Analysis of “Effectively Callback Freeness” for Smart Contracts

Trabajo de Fin de Máster
Curso 2019–2020

Autor
Clara Rodríguez Núñez

Directores
Elvira Albert Albiol
Albert Rubio Gimeno

Máster en Métodos Formales en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Convocatoria: *Julio 2020*
Calificación: *10*

Analysis of “Effectively Callback Freeness” for Smart Contracts

Trabajo de Fin de Máster en Métodos Formales en Ingeniería Informática

Autor
Clara Rodríguez Núñez

Directores
Elvira Albert Albiol
Albert Rubio Gimeno

Máster en Métodos Formales en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Dedicatoria

A mi abuela,
por enseñarme tanto en cada llamada

Agradecimientos

En primer lugar, a Elvira y Albert. Gracias por todo lo que he aprendido y disfrutado estos meses, por el apoyo cuando las cosas no iban bien y por soportarme en cientos de videollamadas, incluso cuando me pasa un tren por encima. Formar parte de un proyecto como este ha sido un autentico privilegio. Gracias también a todos los miembros del grupo COSTA por recibirme tan bien y estar siempre dispuestos a echarme una mano.

Gracias a todas las personas que me han hecho disfrutar de las matemáticas y la informática. A los profesores que he tenido durante estos años, gracias por vuestro esfuerzo y pasión por enseñar, habéis sacado una versión de mí que nunca pensé que existiera. A los amigos que he hecho en esta universidad, gracias por ser los mejores compañeros de viaje. Aun nos quedan muchas locuras por hacer, montañas que escalar y partidos donde no tirar.

Gracias a mi familia. Cuando era una niña y soñaba con ser investigadora me acompañabais cada fin de semana a buscar fósiles a la montaña de detrás de casa. Veinte años más tarde aun no han aparecido dinosaurios en La Cistérniga, pero sí estoy un pasito más cerca de investigar. Gracias por ser sin duda los grandes responsables de ello.

A mi equipo. Gracias por buscarme un sitio en el Antela para mis reuniones y cada jueves fingir que os ibais a colar en ellas, por confiar en mi talento para los triples y mi velocidad al volante. Espero estar muy pronto celebrando esto con vosotras y un montón de manolitos.

A Rocío y Prats, gracias por ser siempre mi equipo. Tengo claro que nuestras cenas han sido un ingrediente clave en este trabajo, incluso las crudiveganas de zanahoria.

Finalmente, gracias a Tom Rosenthal por ser la banda sonora de muchas de las horas dedicadas a este trabajo.

Abstract

Analysis of "Effectively Callback Freeness" for Smart Contracts

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

In general, checking modularity is undecidable– even for programs without loops. This work describes an effective technique for soundly ensuring modularity harnessing SMT solvers. The main idea is to define a constructive version of modularity using *commutativity* and *projection* operations on program segments.

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

The main results in this project have been submitted to the *ACM SIGPLAN conference on Systems, Programming, Languages, and Applications: Software for Humanity (OOPSLA 2020).*

Keywords

Modularity, static analysis, callbacks, commutativity, SMT solvers, DAO attack, Ethereum, smart contracts.

Resumen

Análisis de "Effectively Callback Freeness" para Smart Contracts

Los callbacks son esenciales en muchos entornos de programación, especialmente en los que como Ethereum no permiten estados globales compartidos ni concurrencia. A través de ellos un programa llamado puede llevar de nuevo la ejecución al llamante. A pesar de su efectividad, su uso puede dar lugar a errores. De hecho, muchos de los principales ataques realizados sobre smart contracts explotan su uso. Este trabajo presenta el primer método estático para asegurar *modularidad* en presencia de callbacks y su aplicación sobre algunos de los principales smart contracts. La modularidad de un contrato asegura que llamadas a contratos externos no pueden afectar a su comportamiento. Cabe destacar que se garantiza modularidad sin establecer restricciones sobre la programación.

En general, estudiar la modularidad de un programa es indecidible, incluso cuando no incluye bucles. Este trabajo describe un método efectivo para demostrarla de manera sólida utilizando SMT solvers. La idea clave es el desarrollo de una noción de modularidad basada en la conmutación y proyección de segmentos del programa.

De cara a estudiar la precisión del análisis hemos implementado y aplicado nuestra técnica sobre smart contracts reales (incluyendo un subconjunto de los 150 contratos más llamados en Ethereum). Nuestra implementación decompila el bytecode de los programas a una representación intermedia y después estudia su modularidad a través de consultas a SMT solvers. Los resultados experimentales obtenidos indican que el método puede aplicarse sobre multitud de contratos, y que es capaz de demostrar modularidad en casos donde otros métodos fallan.

Los principales resultados de este trabajo se han presentado a la conferencia internacional *ACM SIGPLAN conference on Systems, Programming, Languages, and Applications: Software for Humanity (OOPSLA 2020).*

Palabras clave

Modularidad, análisis estático, callbacks, conmutatividad, resolutores SMT, ataque DAO, Ethereum, smart contracts.

Contents

List of figures

List of tables

Chapter 1

Introduction

Modularity is a key principle in system design: Encapsulating code and data into different modules which communicate via clearly defined procedural interfaces allows separately designing, developing, understanding, testing, and reasoning about different parts of the system. For example, the fully encapsulated programming model of the Ethereum blockchain allows for any object ("smart contract") to interact with other ones by invoking their methods, but prevents direct access to the other contracts' data. Modularity, however, is not a panacea as demonstrated by the infamous DAO bug [10]. The latter exploited the *callback mechanism* to temporarily steal money.[1] Callbacks occur when a method of a module, say a smart contract, invokes a method of another module, say, another smart contract, and the latter, either directly or indirectly, invokes one or more methods of the former before the original method invocation returns. Callbacks complicate program reasoning (see, e.g., [20]) because they require programmers to consider interleavings of calls to their own code, which, as in concurrent programming [30], can be very tricky. The danger of callback attacks, also called *reentrancy attacks*, led to many suggestions for syntactical program restrictions, e.g., delaying external calls (see, e.g., [12]). However, these restrictions are overly severe and several realistic programs violate them.

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

1.1 The DAO Attack

We motivate our work using the infamous bug in the DAO (Decentralized Autonomous Organization) contract [10]. Figure 1.1 shows a simplified vulnerable *Solidity* contract. The purpose of the DAO contract is to facilitate voting on invest-

[1] The money was "returned" by forking Ethereum blockchain into a new blockchain with a fixed code for the DAO contract.

```
1  pragma solidity ^0.4.24;
2
3  contract Bank {
4    mapping (address => uint) public shares;
5
6    function deposit() payable {
7      /* balance is an alias for address(this).balance */
8      balance += msg.value;
9      shares[msg.sender] += msg.value;
10   }
11
12   function withdraw() {
13     uint256 orig_balance = balance;
14     uint256 orig_shares = shares[msg.sender];
15     if (orig_shares > 0 && orig_balance >= orig_shares) {
16       balance = balance - orig_shares;
17       if (msg.sender.send_money(orig_shares)!=success) {
18         balance = orig_balance; // reverting
19         shares[msg.sender] = orig_shares;
20       }
21       else shares[msg.sender] = 0;
22     }
23   }
24 }
```

Figure 1.1: A Solidity contract illustrating the DAO bug. We write the balance update effects of payable functions and send operations explicitly using the balance variable. The send_money operation is the same as Solidity's send. success represents a success code as returned by send_money. Revert operations are also stated explicitly.

ment proposals by the owners of the DAO (referred to as objects in the following). The contract stores in the variable shares the individual investment for each object as well as the balance variable.

For clarity of the presentation, we avoid using predefined Solidity instructions for money transfer and state reversal, and implement them by explicit updates to the state.

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

The withdraw function allows pulling out all available shares of the object, which is implemented by decreasing the current shares amount from the contract's own balance and transferring it back to the object by means of the send_money in line 17. This is a *call node* where control is relinquished to the callee object. At this point, the callee object might execute a callback. If the call does not fail (programmed as returning success), the object's shares is set to zero. Otherwise the state is reverted to the initial one (then branch).

```
bool attacked;

function send_money(uint value) {
  if (!attacked) {
    attacked = true;
    balance += value;
    Bank.withdraw();
  }
}
```

Figure 1.2: Attacker object stealing money from DAO contract

The DAO was attacked by a "callback loop-hole" in which the receiver object calls back the method withdraw to steal money, in particular, the code of the send_money function is designed to call withdraw again. Figure 1.2 shows a snippet of code that produces such a callback loop-hole and Figure 1.3 shows the exploit trace. Basically, when the attacker receives the control in send_money, it increases its balance and calls back withdraw again.[2] As the shares of the attacker are only updated in line 21 after the send_money has finished, the callback execution of withdraw will find the shares with the initial value and will make another transfer to the attacker. Figure 1.3 depicts the bug in the malicious trace. The presence of the callback violates the invariant balance $\geq \sum$ shares.

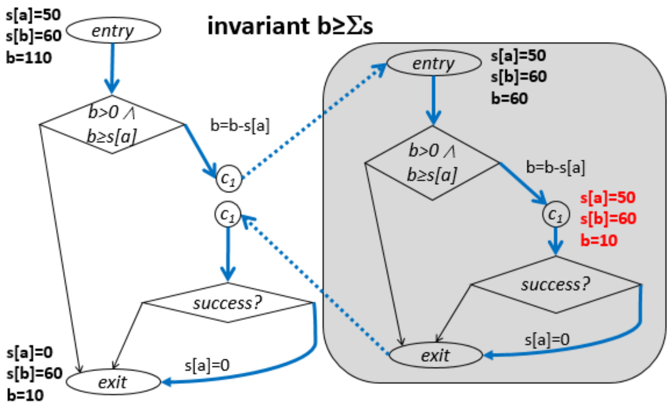


Figure 1.3: The CFG of the withdraw method from the objects in Figure 1.1 and the malicious trace, marked with blue edges (b is balance, s is shares). The area under the grey rectangle pertains to the callback.

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

```
contract Bank {
  mapping (address => uint) public shares;
  bool lock = false;

  function deposit() payable {
    balance += msg.value
    require (!lock);
    shares[msg.sender] += msg.value;
  }

  function withdraw() {
    require (!lock);
    uint256 orig_balance = balance;
    uint256 orig_shares = shares[msg.sender];
    if (orig_shares > 0 && orig_balance >= orig_shares) {
      lock = true;
      balance = balance - orig_shares;
      if (msg.sender.call(orig_shares)!=success) {
        balance = orig_balance;
        shares[msg.sender]= orig_shares;
        lock = false;
      }
      else {
        lock = false;
        shares[msg.sender] = 0;
      }
    }
  }
}
```

Figure 1.4: Solidity contract avoiding the DAO bug. Not verifiable using previous approaches for ECF checking.

Severity of Reentrancy Attacks. The DAO problem is also called 'Reentrancy Attack' since it exploits the non-reentrant nature of the stateful code. The attack is pervasive, e.g., [13, 10], and keeps occurring even after the DAO hack [27, 36, 8]. For example, [6] describe a bug in a test version of Synthetix [31], one of the three top-most valuable crypto assets according to [32]. This bug was identified using the algorithm presented in this dissertation.

Pattern Based Tools. The standard way to identify problems like the DAO is by searching for a common pattern, of 'write-after-call', e.g., [35, 16, 33, 24, 15, 9]. The idea is that if there are no writes to the state after calls, then it is easy to see that the contract is safe against reentrancy attacks. Pattern-based solutions yield many false alarms on existing code, preventing the developers from using these tools.

Example 1 *Consider the contract in Figure 1.4 that illustrates a "contract-locks" solution to avoid callbacks found in real contracts. It uses a boolean state variable* lock *to forbid callbacks such that a callback from a different object to execute* withdraw *will encounter* lock *set to true, and the* require *instruction will prevent the execution of the* withdraw *function.*[3] *Pattern-based tools flag this function as vulnerable to a*

[3] In Solidity, if the condition within the require does not hold, the execution is reverted to the

reentrancy attack, which is not useful to smart contract developers.

1.2 Effectively Callback Freedom (ECF)

In this section we present the notion of **E**ffectively **C**allback **F**ree (ECF) contract introduced by [19] as a semantic way to guarantee immunity to reentrancy attacks. They define the concept of ECF execution and its static extension for contracts. An execution trace of an object is ECF if there exists an equivalent execution trace without callbacks to this object. Hence, an object is ECF if all its possible execution traces are ECF.

[19] also suggest dynamic techniques for checking if an execution is ECF based on conflict between read and write operations: a given execution is ECF if we can reorder its instructions in a way we obtain a callback-free execution such that every pair of read/write operations appears in the same order in both executions. Nevertheless, this approach is over-conservative and flags as dangerous typical solutions for reentrancy attacks, as the one using locks shown in the contract in Figure 1.4.

Finally, we introduce in this section an overview of our approach, motivating the definitions developed in the next chapters. We show the pseudocode of a simplified version of the technique and include examples illustrating its intuition.

1.2.1 Previous approaches

[19] define the notion of **Effectively Callback Free** (ECF) module. Intuitively, a module is effectively callback free if for every trace with a callback, there exists "an equivalent" callback free trace. [19] suggest two definitions of trace equivalence inspired by database theory [7]: (i) semantic equivalence based on final state inspired by final state serializability, and (ii) a syntactic notion of equivalence based on conflict, i.e., reordering based on reads and writes, inspired by conflict-serializability.

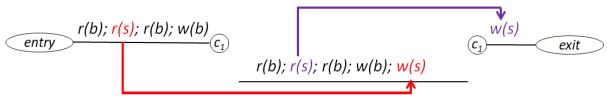
The semantic way to guarantee immunity to reentrancy attacks they suggest is to show that every execution with a callback can also be simulated without callbacks, by making sure that an object is **E**ffectively **C**allback **F**ree (ECF).

A constructive approach to it is to try and reorder callbacks such that they are executed outside the context of a callback, and show that the resulting trace is in some sense equivalent to the original trace. This approach requires checking commutativity of potentially unbounded sequences of operations and each such check is potentially undecidable.

A simple conservative way to check for ECF already suggested in this paper, is to check that an object is ECF by reordering operations without read/write conflicts. This method, called conflict serializability, is the basis for parallelization in modern database systems, e.g., [7]. Two executions are *conflict-equivalent* if every pair of read/write operations appears in the same order in both of these executions. ECF can be ensured, for the considered execution, if we find a callback-free execution (among all the reorderings produced) which is conflict-equivalent to the execution with the callbacks. Consider the malicious execution trace from Figure 1.1 described

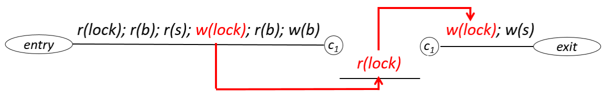
initial state.

above, read (r) and write (w) operations appear on the edges:



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

Conflict serializability is also easier to check using some static analysis techniques, e.g., [34]. Therefore, the original motivation of this work was to implement a static algorithm for checking conflict serializability. Unfortunately, conflict serializability is over-conservative and prohibits valid solutions for reentrancy attacks. For example, the aforementioned Figure 1.4 contains a corrected version of the DAO. The main idea of this corrected code is to deploy a boolean lock preventing unintended callbacks. However, there are traces with callbacks which are not conflict serializable:



Observe that the lock variable is written to before and after the callback, and thus the read of the lock variable in the callback cannot be reordered with respect to either write.

