σ, the basic idea of the analysis is to apply R-movements to construct a callback-free execution whose final state σ' satisfies the relation $\sigma \sqsupseteq_R \sigma'$. Following Example 8, consider the execution of the sequence $\tau_0;\tau_d;\tau_1$ from an initial state σ_0, we denote σ_F to its final state and σ_I to the intermediate state that we reach after executing the callback (i.e., $\sigma_0 - \tau_0;\tau_d - \sigma_I$ and $\sigma_I - \tau_1 - \sigma_F$). It is clear that $\tau_0;\tau_d$ R-commutes for the state σ_0, thus the execution from this state of the trace $\tau_d;\tau_0$ leads to a state σ'_I such that $\sigma_I \sqsupseteq_R \sigma'_I$. The segment τ_1 is R-monotone, hence we have succeeded in constructing a callback-free sequence $\tau_d;\tau_0;\tau_1$ whose execution from σ_0 leads to a state $\sigma_F^{cf} = [shares = [0], balance = 0]$ such that $\sigma_F \sqsupseteq_R \sigma_F^{cf}$ is satisfied, as the segment τ_1 preserves the relation R.

3.3 Outline of the Analysis Algorithm

We introduce a constructive algorithm for reordering (by relying on the commutativity and projection operations of Section 3.2) simple callback traces into R-equivalent callback-free traces such that R-ECF is ensured. The intuition of the algorithm is the following: given a procedure f with a single call node n, it partitions f into two segments: prefix and suffix. The segment prefix represents the set of traces from the start node to the call node n, and the segment suffix the traces from the call node to the end node (the notion of segments is introduced in Definition 1). Then, it considers sequences T of the form prefix ; A ; suffix where $A \in \mathsf{F}^*$. F is the set of all procedures of the program being analyzed. The set F^* represents all possible sequences of unbounded length consisting of procedure calls of our program. We assume that the execution of T from a initial state σ_0 leads to the state σ_1. Our goal is to find subsequences G, H of procedures calls in A such that execution of the callback-free sequence G ; prefix ; suffix ; H from the state σ_0 leads to a state σ'_1 that satisfies the relation $\sigma_1 \sqsupseteq_R \sigma'_1$. A pseudocode of the algorithm for checking this constructive R-ECF definition is given in Fig. 1. It corresponds to a restricted version of the general algorithm presented later that assumes that functions contain at most one call node. We will generalize to any number of call nodes in Sections 3.4 and 3.5. The algorithm receives as parameters the relation R to be used and the procedure $f \in \mathsf{F}$ to be checked (that has a single call node n). It operates by extracting the segments prefix (line 3) and suffix (line 4) that represent the parts of the code of f before and after the call node n, respectively. The algorithm then computes the set of Left (line 5) and Right movers (line 6). From the sets of left and right movers we can construct the subsequences G, H mentioned above. The computation of movers is encapsulated within the functions get_R-left_movers and get_R-right_movers that use the notion of R-movements introduced in the previous section. A procedure g belongs to Left if it R-left-moves with the segment prefix, and it belongs to Right if it R-right-moves with the segment suffix.

Algorithm 1. Pseudocode of an Algorithm for Checking a Procedure With a Single Call Node, That Allows Bidirectional Movement of Callbacks

```
1:  procedure CHECK_R-ECF_SINGLE_CALLNODE(R, f)
2:    let n = get_callnode(f)
3:    let prefix = extract_prefix(f, n)
4:    let suffix = extract_suffix(f, n)
5:    let Left = get_R-left_movers(R, prefix)
6:    let Right = get_R-right_movers(R, suffix)
7:    returncheck_no_move_collisions(R, Left, Right)
8:
9:  procedure CHECK_NO_MOVE_COLLISIONS(R, Left, Right)
10:   let MLeft = check_Right_collisions(R, Right)
11:   let MRight = check_Left_collisions(R, Left)
12:   if (MLeft ∩ MRight) == ∅ then
13:     return R-ECF
14:   else
15:     returnMayNotBe_R-ECF
16:
17: procedure CHECK_RIGHT_COLLISIONS(R, Right)
18:   let MLeft = F \ Right
19:   while MLeft changes do
20:     MLeft = MLeft ∪ get_not_R-right_movers(R, MLeft)
21:   returnMLeft
22:
23: procedure CHECK_LEFT_COLLISIONS(R, Left)
24:   let MRight = F \ Left
25:   while MRight changes do
26:     MRight = MRight ∪ get_not_R-left_movers(R, MRight)
27:   returnMRight
```

In case all procedures are in Left or all are in Right , the program is trivially R-ECF as we can always move all callbacks to the left or to the right respectively, but this does not cover all possible legal movements, i.e., the algorithm 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 callback (referred to as "collision"), we can combine left and right movements. To illustrate the problem of collisions, we consider the program in Example 9 (presented below) and the final-state equivalence relation $R_=$. The procedure g1 must move to the right of the call node, the procedure g2 must move to the left , and g1 cannot move to the right of g2 . Then, for a sequence of callbacks g1; g2 we cannot prove the existence of an equivalent execution using the reordering technique. The function $\mathsf{check_n o_m ove_c ollisions}$ generalizes this check for any potential sequence of callbacks by computing sets MLeft (line 10) and MRight (line 11) that represent the set

of callbacks that cannot move to the right and to the left, respectively. These sets are updated iteratively until a fixed point is reached, starting from the F \ Right and F \ Left sets, and updated in each round using the functions get_not_R-right_movers (line 20) and get_not_R-left_movers (line 26).

Example 9. Consider the following program and the relation $R_{=}$.

```
43 contract coll{
44  uint x, y, z;
45  function f(){
46   x = x * 2;
47   call();
48   y = y * 2;
49  }
50  function g1() {
51  {
52    x = x + 1;
53    z = 0;
54  }
55  function g2()
56  {
57   y = y + 1;
58   z = 1;
59   }
60  }
```

The procedure g1 $R_{=}$-commutes with the suffix (but not with the prefix) of f, and g2 $R_{=}$-commutes with the prefix (but not with the suffix) of f. Then, it is clear that g1 $\in$ F \ Left and g2 $\in$ F \ Right, hence we need to consider the possible collisions between these procedures. The sequence g1; g2 does not $R_{=}$-move, thus the algorithm cannot verify the program.

Importantly, the correctness of the algorithm is only ensured if the program is R-monotone: if a R-monotone program is verified by the algorithm then all executions of simple traces of the program are R-ECF.

Theorem 1. *Given a reflexive and transitive relation R, if the function* check_R-ECF_single_callnode *outlined in Fig. 1 returns R-ECF for all the procedures f of a R-monotone program Pr, then Pr is R-ECF.*

Example 10. Consider the program Pr of Fig. 2. This program contains a single call node in the procedure withdraw, and the prefix and suffix of this procedure are the segments τ_0 and τ_1 defined in Example 4, respectively. First, we consider the relation on states $R_{=}$. If the procedure deposit does not include the require instruction then the procedure deposit belongs to neither Left nor Right, as we discussed in Example 7. Thus, the algorithm cannot verify the program. However, if we consider the relation R introduced in Example 2 then deposit and withdraw R-left-move with the prefix, therefore both of them belong to Left. Hence, the set MRight is empty and the algorithm verifies the program. Pr is R-monotone as we discussed in Example 6, hence it is R-ECF.

