Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts

SHELLY GROSSMAN, Tel Aviv University, Israel
ITTAI ABRAHAM, VMware Research, USA
GUY GOLAN-GUETA, VMware Research, USA
YAN MICHALEVSKY, Stanford University, USA
NOAM RINETZKY, Tel Aviv University, Israel
MOOLY SAGIV, Tel Aviv University, Israel and VMware Research, USA
YONI ZOHAR, Tel Aviv University, Israel

Callbacks are essential in many programming environments, but drastically complicate program understanding and reasoning because they allow to mutate object's local states by external objects in unexpected fashions, thus breaking modularity. The famous DAO bug in the cryptocurrency framework *Ethereum*, employed callbacks to steal $150M. We define the notion of Effectively Callback Free (ECF) objects in order to allow callbacks without preventing modular reasoning.

An object is ECF in a given execution trace if there exists an equivalent execution trace without callbacks to this object. An object is ECF if it is ECF in every possible execution trace. We study the decidability of dynamically checking ECF in a given execution trace and statically checking if an object is ECF. We also show that dynamically checking ECF in Ethereum is feasible and can be done online. By running the history of all execution traces in Ethereum, we were able to verify that virtually all existing contract executions, excluding these of the DAO or of contracts with similar known vulnerabilities, are ECF. Finally, we show that ECF, whether it is verified dynamically or statically, enables modular reasoning about objects with encapsulated state.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Dynamic analysis**;

Additional Key Words and Phrases: Program analysis, Modular reasoning, Smart contracts

ACM Reference Format:
Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *Proc. ACM Program. Lang.* 2, POPL, Article 48 (January 2018), 28 pages. https://doi.org/10.1145/3158136

1 INTRODUCTION

The theme of this paper is enabling modular reasoning about the correctness of objects with encapsulated state. This is inspired by platforms like Ethereum [Wood 2016] that facilitate execution of *Smart Contracts* [Szabo 1997] on top of a blockchain-based distributed ledger [Nakamoto 2008]. A key property in Ethereum Smart Contracts is the lack of global mutable shared state, in contrast

Authors' addresses: Shelly Grossman, Computer Science, Tel Aviv University, Israel; Ittai Abraham, VMware Research, USA; Guy Golan-Gueta, VMware Research, USA; Yan Michalevsky, Stanford University, USA; Noam Rinetzky, Computer Science, Tel Aviv University, Israel; Mooly Sagiv, Computer Science, Tel Aviv University, Israel , VMware Research, USA; Yoni Zohar, Computer Science, Tel Aviv University, Israel.

2475-1421/2018/1-ART48
https://doi.org/10.1145/3158136

to common standard programming environments such as C and Java. A smart contract is analogous to an object with encapsulated state.

However, the Ethereum blockchain, and many other dynamic environments, implement event-driven programming using callbacks. These callbacks are necessary for functionality, but can compromise security. For example, the famous bug in the DAO contract exploited callbacks to steal $150M [Daian 2016].

Indeed, callbacks may break modularity which is essential for good programming style and extendibility. In the context of Blockchain, modularity is even more important since contracts are contributed by different sources, some of which may be malicious. Accordingly, the bug in the DAO allowed an adversarially crafted contract to mutate the DAO's state by calling back to it.

The DAO contract, that implemented a crowd-funding platform, was attacked by a 'callback loop-hole' (to be precisely described below). This attack, the recovery from which required a controversial hard-fork[1] of the blockchain, exhibits a vulnerability that is peculiar to decentralized consensus systems, like Ethereum: in such systems, a buggy contract cannot be updated or fixed (except for extreme measures like hard-forking), which makes validation and verification of smart contracts of even greater importance for this application.