1.2.2 Static Verification of ECF

This work develops a method for statically verifying ECF using commutativity checks (which assure equivalence) while also allowing *projecting away* irrelevant pieces of code. Our starting point is the reduction of [19] for ECF: it shows that if there is a violation (using syntactic conflict equivalence) of the ECF property in a trace with arbitrary nested callback calls then there is one where callbacks are not nested. We generalize this reduction to semantic final state equivalence and develop our techniques for *simple* traces, i.e., ones where the execution of a single method can be interrupted at call nodes by an arbitrary sequence of executions of other procedures, however these interrupting procedures themselves never get interrupted. We prove that a simple trace is ECF by constructing an equivalent callback free trace via a sequence of swapping and removing of all possible different interrupting invocations that might arrive.

Example 2 *Consider trace t_a shown in Figure 1.5. The trace depicts an execution of procedure $h()$ of a module m which is interrupted twice by different callbacks: $h()$ starts executing at its entry point and performs a sequence of primitive commands following its control flow graph (h_1) until it gets to a* call node *(c_1) where it relinquishes control to an external method. At that point, the external method invokes*

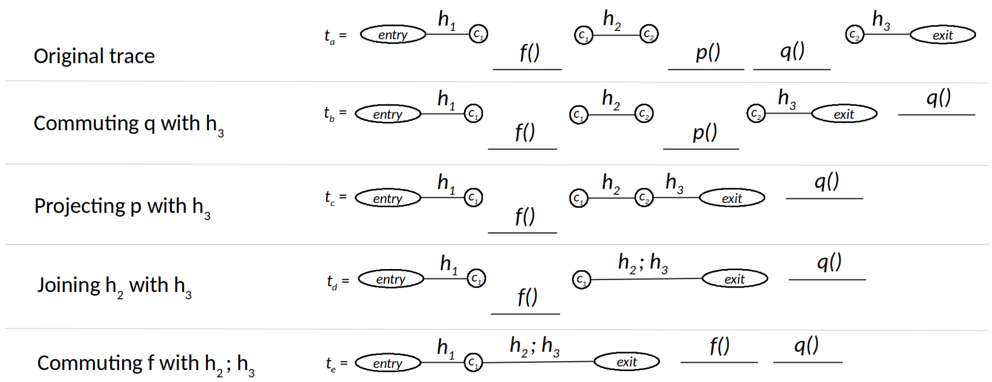


Figure 1.5: Sequence of commutation and projection operations on an example trace.

procedure $f()$ on m thus generating a callback. Control returns to $h()$ only after $f()$ exits and $h()$'s execution continues from c_1 by executing the next sequence of intra-procedural primitive commands (h_2) until another call node (c_2) is reached. At that point the external procedure generates two callbacks by invoking $p()$ and then $q()$. After control returns to $h()$ the sequence h_3 is executed and the execution ends. We turn t_a into the callback free trace t_e by either commuting the subtraces corresponding to the callback calls or projecting *them away. (Note that callback $p()$ is not part of t_e). Trace t_b shows the result of (right) commuting q with h_3. Intuitively, such a transformation is possible if the composed effect of $q; h_3$ is preserved by $h_3; q$ (c.f. Section 3.2). Trace t_c shows a different way to transform the trace, namely by projecting p away. The elimination of p can be done by a (right) projection with h_3, provided the composed effect of $p; h_3$ is preserved by only executing h_3. Alternatively, we can achieve the same goal using (left) projection with h_2, provided the composed effect of $h_2; p$ is preserved by only executing h_2. At this point, we consider the call node c_2 "solved". Once we solved c_2, we can continue with the swapping and projecting operations to the other callbacks. However, we can do better. Note that once we solved c_2 the trace of $h_2; h_3$ is not interrupted. Thus, while we could, for example, try to swap f with h_2 and then with h_3 in order to solve call node c_1, we, instead, try to swap it with the "joined" trace $h_2; h_3$ (t_d). Note that if the separate swaps succeeds it is guaranteed that the swap over the* joint trace *$h_2; h_3$ succeeds too. This is not true, however, the other way around.*

The aforementioned transformation ensures that the resulting trace is final-state equivalent to the original one, i.e., the effect of executing the original trace (t_a) on an initial state σ can be reproduced by executing it on the transformed (callback-free) trace (t_e).

1.2.3 Overview of the technique

We propose a constructive ECF analysis that can be checked using SMT solvers. For the overview, we present a simplified and intuitive version of the definition, that does not show all edge cases. The full definition appears in the main chapters of the work. We partition a trace T which may contain callbacks to subtraces *prefix*,*suffix* and A, such that $T = \mathit{prefix}; A; \mathit{suffix}$ where $A \in F^*$, i.e., all possible sequences of function calls from the object, of unbounded length. Let $\alpha(\cdot)$ denote the multiset of letters in a sequence. The goal is to find disjoint subsequences G, H contained in A, i.e., $\alpha(G) \uplus \alpha(H) \subseteq \alpha(A)$ and $\alpha(G) \cap \alpha(H) = \emptyset$, such that the callback-free sequence $G; \mathit{prefix}; \mathit{suffix}; H$ is final-state equivalent to T.

```
check_ECF_single_callnode(n, f):
prefix = extract_prefix(f,n)
suffix = extract_suffix(f,n)
L = get_left_movers(prefix)
R = get_right_movers(suffix)
// L,R are subsets of F
if (L + R != F) return MayNotBeECF
return check_no_move_collisions(L,R)
```

Figure 1.6: Pseudocode of the algorithm for checking a function with a single callnode.

A pseudocode of the algorithm for checking our constructive ECF definition is given in Figure 1.6. It operates by extracting the code segments that pertain to the 'prefix' and 'suffix' traces of the chosen *callnode* n, where n is the location of function f which yields control to callbacks. The algorithm then computes the set of *left* and *right* movers (similar in spirit to [23]). The left and right movers determine how the subsequences G, H from the above definitions are chosen. To make sure that we can find such G, H for all possible traces A, the set of left and right movers must cover all available functions F. If they do not cover all F, then the function f may not be ECF. In that case, the functions in $F \setminus (L \cup R)$ serve as a witness that explains the potential violation. Interestingly, covering F is not sufficient for proving ECF: the order of the functions in A may affect whether they can be reordered or not. For example, if $g_1 \in R$, $g_2 \in L$, and g_1, g_2 do not commute, then for $A = g_1; g_2$ it is not necessarily true that $g_2; \mathit{prefix}; \mathit{suffix}; g_1$ is equivalent to T. A concrete example is given in Figure 4.2. Hence, the check_no_move_collisions(L,R) takes the sets of left and right movers, and ensures that such conflicts cannot occur by checking commutativity properties of pairs of functions. If two functions do not commute in a way that prevents callback reordering, then it is possible to produce a potential callback sequence that cannot be reordered outside of the call node, and thus may indicate ECF violation. In the following chapters, we explain how the definition and algorithm are generalized to handle the case of multiple call nodes in a function.

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

SMT solvers can be used to soundly reason about commutativity properties, e.g., [1, 4, 37], and we use those in the implementation. Given the known limitations of such solvers in large scale, our chief insight is that for ECF, it is possible to minimize the number of commutativity checks discharged with the SMT solver. This is described in further detail in Chapter 6. To intuitively illustrate how our algorithm operates, and how counterexamples are given, we go back to the buggy code from Figure 1.1. This code contains two functions, one of them containing a single callnode (withdraw). Therefore, the algorithm analyzes whether both functions, withdraw and deposit, can commute with the code segments before and after the callnode, which we denote as withdraw_prefix and withdraw_suffix, resp.

Call node at function withdraw(): line 16

	withdraw()	deposit()
Move before	X	not checked
Move after	X	not checked

ECF check of withdraw() failed due to the following callback trace at line 16:
withdraw();

Figure 1.7: Simple counterexample to ECF produced by the analysis

It can be seen that withdraw does not commute with either withdraw_prefix nor with withdraw_suffix. Thus, the SMT solver shows us traces for violating the commutativity for both, and the conclusion overall would be that withdraw cannot be moved out if it runs as a callback in this callnode. An example summarized output of the analysis is given in Figure 1.7. For the corrected code from Figure 1.4, assuming the algorithm starts by trying to move both functions to the left, then clearly the callbacks can be projected away with respect to the prefix of the callnode–the lock is set to true, and the callbacks have no effect and can be omitted. An example summary is given in Figure 1.8.

Call node at function withdraw(): line 49

	withdraw()	deposit()
Move before	✓	✓
Move after	not checked	not checked

ECF check of withdraw() succeeded.

Figure 1.8: Proof of ECF produced by the analysis

1.3 Objetives and contributions

The goal of this master thesis is to develop a sound static analysis for proving that an object satisfies the ECF property. As we discussed in Section 1.2, there exist techniques for checking if a given execution verifies the ECF property. However, to the best to our knowledge, this is the first work to present a technique for statically verifying this property. We implemented our approach and applied it to real smart contracts, in particular to the 150 most queried contracts in Ethereum, demonstrating the applicability of the technique.

In summary, the master thesis makes the following main contributions:

(1) Semantic commutation and projection, and segment-join operations. We present semantic notions of left/right/zero-projection, that together with the operations of commutation and segment-join (intuitively illustrated in the example in Figure 1.5), lay down our analysis.

(2) Static analysis. We introduce a novel static analysis (intuitively outlined in Figure 1.6) based on proving commutativity and projection between all the fragments of code (or code segments) in between call nodes and *all* other procedures of the module.

(3) Callback invariant. We introduce the new concept of *callback invariant* that can be used within our static framework in a natural way in order to increase its accuracy.

(4) Implementation and evaluation. A prototype of our static analysis algorithm is implemented on top of the EVM bytecode [39] and evaluated on the most called Ethereum contracts and on a realistic decentralized finance application.

The main results in this master thesis have been submitted to the *ACM SIGPLAN conference on Systems, Programming, Languages, and Applications: Software for Humanity (OOPSLA 2020)*[2]. This submission is currently under revision.

1.4 Organization of the Project

The remainder of the work is organized as follows.

Chapter 2 defines the necessary syntax and semantics for the programs that we consider. Chapter 3 introduces the definitions that the static analysis relies on. Essentially, they are segments of code, and the projection and commutation operations on segments. Chapter 4 describes our static analysis technique for checking ECF. Chapter 5 extends the analysis to include the concept of *callback invariant.* Chapter 6 presents the implementation and its evaluation on Ethereum smart contracts, demonstrating the applicability and effectiveness of the proposed technique. Chapter 7 discusses related work and we conclude in Chapter 8 showing the main conclusions and pointing out directions for future research.

Chapter 2

Preliminaries

This chapter introduces some preliminary notions and notations. We present the language we use to formalize our results and define some basic notions like trace or execution.

2.1 Programming language

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

Every procedure has a unique *entry node*, to which no edge leads, and a unique *exit point*, from which no edge leaves. In addition, some of the program locations of a procedure may be *call nodes*. We sometimes refer to call nodes as callback points. Every time a procedure reaches a call node it may invoke arbitrary procedures an arbitrary number of times and then finally *havoc* the value of a specially designated *return variable* r by setting it to an arbitrary value.

Program states $\sigma \in \Sigma$ record the values of the program's global variables, the program counter and the local variables of the currently executing procedure. The state also maintains a stack of the program locations and values of the local variables of pending calls. We assume to have at our disposal a semantic function $[\![\cdot]\!]$ which assigns meaning to transitions $[\![\rho]\!] \subseteq \Sigma \times \Sigma$ as a binary relation over program states. Our programs are deterministic in the sense that at most one output state can be produced by applying a transition (with the exception of the aforementioned havoc transitions) to any input state. The intention is that the program can proceed from the program location n at the *source* of a transition $\rho = \langle n, c : a, n' \rangle$ to the *target* program location n' of ρ only when the program is in an *input state* σ which satisfies c and it then produces an *output state* σ' according to the assignments a

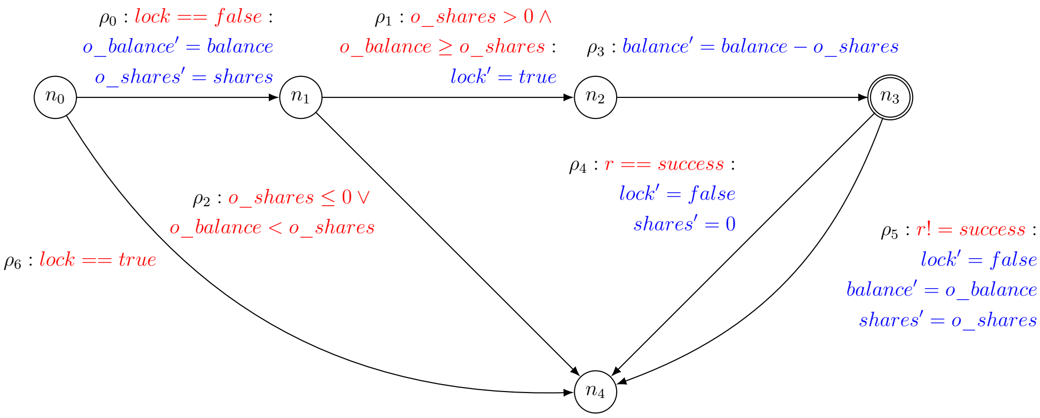


Figure 2.1: TS for withdraw procedure from Fig. 1.4 written in our programming language. Conditions appear in red and assignments in blue.

annotating ρ. Thus, $[\![\rho]\!]$ is comprised of all such pairs of states $\rho = \langle \sigma, \sigma' \rangle$ that define a transition relation. Hence, from now on, we will refer to our CFGs as (a symbolic denotation of) *Transition Systems* (abbreviated as TS). Figure 2.1 depicts the TS of the withdraw procedure from Figure 1.4 where n_0 and n_4 are the entry and exit nodes, resp. We write the assignments annotating edges using two-vocabularies in the standard way: The primed variables v' represent the value of a variable v after the transition and the unprimed version v represents its value before the transition executes. We mark its sole call node (n_3) using a double circle. In our programming language we can describe encapsulated objects as programs defined as the set of TSs for their procedures, and the non-deterministic call mechanism used to represent callbacks. The programming model considered is general enough to define the relevant part of our analysis for most programming languages, and its simplicity helps clarify our presentation.