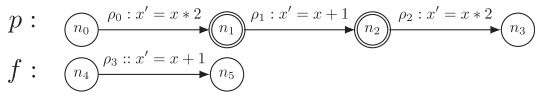
The problem of checking if two segments commute or project is undecidable if the segments can contain loops. However, we can express the complexity of the algorithm outlined in Fig. 1 in terms of these checks. The worst-case occurs when not all functions belong to Left or Right , and the algorithm needs to check the possible collisions between the left and right-movers. This may involve checking the commutation/projection of every pair of functions of the program, therefore the number of commutation checks performed by the algorithm is quadratic with respect to the number of functions.

3.4 Generalization of the Algorithm for Multiple Call Nodes

The case of programs with procedures with several call nodes is more challenging. The first difficulty is that in order to study the possible reorderings of an execution it is not enough to consider the complete prefix and suffix of each one these call nodes (i.e., the segment from the start node to the call node and the one from the call node to the end node of the procedure, respectively) as we need to consider any possible execution that might callback in the intermediate call nodes. For example, the procedure p introduced in the example below (Example 11) contains two consecutive call nodes n_1 and n_2, so it is not sound to consider the segment $\tau_s = \{\rho_1; \rho_2\}$ to check if we can move a callback in the first call node to the end of the execution as the possible callbacks in the node n_2 could *block* this movement, i.e., a procedure g could R-commute with $\rho_1; \rho_2$ but not with $\rho_1; h; \rho_2$ (with h a callback in the second call node). This is the reason why 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 of the segment. For example, given the previous procedure we consider three different segments: one from the start node to n_1, one between the call nodes n_1 and n_2, and finally one from n_2 to the end node.

Second, the presence of multiple call nodes is challenging as we cannot consider the movements of each one of the call nodes in isolation because different call nodes can generate cycles when we try to reorder their callbacks.

Example 11. The following example illustrates how different call nodes can generate cycles when we try to reorder their callbacks. The procedure p contains two consecutive call nodes (n_1 and n_2), and we consider the movements of the procedure f.



If we consider the relation $R_{=}$, then f is only in the set of right movers of the node n_1 as it only $R_{=}$-commutes with its minimal right segment, and it is only in the set of left movers of the node n_2 as it only $R_{=}$-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 forbid cycles by ensuring that if we move a procedure in one direction then it is not going to get blocked in an intermediate call node during the reordering (i.e., once we start moving a procedure to the right (resp. left) then it is not going to reach a call node where it cannot move to the right (resp. left)). Hence, we can only ensure R-ECF if we impose that for any pair of call nodes c_1 and c_2 such that c_2 is reachable from c_1, if a procedure needs to move to the right of c_1 then it can move to the right of c_2, and if it needs to move to the left of c_2 it can move to the left of c_1.

A pseudocode of the algorithm for checking R-ECF in procedures with multiple call nodes is given in Fig. 2. This algorithm can handle programs with any number of call nodes, and follows the same constructive approach that we introduced in the previous section.

The function check_callnode returns the set of procedures that cannot move to the left and right of a given call node n. It follows the intuition of the algorithm for checking procedures with a single call node outlined in Fig. 1: first it extracts the minimal right and left segments of the call node (lines 2 and 3) and computes the procedures that R-left-move with the left segment (Left) and the ones that R-right-move with the right segment (Right). Finally, it calculates the sets of procedures that cannot move to the left (MRight) and to the right (MLeft) taking into account the possible collisions between callbacks.

Finally, the function check_R-ECF_multiple_callnodes checks that there are not cycles like the one in Example 11. For every pair of call nodes n_1 and n_2 such that n_2 is reachable from n_1 (lines 14 and 15), it verifies that the procedures can either move to the right or to the left of the call nodes by checking that there are not procedures that cannot move to the left of n_1 and to the right of n_2 at the same time (line 16). In case this function returns R-ECF then it is always possible to reorder the callbacks of a given execution to generate a callback-free execution whose final state satisfies the relation R with respect to the final state of the original execution. As in the previous section, the correctness of the algorithm is only ensured if the program is R-monotone.

Theorem 2. *Given a reflexive and transitive relation R, if the function* check_R-ECF_multiple_callnodes *outlined in Fig. 2 returns R-ECF for all the procedures f of a R-monotone program Pr, then Pr is R-ECF.*

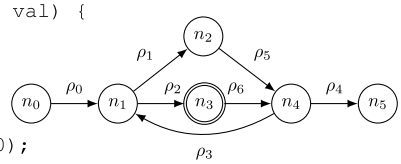
Algorithm 2. Pseudocode of an Algorithm for Checking a Procedure With Multiple Call Nodes, That Allows Bidirectional Movement of Callbacks

```
1:  procedure CHECK_CALLNODE(R, n, f)
2:    let left_segments = extract_minimal_left_segments(f, n)
3:    let right_segments = extract_minimal_right_segments(f, n)
4:    let Left = get_R-left_movers(R, left_segments)
5:    let Right = get_R-right_movers(R, right_segments)
6:    let MRight = check_Left_collisions(R, Left)
7:    let MLeft = check_Right_collisions(R, Right)
8:    returnMRight, MLeft
9:
10: procedure CHECK_R-ECF_MULTIPLE_CALLNODES(R, f)
11:   let callnodes = get_callnodes(f)
12:   for n in callnodes do
13:     let MRight[n], MLeft[n] = check_callnode(R, n, f)
14:   for n_1, n_2 in callnodes do
15:     if n_1 == n_2 or n_2 is reachable from n_1 then
16:       if MLeft[n_2] ∩ MRight[n_1] != ∅ then
17:         returnMayNotBe_R-ECF
18:   returnR-ECF
```

We can use this approach based on considering minimal segments to analyze programs that contain loops, even when they include call nodes inside the loops. In the following example we show how the definition of minimal segments applies to these programs.