Effectively Callback-Free Objects. We identify a natural generic correctness criteria for objects which enables modular reasoning in environments with local-only mutable states, and expect most correct objects to satisfy this requirement. Informally, if an object o calls another object o', and the execution of o' calls o again, this second call to o is defined as a callback. The main idea is to allow callbacks in o only when they cannot affect the serial non-interruptible behavior of o. Thus, such callbacks can be considered harmless and do not affect the set of local reachable states of the object o. In particular, the behavior of such objects is independent of the client environments and of other objects. It is possible to reproduce all behaviors of the object using a most general client and without analyzing external objects.

We say that an execution is **Dynamically Effectively Callback Free** (dECF) when there exists "an equivalent" execution without callbacks which starts in the same state and reaches the same final state. By *equivalent*, we refer to the behavior of a particular object as an external observer may perceive. We say that an object is **Statically Effectively Callback Free** (sECF) when all its possible executions are dynamically ECF. We do not distinguish between dynamic and static ECF when the context is clear. Both definitions are useful. Dynamic ECF in particular is applicable to the blockchain environment, since static ECF is undecidable in the general case. We ran experiments on Ethereum, proving that checking dynamic ECF is inexpensive, and thus can be done efficiently in-vivo. This, combined with Ethereum's built-in rollback feature, would have allowed to prevent the DAO bug from occurring, without invalidating legitimate executions. (In fact, we found just one such legitimate non-ECF contract, discussed in Section 8).

We show that the vulnerable DAO contract is non-ECF while no non-ECF executions are detected after applying the suggested corrections to it. Notice that the ECF notion is similar to the notion of atomic transactions in concurrent systems. Indeed, despite the fact that contract languages do not usually support concurrency, modularity and callbacks require similar kind of reasoning.

The ECF property's usefulness is not limited to bug-finding; once ECF is established, it can be served to simplify reasoning on the object in isolation of other objects: We show that the set of reachable local states in ECF objects can be determined without considering the code of other objects and thus enable modular reasoning. This modular reasoning can be performed automatically using abstract interpretation e.g., as suggested in Logozzo [2009] or by using deductive verification

[1] Which can be thought of as taking an agreed history of transactions, and manually change it.

which is supported by Dafny [Leino 2010]. We demonstrate this by verifying an interesting invariant of the DAO contract. (See Section 2).

Online Detection of ECF executions. A naïve detection of dECF may be costly because of the need to enumerate subexecution traces. Therefore, we develop an effective polynomial online algorithm for checking if an execution is ECF. The main idea is to detect conflicting memory accesses and utilize commutativity in an effective manner. We integrated the algorithm into the *Ethereum Virtual Machine (EVM)* [Wood 2016]. We ran the algorithm on all executions kept in the Ethereum blockchain until 23 June 2017, and demonstrate that: (i) the vulnerable DAO contract and other buggy contracts are non-ECF. (ii) very few correct contracts are non-ECF, (iii) callbacks are not esoteric and are used in many contracts, and (iv) the runtime overhead of our implementation is negligible and thus can be integrated as an online check.

This online detection can thus be used to prevent incidents like the theft from the DAO at the cost of slightly more restricted form of programming.

As far as we are aware, our tool is also the most precise and effective tool for finding such vulnerable behaviors due to callbacks. We compared it to the Oyente tool [Luu et al. 2016; Melonport 2017], by giving it both ECF and non-ECF contracts based on the DAO object (Figure 1). We found that it has false positives, as it detects a 'reentrancy bug' (the common name of the DAO vulnerability in the blockchain community) for any one of the fixes that render our example contract ECF.

Decidability of sECF for objects. We also consider the problem of checking sECF algorithmically. Obviously, since modern contract languages, such as Solidity [Ethereum Foundation 2017b], support Turing complete languages, checking if a contract is ECF is undecidable.

We show that checking that a contract is sECF in a language with finite local states is decidable. This is interesting since many contracts only use small local states or maps with uniform data independent accesses. Technically, this result is non-trivial since the nesting of contract calls is unbounded, and since ECF requires reasoning about permutations of nested invocations. The reason for the decidability is that non-ECF executions which occur in high depth of nesting must also occur in depth 2.