2.2 Traces

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

subscript in what follows).

Example 3 *In the program shown in Figure 2.1, we have, for instance, that* $TR(n_0, n_3) = \{\rho_0; \rho_1; \rho_3\}$, $TR(n_0, n_4) = \{\rho_0; \rho_1; \rho_3; \rho_4,\ \ \rho_0; \rho_1; \rho_3; \rho_5,\ \rho_0; \rho_2,\ \rho_6\}$, *and* $TR(n_3, n_4) = \{\rho_4, \rho_5\}$.

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

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

Example 4 *Examples of traces without callbacks from* n_0 *to* n_4 *are shown in Ex. 3 in* $TR(n_0, n_4)$. *Examples with callbacks would be (the callback trace is underlined):* $\rho_0; \rho_1; \rho_3; \underline{\rho'_6}; \rho_4$ *where* ρ'_6 *is a callback trace, or* $\rho_0; \rho_1; \rho_3; \underline{\rho'_0; \rho'_2}; \rho_4$. *However, the latter would be pruned out by the execution since it is not feasible to execute* ρ'_0 *at this point as* ρ_1 *sets* lock *to* true *and hence the condition in* ρ'_0 *does not hold.*

2.3 Executions

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

Definition 1 ($\simeq_{FS}$) *Executions* ξ_1 *and* ξ_2 *are final state equivalent, written* $\xi_1 \simeq_{FS} \xi_2$, *if* $start(\xi_1) = start(\xi_2)$ *and* $end(\xi_1) = end(\xi_2)$.

It is now possible to use the above notations to define ECF for both executions (dynamic) and programs (static), similarly to [19].

Definition 2 ($dECF_{FS}$) *A complete well-formed execution* ξ *is* effectively callback-free, *written* $\xi \models dECF_{FS}$, *if it is final state equivalent to a complete callback-free execution.*

Definition 3 ($sECF_{FS}$) *A program Pr is* effectively callback-free *(denoted* $P \models sECF_{FS}$*) if every complete well-formed execution of Pr is effectively callback free.*

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

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

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

Chapter 3

Segments, Projection and Commutation

This chapter introduces auxiliary definitions that the static analyses in Chapter 4 rely on, namely segments of code, and the projection and commutation operations on segments. As usual, the static analysis handles many traces at once: the concept of *segment* will allow us to characterize all traces that can arise from using the fragment of code in the segment. In order to explain the intuition of our operations, we consider a simple complete well-formed trace which is not callback-free $t_1; t_f; t_2$, where t_f is a function and $t_1; t_2$ is a function as well. (Note that $\mathit{end}(t_1) = \mathit{start}(t_2)$ is a call node.) We say that t_1 is the left subtrace, and t_2 is the right subtrace, and denote by τ_1, τ_f and τ_2 the segments to which t_1, t_f and t_2, resp., belong. Our technique aims at guaranteeing ECF by proving that the final state of an execution of $t_1; t_f; t_2$ is the same as the final state of an execution of either $\tau_1; \tau_2; \tau_f$ or $\tau_f; \tau_1; \tau_2$ or $\tau_1; \tau_2$ (when starting from the same initial state). In order to prove the equivalence, we define *projection* and *commutation* of pairs of segments. Applying these operations guarantees that the resulting state is the same *and* that in all *feasible states* from which the original segment sequence can start and fully execute, so can the new one. Informally, the projection operation applied on τ_1 and τ_f ensures that an execution of $\tau_1; \tau_f$ leads to the same state as an execution of τ_1 alone. If it holds, we have proven ECF for the considered sequence. Commutation ensures that an execution of $\tau_1; \tau_f$ results in the same state as an execution of $\tau_f; \tau_1$.

3.1 Basic definitions on segments

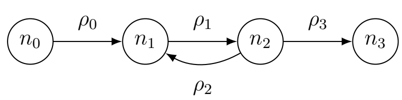
Segments represent potentially unbounded number of traces, going between start, exit, and call nodes. In the definition for segments, we refer to the start and exit nodes of a procedure as call nodes too. In the rest of the chapter, we assume to be working with an arbitrary fixed program Pr.

Definition 4 (Segment) *Given two call nodes n and n', the* segment *between n and n' is the set of traces $TR(n, n')$. A segment $TR(n, n')$ is a* function *if n is the start node of a procedure and n' is its exit node. The set of function segments of a program Pr is denoted by $F(Pr)$. A segment belongs to a procedure p if its start and exit nodes belong to p.*

Example 5 *The segment for the program shown in Figure 2.1 for n_0 and n_3 is $\tau_0 = \{\rho_0; \rho_1; \rho_3\}$, for n_3 and n_5 is $\tau_1 = \{\rho_4 \ , \ \rho_5\}$ and for n_0 and n_4 is $\tau_2 = \{\rho_0; \rho_1; \rho_3; \rho_4 \ , \ \rho_0; \rho_1; \rho_3; \rho_5 \ , \ \rho_0; \rho_2 \ , \ \rho_6\}$, where τ_2 is a function segment, since its traces go from the start node n_0 to the end node n_4.*

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

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



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

Definition 5 *Given a segment τ, we say that $\sigma - \tau - \sigma'$ if and only if there exist a trace $t \in \tau$ such that $\sigma - t - \sigma'$.*

3.1.1 Segment-sequences

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

Definition 6 (Segment-sequence) *A* segment-sequence *is a non-empty sequence of segments of the program. A segment-sequence is* well-formed *if the end node of each segment is the initial node of the next one.*

Following the example shown in Example 5, the segment-sequence for the execution trace $\rho_0; \rho_1; \rho_3; \rho_6'; \rho_4$ would be $\tau_0; \tau_2'; \tau_1$, where we have primed the segment τ_2' of the callback procedure.

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

Definition 7 *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,...,t_n such that for every $i = 1, \ldots, n$ we have that $t_i \in \tau_i$.*

Definition 8 *Given a segment-sequence π, we say that $\sigma - \pi - \sigma'$ if and only if there exist a trace t represented by π such that $\sigma - t - \sigma'$.*

3.2 Commutation and projection

We define the following concepts about commutativity and projection.

Definition 9 (Commutation) *Given two segments τ_1 and τ_2, we say that τ_1 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 $\sigma' = \sigma''$.*

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

Definition 10 (Left-projection) *Given two segments τ_1 and τ_2, we say that τ_1 left-projects with τ_2 for the state $\sigma \in Feasible(\tau_1; \tau_2)$ if and only if if $\sigma - \tau_1; \tau_2 - \sigma'$ and $\sigma - \tau_1 - \sigma''$ then $\sigma' = \sigma''$.*

Definition 11 (Right-projection) *Given two segments τ_1 and τ_2, we say that τ_1 right-projects with τ_2 for the state $\sigma \in Feasible(\tau_1; \tau_2)$ if and only if $\sigma \in Feasible(\tau_2)$ and if $\sigma - \tau_1; \tau_2 - \sigma'$ and $\sigma - \tau_2 - \sigma''$ then $\sigma' = \sigma''$.*

Consider the segments τ_0 and τ_2 defined in Example 5. τ_0 represents the traces of withdraw until the callnode point. τ_2 is representing the withdraw function. We study whether they commute or project in order to prove ECF for traces of withdraw that have withdraw called as a callback. τ_0 does not commute over τ_2 since there is an initial state where the final values of the balance variable could be different: $\tau_0; \tau_2$ does not decrement balance a second time in the callback τ_2 due to the lock being set in τ_0, while $\tau_2; \tau_0$ may fully execute the first withdraw, decrementing balance, after which the trace in τ_0 decrements balance again. However, we have left-projection as $\tau_0; \tau_2$ leads to the same state as τ_0 (because the lock is taken when τ_2 executes and there is only one decrement of balance).

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

Definition 12 (Left-movement) *Given two segments τ_1 and τ_2, we say that $\tau_1; \tau_2$ left-moves if and only if for all $\sigma \in Feasible(\tau_1; \tau_2)$ we have that either τ_1 commutes or left-projects with τ_2 for the state σ.*

Definition 13 (Right-movement) *Given two segments τ_1 and τ_2, we say that $\tau_1; \tau_2$ right-moves if and only if for all $\sigma \in Feasible(\tau_1; \tau_2)$ we have that either τ_1 commutes or right-projects with τ_2 for the state σ.*

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

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

Definition 14 (Zero-projection) *Given two segments τ_1 and τ_2, we say that τ_1 zero-projects with τ_2 for the state $\sigma \in Feasible(\tau_1; \tau_2)$ if and only if, if $\sigma - \tau_1; \tau_2 - \sigma'$, then $\sigma = \sigma'$.*

Zero-projection expresses that two segments transition from a state σ to a final state equivalent to σ. For example, assuming we are in a state where $0 \leq x \leq 1000$, the segments $\tau_1 : x' = x * 2$ and $\tau_2 : x' = x/2$ zero-project, but they do not left or right-project or commute.

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

Definition 15 (Movement) *Given two segments τ_1 and τ_2, we say that $\tau_1; \tau_2$ moves if and only if for all $\sigma \in Feasible(\tau_1; \tau_2)$ we have that either τ_1 commutes, right-projects, left-projects or zero-projects with τ_2 for the state σ.*

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

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

Chapter 4

Static analysis

This chapter presents our static analysis to prove that a given program satisfies the $sECF_{FS}$ property. We first introduce in Section 4.1 the basic approach to prove that one call node is *solvable* in isolation, i.e., it does not break the ECF property. In order to handle all call nodes in the program, we extend in Section 4.2 our approach with an operation that, once a call node has been solved, allow us to join its left and right segments to gain further accuracy. Finally, section 4.3 presents the treatment of *revert* executions which undo all changes and revert to the initial state.

Our techniques have to ensure that given a trace we can always find an alternative callback-free one. To this end, we first prove that if we can *solve* (i.e. find a final state equivalent callback-free trace) all traces with callbacks only at *depth* one (i.e. no callbacks inside another callback), then we can solve all traces. Moreover, we only have to show that we can solve traces where all callbacks occur inside a single function, considering all its call nodes. This result generalizes to final state equivalence the reduction to simples traces of [19] that was based on conflict-equivalence.

Definition 16 (simple trace) *Given a trace $t_1; \ldots; t_n$ with $t_i \in TR$, the depth of t_i in $t_1; \ldots; t_n$ is the number of entry nodes visited minus the number of exit nodes visited in $t_1; \ldots; t_{i-1}$. The depth of the trace is the highest depth of all its t_i. A trace is* simple *if: (1) it is of depth one, and (2) after removing all t_i that are callbacks we obtain a trace $t_{i_1}; \ldots; t_{i_m}$ that is a trace of a procedure p of the program, and we say that it is a simple trace of p.*

Lemma 1 *If all executions of simple traces of a program Pr are $dECF_{FS}$ then Pr is $sECF_{FS}$.*

Proof: *In Section A.1.* $\square$

The proofs of all our results are provided in Appendix A. Some of them require auxiliar lemmas and definitions, so for readability reasons we have decided to keep them in an appendix.

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

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

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

4.1 Solvable Call Node

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

In this definition we consider that the initial and end nodes of a procedure are call nodes too, as we did before we introduce the definition of segment in Def. 4

Definition 17 (Minimal left/right segments) *Given a call node c of a procedure p of Pr with a set of call nodes C, we define the set of minimal left/right segments resp. as follows:*

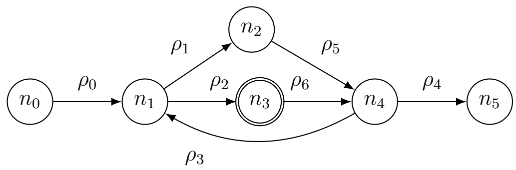
- $SLeft(c) = \cup_{c'}\{TR(c', c) | \forall t = \rho_1; \ldots; \rho_n \in TR(c', c), \forall j \in \{2 \ldots n\}.\ source(\rho_j) \notin C\}$
- $SRight(c) = \cup_{c'}\{TR(c, c') | \forall t = \rho_1; \ldots; \rho_n \in TR(c, c'), \forall j \in \{1 \ldots n-1\}.\ target(\rho_j) \notin C\}$

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

Example 6 *Let us illustrate these sets on the examples of the work. First, we consider the example in Fig. 1.4, which is the fixed DAO, and whose TS is given in Fig. 2.1. Here, in addition to the initial node n_0 and the final node n_4, there is a single call node n_3. Then, $SLeft(n_3) = \{\{\rho_0; \rho_1; \rho_3\}\}$ and $SRight(n_3) = \{\{\rho_4; \rho_5\}\}$. For the original DAO problem in Fig. 1.1 (where there is no use of the* `lock` *variable), we have the same $SLeft(n_3)$ and $SRight(n_3)$ since its TS is like Fig. 2.1, but omitting transition ρ_6 and all conditions or assigments involving the* `lock` *variable.*

Example 7 *We can apply these notions to call nodes that appear in loops. Consider a function with one call node in the loop:*

```
function loop1(int val) {
  int aux = 0;
  do {
    if (val != 0){
      aux += val;
      val++;
    }
    else{
      aux = call();
    }
  }
  while (aux < 10);
}
```



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

$$\mathit{SLeft}(n_3) = \{\{\rho_0;\rho_3\ ,\ \rho_0;t;\rho_2\ ,\ \rho_0;t;t;\rho_2\ ,\ \dots\},\{\rho_6;\rho_3\ ,\ \rho_6;t;\rho_2\ ,\ \rho_6;t;t;\rho_2\ ,\ \dots\}\}$$
$$\mathit{SRight}(n_3) = \{\{\rho_6;\rho_4\ ,\ \rho_6;t;\rho_4\ ,\ \rho_6;t;t;\rho_4\ ,\ \dots\},\{\rho_6;\rho_3\ ,\ \rho_6;t;\rho_2\ ,\ \rho_6;t;t;\rho_2\ ,\ \dots\}\}$$

The static analysis needs to consider sequences of n callbacks, e.g., of the form $\tau_1; f_1; \dots; f_n; \tau_2$, where the f_i (for $i = 1, \dots, n$) are function segments for the callbacks to all n different procedures in the program. As we do not know which call(s) might arrive at runtime, all permutations of the f_i must be considered. Thus, we cannot just apply the operations for movements in Chapter 3 to each of the functions since it could be the case that, for instance, $f_1;\tau_2$ right-moves (but $\tau_1; f_1$ does not left-move) and $\tau_1; f_n$ left-moves (but $f_n;\tau_2$ does not right-move). A necessary condition in this case is that $f_1; f_n$ must move as well, since f_1 may appear before f_n. However, it is insufficient since there are additional calls in the middle $(f_2, \dots, f_{n-1})$ whose own ability to move with τ_1 and τ_2 must be preserved independently of f_1 and f_n. Therefore, this imposes additional movement properties of f_1 over all of $f_2, \dots, f_n$ and of f_n over $f_1, \dots, f_{n-1}$. The example in Fig. 4.2 illustrates this situation for only two calls. There, we have a single call node n_1, τ_1 is the segment that contains only the trace with ρ_0 and τ_2 is the segment that contains only the trace with ρ_1. Thus, although f_1 commutes with τ_2 (but not with τ_1) and f_2 commutes with τ_1 (but not with τ_2), because $f_1; f_2$ does not move, any trace represented by the segment-sequence $\tau_1; f_1; f_2; \tau_2$, does not have a final state equivalent callback-free trace, and hence the program is not ECF. This is the reason why we must require $f_1; f_2$ to move.