Example 12. Consider the procedure loop1 that contains the call node n_3 in the loop. The left segments of this call node represent traces from a call node (the initial n_0 or n_3) to n_3 and its right segments from n_3 to a call node (n_5 or n_3).

```
61 function loop1(int val) {
62   int aux = 0;
63   do {
64     if (val != 0)
65       aux += val;
66     else
67       aux = call();
68   } while (aux < 10);
69 }
```



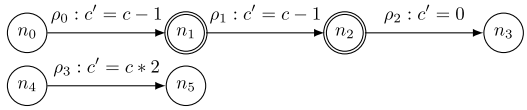
We first consider the segment that goes from n_0 to n_3: it contains all the traces between these two nodes that do not include any other call node apart from themselves. There might be an unbounded number of such traces since we can take the path $\rho_1;\rho_5;\rho_3$ as many times as we like before 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 $t = \rho_1;\rho_5;\rho_3$,

$$\begin{aligned} SLeft &= \{\{\rho_0;\rho_2\ ,\ \rho_0;t;\rho_2\ ,\ \rho_0;t;t;\rho_2\ ,\ \ldots\},\\ &\quad\ \{\rho_6;\rho_3;\rho_2,\ \rho_6;\rho_3;t;\rho_2\ ,\ \ldots\}\}\\ SRight &= \{\{\rho_6;\rho_4\ ,\ \rho_6;\rho_3;\rho_1;\rho_5;\rho_4\ ,\ \ldots\},\\ &\quad\ \{\rho_6;\rho_3;\rho_2\ ,\ \rho_6;\rho_3;t;\rho_2\ ,\ \ldots\}\} \end{aligned}$$

3.5 Segments Join

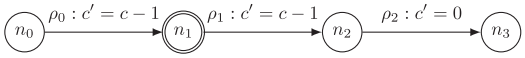
The technique we have considered in the previous section is powerful, but it can be more accurate if, once we have solved a call node (i.e., removed all the callbacks that entered into it), we allow "joining" its left and right segments. For instance, consider the procedure p defined in Example 11. If we prove that all procedures are moving with respect to call node n_2 (via commutation or projection with ρ_1 and ρ_2), then we have proven that no callback interrupts the two segments ρ_1, ρ_2 thus we can elide them into a single segment. The reason for performing the join operation is that having larger segments leads to strictly more accurate results. The following example shows a case where we gain accuracy by joining segments:

```
70 contract Example_no_ECF {
71   uint c;
72   function discount2(){
73     c = c - 1;
74     call();
75     c = c - 1;
76     call();
77     c = 0;
78   }
79   function mult(){
80     c = c * 2;
81   }
82 }
```



Example 13. Consider the program above whose procedure discount2 has three transitions and two call nodes while the procedure *multiply* has a single transition and no call nodes, and the final-state equivalence relation $R_{=}$. 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 mul for the

function segment $\{\rho_3\}$ of multiply. Now, the segment-sequence representing our trace is $\tau_0; mul; \tau_1; mul; \tau_2$. We start by handling the second call node, n_2, first. We $R_=$-right-project $mul; \tau_2$ to τ_2 and obtain the new segment-sequence (that is final-state equivalent to our original trace) $\tau_0; mul; \tau_1; \tau_2$. But now we cannot go further and solve n_1 since we cannot apply any $R_=$-movement on $\tau_0; mul$ or $mul; \tau_1$. 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 we have removed all the callbacks that entered into it, then our CFG would be:



Hence, the right segment of n_1 is the segment $\tau_{1;2} = \{\rho_1; \rho_2\}$, which is the join of segments τ_1 and τ_2, thus the sequence we have to consider now is $\tau_0; mul; \tau_{1;2}$. Then, we can $R_=$-right-project $mul; \tau_{1;2}$ to $\tau_{1;2}$, and obtain the sequence $\tau_0; \tau_{1;2}$.

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. For example, if we consider again the procedure discount2 and try to solve the call node n_1 first, we cannot verify that the program is $R_=$-ECF as the procedure multiply does not commute or project with the left and right segment of the call node (τ_0 and τ_1 resp.).

Algorithm 3. Pseudocode of an Algorithm for Checking a Single Call Node Following a Given Order O

```
1:  procedure CHECK_R-ECF_CALLNODE(R, n, f, C)
2:    let left_segments = extract_left_segments_order(f, n, C)
3:    let right_segments = extract_right_segments_order(f, n,
    C)
4:    let Left = get_R-left_movers(R, left_segments)
5:    let Right = get_R-right_movers(R, right_segments)
6:    let MRight = check_Left_collisions(R, Left)
7:    let MLeft = check_Right_collisions(R, Right)
8:    returnMRight ∩ MLeft == ∅
9:
10: procedure check_ECF_ordered(R, f, O)
11:   let remaining_c = get_callnodes(f)
12:   for smallest c in remaining_c according to O do
13:             let is_solved = check_R-ECF_callnode(R,c,f,
    remaining_c)
14:     if not is_solved then
15:       returnMayNotBe_R-ECF
16:     let remaining_c = remaining_c \ { c }
17:   returnR-ECF
```

We can verify that a program is R-ECF if we can find an order for solving all its call nodes. The difference with the previous algorithms is that we check the call nodes one by one following the order O, and join their left and right segments once we prove them (e.g., we need to follow the order $O = n_2 < n_1$ to prove that discount2 is safe as we need to join the left and right segments of the node n_2 before checking n_1). In this case it is *not* necessary to add conditions to ensure that we are not generating cycles. Once we remove the solved call nodes, it is not possible to generate cycles.

The function check_R-ECF_callnode checks if a call node n of a procedure f is R-ECF considering that the nodes in C are the only call nodes of the procedure. Hence, extract_left_segments and extract_right_segments (lines 2 and 3) compute the minimal segments of the node n only considering the nodes in C as call nodes (i.e., segments go from a node of C to n and do not traverse any other node in C), and then the procedure checks if the call node is solvable by computing the sets of left and right movers (MLeft and MRight).

Given a total ordering of the call nodes $O = n_{i_1} < n_{i_2} < \ldots < n_{i_k}$, the function check_R-ECF_ordered applies this check to all the call nodes of a procedure f by following the order O (i.e., it first proves the smallest and so on). Initially, it considers that we have not removed any call node, so the set of remaining call nodes remaining_c contains all the call nodes of the procedure (line 11). Then, the function considers the call nodes in remaining_c one by one following the order O and studies if they are R-ECF (line 13). Once the algorithm checks that a node c is R-ECF it considers that it is no longer a call node as we can move all its callbacks to other call nodes (via commutation or projection), thus it removes the node from the set of remaining call nodes (line 16).

For example, we consider the procedure discount2 of Example 13, and the order $O = n_2 < n_1$. The right segment of the node n_1 according to this order is the segment $\tau_{1;2} = \{\rho_1; \rho_2\}$ as the set of remaining call nodes when we study n_1 is $\{n_1\}$, but if we consider the order $O = n_1 < n_2$ then the right segment of the node is $\tau_1 = \{\rho_1\}$ as the set of remaining call nodes is $\{n_1, n_2\}$.

Theorem 3. *Given a reflexive and transitive relation R, if there exists an order O such that the function* check_R-ECF_ordered *outlined in Fig. 3 returns R-ECF for all the procedures f of a R-monotone program Pr, then Pr is R-ECF.*

Example 14. Consider the procedure discount2 of Example 13, if we take O as $n_2 < n_1$ we have that the only right segment of n_2 is $\tau_2 = \{\rho_2\}$, and the only right segment of n_1 is $\tau_{1;2} = \{\rho_1; \rho_2\}$. Thus, both discount2 and multiply belong to Right of n_2 and n_1 resp. as they $R_=$-move with their right segments, hence the program is $R_=$-ECF.

This approach is strictly more accurate than Fig. 2. In fact, if we can verify a program using Fig. 2, then we can verify it using Fig. 3 choosing any order for solving the call nodes.