Main Results. Our results can be summarized as follows:

(1) We define a general safety property, called ECF, for objects (sECF) and executions (dECF). Our definition is inspired by the Blockchain environment but it may also be useful for other environments with encapsulated states, such as Microservices.
(2) We show that objects with encapsulated data, under the assumption that they satisfy ECF, can be verified using modular reasoning in a sound manner.
(3) A stronger notion of ECF, based on conflict-equivalence, enabling efficient verification of dECF in real-life environments, and for which sECF is decidable for programs with finite state and unbounded stack.
(4) A polynomial time and space algorithm for online checking of dECF and prototype implementation of it as a dynamic monitor of dECF, built on top of an Ethereum client.
(5) Evaluation of the algorithm on the entire history of the Ethereum blockchain (both main and 'Classic' forks, see Section 8). The monitor detects true bad executions (the infamous DAO and others) as non-ECF, and has near-zero false positives. Based on this result, it can be inferred that, in practice, most non-ECF executions correspond to bad executions. We also show that our monitor has a very small runtime overhead. By retroactively running the dECF monitor on the available history, we were able to prove its effectiveness in preventing the exploitation of the vulnerability in the DAO, and even more importantly, the feasibility of leveraging it in other applications, e.g., simplifying modular contract verification.

```
Object DAO
  Map<Object, int> credit
  int balance
  Invariant (sum o: credit[o]) = balance

  Method withdrawAll(Object o)
    2: if (oCredit > 0)
      // 2.5: credit[o] = 0
      3: this.balance -= oCredit
      4: o.pay(oCredit)
      5: credit[o] = 0

Method deposit(Object o, int amount)
    6: credit[o] += amount
    7: this.balance += amount
```

Fig. 1. A contract illustrating the DAO bug. The representation invariant may be violated by callbacks from malicious contracts. Line 2.5 fixes the bug.

```
Object GoodClient
  Object Dao, int balance
  Method init(Object dao)
    1: this.Dao = dao
  Method pay(int profit)
    2: this.balance += profit
  Method depositCredit(Object dao, int amount)
    3: Dao.deposit(this, amount)
  Method getCredit(Object dao)
    4: Dao.withdrawAll(this)
```

(a) An innocent client using the DAO object without violating its representation invariant.

```
Object Attacker
  Object Dao, bool stop, int balance
  Method init(Object dao)
    1: Dao = dao
    2: stop = false
  Method pay(int profit)
    3: this.balance += profit
    4: if (!stop)
       5: stop = true
       6: Dao.withdrawAll(this)
    7: stop = false
```

(b) A snippet of an `Attacker` object. It is stealing money from the DAO object by violating its representation invariant.

Fig. 2. An innocent and a malicious client using the DAO object

2 OVERVIEW

This section provides some necessary background and an informal overview of our approach.

2.1 The DAO Bug

Figure 1 shows pseudocode illustrating the vulnerability in the DAO [2]. The contract stores a `credit` for each object, as well as the current `balance`.[3] The `credit` represents individual investments per object. To align with the Ethereum terminology, the unit of currency represented by `credit` and `balance` is called *ether*. The contract maintains a representation invariant, where the sum of the credits equals to the current balance, i.e.,

$$\Sigma_{o \in \mathrm{dom}(credit)} credit[o] = balance \tag{1}$$

The contract offers two methods for manipulating states: `deposit` for depositing money and `withdrawAll` for withdrawing all available funds of a specific object.

[2] *DAO* is acronym for *decentralized autonomous organization*, and its purpose is to facilitate voting on proposals and on investments by the owners of the DAO.

[3] In programming languages like Solidity, balance is a predefined field of every contract, maintained by the runtime system. We write it explicitly for clarity.