The aforementioned situation requires leveraging the projection and commutation operations to handle multiple callbacks at a call node. Basically, we classify in Def. 18 the calls at this node as either *left-solvable* (commute or project with the

```
contract Example_no_ECF {
  uint c;
  uint s;

  function inc() {
    c = c+1;
    call();
    s = s+1;
  }

  function f_1() {
    s = s+1;
    c = 0;
  }

  function f_2() {
    s = 0;
    c = c+1;
  }
}
```

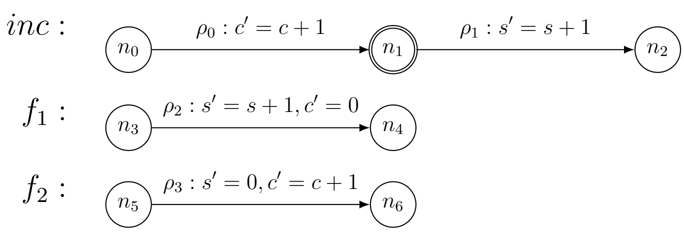


Figure 4.2: Example of functions f_1 and f_2 that do not commute. The contract is not ECF (trace $\rho_0; \rho_2; \rho_3; \rho_1$)

minimal left segment) and/or *right-solvable* (commute or project with the minimal right segment), and then Def. 19 requires movement properties for those that are exclusively left- or right-solvable.

Definition 18 *Given a call node c of a procedure p, we define sets of function segments $\mathit{Left}(c)$ and $\mathit{Right}(c)$ as follows:*

1. *for every function g in Pr we have that $g \in \mathit{Left}(c)$ iff $\tau; g$ left-moves for all $\tau \in \mathit{SLeft}(c)$.*

2. *for every function g in Pr we have that $g \in \mathit{Right}(c)$ iff $g; \tau$ right-moves for all $\tau \in \mathit{SRight}(c)$.*

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

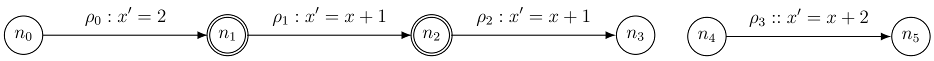
Definition 19 *Given a call node c of a procedure p, and denoting the set of functions of Pr by $F(Pr)$, we define sets of function segments $MLeft(c)$ and $MRight(c)$ using the least fixed point operator as follows:*

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

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

Definition 20 (Solvable call node) *Given a program Pr, we say that a call node c of Pr is solvable if $MLeft(c) \cap MRight(c) = \emptyset$.*

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



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

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

Definition 21 ($sECF_{SS}$) *Given a program Pr, it is $sECF_{SS}$ if and only if for all procedures p in Pr with call nodes C we have that, for every c, c' in C such that c' is reachable from c or $c' = c$, it holds that $MRight(c) \cap MLeft(c') = \emptyset$.*

Example 8 *Consider again the example in Figs. 1.4 and 2.1 which is sECF. In Example 6, we have seen that $SLeft(n_3) = \{\{\rho_0; \rho_1; \rho_3\}\}$ and $SRight(n_3) = \{\{\rho_4; \rho_5\}\}$. Now let τ_d be the function segment of* deposit *and τ_w be the function segment of* withdraw*. We have that $Left(n_3) = \{\tau_d, \tau_w\}$ as for both $\{\rho_0; \rho_1; \rho_3\}; \tau_d$ and $\{\rho_0; \rho_1; \rho_3\}; \tau_w$ left project to $\{\rho_0; \rho_1; \rho_3\}$, since ρ_1 sets* lock *to* true *(which is not changed in ρ_3), and in such state both* deposit *and* withdraw *do nothing. Then all functions are in $Left(n_3)$ and hence the program is $sECF_{SS}$.*

Now, we show why the example in Fig. 1.1 (which is not ECF) is not $sECF_{SS}$. As seen in Example 6 we have that $SLeft(n_3) = \{\{\rho_0; \rho_1; \rho_3\}\}$ and $SRight(n_3) = \{\{\rho_4; \rho_5\}\}$, and recall that we do not use the lock *variable and we do not have transition ρ_6. Here, we have that τ_w neither belong to $Left(n_3)$ nor to $Right(n_3)$, since without using* lock*, we cannot project or commute.*

Theorem 1 *If a program is $sECF_{SS}$ then it is $sECF_{FS}$.*

Proof: *In Section A.2.* □

4.2 Segments Join

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

```
contract Ex_need_join {
  uint c;

  function discount2() {
    c = c − 1;
    call();
    c = c − 1;
    call();
    c = 0;
  }

  function multiply(){
    c = c ∗ 2;
  }
}
```

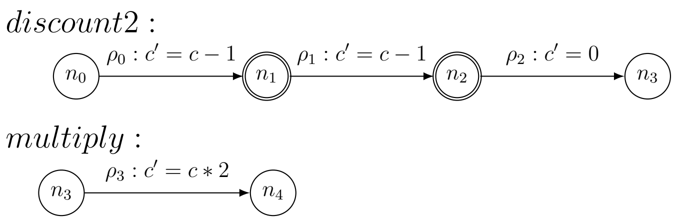


Figure 4.3: ECF contract that requires call node removal and cannot be proven using minimal segments

Example 9 *Consider the example in Fig. 4.3 whose procedure* discount2 *has three transitions and two call nodes, namely n_1 and n_2 (where callbacks can enter), while the function* multiply *has a single transition and no call nodes. Assume that our trace has a callback (to* multiply*) at each call node: $\rho_0; \rho_3; \rho_1; \rho_3'; \rho_2$ (we have primed the second use of* multiply*). The minimal segments of* discount2 *are (i) the set of traces from n_0 to n_1, i.e. $\tau_0 = \{\rho_0\}$, (ii) the set of traces from n_1 to n_2, i.e. $\tau_1 = \{\rho_1\}$, and (iii) the set of traces from n_2 to n_3, i.e. $\tau_2 = \{\rho_2\}$. We use f for the function segment $\{\rho_3\}$ of* multiply*. Now, the segment-sequence representing our trace is $\tau_0; f; \tau_1; f; \tau_2$. We start by handling the second call node, n_2, first. We can do either commutation of f over τ_2 or we can do right-projection of $f; \tau_2$ to τ_2, e.g., in the latter we have solved the call node n_2, and the new segment-sequence (representing final state equivalent traces to our trace) is $\tau_0; f; \tau_1; \tau_2$. But now we cannot go further and solve n_1 since we cannot apply any projection or commutation on $\tau_0; f$ or $f; \tau_1$. However, if we* *use the fact that n_2 has already been solved, we can consider that n_2 is no longer a call node, since it does not have callbacks in it, then our transition system would be:*

$$n_0 \xrightarrow{\rho_0 : c' = c-1} n_1 \xrightarrow{\rho_1 : c' = c-1} n_2 \xrightarrow{\rho_2 : c' = 0} n_3$$

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

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

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

We will thus consider that we can apply an operation to *remove call nodes* that enables a more accurate static analysis for procedures with multiple call nodes. However, once we introduce this operation, the order in which call nodes are solved might affect the accuracy of the analysis results. Assume we have a segment-sequence π with k callbacks ($n_1, \ldots, n_k$ ordered by their position at the execution). We establish a new order in which they are solved, by means of a permutation $i_1, \ldots, i_k$ of $1, \ldots, k$ which indicates that we will solve the callback nodes in the order $n_{i_1}, \ldots, n_{i_k}$. For instance, the order $2, 1$ leads to a solution in Example 9. The general concept we have is an order $<_O$ that indicates when a call node is solved before another, i.e. if $c' <_O c$ then we know that c' has been solved when we solve c. This means that when checking if c is solvable we have to first remove as call nodes from the transition systems all those call nodes c' such that $c' <_O c$. Now, we present a generalization of the $sECF_{SS}$ property to the case where we solve the call nodes in a given order. First we define the notion of solvable call node for a given order $<_O$.

Definition 22 (Orderly solvable call node) *Given a program Pr and an order $<_O$ on the call nodes of Pr. We say that a call node c of Pr is solvable wrt. $<_O$ if c is solvable after removing as call nodes from Pr all $c' <_O c$.*

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

Definition 23 ($sECF_{OS}$) *We say that a program Pr is $sECF_{OS}$ if there exists an order $<_O$ for the call nodes C of Pr such that all $c \in C$ are solvable with respect to $<_O$.*

Theorem 2 *If a program is $sECF_{OS}$ then it is $sECF_{FS}$.*

Proof: *In Section A.3.* □

Example 10 *Consider the example in Fig. 4.3 for the function* discount2 *whose TS is in Ex. 9, taking O as $n_2 <_O n_1$, we have that $SLeft(n_2) = \{\{\rho_1\}\}$ and $SRight(n_2) = \{\{\rho_2\}\}$, and $SLeft(n_1) = \{\{\rho_0\}\}$ and $SRight(n_1) = \{\{\rho_1; \rho_2\}\}$. Now, we can prove that both* discount2 *and* multiply *belong to $Right(n_2)$ and to $Right(n_1)$.*

The next result proves that the $sECF_{OS}$ approach is strictly more precise than $sECF_{SS}$. Moreover, it proves that if a program is $sECF_{SS}$ we can use any order to solve its call nodes.

Theorem 3 *If a program Pr is $sECF_{SS}$ then for any order $<_O$ for the call nodes C of Pr all $c \in C$ are solvable with respect to $<_O$.*

Proof: *In Section A.4.* □

4.3 Treatment for Revert Operations

Some environments, like Ethereum, include a *revert* operation that undoes all changes made in the current call and all its callbacks. We formalize them by means of *reverting transitions* that do not annotate the edge with assignments but rather with a *revert* label and its target location is always an exit point. It is clear that any execution of a simple trace t that reverts is $dECF_{FS}$: the final state is the same as the initial one, thus the execution is final-state equivalent to the empty one. We need to improve our $sECF_{SS}$ and $sECF_{OS}$ approaches taking into account this situation as the following example illustrates.

```
contract Ex_revert {
  uint c;

  function f() {
    c = 2;
  }

  function rev() {
    c = 1;
    call();
    if (c != 1)
      revert();
  }
}
```

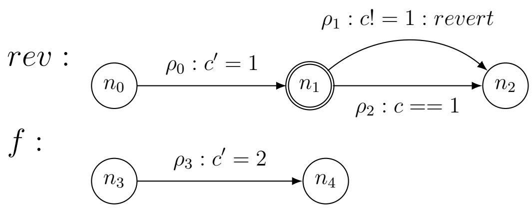


Figure 4.4: Example of contract with *revert* instructions. The program is ECF.

Example 11 *The contract in Fig. 4.4 is $sECF_{FS}$. We can check that any possible execution ξ of the program from an initial state σ either ends at the state $[c = 1]$, $[c = 2]$ or reverts. Hence, there exist complete callback-free executions equivalent to ξ: the execution of the function* rev*,* f *and the empty execution respectively. However, it is not $sECF_{OS}$. There is a single call node n_1 and the sets $SLeft(n_1)$ and $SRight(n_1)$*

only contain the segments $\tau_0 = \{\rho_0\}$ *and* $\tau_1 = \{\rho_1\ ,\ \rho_2\}$*, respectively. It is clear that the function* rev *belongs to* $Left(n_1)$ *and* $Right(n_1)$*, but* f *does not belong to any of this sets, as* $\tau_0; f_1$ *and* $f_1; \tau_1$ *does not left or right-move, so the contract is not* $sECF_{OS}$.

The example shows that working only with commutation or projection of segments is not appropriate for executions that revert. We take a more general definition of right-movements that gives a special treatment to reverting executions: the revert/right-movements. We only need to modify the definition of right-movement because it is impossible that the left-segment of a call node reverts, as the target node of a revert transition is always an exit point.

Definition 24 (Revert/Right-movement) *Given two segments* τ_1 *and* τ_2*, we say that* $\tau_1; \tau_2$ *reverts or right-moves if and only if for all* $\sigma \in Feasible(\tau_1; \tau_2)$ *either* τ_1 *commutes or right-projects with* τ_2 *for the state* σ *or the execution of* $\tau_1; \tau_2$ *from the state* σ *reverts.*

Now, we adapt the $sECF_{SS}$ and $sECF_{OS}$ approaches by using the revert/right-movements in order to check the functions that belong to $Right(c)$, instead of the right-movements.

Definition 25 *Given a call node* c *of a procedure* p*. We define sets of function segments* $Left(c)$ *and* $Right(c)$ *as follows:*

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

2. *for every function* g *in* Pr *we have that* $g \in Right(c)$ *iff* $g; \tau$ *reverts or right-moves for all* $\tau \in SRight(c)$.

Example 12 *Consider again the example in Fig. 4.4. The execution of* $f_1; \tau_1$ *from any state reverts, thus* $f_1; \tau_1$ *reverts or right moves. Hence,* $f_1 \in Right(c_1)$ *and the program is* $sECF_{OS}$.