A possibility for checking if a program is R-ECF is to try all orderings and find if one of them is able to verify the program. In practice, trying all possible call node orders, given that there are procedures that have over 10 call nodes, may be too expensive due to the number of required SMT queries. Our implementation follows a predetermined call node ordering: going linearly from latest (in program-order) call nodes to earlier call nodes, as it 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, this approach would lead to faster proofs of R-ECF.

Finally, let us mention that we can apply the notion of *callback invariants* introduced by [1] to improve the accuracy of the three algorithms outlined in this section. As a

standard invariant, a callback invariant of a call node c 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. Thus, we can exploit the information given by callback invariants to improve the precision of the commutation and projection checks of our algorithms (e.g., if $I = x \geq 0$ is a callback invariant of the call node c, then it is enough to consider the states that satisfy I to check if a procedure R-right-moves with the suffix of c).

4 Property-Guided Verification Using R-ECF

We introduce in this section an important application of R-ECF for proving invariant properties that ensure safety in the presence of for callbacks. The problem is challenging as it is not enough to verify that the properties are preserved by callback-free executions, but we need to consider all possible sequences of callbacks as they can generate unexpected behaviors.

Example 15. Let us consider the (simplified) attacked DAO, that is like the program in Fig. 2 without including the require instructions of the procedures deposit and withdraw (line 7 and 13 resp.). The use of callbacks introduces unexpected behaviors that are actually malicious as they do not preserve the solvency property established in Section 1.1 that ensures that the program always has enough balance to give back to its clients the shares that they have in their accounts:

$$I(\sigma) = \sigma[balance \geq \sum shares]$$

The property I is preserved by the procedures deposit and withdraw, so it is preserved by any execution of the program that does not contain callbacks. However, there are executions with callbacks that do not preserve the property. For example, the execution in which we invoke withdraw from the initial state $\sigma = \{balance = 20, shares = [10, 10]\}$ and then callback to withdraw again leads to the final state $\sigma_1 = \{balance = 0, shares = [0, 10]\}$, and $I(\sigma_1)$ does not hold.

Before introducing our approach, we define the notions of program invariant and callback-free program invariant to distinguish the properties that are preserved by any execution and by any callback-free execution, respectively.

Definition 12 (Program invariant). *Given a program Pr and a property on states I, the property I is a program invariant of Pr if for any execution of the program $\sigma - t - \sigma_1$, the states σ and σ_1 satisfy $I(\sigma) \Rightarrow I(\sigma_1)$.*

Definition 13 (Callback-free program invariant). *Given a program Pr and a property on states I, the property I is a callback-free program invariant of Pr if for any callback-free execution of the program $\sigma - t - \sigma_1$, the states σ and σ_1 satisfy $I(\sigma) \Rightarrow I(\sigma_1)$.*

It is clear that any program invariant is a callback-free program invariant, but the reverse does not hold. For instance, the property $I(\sigma) = \sigma[balance \geq \sum shares]$ is a callback-free invariant of the attacked DAO, but it is not a program invariant. However, if a program is ECF, then any execution of the program is final-state equivalent to a callback-free execution. Thus, properties preserved by callback-free executions are also preserved by executions with callbacks.

Lemma 3. *Given a program Pr and a property on states I, if Pr is ECF and I is a callback-free program invariant of Pr, then I is a program invariant of Pr.*

We generalize this result for R-ECF in the next theorem that constitutes the main result of this section. R-ECF is a weaker property than ECF, hence we need to define the requirements that a relation has to fulfill to be suitable to verify a property I.

Theorem 4. *Given a property on states I and a preorder relation on states R such that for all states σ and σ', we have $(\sigma \sqsupseteq_R \sigma' \wedge I(\sigma')) \Rightarrow I(\sigma)$. Then, for any program Pr, if I is a callback-free program invariant of the program and Pr is R-ECF, I is a program invariant of Pr.*

It is not necessary to add the R-monotonicity requirement in this theorem as the result is based on the general definition of R-ECF (Definition 6) that does not require the R-motononicity of the program. In case a program is R-ECF wrt. a non-monotone relation, the result in Theorem 4 holds although our algorithms are not able to verify that the program is R-ECF.

4.1 Guiding the Choice of Relations From Invariants

We introduce now results that enable the automatic synthesis of relations that satisfy the requirements introduced in Theorem 4 with respect to a given property on states.

The first result shows that we can generate appropriate relations satisfying the requirements for any property that can be expressed via an inequality.

Proposition 1. *If I is a property that can be expressed using an inequality $\sigma[expr] \geq 0$ then the relation $\sigma \sqsupseteq_R \sigma' = \sigma[expr] \geq \sigma'[expr]$ is a preorder and satisfies: $(\sigma \sqsupseteq_R \sigma' \wedge I(\sigma')) \Rightarrow I(\sigma)$.*

The following proposition handles the case of properties that can be expressed as the conjunction of other properties, and proposes a candidate relation for satisfying the requirements.

Proposition 2. *If I is a property such that $I = I_1 \wedge \ldots \wedge I_n$ with $R_1, \ldots, R_n$ being preorder relations such that $\sigma \sqsupseteq_{R_i} \sigma' \wedge I_i(\sigma') \Rightarrow I_i(\sigma)$ then the relation $\sigma \sqsupseteq_R \sigma' = \sigma \sqsupseteq_{R_1} \sigma' \wedge \ldots \wedge \sigma \sqsupseteq_{R_n} \sigma'$ is a preorder and satisfies: $(\sigma \sqsupseteq_R \sigma' \wedge I(\sigma')) \Rightarrow I(\sigma)$.*

Finally, the proposition below proves that given any property I, we can always consider the relation $\sigma \sqsupseteq_R \sigma' = (I(\sigma') \Rightarrow I(\sigma))$ as it verifies the requirements introduced in Theorem 4.

Proposition 3. *Given a property on states I, the relation $\sigma \sqsupseteq_R \sigma' = (I(\sigma') \Rightarrow I(\sigma))$ is a preorder and satisfies: $(\sigma \sqsupseteq_R \sigma' \wedge I(\sigma')) \Rightarrow I(\sigma)$.*

Intuitively, given a program Pr and property I, Pr is not always R-monotone with respect to the relations that we can define using the previous propositions, even when I is a program invariant of Pr. In fact, it may be R-monotone for

some of the relations but not for others, as the following example shows.