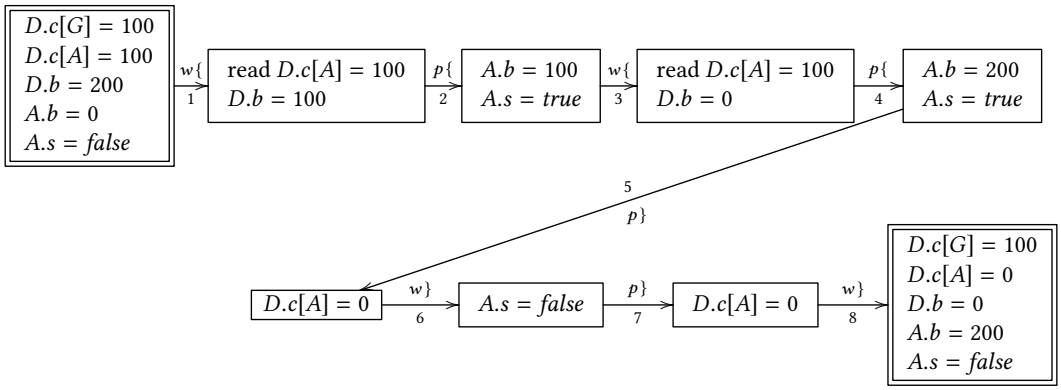

Fig. 3. A trace of calls illustrating an attack on the DAO. Nodes are labeled by local changed states and edges are labeled by actions and by the corresponding order in the original trace. D denotes the DAO, G denotes a `GoodClient` and A is an `Attacker` object. w denotes the `withdrawAll` operation and p denotes the pay operation. b is a shorthand for `balance`, c is a shorthand for `credit`, and s is a shorthand for `stop`.

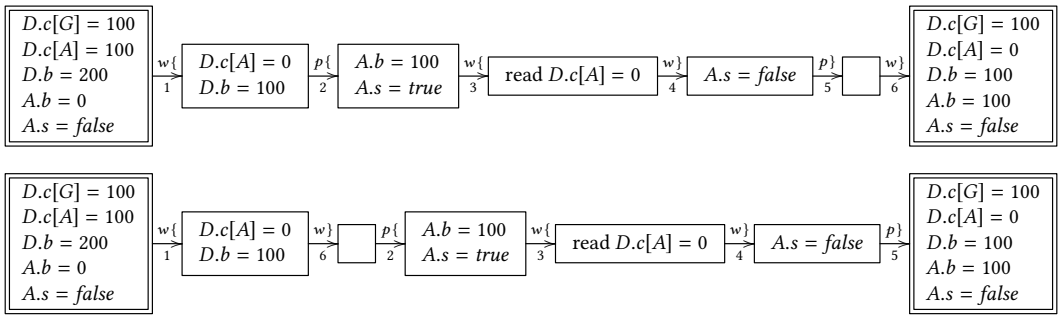

Fig. 4. Two traces of calls illustrating the original and callback-free versions of a failed attack on the ECF version of the DAO. Nodes are labeled by local changed states and edges are labeled by actions and by the corresponding order in the original trace. D denotes the DAO, G denotes a `GoodClient` and A is an `Attacker` object. w denotes the `withdrawAll` operation and p denotes the pay operation. b is a shorthand for `balance`, c is a shorthand for `credit`, and s is a shorthand for `stop`.