Chapter 5

Callback invariant

Motivated by challenging contracts found in the Ethereum environment (similar to the one in Example 13 to follow), we introduce the notion of *callback invariant* as a way to increase the accuracy of the $sECF_{SS}$ and $sECF_{OS}$ approaches. A callback invariant is a property that holds when we first arrive at the call node, but also after executing any possible sequence of callbacks. The notion of callback invariant can be extended to several call nodes, having an invariant per call node. Note that we can always take *true* as invariant in a call node if we do not need it. Then, taking *true* as a (fictitious) invariant for the initial node, we have that the invariants must be preserved by all transitions between two callnodes (or the initial node) and they need to be preserved when executing all functions. Being precise:

Definition 26 *Given a procedure p with call nodes C and start node n_0, we say that $I(C)$, from nodes to properties, is callback invariant of C, if, taking $I(n_0) = true$, we have that*

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

Example 13 (Monotone lock) *The contract appearing in Figure 5.1 is a simplification of the Synthetix case study (a fragment of it is shown in Section 6.2) with no loops. It uses a counter to prevent callbacks that can lead to harmful results. This contract only has two call nodes: n_2 and n_3. The node n_2 is solvable according to the $sECF_{OS}$ approach, but n_3 is not. The minimal segments of the node n_3 are SLeft(n_3) only containing the segment $\tau_l = \{\rho_0; \rho_3\}$ and SRight(n_4) containing $\tau_r = \{\rho_5; \rho_6,\ \rho_5; \rho_7\}$. This node is not solvable: the function* exchange *does not left-move nor revert/right-move with the segments τ_l and τ_r, resp. The states that are problematic for the right-movements are only the ones where $\sigma[count] = \sigma[lc] - 1$. For any other state, after executing* exchange *we will obtain a state σ' such that $\sigma'[count] \neq \sigma'[lc]$, thus the execution will* revert. *Hence, if we could prove that no execution gets to the call node n_3 in the problematic state described above, we would be able to prove that the contract is $sECF_{FS}$.*

```
pragma solidity ^0.4.24;
contract Bank {
  mapping (uint => uint) public
       deposits;
  uint initIndex;
  uint count = 0;

  function exchange(uint remaining) {
    count += 1;
    uint lc = count;
    deposit = deposits[initIndex];
    if(deposit == 0){
      initIndex++;
    }
    else if(deposit > remaining){
      uint newAmount= deposit −
           remaining;
      deposits[initIndex] =
           newAmount;
      user.send(remaining);
    }
    else{
      deposits[initIndex] = 0;
      user.send(deposit);
      initIndex++;
    }
    require(lc == count);
  }
}
```

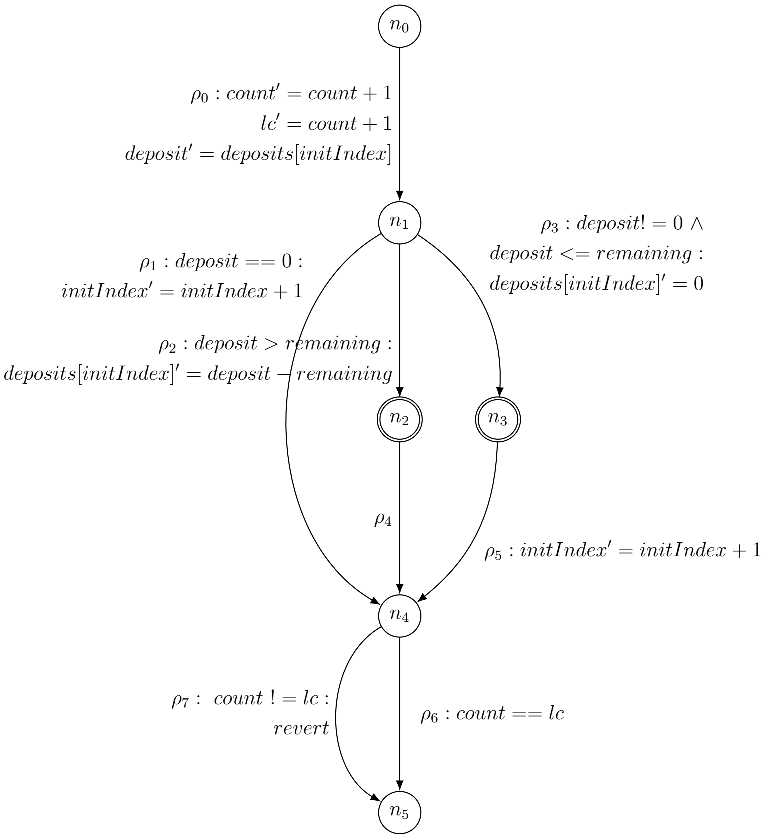


Figure 5.1: Simplified Synthetix contract non verifiable using the $sECF_{OS}$ approach

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

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

Definition 27 (Left-movement with precondition) *Given two segments τ_1 and τ_2, and a property P, we say that $\tau_1; \tau_2$ left-moves assuming the precondition P if and only if for all $\sigma \in Feasible(\tau_1; \tau_2)$ such that σ satisfies P we have that either τ_1 commutes or left-projects with τ_2 for the state σ.*

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

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

Then, we just have to adapt the definitions of $Left(c), Right(c)$ to use this new movements and take the sets $MLeft(c)$ and $MRight(c)$ according to them.

Definition 28 *Given a procedure p with call nodes C, a call node $c \in C$ and I from nodes to properties. We extend function $Left(c, I)$ and $MLeft(c, I)$ to work with invariants as follows:*

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

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

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

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

Definition 29 ($sECF_{IOS}$) *Given a program Pr, it is $sECF_{IOS}$ if and only if there exist an order $<_O$ for the call nodes C of Pr and a callback invariant I of C such that all $c \in C$ are solvable assuming I with respect to $<_O$.*

Theorem 4 *If a program is $sECF_{IOS}$ then it is $sECF_{FS}$*

Proof: *In Section A.5.* □

Finally, we can prove that the above contract is ECF. The property I is a callback invariant of the call node n_3 and $MLeft(n_3, I) = \emptyset$, as the call node n_2 is also solvable, we conclude that the contract is $sECF_{IOS}$.

Chapter 6

Implementation and Experimental Evaluation

Our implementation decompiles smart contracts given as EVM bytecode and produces code in an intermediate representation amenable to static analysis and the generation and discharge of verification conditions using SMT solvers, such as Z3 [14]. Furthermore, since the EVM bytecode does not contain a notion of procedures or functions, and the Solidity compiler generates generic 'dispatch' code to jump to the appropriate function code, we split out the function implementations from the large EVM bytecode. Currently, we have bounded support for loops using finite unrolling, we are working on the general extension.

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

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

We have run our benchmarks on an Amazon AWS c5n.2xlarge machine. The SMT solver used is Z3, with a timeout of 60 seconds per query. To each callnode we set a timeout of 5 minutes for analyzing it, requiring all needed SMT queries to run within the time span. Callnodes are detected in a conservative manner–any instance of a call instruction, except for STATICCALL, is considered a callnode. The STATICCALL instruction is not considered a callnode because it enforces the VM to avoid any writes to the global state in all calls until the STATICCALL returns, and therefore trivially projects. This method assumes a completely open environment, that considers only the contract checked. As we show later, many contracts use other contracts as libraries and thus establish properties that should hold when one contract calls the library. In that case, it is possible to ignore certain callnodes, because the callee contract is guaranteed not to trigger a callback. In result, this would lead to a greater number of verified contracts (those marked * in Table 6.1).

Delegate calls Two special instructions in the EVM bytecode are DELEGATE-CALL and CALLCODE. These instructions allow to execute 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 callnodes in order to prove ECF, but it should be noted that if a contract contains such delegating instructions, then ECF does not guarantee sound modular reasoning.

6.1 Experimental evaluation

To validate the usefulness of our approach, we picked a benchmark of smart contracts that are often used and invoked. To that end, we extracted the top-150 contracts based on volume of usage, as of December 31st, 2019 [1]. A total of 132 contracts in total were successfully decompiled, but 38 contracts did not contain callnodes. Since the ECF property that we check is based on the results for all functions, we give in Table 6.1 the summarized results for all functions extracted out of all contracts. Section 6.1.1 gives detailed results for 94 contracts that were successfully decompiled and have at least one function with a callnode.

Out of the total 2733 functions extracted, 386 contained callnodes, and thus are candidates to ECF checking. Out of these 386 functions, 238 are verified to be ECF, 105 are reported as violating ECF, and 43 time out before a definite answer is returned. We manually analyzed 72 of the violations (for the other 33, 16 did not have source code and 17 were too complex for human reasoning). 10 functions are confirmed to be true violations. 36 are violated because of the over-conservative choice of callnodes. After studying the contract systems, we believe those callnodes can be omitted (and thus become ECF verified). 19 are violated due to over-approximations in the implementation, and we plan to re-run those tests after the accuracy is improved. 7 were violated but they are not $sECF_{OS}$, and thus cannot be proved to be ECF using our approach.

6.1.1 Detailed results

Table 6.2 details our results for the 94 contracts out of the 150 that were successfully decompiled and have at least one function with a callnode.

The table shows for each contract: its size (in number of edges in the CFG of the decompiled bytecode), the number of non-read only functions, the number of functions with callnodes, the total number of callnodes in all functions, the number of SMT queries performed by the implementation, the time it took to process the contract (in seconds), and the number of functions that were verified to be ECF,

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

	# fs	% all fs	% fs w. CN	Avg. T (sec.)
ECF Verified	238	8.7	61.7	30
ECF Violated	105	3.8	27.2	132
Timeout	43	1.6	11.1	1240

Analysis of violations	
Confirmed violations	10
Could not manually check	33
FPs due to callnode choice*	36
FPs to $sECF_{OS}$	7
FPs due to implementation	19

Table 6.1: Summarized ECF results. 'CN' stands for 'callnode', and 'f' for 'function'. We only consider functions that are candidates to ECF checking (>0 CNs).

out of functions with callnodes. Contracts with the DELEGATECALL instruction are marked with a *.

The complexity of the ECF check depends on both the number of non-read only functions for which movement checks are required, and the number of callnodes that have to be processed, listed in *non-RO* and *CNs* columns. The *fs with CNs* column lists the number of functions for which we need to determine if they are ECF or not. We present these values for each contract. The last three columns shows how many functions were found to be ECF, non-ECF, or that the implementation timed-out, out of all functions with callnodes.

Finally, the table lists the running time for analyzing the contract (in seconds) as well as the number of required calls to the SMT solver.

Table 6.2: Results for 94 contracts with callnodes.

ID	Size	non-RO	fs with CNs	CNs	Queries	Time	ECF	Non-ECF	Timeout
1*	86	4	1	1	0	0	1	0	0
2	166	7	1	1	0	0	1	0	0
3	431	16	2	2	0	0	2	0	0
5	345	11	1	1	0	0	1	0	0
6	1105	26	11	15	162	2603	5	4	2
7	2448	29	17	33	229	6081	4	6	7
9	40	2	2	2	0	0	2	0	0
10	188	8	2	2	0	0	2	0	0
11	241	6	2	2	0	0	2	0	0
12	143	6	1	1	0	0	1	0	0
13	723	15	8	10	16	46	0	8	0
15	2567	14	12	14	20	2227	0	8	4
16*	2	1	1	1	0	0	1	0	0
17	538	16	3	4	103	220	3	0	0
18	294	11	1	1	0	0	1	0	0
19	709	19	11	17	29	1670	5	5	1
20	120	2	1	1	0	0	1	0	0
22	1349	19	5	5	16	798	1	4	0
23	538	16	3	4	103	230	3	0	0
24	702	9	4	8	5	817	2	1	1
27*	584	14	3	4	50	155	2	1	0
28	126	11	4	4	0	0	4	0	0
29	2369	12	10	12	16	1882	0	6	4
33	452	16	10	14	209	459	10	0	0
34	383	7	2	3	24	191	2	0	0
35	2533	38	18	105	712	31796	5	2	11

Continued...

ID	Size	non-RO	fs with CNs	CNs	Queries	Time	ECF	Non-ECF	Timeout
36	169	5	2	2	0	0	2	0	0
37	169	7	2	2	0	0	2	0	0
39	638	15	1	1	0	0	1	0	0
41	129	7	1	1	0	0	1	0	0
42	208	6	1	1	0	0	1	0	0
43	2246	32	15	68	128	6954	1	10	4
44	260	11	1	1	0	0	1	0	0
47	159	6	1	1	0	0	1	0	0
48	215	6	1	1	0	0	1	0	0
51	143	6	1	1	0	0	1	0	0
52	156	6	1	1	0	0	1	0	0
53	875	25	7	7	0	0	7	0	0
54	156	6	1	1	0	0	1	0	0
56	72	1	1	4	2	5	0	1	0
57	157	3	3	18	68	1405	2	1	0
58	648	9	4	5	4	70	3	1	0
59	1110	21	4	4	36	300	3	0	1
61	375	12	3	3	0	0	3	0	0
62	1168	22	14	48	290	5650	9	5	0
63	209	8	1	1	0	0	1	0	0
64*	1387	78	52	82	2440	3621	48	4	0
65	630	16	10	13	56	687	8	1	1
66	479	12	7	7	14	21	0	7	0
67	266	11	1	1	0	0	1	0	0
69	506	16	9	9	18	35	0	9	0
70	629	16	7	8	8	740	4	2	1
71	192	8	1	1	0	0	1	0	0
72*	153	6	6	7	0	0	6	0	0
75	140	4	1	1	0	0	1	0	0
76	42	1	1	4	2	5	0	1	0
77	161	4	1	1	0	0	1	0	0
78	1153	19	5	5	0	0	4	1	0
80	282	11	2	2	4	8	0	2	0
81	1286	12	3	3	0	303	2	0	1
83	350	7	2	3	24	90	2	0	0
84	279	7	1	1	0	0	1	0	0
89	350	14	2	2	0	0	2	0	0
90	341	13	1	1	0	0	1	0	0
91	13	1	1	1	0	0	1	0	0
92	132	6	1	1	0	0	1	0	0
93	151	4	1	1	0	0	1	0	0
97	161	4	1	1	0	0	1	0	0
98	181	5	4	17	13	280	0	4	0
99	1185	11	7	13	63	470	2	5	0
100	456	11	5	5	2	34	4	1	0
101	327	10	4	4	20	418	3	0	1
102	289	9	1	1	10	17	0	1	0
104	331	11	2	2	0	0	2	0	0
105	170	6	1	1	0	0	1	0	0
107	577	8	1	1	0	0	1	0	0
108	475	16	7	7	4	10	6	1	0
109	290	7	1	1	0	0	1	0	0
110	646	13	1	1	0	0	1	0	0
114	39	1	1	4	2	8	0	1	0

Continued...

ID	Size	non-RO	fs with CNs	CNs	Queries	Time	ECF	Non-ECF	Timeout
116	780	7	2	2	0	0	2	0	0
117*	153	6	6	7	0	0	6	0	0
119	215	9	1	1	0	0	1	0	0
120	470	11	1	1	0	0	1	0	0
122	188	8	1	1	0	0	1	0	0
123	270	10	1	1	0	0	1	0	0
124	1226	15	4	12	0	3361	2	0	2
125	185	6	6	7	0	0	6	0	0
126	313	13	1	1	0	0	1	0	0
128	129	7	1	1	0	0	1	0	0
129	191	9	3	3	0	0	3	0	0
130	39	1	1	4	2	6	0	1	0
131	45	1	1	4	2	6	0	1	0
132	279	8	5	7	24	675	3	0	2

End of table.