Example 16 (Non-monotone relations). We consider the following program Pr and the property $I = (x \geq 10)$ that is a callback-free program invariant of Pr, and study the R-monotonicity of the program with respect to different relations that we can generate using the results of this section:

```
83 contract not_monotone{
84   int x;
85   function f(){
86     x = 5;
87     call();
88     if(x >= 0)
89       x = 10;
90   }

91   function g(){
92     if(x == 5)
93       x = 0;
94   }
95 }
```

1) We consider the relation $\sigma \sqsupseteq_{R_1} \sigma' = \sigma[x] \geq \sigma'[x]$. This relation is a preorder, but Pr is not R_1-monotone as the procedure g is not R_1-monotone.
2) We consider the relation $\sigma \sqsupseteq_{R_2} \sigma' = (I(\sigma') \Rightarrow I(\sigma))$. In this case, the procedures f and g are R_2-monotone, but the suffix of the procedure f is not, thus Pr is not R_2-monotone.
3) We consider the relation R_3: $\sigma \sqsupseteq_{R_3} \sigma' \Leftrightarrow (I(\sigma') \Rightarrow I(\sigma)) \wedge (\sigma'[x] \geq 0 \Rightarrow \sigma[x] \geq 0)$. Pr is R_3-monotone, e.g., the suffix of the procedure f is R_3-monotone as for any two states σ_0 and σ'_0 such that $\sigma_0 \sqsupseteq_{R_3} \sigma'_0$ holds, $\sigma'_0[x] \geq 0 \Rightarrow \sigma_0[x] \geq 0$. Thus, in case $\sigma'_0[x] \geq 0$ executing the suffix from the states σ_0 and σ'_0 leads to the final states σ_1 and σ'_1 resp. that satisfy $\sigma_1[x] \geq 10$ and $\sigma'_1[x] \geq 10$, so $\sigma_1 \sqsupseteq_{R_3} \sigma'_1$ holds. In case $\sigma'_0[x] < 0$, the suffix does not modify the state ($\sigma'_1 = \sigma'_0$), hence $\sigma_1 \sqsupseteq_{R_3} \sigma'_1$ trivially holds.

The overall approach thus works by first using one of the algorithms outlined in Section 3 to verify that a program Pr is R-ECF, and then using the relation R to prove that the property I is a program invariant of Pr. For the above example, the first algorithm is enough to verify that the program Pr is R_3-ECF, as the procedure f contains a single call node and both f and g R_3-right-move with its suffix. Finally, I is a callback-free program invariant of Pr, R_3 is a preorder relation, and they satisfy that $(\sigma \sqsupseteq_{R_3} \sigma' \wedge I(\sigma')) \Rightarrow I(\sigma)$. Thus, I is a program invariant of Pr.

Example 17. This example shows how we can apply our approach to verify that the property $\sigma[balance \geq \sum shares]$ is a program invariant of the code in Fig. 2, (this property was not preserved by the vulnerable code and cannot be proven by [1]). We consider the program Pr, the relation R introduced in Example 2, and the property on states $I(\sigma) = \sigma[balance \geq \sum shares]$. For any two states σ, σ', if $\sigma \sqsupseteq_R \sigma'$ and $I(\sigma')$ are satisfied, then $\sigma[balance - \sum shares] \geq \sigma'[balance - \sum shares] \geq 0$. Thus, $I(\sigma)$ is satisfied. The program Pr is R-ECF and I is a callback-free program invariant of Pr, hence I is a program invariant of Pr.

Finally, note that the relation $\sigma \sqsupseteq_R \sigma' = (I(\sigma') \Rightarrow I(\sigma))$ is the weakest relation that satisfies $(\sigma \sqsupseteq_R \sigma' \wedge I(\sigma')) \Rightarrow I(\sigma)$, but sometimes it is not adequate for verifying an invariant I. In fact, in Example 16, the program Pr is only monotone wrt. R_3, that is a relation stronger than R_2 but weaker than R_1 (and Pr is not monotone wrt. any of these relations).

5 Implementation and Experimental Results

We implemented a generic framework for decompiling EVM bytecode [29] to an intermediate representation, enabling the implementation of static analyses as well as generating and discharging verification conditions using SMT solvers, such as Z3 [10] and CVC4 [3]. 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. Furthermore, users can write relations for checking R-ECF. The language for specifying the relations uses comparison operators to express equalities and inequalities between arithmetic expressions ($=, \leq, \geq, <, >$). In order to build or combine these expressions, the language accepts basic arithmetic and boolean operators ($\wedge, \vee, \Rightarrow, +, *, \ldots$). The relations may refer to primitive member fields in contracts. In addition, the user can add conjuncts to the relation for expressing equality over all keys of a Solidity [11] mapping.[4]

Motivated by the real smart contracts analyzed, we picked a benchmark consisting of the top 150 most used smart contracts in the Ethereum blockchain. The contracts were found using BigQuery[5] that allows making SQL-like queries on the blockchain contracts of Ethereum and it is updated on real-time. The query was made on 2019-12-31 and asked for the address and bytecode of the 150 contracts that had had more transactions up-to that date.

To scale the experiment, we started by checking the equality relation $\equiv$ as our choice of R. This choice relieved us of checking monotonicity since the equality relation over all states is monotone for all programs. Additionally, if we manage to prove R-ECF for the equality relation, it means any invariant can be checked on the callback free traces. For the examples that could not be proven R-ECF with the equality relation because it is too strict, we wrote the desired invariants and their suitable relations, and re-ran the algorithm on the customized settings.

The actual algorithm implemented is based on Fig. 3, but with a predetermined call node ordering: going linearly from latest (in program-order) call nodes to earlier call nodes. The considerations for choosing this particular approach are:

- The ordered approach is strictly more precise than the minimal segments approach, thanks to join operations.
- Nevertheless, trying all possible call node orders, given that some functions may 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

4. We expressed summations over mappings by instrumenting the code to track the total sum of elements using ghost variables.

5. https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics

TABLE 1
Summarized ECF Results. 'CN' Stands for 'Call Node', and 'f' for 'Function'

	# 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
Failed to verify as ECF	11

performed. For these contracts, the approach would lead to faster proofs of R-ECF.

We have run all of 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. Another call node optimization can be done on contracts that invoke library contracts, that are guaranteed not to trigger a callback. In such cases, it is possible to ignore these call nodes, leading 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 R-ECF, but importantly it should be established that the state accessed by the the callee is separate from the caller's for soundness.

5.1 Experiments in a Realistic Setting

To validate the usefulness of our approach in a realistic setting, we conducted two experiments. First, we picked as a benchmark set the most used and invoked smart contracts, taking the top-150 contracts based on volume of usage, as of December 31st, 2019.[6] Second, we took a case-study of a callback related bug, and proved the bug patch as R-ECF. The case-study presents a contract which was drained \$7 million recently by attackers, using malicious callbacks behavior [16]. We show a non-trivial fix that renders an important invariant correct even in the presence of callbacks, while still allowing meaningful behaviors with callbacks that cannot be otherwise implemented. The proof of the invariant implies that such thefts cannot occur in the corrected code.

Top used contracts benchmark

Out of the top-150 contracts benchmark, a total of 132 contracts were successfully decompiled, but 38 of them do not contain call nodes and were therefore 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. As mentioned earlier, for this experiment we used $\equiv$ as our preorder.

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—not only do these contracts violate ECF, the non-ECF behavior is buggy according to natural invariants of the contracts[7]. 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 for ECF, since we found that the functions in fact do satisfy the original ECF property, but it cannot be proven using the ordered approach—namely, it is possible to construct an equivalent execution using a different function from the one being checked.