Figure 2a shows a simple client illustrating the expected usages of the DAO object. Figure 2b shows a simple attack on the DAO object. The code callbacks to the DAO method `withdrawAll` to steal money. Figure 3 depicts a concrete trace of attacking the DAO assuming that the DAO's initial balance is 200 ether. We reached that state after a `GoodClient` object and an `Attacker` object deposited each 100 ether. In the first call to `withdrawAll`, the attacker will get the amount he invested originally in the attack (100 ether). The DAO then calls to the `Attacker` object's pay method, which increases the attacker's balance by 100 ether, and calls `withdrawAll` again. The pay method is designed to call `withdrawAll` at most once in a trace by updating the `stop` variable, and avoid infinite recursion.[4] The code of `withdrawAll` in the second run will transfer an additional 100 ether from the DAO

[4]For clarity, we avoid technical discussion of the semantics of executions and exceptions in Ethereum/Solidity, to allow us to focus on the ECF property.

object to the attacker. In the end of the trace, the DAO was depleted of its funds completely, and the attacker managed to illegitimately receive the funds that belonged to GoodClient.

2.2 Effectively Callback Free Contracts

In principle, semi-automatic program verification and abstract interpretation can be used to verify the absence of malicious attacks like the one in the Attacker object. However, this requires reasoning about the whole code[5]. This paper advocates a different solution by exploring modularity. The idea is to require stronger conditions from the contracts which prevent the need to reason about other objects at all.

Specifically, we define the notion of **effectively callback free** (ECF) objects. Our definition is inspired by Blockchain contracts but is applicable to enforce modularity in other environments with local states.

We say that an execution of an object with an initial state s_0 and final state s is **Dynamically Effectively Callback Free** (DECF) when there exists "an equivalent" execution of the contract without callbacks which starts in the same initial state s_0 and reaches the final state s. We say that an object is **Statically Effectively Callback Free** (SECF) when all its possible executions are effectively callback free.

The DAO object is not ECF. For example, the trace depicted in Figure 3 cannot be reproduced without callbacks to reach the same state. In contrast, the fix to the DAO object by uncommenting line 2.5 and deleting line 5 makes the contract ECF. This contract is now ECF since all its traces can be reordered to avoid callbacks. For example, Figure 4 shows a trace of an attempt to perform an attack similar to the attack in Figure 3 and its corresponding reordering that avoids callbacks. Note, that in the reordered trace, withdrawAll did not execute line 4. Omitting calls is allowed for the sake of proving an execution is ECF, as our goal is to be able to reproduce, assuming there are no callbacks, the same behaviors that are feasible with callbacks.

2.3 Online ECF Detection

It is possible to check in a naïve way that an execution is ECF by recording the trace and checking the ECF property at the end of the execution, by enumerating all possible permutations. However, this is costly both in space and in time, since the number of permutations grows exponentially with the size of the trace. In particular, it is hard to see if such a solution can be integrated into a virtual machine.

In order to obtain a feasible online algorithm, we check a stronger requirement than ECF, which is inspired by conflict serializability of database transactions. The main idea is to explore commutativity of operations for efficient online checking of a correctness condition which guarantees that the callback-free trace results in the same state as the original trace.

Consider a trace π with potential callbacks and a reordered trace π' which does not include callbacks. π' is not necessarily feasible, unless we permit to ignore external calls by objects and force the clients to perform these calls instead. We say that π and π' are conflict equivalent if every pair of conflicting read/write operations in π appear in the same order in π'. Operations conflict when they are not commutative. Commutativity is mechanically checked by comparing the read and the write sets of operations, and forbidding intersection of read/write conflicts. For example, in Figure 3, the read operation of D.c[A] in the withdrawAll action labeled 1 (lines 1-3) does not commute with the write operation of D.c[A] in the withdrawAll action labeled 5 (line 5). However, in Figure 4, depicting a trace of the ECF version of DAO, the operations in the withdrawAll action

[5]In the case of Ethereum, it is in fact impossible to reason about the whole code, as new contracts can be added at any time, and these contracts could interact with the contract being checked.

labeled 3 (lines 1-2) commute with the operations in the `withdrawAll` action labeled 6, which has an empty read and write sets (no code was executed). The information regarding commutativity of different subtraces is used to build a constraint graph on the ordering of object invocations. When this constraint graph contains no cycles, it is possible to perform topological sort to find a concrete callback-free trace. A full description of the algorithm and its complexity is available in Section 7.