6.2 Challenging real case study

The vast majority of the contracts analysed in Table 6.1 are rather simple. Therefore, the reader may conclude that all smart contracts are simple, which is not our experience. Some of the valuable smart contracts actually implement complex logic, which makes checking ECF and other properties quite hard. One such example is the reentrancy bug [6] in *Synthetix* [31]–a high-volume De-Fi[2] application.[3] In Figures 6.1 and 6.2 an excerpt of the buggy code and two potential fixes preventing callbacks are given. Our technique can mechanically verify both: one of them as-it-is, the other using callback invariants. To the best of our knowledge, none of the techniques available are able to show that immunity to reentrancy attacks is true for the fixed contract. Indeed, we also compared our implementation to other existing tools whose premise is to handle 'reentrancy bugs': *Securify2* [35] and *Slither* [15]. Notably the properties checked by these tools are more restrictive than ECF: *Securify* and *Slither* check that there are no global state updates following a call instruction. When we ran this case study (as well as our lock-based example of Figure 1.4), *Securify* and *Slither* both failed to show that it is actually safe (*Securify* times out after hours of running on the Amazon machine). The same holds for the simplified version of our case study as appears in Figure 5.1.

Excerpt of Solidity code for our case study. The code uses *modifiers* to prevent callbacks that can lead to harmful results. When the code of a function using a modifier is invoked (observe that this is stated in the function header), the modifier is executed by replacing the hole "_" by the code of the invoked function.

[2]Decentralized Finance

[3]according to [32], rated 2nd in locked USD value, with $116.7M locked as of May 5th, 2020.

```
function exchangeEtherForSynths() public payable nonReentrant rateNotStale(ETH) notPaused returns (uint) {
  require(msg.value <= maxEthPurchase);
  uint ethToSend;
  uint requestedToPurchase = msg.value.multiplyDecimal(exchangeRates().rateForCurrency(ETH));
  uint remainingToFulfill = requestedToPurchase;
  for (uint i = depositStartIndex; remainingToFulfill > 0 && i < depositEndIndex; i++) {
    synthDeposit memory deposit = deposits[i];
    if (deposit.user == address(0)) {
      depositStartIndex = depositStartIndex.add(1);
    }
    else {
      if (deposit.amount > remainingToFulfill) {
        uint newAmount = deposit.amount.sub(remainingToFulfill);
        deposits[i] = synthDeposit({user: deposit.user, amount: newAmount});
        totalSellableDeposits = totalSellableDeposits.sub(remainingToFulfill);
        ethToSend = remainingToFulfill.divideDecimal(exchangeRates().rateForCurrency(ETH));
        if (!deposit.user.send(ethToSend)) {
          fundsWallet.transfer(ethToSend);
        }
        synthsUSD().transfer(msg.sender, remainingToFulfill);
        remainingToFulfill = 0;
      }
      else if (deposit.amount <= remainingToFulfill) {
        delete deposits[i];
        depositStartIndex = depositStartIndex.add(1);
        totalSellableDeposits = totalSellableDeposits.sub(deposit.amount);
        ethToSend = deposit.amount.divideDecimal(exchangeRates().rateForCurrency(ETH));
        if (!deposit.user.send(ethToSend)) {
          fundsWallet.transfer(ethToSend);
        }
        synthsUSD().transfer(msg.sender, deposit.amount);
        remainingToFulfill = remainingToFulfill.sub(deposit.amount);
      }
    }
  }
  if (remainingToFulfill > 0) {
    msg.sender.transfer(remainingToFulfill.divideDecimal(exchangeRates().rateForCurrency(ETH)));
  }
  return requestedToPurchase.sub(remainingToFulfill);
}
```

Figure 6.1: The code of the exchangeEtherForSynths function. Without a lock as defined by the nonReentrant modifier in any one of the proposed fixes in Figure 6.2, it is not ECF.

```
/* Fix 1 (Simple Lock) */
bool l;
modifier nonReentrant() {
  require(!l);
  l = true;
  _;
  l = false;
}
```

```
/* Fix 2 (Monotone Lock) */
uint256 count;
modifier nonReentrant() {
  count += 1;
  uint256 lc = count;
  _;
  require(lc == count);
}
```

Figure 6.2: Two fixes for the exchangeEtherForSynths function.

Chapter 7

Related Work

We have presented a novel static analysis that proves modularity of the contract for any execution and can be applied to ensure effective-callback freedom prior to deployment. Reentrancy attacks have led to the most severe exploits in the blockchain and, as we have shown in the work, general techniques for ensuring modularity of programming languages can be used to detect ECF violations and avoid these malitious attacks. This kind of reentrancy problems were pinpointed as a possible source of correctness bugs [24, 3]. As discussed in Chapter 1, our work is inspired by that of [19] who pioneered the idea of ECF as means to immune modules (contracts) from reentrancy attacks and enable modular reasoning. However, the analysis of [19] is dynamic hence it cannot be used to verify ECF. In the rest of this chapter, we review other closely related work.

[25] present a framework, called FSolidM [9], that allows preventing reentrancy via a built-in locking mechanism. In contrast, we present a technique for verifying ECF, and thus the absence of reentrancy bugs, is language-agnostic while allowing judicious use of callbacks. [17] survey on recent theories and tools for formal verification of Ethereum smart contracts focusing on the F*-formalized small-step semantics presented by [18] and its Horn clauses-based abstraction. Most relevant to our work is over-approximation of the single-reentrancy property [29, 18] which, intuitively, states a contract is single-entrant if it cannot perform any more calls once it has been reentered. This restriction, however does not mean that callbacks may not have unique behaviors which cannot be exposed in callback-free executions. [35] report of a parametric static verification tool which can detect whether a contract violates a given security property encoded as a bad pattern in the contract's data-flow graph. To detect reentrancy-related bugs, they use a pattern which forbids writes after calls. Thus, their restrictions are more severe even than the ones imposed by conflict-based ECF. [21] identified a family of bugs in blockchain-based smart contracts, dubbed event-ordering (or EO) bugs, which are related to the dynamic ordering of contract events, i.e. calls of its functions. However, in contrast to our work, the ordering they investigate is between different transactions while our focus is on errors which occur within one transaction. Thus, the class of bugs we are after does not overlap with theirs. Also, our tool is static while theirs is based on dynamic (symbolic) testing. In MAIAN [26] the authors present a symbolic execution

tool for detecting contracts vulnerabilities such as ether leaking. Such vulnerbilities may intersect with reentrancy vulnerabilities (for example, the DAO's reentrancy attack leads to leaked ether).

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

[28] employ taint analysis on Ethereum traces to detect reentrancy vulnerabilities. The dynamic check implemented there is more precise than the as-of-then static analysis tools and its performance is similar to [19] for non CREATE-generated callnodes. (the latter did not include CREATE as a callnode candidate). A work by [16] define a language for patterns in Ethereum transactions representing malicious behaviors, and an instrumented Ethereum client that can detect such patterns in-vivo. Patterns can be added and removed based on voting in a smart contract. 4 out of 6 patterns presented in [16] are related to reentrancy vulnerabilities. Of most relevance to our work is the comparison between pattern-based detection of malicious attacks and semantic equivalence checking. In both the dynamic and static settings, the pattern-based approach can easily lead to over-approximation and false positives, while on the other hand not giving full clarity about the actual immunity of the code to malicious callbacks. In contrast, our approach, while more expensive computationally, gives strong guarantees about callbacks not being able to influence the execution in unexpected ways, while also being more resistant to false positives.

As [30] note when discussing the similarity of smart contracts to concurrent objects, enabling modular verification is one of the highlighted challenges. A key benefit of our semantic equivalence based approach, when compared to pattern-based techniques, is that it enables to modularly check properties of ECF contracts. For example, [38] present VeriSol, a tool for static verification of smart contracts against a state machine model specification and an access-control policy. The analysis is capable of inferring *contract invariants*—properties of the state of the contract which are true when none of its procedures is pending. However, the analysis is not modular. We believe that our approaches can be combined so that once the contract is verified as ECF, VeriSol can infer its class invariants in a sound modular way. A different approach to modularity is given in [11] by Cecchetti et al., defining reentrancy as an information-flow property, and reentrancy security as a property that guarantees invariants inductiveness even in the presence of callbacks. Their approach has the benefit of finer-grained policies, enabling supporting systems that consist of multiple contracts, but also requires the user to annotate 'critical sections' in the code.

Complementary approaches to modularity is to check an invariant of the program, e.g., [22, 5].

As regards the state equivalence check, we have implemented an SMT-based technique similar to the ones proposed to check commutativity in the context of model checking of concurrent programs (see, e.g., [37, 1]). However, our method is generic wrt. the particular check used and we will benefit for future improvements in this domain. For example, Bansal et al. [4] present a refinement-based technique for synthesizing commutativity conditions for operations on representations (implementations) of abstract data types (ADTs). The algorithm is generalized to handle left/right-movers [23]. We utilize commutativity checks as a "black box" in our algorithms. Thus, in that respect our works are complementary. Nevertheless, the projection checks and the gradual simplification of the commutativity checks done in the treatise algorithm are novel.

Chapter 8

Conclusions and Future Work

Reentrancy attacks represent one of biggest threats to smart contracts, as we discussed in Chapter 1. They exploit the use of callbacks to break the modularity of the contracts, generating unexpected behaviours. The main contribution of this master thesis is the first static analysis for verifying ECF.

First, we introduced the notions of segment and the commutation, projection and segment-join operations. These definitions are novel and allow us to characterize and work with all traces that can arise from a fragment of code, instead of working with individual traces. This is one of the key points of our static analysis.

Second, we presented our static analysis. It is based on proving commutativity and projection between all fragments of code between call nodes and the procedures of the module, that are the possible callbacks. As we discussed in Section 1.2, there exist techniques for proving that a given execution is ECF; however, to the best to our knowledge, this is the first work to present a technique for statically verifying this property. Our analysis can help developers with feedback on potential vulnerabilities of the contract prior to deployment. This is fundamental in environments like Ethereum, where once a contract is deployed it can not be modified.

We also introduced the notion of callback invariant as a way to increase the accuracy of the analysis. Callback invariants are properties that hold when we first arrive at the call node, but also after executing any possible sequence of callbacks. We extended our static analysis to take into account the information that callback invariants give us to check the commutation and projection of the callbacks more precisely.

Our experimental results show that our approach can be applied to many real contracts, and it is able to prove modularity where other methods fail. Our technique is able to verify typical solutions to avoid reentrancy attacks, like the ones shown for the Synthetix contract in Figure 6.2, that can not be proven using previous approaches.

8.1 Future work

We have found several research directions we would like to follow in future work. The first direction is related to the implementation. As we discussed in Chapter 6, the actual algorithm is based on $sECF_{OS}$ but always considers the later-to-early callnode order. We would like to extend our implementation considering all possible orders. This represents a challenging problem as studying all orders may be impractical due to the number of required SMT queries. However, Theorem 3 suggests that it is possible to reduce the number of queries needed to study an order by taking advantage of the results of other orders already checked. Moreover, we would like to integrate in the implementation techniques for generating callback invariants. The actual implementation checks and takes into account the invariants that are given, but does not generate them.

The second direction is applying notions we have introduced as segments or callback invariants to other problems. They are not restricted to modularity checks, so we would like to study new possible applications. Generalizing these ideas and expressing new problems in terms of them represents an exciting future work.

Finally, the third direction is related to our experimental evaluation. We picked as benchmark set the most used smart contracts and checked that most of them are ECF. The majority of these contracts are rather simple, so as future work we would like to consider benchmark sets containing valuable contracts implementing complex logic, like the *Synthetix* contract that we presented in Section 6.2.

Bibliography

[1] Albert, E., Gómez-Zamalloa, M., Isabel, M. and Rubio, A. Constrained dynamic partial order reduction. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, 392–410. 2018.

[2] Albert, E., Grossman, S., Rinetzky, N., Rodríguez, C., Rubio, A. and Sagiv, M. Taming callbacks for smart contract modularity. 2020.

[3] Atzei, N., Bartoletti, M. and Cimoli, T. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, 164–186. Springer-Verlag New York, Inc., New York, NY, USA, 2017. ISBN 978-3-662-54454-9.

[4] Bansal, K., Koskinen, E. and Tripp, O. Automatic generation of precise and useful commutativity conditions. In *Tools and Algorithms for the Construction and Analysis of Systems* (edited by D. Beyer and M. Huisman), 115–132. Springer International Publishing, Cham, 2018. ISBN 978-3-319-89960-2.

[5] Beillahi, S. M., Ciocarlie, G., Emmi, M. and Enea, C. Behavioral simulation for smart contracts. To appear, 2020.

[6] Bernardi, T., Dor, N., Fedotov, A., Grossman, S., Nutz, A., Oppenheim, L., Pistiner, O., Sagiv, M., Toman, J. and Wilcox, J. Preventing reentrancy bugs - another use case for formal verification. 2020. `https://www.certora.com/blog/reentrancy.html`.

[7] Bernstein, P. A., Hadzilacos, V. and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5.

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

[9] Brent, L., Grech, N., Lagouvardos, S., Scholz, B. and Smaragdakis, Y. Ethainter: A smart contract security analyzer for composite vulnerabilities. To appear, 2020.

[10] Buterin, V. Critical update re: Dao vulnerability. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/, 2016. [Online; accessed 2-July-2017].

[11] Cecchetti, E., Yao, S., Ni, H. and Myers, A. Securing smart contracts with information flow. In *Third International Symposium on Foundations and Applications of Blockchain 2020*. 2020.

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

[13] Daian, P. 2016.

[14] De Moura, L. and Bjørner, N. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 337–340. Springer-Verlag, Berlin, Heidelberg, ???? ISBN 3-540-78799-2, 978-3-540-78799-0.

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

[16] Ferreira Torres, C., Baden, M., Norvill, R. and Jonker, H. ægis: Smart shielding of smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS 19, 2589–2591. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450367479.

[17] Grishchenko, I., Maffei, M. and Schneidewind, C. Foundations and tools for the static analysis of ethereum smart contracts. In *Computer Aided Verification* (edited by H. Chockler and G. Weissenbacher), 51–78. Springer International Publishing, Cham, 2018. ISBN 978-3-319-96145-3.

[18] Grishchenko, I., Maffei, M. and Schneidewind, C. A semantic framework for the security analysis of ethereum smart contracts. In *Principles of Security and Trust* (edited by L. Bauer and R. Küsters), 243–269. Springer International Publishing, Cham, 2018. ISBN 978-3-319-89722-6.

[19] Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M. and Zohar, Y. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL*, Vol. 2(POPL), 48:1–48:28, 2018.

[20] Hernandez, F. Understanding callbacks and promises. https://dev.to/_ferh97/understanding-callbacks-and-promises-3fd5, 2019. [Online; accessed 14-May-2020].

[21] Kolluri, A., Nikolic, I., Sergey, I., Hobor, A. and Saxena, P. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, 363–373. ACM, New York, NY, USA, 2019. ISBN 978-1-4503-6224-5.

[22] Li, A., Choi, J. A. and Long, F. Securing smart contract with runtime validation. To appear, 2020.

[23] Lipton, R. J. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, Vol. 18(12), 717–721, 1975. ISSN 0001-0782.

[24] Luu, L., Chu, D.-H., Olickel, H., Saxena, P. and Hobor, A. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 254–269. ACM, New York, NY, USA, 2016. ISBN 978-1-4503-4139-4.