The contracts exhibiting these violations were selected for testing the usability of the parametric R-ECF approach, by us analyzing the contracts and writing invariants and preorders for use in R-ECF. Therefore, we focused on these 11 contracts to show the benefits of our property-guided approach. We manually analyzed the contracts and identified properties that describe the integrity of the system. In most cases, the properties are similar to the one we used for our running example: $I(\sigma) = \sigma[balance \geq \sum shares]$. Many of these contracts function as "banks", thus it is fundamental to ensure that they

6. up to Ethereum blockchain block number 9193265 until 2019-12-31 23:59:45 UTC

7. We have contacted the code owners but did not get a response from them.

TABLE 2
Results of Applying R-ECF to Prove Properties That Describe the Integrity of Contracts That are not ECF. We Denote by R_1 the Relation $I(s_2) \Rightarrow I(s_1)$, by R_2 the Relation $R_1 \wedge \texttt{Authority}(s_1)$, and by R_3 the Relation `Offers` Represents the Same Multiset

ID	Invariant	R	Result
0e6	$\texttt{\_supply} + \texttt{currentAirdropAmount} - \texttt{airdropBSupply} = \Sigma\texttt{\_balances}$	R_2	Verified
86f	$\texttt{\_supply} = \Sigma_{\texttt{x}}\texttt{\_balances[x]}$	R_1	Verified
89d	$\texttt{\_supply} = \Sigma_{\texttt{x}}\texttt{\_balances[x]}$	R_1	Verified
359	$\texttt{totalSupply} = \Sigma_{\texttt{x}}\texttt{balances[x]}$	R_1	Verified
d0a	$\texttt{EOS.supply} = \Sigma_{\texttt{x}}\texttt{balances[x]} \wedge \forall\texttt{p.dailyTotals[p]} = \Sigma_{\texttt{x}}\texttt{userBuys[p][x]}$	R_1	Non-exp.
2a0	$\texttt{balance} \geq \Sigma_{\texttt{x}}\texttt{tokens[0][x]}$	R_1	Not mon.
14f	N/A	R_3	Non-exp.
397	N/A	R_3	Non-exp.
d26	$\texttt{totalSupply} = \Sigma_{\texttt{x}}\texttt{balances[x]}$	R_1	Verified
d1c	$\texttt{jackpotSize} + \texttt{lockedInBets} \leq \texttt{balance}$	R_1	Not mon.
d1c	$\texttt{jackpotSize} + \texttt{lockedInBets} \leq \texttt{balance}$	R_1	Not mon.

always have enough balance to give back to their clients. From those invariants, we derived relations that satisfy Theorem 4, following the method used in Section 4.1. We then applied our tool for checking R-ECF against this benchmark.

Table 2 presents the results of the evaluation. Five contracts out of 11 were proven R-ECF with respect to the chosen relation, meaning that it suffices to prove the invariants on callback free executions of the contracts. Three contracts could not be proven because our language for expressing relations is not rich enough: In two of them, we need to express an equality between multisets defined by the collection of values in a Solidity mapping, which is beyond the reach of our expression language at the time of writing. For the third we have an invariant that connects the sum of balances in the contract to a `supply` variable in another contract. One contract could not be proven R-ECF because monotonicity of the relation could not be proven. For two other contracts the relations that we wrote are not monotonic.

All in all, we argue that our approach provides a new look into the callback problem that will be key to prove callback safety of contracts. As witnessed in our experiments, failing to prove the desired invariants and/or the monotonicity of the relation allows the user to understand the potential effect of the callbacks on the program state and integrate her feedback into the verification process.

Threats to validity

The main threat to validity in our experiments is that in some cases we had to instrument the code. Specifically, we introduced ghost variables to track properties involving the (unbounded) sum of elements in a data structure. In addition, the intended behavior of the contract, as captured by the verified invariant, was prescribed by us. It is possible that the authors of the contracts had other properties in mind. In general, even if the invariants were given by the programmers, there is the inherent gap between the intended behavior and its formalization by an invariant. (However, this is always the case when user-specification is needed.) Also, while we rigorously inspected our implementation, we did not verify it formally. Thus, it might contain latent bugs. However, our technique produces verification conditions which amount to claims about the commutativity of code blocks with respect to a given order relation, and these claims may be verified by an external tool. Finally, our approach is sound, but not complete, i.e., our tool may report the existence of a possible bug, where in fact no such bug exists.

```
96  contract Origin{
97    uint totalCoins;
98    uint ousdSupply;
99    mapping(address => uint) credits;
100   uint sumCredits;
101   uint creditsPerToken;
102   // uint outstanding;
103
104   function mint(
105     address to,
106     uint amount) private {
107     //amount = min(amount,outstanding)
108     //outstanding -= amount;
109     uint creditAmount = amount*
            creditsPerToken;
110     credits[to] += creditAmount;
111     sumCredits += creditAmount;
112     ousdSupply += amount;
113   }
114
115   function rebase() public {
116     // require(outstanding == totalCoins-
            ousdSupply);
117     // outstanding = 0;
118     if (totalCoins > ousdSupply) {
119       ousdSupply = totalCoins;
120       creditsPerToken = sumCredits/ousdSupply
            ;
121       require (creditsPerToken > 0);
122     }
123   }
124
125   function depositMultiple(
126     address[] assets,
127     uint[] amounts) public {
128     require (assets.length == amounts.length)
            ;
129     uint total = 0;
130     for (uint j = 0; j < assets.length; j++)
            {
131       total += amounts[j];
132     }
133     rebase();
134     for (uint i = 0; i < assets.length; i++)
            {
135       totalCoins += amounts[i];
136       // outstanding += amounts[i];
137       assets[i].transferFrom(
138         msg.sender, this, amounts[i]);
139     }
140     mint(msg.sender, total);
141 } }
```

Fig. 5. A simplified version of the Origin code. The invariant maintained is $totalCoins \geq ousdSupply$. The `private` modifier ensures that `mint` can only be called by functions within the contract, and thus in particular, cannot be called as a callback. The `totalCoins` can be changed by functions not presented in this simplified version.

5.2 Case Study: R-ECF for non Trivial Solvency

We present a case study based on the Origin code that was hacked for $7 million [16] in order to demonstrate how our approach and tool can be used to produce safe code without hamstringing callbacks. In Fig. 5 we show a snippet of the contract, with the lines used to fix it marked as comments. It implements a 'rebasing' *stablecoin* called *OUSD*. The contract is designed so that a single OUSD token is always worth $1, thus its value is stable. This value is backed by users' deposit of other stablecoin assets, which also have unit worth of $1. In code omitted from here, those assets are invested, gaining interest, and can be later *rebased* to

distribute the revenues among all token holders. To maintain the ownership of the token, the contract tracks units of *credits*, and uses a ratio denoted as *creditsPerToken* to convert credit amounts to token amounts and vice versa: `totalCoins` holds the total amount of token that the contract has (including revenues from interest). `ousdSupply` tracks how many of the tokens were distributed to the depositors. The *rebase* operation distributes the new wealth to all depositors by decreasing the credits per token ratio to increase the worth of credits, and updates the new supply of OUSD.