We integrated this algorithm into the EVM (the *Ethereum Virtual Machine*) and applied it to all available executions in the blockchain. The results are summarized in Section 8. They indicate that the vast majority of non-ECF executions come from erroneous contracts. They also indicate that the runtime overhead of our instrumentation is neglectable. From these encouraging results, we concluded that if the ECF check was part of the Ethereum protocol, it could have prevented the vulnerability in the DAO from being exploited. Its clearly beneficiary for an environment like Ethereum, which handles sensitive financial transactions, and in which code is virtually impossible to upgrade.

2.4 Deciding ECF Contracts

We also investigated the possibility to verify at compile-time that a contract is ECF (the sECF property). In general, this is undecidable, since languages such as *Solidity*, a high-level front-end to EVM bytecode, are Turing-complete. However, we show that for contracts with finite local states, checking ECF is decidable. This result is non-trivial as the model allows for an unbounded stack length. The decision procedure devised provides insight on additional techniques for checking ECF in practice.

2.5 Verifying Properties of ECF

In the paper, we show that reasoning about ECF contracts can be performed in a modular fashion. The local reachable states of an ECF contract are only affected by the code of the contract, and cannot be changed by external contracts.

This is useful for program verification and program analysis. For program verification, it means that external calls are treated as a non-deterministic operation that may return an arbitrary value, but cannot change the local state. We utilized this property using Dafny [Leino 2010], to verify correctness of the revised DAO object from Figure 1 (including line 2.5, excluding line 5). When doing so, we ignored the call in line 4, because the return value was not used. We provide a deeper discussion on verifying this example using Dafny in Section 6.

2.6 Summary of the Rest of This Paper

The paper is organized as follows: In Section 3 we formally present the syntax and semantics of our programming language for contracts, called **SMAC**. The ECF property and its different 'flavors' (dynamic vs. static, and notions of equivalence) are presented in Section 4. We discuss decidability results for ECF in Section 5. Section 6 shows the application of the ECF property to achieve modular object-level analysis. The algorithm for online verification of DECF is given in full in Section 7. Results of experiments of running the algorithm on the Ethereum blockchain are presented in Section 8 as well as extensive discussion. Related work is provided in Section 9 and we conclude in Section 10.

3 PROGRAMMING LANGUAGE

We formalize our results for **SMAC**, a simple imperative object-based programming language with pass-by-value parameters with integer-typed local variables and data members (fields). For simplicity, and without loss of generality, every method has a single formal parameter named `arg` and returns a value by assigning it to a designated variable `ret`. Even though we present our

$$
\begin{array}{lcll}
c & \in & \mathsf{PCmd} & \stackrel{\text{def}}{=} \quad x := e \mid F := x \mid x := F \mid \mathtt{assert}(b) \mid x := o(e) \mid \mathtt{skip} \mid \mathtt{enter} \mid \mathtt{return} \\
C & \in & \mathsf{Cmd} & \stackrel{\text{def}}{=} \quad c \mid C\,;\, C \mid \mathtt{if}\ b\ \mathtt{then}\ C\ \mathtt{else}\ C \mid \mathtt{while}\, b\, \mathtt{do}\ C \\
K & \in & \mathsf{Contract} & \stackrel{\text{def}}{=} \quad k\colon \overline{f}\ \ \mathtt{enter}\ \mathtt{var}\ \overline{x}\ \ C\ \ \mathtt{return}
\end{array}
$$

Fig. 5. Syntax.