[25] Mavridou, A. and Laszka, A. Tool demonstration: Fsolidm for designing secure ethereum smart contracts. In *Principles of Security and Trust* (edited by L. Bauer and R. Küsters), 270–277. Springer International Publishing, Cham, 2018. ISBN 978-3-319-89722-6.

[26] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P. and Hobor, A. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, 653–663. 2018.

[27] Palmer, D. Spankchain loses $40k in hack due to smart contract bug. `https://www.coindesk.com/spankchain-loses-40k-in-hack-due-to-smart-contract-bug`, 2018. [Online; accessed 11-May-2020].

[28] Rodler, M., Li, W., Karame, G. O. and Davi, L. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[29] Schneidewind, C., Scherer, M., Grishchenko, I. and Maffei, M. ethor: Practical and provably sound static analysis of ethereum smart contracts. To appear, 2020.

[30] Sergey, I. and Hobor, A. A concurrent perspective on smart contracts. In *Financial Cryptography and Data Security* (edited by M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore and M. Jakobsson), 478–493. Springer International Publishing, Cham, 2017.

[31] Synthetix. Synthetix - decentralised synthetic assets. 2020. `www.synthetix.io`.

[32] The Concourse Open Community. Defi pulse. `https://defipulse.com/`, 2019. [Online; accessed 11-May-2020].

[33] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E. and Alexandrov, Y. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 9–16. 2018.

[34] Tripp, O., Manevich, R., Field, J. and Sagiv, M. JANUS: exploiting parallelism via hindsight. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012* (edited by J. Vitek, H. Lin and F. Tip), 145–156. ACM, 2012.

[35] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F. and Vechev, M. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, 67–82. ACM, New York, NY, USA, 2018. ISBN 978-1-4503-5693-0.

[36] Turley, C. imbtc uniswap pool drained for $300k in eth. `https://defirate.com/imbtc-uniswap-hack/`, 2020. [Online; accessed 11-May-2020].

[37] Wang, C., Yang, Z., Kahlon, V. and Gupta, A. Peephole partial order reduction. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, 382–396. 2008.

[38] Want, Y., Lahiri, S., Chen, S., Pan, R., Dillig, I., Bprb, C. and Naseer, I. Formal specification and verification of smart contracts for azure blockchain. 2019. ArXiv:1812.08829v2.

[39] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. `http://gavwood.com/paper.pdf`, 2016. [Online; accessed 5-July-2017].

Appendix **A**

Proofs

A.1 Proof of Lemma 1

We first establish a simple result about complete well-formed traces:

Lemma 2 *Let t be a complete well-formed trace of depth d then either t is callback-free or $t = t_1; t_1^s; t_2; t_2^s; \ldots; t_{n-1}^s; t_n$ with t_i^s simple traces and $t_1; t_2; \ldots; t_{n-1}; t_n$ a complete well-formed trace of depth $d-1$.*

Proof: *Proof by induction on the length of the trace.*

□

Lemma 3 *If all executions of simple traces of a program Pr are $dECF_{FS}$ then Pr is $sECF_{FS}$.*

Proof: *We prove that any execution ξ is $dECF_{FS}$ by induction on the depth of the executed trace t.*

If t is callback-free then the execution is trivially $dECF_{FS}$.

Otherwise, we can apply Lemma 2, hence $t = t_1; t_1^s; t_2; t_2^s; \ldots; t_{n-1}^s; t_n$ with t_i^s simple traces and $t_1; t_2; \ldots; t_{n-1}; t_n$ a complete trace of depth $d-1$.

We denote the initial and final states of ξ by $\sigma_1 = start(\xi)$ and $\sigma_n = end(\xi)$. Then, for each $i = 1, \ldots, n-1$ there exist intermediate states σ_i and σ_i^s such that $\sigma_i - t_i - \sigma_i^s$ and $\sigma_i^s - t_i^s - \sigma_{i+1}$.

We are assuming that all executions of simple traces of the program are $dECF_{FS}$, hence for each one of executions $\sigma_i^s - t_i^s - \sigma_{i+1}$ there exists a complete callback-free trace $\overline{t_i}$ such that $\sigma_i^s - \overline{t_i} - \sigma_{i+1}$.

Finally, we consider the trace $\overline{t} = t_1; \overline{t_1}; \ldots; \overline{t_{n-1}}; t_n$. It is clear that $\sigma_1 - \overline{t} - \sigma_n$, hence this execution is final state equivalent to ξ. Moreover, $\overline{t}$ is complete well-formed and has depth $d-1$.

□

A.2 Proof of Theorem 1

A.2.1 Auxiliary proofs and definitions

Definition 30 ($\leq_F$) *Given two callback-free segment sequences* $\pi_1 = f_1; \ldots ; f_n$ *and* $\pi_2 = f'_1; \ldots ; f'_m$*, then* $\pi_2 \leq_F \pi_1$ *if and only the multisets that contain their functions verify* $\{f'_1, \ldots , f'_m\} \subseteq \{f_1, \ldots , f_n\}$.

Definition 31 ($D(\pi)$) *Given a program* Pr*, a callback-free segment sequence* $\pi = f_1; \ldots ; f_n$ *and two disjoint sets FLeft and FRight such that* $FLeft \cup FRight = F(Pr)$*, we define the value*

$$D(\pi) = \sum_{1=1\ldots n} d_i(\pi) \tag{A.1}$$

with $d_i(\pi) =$

$$d_i(\pi) = \begin{cases} 0 & \text{if } f_i \in FRight \\ \#j : 1 \leq j < i \wedge f_j \in FRight & \text{if } f_i \in FLeft \end{cases} \tag{A.2}$$

We are going to use this value to control the number of functions that are not correctly placed. The idea is that we have two kind of functions: the ones that have to be placed at the left of the sequence and the ones that have to be placed at the right. In case a function f_i has to be placed at the left, $d_i(\pi)$ expresses the number of misplaced functions with respect to f_i (they have to be placed at the right but appear before this function).

We can easily check that $D(\pi) = 0$ if and only if all the functions are correctly placed (there exists $k \in \{0..n\}$ such that $f_i \in FLeft$ if $i \leq k$ and $f_i \in FRight$ if $i > k$).

Proposition 1 *Given a program* Pr *and two sets FLeft and FRight such that*

- $FLeft \cap FRight = \emptyset$
- $FLeft \cup FRight = F(Pr)$
- *for all* $f \in FLeft$ *and* $g \in FRight$ $g; f$ *moves*

Then, given a callback-free execution ξ *of a sequence of functions* $f_1; \ldots , f_n$ *there exists a sequence of functions* $f'_1; \ldots ; f'_m$ *whose execution is final-state equivalent to* ξ *such that* $f'_1; \ldots ; f'_m \leq_F f_1; \ldots ; f_n$ *and* $D(f'_1; \ldots ; f'_m) = 0$.

Proof: *We are using the notation* $\pi_0 = f_1; \ldots ; f_m$*. We prove the result by induction on* $D(\pi_0)$.

The case $D(\pi_0) = 0$ *is trivial. So, we assume* $D(\pi_0) > 0$*. Then, there exists a value* i *such that* $f_i \in FRight$ *and* $f_{i+1} \in FLeft$.

Then, we consider the states σ_i, σ_{i+1} *before and after executing* $f_i; f_{i+1}$ *in the original execution, respectively.*

The function $f_i \in FRight$ *and* $f_{i+1} \in FLeft$*, hence* $f; g$ *moves. We distinguish four possibilities: commutation, left-projection, right-projection and zero-projection.*

- *If* f_i *and* f_{i+1} *commute for the state* σ_i *then* $\sigma_i - f_{i+1}; f_i - \sigma_{i+1}$.

  We consider the execution of the segment-sequence π_1 *where we substitute* $f_i; f_{i+1}$ *by* $f_{i+1}; f_i$ *then* $\sigma_I - \pi_1 - \sigma_F$. *Moreover, it is clear that* $D(\pi_1) < D(\pi_0)$.

- *If* f_i *left-projects with* f_{i+1} *for the state* σ_i *then* $\sigma_i - f_i - \sigma_{i+1}$. *Following the same reasoning, we denote by* π_1 *the segment sequence where we substitute* $f_i; f_{i+1}$ *by* f_i *then* $D(\pi_1) < D(\pi_0)$ *and* $\sigma_I - \pi_1 - \sigma_F$.

- *The cases of right and total-projections are analogous.*

We have proved that we can obtain a segment-sequence π_1 *that represents a trace t whose execution is final state equivalent to the original execution, such that* $\pi_1 \leq_F \pi_0$ *and* $D(\pi_1) < D(\pi_0)$. *Hence, the proposition holds by induction.*

□

Lemma 4 *Existence of* $FLeft_i$ *and* $FRight_i$

If a program Pr *is* $sECF_{SS}$ *then for any simple segment-sequence* π *such that all its segments are minimal and it traverses the call nodes* $c_1, \ldots, c_n$ *following this order (first* c_1*, next* c_2*,...) there exists a family of pairs of functions sets* $\{(FLeft_i, FRight_i)\}_{i=1,\ldots,n}$ *such that:*

- $FLeft_i \cup FRight_i = F(Pr)$
- $FLeft_i \cap FRight_i = \emptyset$
- $FLeft_i \subseteq Left(c_i)$ *and* $FRight_i \subseteq Right(c_i)$
- *If* $i \leq j$ *then* $FLeft_j \subseteq FLeft_i$ *and* $FRight_i \subseteq FRight_j$
- *For all* $f_1 \in FLeft_i, f_2 \in FRight_i$, $g_2; g_1$ *moves.*

Proof: *We take for each* $i = 1, \ldots, n$:

$FRight_i = \bigcup_{j=1,\ldots,i} MRight(c_j)$

$FLeft_i = F(Pr) \setminus FRight_i$

Let us check that this family of sets verifies the conditions:

- *The two first conditions are trivial.*
- *Now, we prove that* $f \notin Right(c_i)$ *implies that* $f \notin FRight(c_i)$.

  If $f \notin Right(c_i)$ *then* $f \in MLeft(c_i)$. *We are assuming that the program* Pr *is* $sECF_{SS}$, *hence* $f \notin MRight(c_j)$ *for all* $j : 1 \leq j \leq i$ *(as* c_i *is reachable from* c_j*). Then,* $f \notin \bigcup_{j=1,\ldots,i} MRight(c_j) = FRight_i$.

  The proof of $f \notin Right(c_i) \Rightarrow f \notin FRight(c_i)$ *is analogous.*

- *If* $i \leq j$ *then* $FRight_i = \bigcup_{k=1,\ldots,i} MRight(c_k) \subseteq \bigcup_{k=1,\ldots,j} MRight(c_k) = FRight_j$.

  On the other hand, $FLeft_i = F(Pr) \setminus FRight_i$ *hence* $FLeft_j \subseteq FLeft_i$.

- *Finally, we check that for all* $f_1 \in FLeft_i$ *and* $f_2 \in FRight_i$, $f_2; f_1$ *moves.*

  The function f_2 *belongs to* $FRight_i$, *hence there exists* $j \leq i$ *such that* $f_2 \in MRight(c_j)$. *If* $f_2; f_1$ *does not move,* f_1 *belongs to* $MRight(c_j)$ *too.*

  Hence, $f_1 \in FRight_i$ *and we get a contradiction:* $f_1 \in FLeft_i \cap FRight_i$ *but this intersection is empty.*

□

A.2.2 Proof $sECF_{SS}$ implies $sECF_{FS}$

Definition 32 ($V(\pi)$) *Let* π *be a segment-sequence representing a simple trace,* $\pi = f_0^1; \ldots; f_0^{m_0}; \tau_1; f_1^1; \ldots; f_1^{m_1}; \tau_2; \ldots; \tau_{n+1}; f_{n+1}^1; \ldots; f_{n+1}^{m_{n+1}}$

We are using the notation $c_1, \ldots, c_n$ *for the call nodes it traverses and* $f_i^1; \ldots; f_i^{m_i}$ *to denote the, maybe empty, sequence of callbacks made in the call node* c_i.

Given a family of pairs of functions sets $\{(FLeft_i, FRight_i)\}_{i=1,\ldots,n}$ *such that:*

- $FLeft_i \cup FRight_i = F(Pr)$
- $FLeft_i \cap FRight_i = \emptyset$

we define the value $V(\pi)$ *as:*

$$V(\pi) = \sum_{i=1,\ldots,n} v_i(f_i^1; \ldots; f_i^{m_i}) \tag{A.3}$$

with $v_i(f^1; \ldots; f^m) = \sum_{j=1,\ldots,m} v_i(f^j)$ *and*

$$v_i(f) = \begin{cases} i & \text{if } f \in FLeft(c_i) \\ n+1-i & \text{if } f \in FRight(c_i)\} \end{cases} \tag{A.4}$$

We can easily check the following facts:

- A simple segment sequence π is callback-free if and only if $V(\pi) = 0$.
- We consider two simple segment sequences π and π' that only differ in the callbacks made in the node c_i. The first one calls to the sequence of functions $\pi_i = f_1; \ldots; f_n$ and the second one to $\pi'_i = f'_1; \ldots; f'_m$. If they verify that $\pi'_i \leq_F \pi_i$ then $V(\pi') \leq V(\pi)$.

Theorem 5 ($sECF_{SS} \Rightarrow sECF_{FS}$) *If a program* Pr *is* $sECF_{SS}$ *then it is* $sECF_{FS}$.

Proof: *We are proving that the execution of any simple trace of the program is* $dECF_{FS}$. *This condition is enough to prove that the program is* $sECF_{FS}$ *according to the Lemma* 3.

Given the execution of a simple trace $\xi = \sigma_I - t - \sigma_F$, *we consider a segment-sequence of minimal segments* π *that represents its trace* t. *We assume that it traverses the call nodes* $c_1, \ldots, c_n$ *following this order (first it gets to* c_1, *then to* $c_2, \ldots$*).*

According to the Lemma 4, *there exists a family of sets* $\{(FLeft_i, FRight_i)\}_{i=1,\ldots,n}$ *that verifies the properties established in the lemma. We consider the value of* $V(\pi)$ *according to this family of sets.*

- *If $V(\pi) = 0$ there are not callbacks in the segment-sequence, hence the execution trivially is $dECF_{FS}$.*

- *If $V(\pi) > 0$ there exists at least one callback, we assume it is made in the node c_i. We consider the sequence of callbacks made in the node c_i and denote it by $\pi_i = f_i^1; \ldots; f_i^{m_i}$.*

  Next, we take the states σ_1 and σ_2 that are, respectively, the state of the contract in the original execution before and after executing this sequence of callbacks. We apply the Lemma 1 on the execution $\sigma_1 - \pi_i - \sigma_2$ with respect to the sets $FLeft_i$ and $FRight_i$. Then, there exists a segment-sequence $\overline{\pi_i} = \overline{f_i^1}; \ldots; \overline{f_i^r}$ such that $\sigma_1 - \overline{\pi_i} - \sigma_2$ and $D(\overline{\pi_i}) = 0$. Moreover, it verifies that $\overline{\pi_i} \leq_F \pi_i$.