The hack on the contract involved calling an external contract that unexpectedly triggers a callback. In depositMultiple the depositer provides an array of stablecoin identifiers, and amounts (in $) to be deposited out of each stablecoin. The code first sums over the stablecoins how much money enters the contract. Then, it rebases any unaccounted revenues from previous transactions. Only then it may transfer the new assets into the contract by again looping over the assets and calling their transferFrom method. Afterwards, it can mint new OUSD based on the previously computed amount sum. The attack exploited the fact that between the rebase and mint operations there could be funds stored in the contract that are not owned by any user—especially after the first iteration of the loop, in which a positive amount of tokens was transferred to the contract and `totalCoins` was updated to reflect the new balance. Thus, if another rebase occurs between the transferring of funds via transferFrom and the mint, the contract would be fooled to think that the deposited money was gained by interest, and decrease the credits per token ratio. After the callback returns, the contract would mint the attacker's deposit according to the new ratio. However, any previous amount the attacker had would be worth more due to the new credits per token ratio. As a result, `ousdSupply` would be bigger than `totalCoins` and the contract would not be able to pay back all the depositors if everybody were to withdraw.

The core issue here is that transferred funds are accounted twice: both in a premature rebase (where they are mistaken for interest gains) and in minting (where they are generated and allocated to the account who deposited them). The suggested fix to the code fixes the double accounting of deposited amounts by keeping track of the outstanding supply not assigned to any particular user. Tracking the outstanding supply makes sure that the invariant that the total number of stablecoins deposited is greater or equal to the supply of OUSD, holds using our R-ECF framework. However, the fix does not render this contract ECF. This is because the final credits per token ratio can be different depending on the sequence of callbacks invoked. Specifically, any attempt to attack the contract would end up with the attacker losing money.

Our tool is able to ensure the safety of the fixed program by proving that all executions preserve the desired invariant ($totalCoins \geq ousdSupply$), while previous approaches based on ECF fail.

5.3 Comparison to Other Tools

We compared our implementation to other existing tools whose premise is to handle 'reentrancy bugs': *Securify2* [25] and *Slither* [12]. 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 our case study (as well as our lock-based example of Fig. 2, *Securify* and *Slither* both failed to show that it is actually safe (*Securify* times out after hours of running on the Amazon machine). In addition, neither *Securify* nor *Slither* were able to prove the correctness of the lock-based example of Fig. 2.

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 bytecode. Because of that we could only compare *Securify* to 10 contracts, and the results were aligned with ours in nine 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.

There are strong connections between our work and the ECF framework [1]: We both establish the correctness of a program by considering only callback-free executions. However, despite the similarity in names, there is a profound difference between the two frameworks. ECF allows establishing a generic correctness condition that ensures that callbacks do not introduce new behaviors, which, in our experience, amounts in most cases to verify that certain "callback protection" mechanisms are utilized correctly. Indeed, when looking at the benchmarks used to verify ECF we see that in most cases the contracts were correct due to the introduction of locks and command reorderings which push calls to other contracts to the end of the method. These procedures account for most contracts in the wild, where programmers do not wish to burden themselves with reasoning about complicated behaviors and tricky invariants. In contrast, the R-ECF framework proves a custom program-specific property, and as such, we intend it to be used by developers of systems in which unique behaviors of callbacks are allowed, and the correctness of the program lies in preserving a particular relation between executions with callbacks and callback-free ones. It stands to reason, that the number of the latter kind of programs is a mere fraction of the former. Thus, it is no surprise that we found a very small number of non-ECF examples. It is encouraging, however, that in most of these cases R-ECF is applicable in practice. Indeed, we were able to prove that five of the 11 functions that we were not able to verify as ECF are correct with respect to R-ECF and a suitable ordering relation, improving by 2% the number of verified functions with respect to ECF.

6 Conclusions and Related Work

Since the advent of smart contracts, reentrancy problems were identified as a dangerous source of correctness bugs [2], [17]. Our work contributes to the line of research started by Grossman *et al.* [15], who introduced ECF as a

means to ensure that contracts are immune to reentrancy attacks and to enable modular reasoning. However, the analysis of [15] is dynamic, hence it cannot be used to verify ECF safety for all executions. In this article, we have proposed a static R-ECF analysis that, as already discussed in detail along the article, enables more relaxed modularity relations and relies on them to prove contract invariants. In the rest of this section, we review other closely related work.

Focusing on the domain of smart contracts, Want *et al.* [28] presents a tool for static verification, called VERISOL, that allows inferring invariant properties preserved by all procedures of the contract. However, the analysis does not consider reentrant calls, thus executions with callbacks may not preserve these properties (according to our terminology, the inferred properties are callback-free program invariants, but they may not be program invariants). We believe that our approaches are complementary as once an invariant property is inferred by VERISOL, we can apply R-ECF to ensure that the property is also preserved by executions with callbacks. Cecchetti *et al.* [7] presents a different approach to verifying invariant properties in the presence of callbacks based on information flow control. This approach considers that a contract is reentrancy-secure if allowing reentrancy does not change which properties are invariants, i.e., if all callback-free program invariants of the contract are program invariants.

Mavridou *et al.* [18] presents a framework, called FSolidM, that prevents reentrancy using a built-in locking mechanism. In contrast, we present a technique for verifying the absence of reentrancy bugs that allows the benign use of callbacks. Schneidewind *et al.* [21] and Grishchenko *et al.* [14] introduce an over-approximation of the single-reentrancy property which, intuitively, states that 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 dangerous behaviors which cannot be reproduced by callback-free executions. Tsankov *et al.* [25] reports on 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 bugs, they use a pattern which forbids writes after calls. Thus, their restrictions are more severe than the ones imposed by R-ECF. Similar patterns are used by [12], [24].

A key benefit of our semantic equivalence based approach, when compared to pattern-based techniques, is that it enables to modularly check properties of contracts, that is one of the challenges of smart contracts verification as Sergey *et al.* [22] note when discussing the similarity of smart contracts to concurrent objects.

A deductive methodology for verifying module (object) invariants is presented in [19]. The main idea is to associate every module with a boolean ghost field *valid* which indicates whether the module satisfies the module invariant or not. Procedures which assume the module invariant as a precondition require the user to prove that its valid field is true when they are invoked. Procedures which mutate the module's state set the valid field to false before any mutation and set it back to true after the invariant is reestablished. As a result, it is not possible to prove the module is correct if it is used in the context of a program that invokes a procedure of the module as a callback while the module's state is being updated. We, on the other hand, do not assume that the user respects the module's specification and establish properties that hold in arbitrary executions, and, in particular, allow for state-mutating callbacks.