$$
\begin{array}{llll}
\rho \in \mathcal{E} = \mathsf{Var} \rightharpoonup_{\mathit{fin}} \mathbf{Val} & \text{Local states} & \Gamma \in \mathbf{Stack} = \overline{\mathbf{Frame}} & \text{Stacks} \\
\psi \in \Psi = \mathsf{Fld} \rightharpoonup_{\mathit{fin}} \mathbf{Val} & \text{Object states} & \sigma \in \mathbf{Store} = \mathsf{Cnt} \rightharpoonup_{\mathit{fin}} \Psi & \text{Stores} \\
\gamma \in \mathbf{Frame} = \mathsf{Cnt} \times \mathsf{Cmd} \times \mathcal{E} & \text{Frames} & s \in \mathbf{State} = \mathbf{Stack} \times \mathbf{Store} & \text{States}
\end{array}
$$

Fig. 6. Semantic domains.

theoretical development for contracts in **SMAC**, for readability we use a Java-like notation in our examples, which can be easily desugared.

3.1 Syntax

Figure 5 defines the syntax of **SMAC**. We assume infinite syntactic domains of $k \in \mathsf{Cnt}$, $f \in \mathsf{Fld}$, and $x \in \mathsf{Var}$ *contract identifiers*, *field names*, and *variable identifiers*, respectively. A contract K is identified by of a (unique) *contract identifier* k, and contains a sequence of *field* definitions $\overline{f}$ and a (single nameless) *contract method*. The contract method is comprised of a sequence of *local variable* definitions $\overline{x}$ and a command $C \in \mathsf{Cmd}$. C may be a *primitive command* $c \in \mathsf{PCmd}$ or a *compound command*, i.e., a sequential composition of commands, a conditional, or a loop. A primitive command $c \in \mathsf{PCmd}$ may be either an assignment of an expression e to a local variable x ($x := e$), an assignment of the value of a local variable x to a field F ($F := x$), an assignment of the value of a field F to a local variable x ($x := F$), an *assert* command ($\mathtt{assert}(b)$), a call to a contract method with a single argument e, keeping the returned value in a local variable x ($x := o(e)$), or a `skip` command. Each contract has a single method, thus methods are not named, and may be colloquially referred to using the name of their contract. We also use the terms 'contract' and 'object' interchangeably.

3.2 Semantics

SMAC has a rather mundane stack-based operational semantics, which handle method calls using a *stack* of activation records (*frames*), and uses a *store* to record the values stored in object fields. We refer to a state in which the stack is empty as a *quiescent* state and to a non-quiescent state as an *active* state. Once the execution reaches a *quiescent* state, *any* object method may start running. We refer the reader's attention to three important points: (i) contract states are encapsulated: A contract o can only access its own fields, (ii) local variables are private to their invocation, and (iii) once a contract method is invoked, the semantics is deterministic.

States. Figure 6 defines the semantic domains. A *state* $s = \langle \Gamma, \sigma \rangle$ is a pair comprised of a (possibly empty) *stack* of *frames* $\Gamma \in \mathbf{Stack}$ and a *store* $\sigma \in \mathbf{Store}$, denoted by $\Gamma(s) = \Gamma$ and $\sigma(s) = \sigma$, respectively. The *depth* of a state s, denoted by $\mathrm{Depth}(s)$, is the number of elements in its stack, i.e., $\mathrm{Depth}(s) = |\Gamma(s)|$.

We denote the *top* of the stack in an active state $s = \langle \Gamma, \sigma \rangle$ by $\mathrm{top}(s) = \Gamma(1)$. Intuitively, $\mathrm{top}(s)$ contains the *local state* of the *active* (i.e., currently executing) contract method, while the other frames record the locals states of *pending* calls to contract methods. A *frame* $\gamma = (o,\ c,\ \rho)$ records the *local state* of (a call to the contract method of) an object. Formally, γ is a triple comprised