  We consider now a segment-sequence $\overline{\pi}$ similar to the original one but substituting the sequence of callbacks made in the node c_i by $\overline{\pi_i}$. This segment-sequence verifies that $\sigma_I - \overline{\pi} - \sigma_F$ and $V(\overline{\pi}) \leq V(\pi)$.

  Now we distinguish three possibilities:

  - *If the new sequence of callbacks $\overline{\pi_i}$ is empty, then it is clear $D(\overline{\pi}) < D(\pi)$.*
  - *If it is not empty and its first function $\overline{f_i^1}$ belongs to $FLeft(c_i)$ then this function left-moves with the segment τ_i at the left of the call node c_i ($\tau_i; \overline{f_i^1}$ left-moves).*

    We consider the states σ_3 and σ_4 before and after executing $\tau_i; \overline{f_i^1}$, respectively, we have that $\sigma_3 - \tau_i; \overline{f_i^1} - \sigma_4$.
    We know that τ_i either left-projects or commutes with f_i^1 for the state σ_3, hence we distinguish between these two cases:

    - *If it left-projects, then $\sigma_3 - \tau_i - \sigma_4$. So, if we take the segment-sequence π_1 where we substitute $\tau_i; f_i^1$ by τ_i we have that $\sigma_I - \pi_1 - \sigma_F$. Moreover, $V(\pi_1) < V(\overline{\pi}) \leq V(\pi)$.*
    - *The case of commutation is similar. It is clear that $\sigma_3 - f_i^1; \tau_i - \sigma_4$. Hence, the segment-sequence π_1 where we move the first callback f_i^1 from the beginning of the callbacks of the call node c_i to the end of the callbacks of node c_{i-1}, then $\sigma_I - \pi_1 - \sigma_F$. Moreover, $f \in FLeft(c_i) \subseteq FLeft(c_{i-1})$ hence $V(\pi_1) = V(\overline{\pi}) - i + (i-1) < V(\pi)$*

    In both cases we get a segment sequence that represents a trace whose execution is final-state equivalent to ξ such that $V(\pi_1) < V(\pi)$.
  - *If $\overline{f_i^1}$ does not belong to $FLeft(c_i)$ then all the callbacks belong to $FRight(c_i)$. We consider then the last callback $\overline{f_i^r}$ and follow an analogous reasoning.*

We have proved that there exists a segment-sequence π_1 such that $V(\pi_1) < V(\pi)$ and it represents a trace t whose execution is final-state equivalent to ξ. Hence, we conclude that there exist a callback-free execution $\xi' = \sigma_I - t' - \sigma'_F$ final state equivalent to π, hence ξ is $dECF_{FS}$.

□

A.3 Proof of Theorem 2

Theorem 6 *If a program Pr is $sECF_{OS}$ then it is $sECF_{FS}$*

Proof: *We are going to prove that the execution of any simple trace is $dECF_{FS}$. We do this by induction on the number of call-nodes that have not been solved. We say that a call node c_i is solved according to an order $<_O$ if c_i and all the nodes smaller than it do not have callbacks.*

We are assuming that the contract is $sECF_{OS}$ hence there exists an order $<_O$ for the call nodes s.t. all of them are solvable wrt. $<_O$.

Let $\xi = \sigma_I - t - \sigma_F$ be a simple execution, we assume it traverses the call nodes $c_1, \ldots, c_n$ following this order (first c_1, then c_2, ...). Then, we distinguish two possibilities:

1. *If all call nodes have been solved, the execution has not callbacks hence it is trivially $dECF_{FS}$.*

2. *Otherwise, we take c_i the smallest call node with callbacks according to the order $<_O$. Hence, not callbacks are made in any node c_j such that $c_j <_O c_i$. We denote the sequence of callbacks that take place in c_i by $\pi_i = f_i^1; \ldots; f_i^m$.*

   First, take the sets FLeft $=$ MLeft$^O(c_i)$ and FRight $= F(Pr) \setminus$ MLeft$^O(c_i)$. The call node c_i is solvable, hence all the nodes in FLeft belong to Left$^O(c_i)$, all the nodes in FRight belong to Right$^O(c_i)$ and for all $g \in$ FLeft, $f \in$ FRight $f; g$ moves.

   Now, we apply the Lemma 1 and obtain the segment-sequence $\overline{\pi_i} = \overline{f_i^1}; \ldots; \overline{f_i^r}$ such that $\sigma_I - \overline{\pi_i} - \sigma_F$ and $D(\pi_i) = 0$. Then, there exists $k \in 0, \ldots, m$ such that $\overline{f_j^i} \in$ FLeft for all $j \leq k$ and $\overline{f_j^i} \in$ FRight for all $j > k$. We have sorted the callbacks inside the call-node correctly, now we have to move them away from it (to its left or right respectively).

   Let c_l be the last non-solved call-node that appears before c_i (we are considering the initial node and end nodes of the function as non-solved call-nodes for this definition). It verifies that $l < i$, $c_l \geq_O c_i$ and for all $l : l < j < i$ we have that $c_j < c_i$. We take $\tau_l = TR(c_l, c_i)$ it is clear that $\tau_l \in$ SLeft$^O(c_i)$. Moreover, in the original execution there are not callbacks made in call nodes between c_l and c_i, hence the trace followed between these nodes belongs to τ_l.

   Now, we move all the functions $\overline{f_1^i}, \ldots, \overline{f_k^i}$. We know that all of them belong to Left$^O(c_i)$, hence $\tau_l; \overline{f_i^l}$ left-moves for all them. Applying a similar reasoning to the one we followed in the proof of Theorem 5, we commute or project each one of them one by one, obtaining a segment sequence where all these functions have been either projected or moved to the call-node c_l.

   We can apply the same reasoning for the functions in FRight: let c_r be the first non-solved call-node that appears after c_i. It verifies that $i < r$, $c_r \geq_O c_i$ and for all $l : i < j < r$ we have that $c_j < c_i$. We take the segment between these two nodes $\tau_r = TR(c_i, c_r)$. As $c_i \leq_O c_r$ and the call-nodes between them are smaller than c_i, it verifies that $\tau_r \in$ SRight$^O(c_i)$.

Now we consider the functions $\overline{f_i^{k+1}}, \ldots, \overline{f_i^m}$. *We know that all of them belong to* $Right^O(c_i)$, *hence* $\overline{f_i^r}; \tau_r$ *right-moves for all them. As we said in the previous case, we can get a segment sequence where all these functions have been either projected or moved to the call-node* c_r *that represents a trace whose execution is final-state equivalent to* ξ.

This trace does not have callbacks in nodes c_j *such that* $c_j \leq_O c_i$, *hence all these nodes are solved. Then, the number of not solved call-nodes is strictly smaller than in the original execution* ξ. □

A.4 Proof of Theorem 3

The next two results prove that the ordered approach is at least as precise as the minimal segments approach.

Lemma 5 *Given an order* $<_O$ *and a function* $g \in F(Pr)$, *then if* $f \in Left(c_i)$ *for all* $c_i \leq_O c_n$ *such that* c_n *is reachable from* c_i *then* $f \in Left^O(c_n)$ *and if* $f \in Right(c_i)$ *for all* $c_i \leq_O c_n$ *such that* c_i *is reachable from* c_n *then* $f \in Right^O(c_n)$.

Proof: *We are proving the result for* $Left^O(c_n)$, *the case* $Right^O(c_n)$ *is analogous.*

Let us consider $\tau \in SLeft^O(c_n)$, *this set was built considering that the nodes* $c_i <_O c_n$ *are not call-nodes. Then, it is clear that* τ *is going to start at a call-node* c_j *such that* $c_j \geq_O c_i$, *traverse some nodes* $c_1, \ldots, c_m$ *smaller than* c_n *and finally end in* c_i.

So, we divide τ *into minimal segments:*

$\tau = \tau_1; \tau_2; \ldots; \tau_m \tau_n$

We have that $\tau_1 \in SLeft(c_1), \ldots, \tau_m \in SLeft(c_m), \tau_n \in SLeft(c_n)$ *with* $c_1, \ldots, c_m$ *nodes such that all of them are smaller than* c_n *according to the order* $<_O$.

The result we want to obtain is that $\tau; g$ *left-moves. Then, for any state* $\sigma_I \in Feasible(\tau)$, *we need to prove that* τ *either left-projects or commutes with* g *for the state* σ_I.

We consider that $\sigma_I - \tau; g - \sigma_F$. *Then, there exists a state* σ_1 *such that* $\sigma_I - \tau_1; \ldots; \tau_m - \sigma_1$ *and* $\sigma_1 - \tau_n - \sigma_F$.

We have that $\tau_n \in SLeft(c_n)$ *and* $g \in Left(c_n)$, *hence* τ_n *commutes or left-projects with* g *for the state* σ_1.

1. *If they left-project,* $\sigma_1 - \tau_n - \sigma_F$. *Then,* $\sigma_1 - \tau - \sigma_F$, *hence* τ *left-projects with* g.

2. *If they commute,* $\sigma_1 - g; \tau_n - \sigma_F$. *Then, we have that* $\sigma_1 - \tau_1; \ldots; \tau_m; g; \tau_n - \sigma_F$.

   Now, we can repeat the same reasoning for each one of the intermediate segments, hence finally the execution $\sigma_I - \tau; g - \sigma_F$ *is final-state equivalent to either* $\sigma_I - \tau - \sigma'_F$ *or* $\sigma_I - g; \tau - \sigma''_F$ *(in case it commutes with all the intermediate segments).*

We have proved that $\tau; g$ left-moves for any $\tau \in SLeft^O(c_n)$, hence $g \in Left^O(c_n)$. □

Theorem 7 *Given a program Pr, if a subset of call-nodes $c_1, \ldots, c_m = C' \subseteq C$ verify the $sECF_{SS}$ property then for any order $< O$ such that $c_1 <_O c_2 <_O \ldots <_O c_m <_O c$ for all $c \in C \setminus C'$ we have that all the nodes of C' are solvable wrt. $< O$.*

Proof: *Let us take $c_i \in C'$. In order to prove that this node is solvable we need to prove that $MLeft^O(c_i) \cap MRight^O(c_i) = \emptyset$.*

If $g \in MLeft^O(c_i)$ then either $g \notin Right^O(c_i)$ or there exists a function $g_1 \in MLeft^O(c_i)$ such that $g; g_1$ does not move. We can repeat the same reasoning for g_1 and so on, until we eventually get a function g_n such that $g_n \notin Right^O(c_i)$.

Then, according to the Lemma 5, $g_n \notin Right(c_j)$ for a call-node $c_j \leq_O c_i$ such that c_j is reachable from c_i, hence $g_n \in MLeft(c_j)$. Now, we take the reverse path: if $g_n \in MLeft(c_j)$ and $g_{n-1}; g_n$ does not move hence $g_{n-1} \in MLeft(c_j)$, hence $g \in MLeft(c_j)$.

We follow an analogous reasoning for $MRight^O(c_i)$, hence if $f \in MRight^O(c_i)$ then $f \in MRight(c_k)$ for a call-node $c_k \leq_O c_i$ such that c_i is reachable from c_k.

Finally, we observe that if $g \in MLeft^O(c_i) \cap MRight^O(c_i)$ then $g \in MRight(c_k) \cap MLeft(c_j)$ with c_j reachable from c_k and $c_k, c_j \in C'$, hence the nodes of C' would not verify the ECF_{SS} property. □

A.5 Proof of Theorem 4

Proposition 2 *Given a program Pr, two sets FLeft and FRight and a callback invariant I such that*

- *$FLeft \cap FRight = \emptyset$*
- *$FLeft \cup FRight = F(Pr)$*
- *for all $f \in FLeft$ and $g \in FRight$ $g; f$ moves assuming the invariant $I(c)$*

Then, given a callback-free execution ξ of a callbacks made in the node c $f_1; \ldots, f_n$ there exists a sequence of callbacks made in that node $f'_1; \ldots; f'_m$ whose execution is final-state equivalent to ξ such that $f'_1; \ldots; f'_m \leq_F f_1; \ldots; f_n$ and $D(f'_1; \ldots; f'_m) = 0$.

Proof: *It is analogous to 1.*

We just have to take into account that the state σ_i satisfies $I(c)$, hence the functions f_i and f_{i+1} either commute or project for this state. □

Theorem 8 *If a program Pr is $sECF_{IOS}$ then it is $sECF_{FS}$*

Proof: *The contract Pr is* $sECF_{IOS}$ *hence there exists an order* $<_O$ *and callback invariants I for the nodes of the program C s.t. all the call nodes are solvable wrt.* $<_O$.

Let $\xi = \sigma_I - t - \sigma_F$ *be a simple execution, we assume it traverses the call nodes* $c_1, \ldots, c_n$ *following this order (first* c_1*, then* c_2*, ...). We denote by* c_0 *the start node.*

First, we prove that during the execution the property $I(c_i)$ *holds when we first arrive at the node* c_i*, and it also holds after the sequence of callbacks made in this node. We prove this result by induction.*

We assume that the result holds for the nodes $c_0, \ldots, c_{i-1}$*, then after execution the callbacks made in the node* c_{i-1} *we are in a state* σ_{i-1} *that satisfies* $I_{c_{i-1}}$*. Then, we consider the trace* t_i *followed between the nodes* c_i*,* c_{i-1} *in the execution* ξ*. The property* I_{c_i} *is a call node invariant hence if* $\sigma_{i-1} - t_i - \sigma_i$ *then* σ_i *satisfies* I_{c_i}*. Then, it is clear that* I_{c_i} *holds when we first arrive at the node* c_i *and it also holds after executing any callback.*

Finally, we just have to follow an analogous reasoning to the one we used to prove that $sECF_{OS} \Rightarrow sECF_{FS}$*. In case there is a not solved node* c_i*, we apply the proposition above to get a sequence of callbacks* $\overline{\pi_i}$ *such that* $D(\overline{\pi_i}) = 0$*. Then, we take the same* τ_l *and* τ_r *segments and move the callbacks to the nodes* c_l *and* c_r*.*

When we are trying to move a callback $\overline{f_i^k}$ *(we assume it belongs to* $FRight$*, the case* $\overline{f_i^k} \in FLeft$ *is analogous), we have to take into account that* $I(c_i)$ *is a call node invariant. We consider the states* σ_1 *and* σ_2 *before and after executing* $\overline{f_i^k}; \tau_r$ *in the original execution, respectively. I is a call node invariant, hence* σ_1 *satisfies* $I(c_i)$*. Then, it is clear that* $\overline{f_i^k}; \tau_r$ *either commutes or right-projects for the state* σ_i*.*

$\square$