As reentrant calls behave in the same way as thread interleavings (i.e., they might change the global state when they interrupt), we also need to compare our work to verification of concurrent programs. The closest work to the ECF analysis is the atomicity analysis of Flanagan and Qadeer [13] and Wang *et al.* [27]. These analyses try to prove that all executions of the program are equivalent to one in which those code blocks execute without interruption by other threads. Same as Albert *et al.* [1], equivalence in this framework is checked by means of final-state equivalence. It will be interesting to study whether such atomicity analyses can be relaxed to use weaker equivalence relations as we have proposed in this article. Also, [13], [27] do not allow bidirectional movements that could lead to further accuracy. Another related work is Sousa and Dillig [23]: Def. 6 is similar to the *cartesian Hoare triple* of [23] used for verifying k-safety properties. The similarity is that both definitions rely on a generic relation R applied on two fragments of code to prove the equivalence of executions. The main difference as regards the problem definition is that we apply R-ECF on a single program with reentrant calls from an initial state, while [23] applies the cartesian Hoare triples over different programs possibly starting from different initial states. The constructive analysis in [23] is fundamentally different from ours, as it relies on a series of logic rules defined at the level of simple instructions, and it does not include our powerful commutation and projection operators defined over segments of code. We believe that our operators could strengthen their k-safety analysis and it is plan of our future research to investigate it.

REFERENCES

[1] E. Albert, S. Grossman, N. Rinetzky, C. Rodríguez-Núñez, A. Rubio, and M. Sagiv, "Taming callbacks for smart contract modularity," *Proc. ACM Program. Lang.*, vol. 4, pp. 1–30, 2020.

[2] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in *Pro. 6th Int. Conf. Princ. Secur. Trust*, 2017, pp. 164–186.

[3] C. W. Barrett *et al.*, "CVC4," in *Proc. Comput. Aided Verification 23rd Int. Conf.*, 2011, pp. 171–177.

[4] T. Bernardi *et al.*, "Preventing reentrancy bugs - another use case for formal verification," 2020. [Online]. Available: https://www.certora.com/blog/reentrancy.html

[5] A. Bizga, "A hackers' dream payday: Ledf.me and uniswap lose $25 million worth of cryptocurrency," 2020. Accessed: May 11, 2020. [Online]. Available: https://securityboulevard.com/2020/04/a-hackers-dream-payday-ledf-me-and-uniswap-lose-25-million-worth-of-cryptocurrency/

[6] V. Buterin, "Critical update re: Dao vulnerability," 2016. Accessed: Jul 2, 2017. [Online]. Available: https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/

[7] E. Cecchetti, S. Yao, H. Ni, and A. Myers, "Securing smart contracts with information flow," in *Proc. 3rd Int. Symp. Found. Appl. Blockchain*, 2020, pp. 1–5.

[8] Consensys, "Ethereum smart contract best practices," 2019. Accessed: May 14, 2020. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/

[9] P. Daian, "Analysis of the dao exploit," 2016. [Online]. Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[10] L. D. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Theory Pract. Softw. 14th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.

[11] Ethereum, Solidity, 2018. [Online]. Available: https://solidity.readthedocs.io
[12] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2019, pp. 8–15.
[13] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2003, pp. 338–349.
[14] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Proc. Int. Conf. Princ. Secur. Trust*, 2018, pp. 243–269.
[15] S. Grossman *et al.*, "Online detection of effectively callback free objects with applications to smart contracts," *Proc. ACM Program. Lang.*, vol. 2, pp. 1–28, 2018.
[16] M. Liu, "Urgent: Ousd was hacked and there has been a loss of funds," 2020, Accessed: Jan. 29, 2021. [Online]. Available: https://medium.com/originprotocol/urgent-ousd-has-hacked-and-there-has-been-a-loss-of-funds-7b8c4a7d534c
[17] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.
[18] A. Mavridou and A. Laszka, "Tool demonstration: Fsolidm for designing secure ethereum smart contracts," in *Proc. Int. Conf. Princ. Secur. Trust*, 2018, pp. 270–277.
[19] D. A. Naumann, "Assertion-based encapsulation, object invariants and simulations," in *Proc. Int. Symp. Formal Methods Compon. Objects*, 2005, pp. 251–273.
[20] D. Palmer, "Spankchain loses $40k in hack due to smart contract bug," 2018, Accessed: May 11, 2020. [Online]. Available: https://www.coindesk.com/spankchain-loses-40k-in-hack-due-to-smart-contract-bug
[21] C. Schneidewind, M. Scherer, I. Grishchenko, and M. Maffei, "*Ethor: Practical and provably sound static analysis of ethereum smart contracts*," in *Proc. 2020 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 621–640.
[22] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, 2017, pp. 478–493.
[23] M. Sousa and I. Dillig, "Cartesian hoare logic for verifying K-safety properties," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2016, pp. 56–69.
[24] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *Proc. IEEE/ACM 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2018, pp. 9–16.
[25] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 67–82.
[26] C. Turley, "imbtc uniswap pool drained for $300k in eth," 2020, Accessed: May 11, 2020. [Online]. Available: https://defirate.com/imbtc-uniswap-hack/
[27] L. Wang and S. D. Stoller, "Static analysis of atomicity for programs with non-blocking synchronization," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2005, pp. 61–71.
[28] Y. Want *et al.*, "Formal specification and verification of smart contracts for azure blockchain," 2019, *arXiv:1812.08829v2*.
[29] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2016, Accessed: Jul. 5, 2017. [Online]. Available: http://gavwood.com/paper.pdf

Elvira Albert is a professor with the Complutense University of Madrid, Spain, where she leads a research group, the COSTA team, currently made up of eight senior researchers and four PhD students. She is author of around 150 publications in international journals and volumes related to the topics of validation and verification of sequential and concurrent programs. She has been invited speaker with major conferences such as the ICST, ICLP, LPAR, and FM.

Shelly Grossman is currently working toward the PhD degree from the School of Computer Science, Tel Aviv, under the supervision of prof. Mooly Sagiv and prof. Noam Rinetzky. She is working on techniques for reasoning about programs in the presence of callbacks, and is the lead author of the POPL'18 paper about online detection of callback related vulnerabilities.

Noam Rinetzky is an associate professor with the School of Computer Science, Tel Aviv. He develops techniques and tools that aim to ensure the safety and correctness of sequential and concurrent software systems. He is the author of around 60 papers published in the most prestigious conferences and journals, including POPL, PLDI, CAV and the *Journal of the ACM*.

Clara Rodríguez-Núñez is currently working toward the PhD degree in computer science with the Complutense University of Madrid under the supervision of prof. Elvira Albert and prof. Albert Rubio. She is working on static analysis of smart-contracts, including techniques for ensuring modularity in the presence of callbacks.

Albert Rubio received the PhD degree in computer science from the Technical University of Catalonia, in 1994. He is professor with the Complutense University of Madrid since 2019. He was previously full professor with the Technical University of Catalonia since 2008. He is a author of around 70 papers published in the most prestigious conferences and journals, including LICS, CAV or the *Journal of ACM*. He is a coauthor of a chapter of the *Handbook of Automated Reasoning*.

Mooly Sagiv is professor with the School of Computer Science, Tel Aviv, and CEO of Certora, a startup company providing formal verification of smart contracts. His fields of interest include shape analysis, static program analysis, abstract interpretation, data-flow analysis, and smart contracts. He is author of around 200 papers published in the most prestigious conferences and journals. He has been awarded ACM SIGSOFT retrospective impact paper award (2011), microsoft outstanding collaborator award (2016), and ACM fellow (2016).