$$
\begin{array}{rcll}
\langle \epsilon, \sigma\rangle & \Rightarrow & \langle (o,\ \kappa(o),\ [\texttt{arg} \mapsto n]),\ \sigma\rangle & o \in \mathrm{dom}(\sigma), n \in \mathbb{N} \\
\langle (o,\ \texttt{return},\ \rho),\ \sigma\rangle & \Rightarrow & \langle \epsilon, \sigma\rangle & \\
\langle (o,\ x \coloneqq o'(e),\ \rho)\cdot\gamma,\ \sigma\rangle & \Rightarrow & \langle (o', \kappa(o'), [arg \mapsto \rho(e)])\cdot(o,\ x \coloneqq res,\ \rho)\cdot\gamma, \sigma\rangle & \\
\langle (o',\ \texttt{return},\ \rho')\cdot\Gamma, \sigma\rangle & \Rightarrow & \langle (o,\ \texttt{done},\ \rho[res \mapsto \rho'(\texttt{ret})])\cdot\Gamma, \sigma\rangle &
\end{array}
$$

Fig. 7. Operational semantics with respect to a given context $\kappa = \mathrm{Cnt} \rightharpoonup_{fin} \mathrm{Cmd}$ mapping all contracts to their codes. ρ is naturally extended for expressions over variables in LVar.

of an *object identifier*, denoted by $o(\gamma) = o$, a command, denoted by $C(\gamma) = c$, which the method needs to execute, and a *local environment* $\rho \in \mathcal{E}$, denoted by $\rho(\gamma) = \rho$, which assigns values to the invocation's local variables. A *store* $\sigma \in \mathbf{Store}$ is a mapping from a finite number of object identifiers to their *object state*.

Transition relations. We formalize the semantics of our programming language using a *transition relation*. A *transition* is a triple $\tau = (\iota, s, s') \in Tr \subseteq I \times \mathbf{State} \times \mathbf{State}$ comprised of a *transition identifier* ι, denoted by $\iota(\tau)$, a *source state* s, denoted by $\mathrm{src}(\tau)$, and *target* state s', denoted by $\mathrm{trg}(\tau)$. For clarity, we sometimes write a $\tau = (\iota, s, s')$ as $s \Rightarrow_\iota s'$. We denote the active object of the transition by $o(\tau) = o(\mathrm{top}(\mathrm{src}(\tau)))$, or o_{main} if it starts in a quiescent state. We define for each transition the *primitive step*, which is the primitive command that justifies the change in the transition's states, denoted by $c(\tau) \in \mathrm{PCmd}$.

The meaning of primitive and compound commands is standard, and thus omitted. We only mention that primitive commands can only use local variables taken from the top stack frame, and that only the fields of the *current object* can be accessed.

Figure 7 defines meaning of method calls and returns. When an object o is called from a quiescent state, a new stack frame is pushed to the currently empty stack. The frame determines that the active object is o, the command executing is the code $\kappa(o)$ of o, and the local environment for the invocation is the assignment of the value of n to `arg`. The last command in $\kappa(o)$ is always a `return`, after which the frame is popped, leading to a quiescent state. When a call $x \coloneqq o'(e)$ is made from an active state, a new stack frame is pushed as in the previous case. We note that the local environment is initialized by assigning to `arg` the value of e in the local environment belonging to the caller, $\rho(e)$. To handle retrieval of the return value from the callee, the command in the caller is modified to assign to x the value of *res*. When the callee invocation of o' finishes, the command in the top frame is `return` and we let ρ' denote the local environment of the callee. The control transfers back to the caller object o, and the value of *res* is set to be the value of `ret` in ρ'. The assigned value of *res* is then automatically assigned to x, as determined by the operational semantics of the call. The primitive step associated with a call is `enter`, and with a return is `return`. The `return` step in the callee is reduced to a `done` step in the caller, which behaves like a `skip` command.

Executions. An *execution* $\pi = \pi(1) \dots \pi(|\pi|)$ is a finite sequence of transitions coming from Tr. An execution π is *well-formed* if the target state of every transition is the source state of the following one, i.e., $\forall i \in \{2..|\pi|\}.\ \mathrm{trg}(\pi(i-1)) = \mathrm{src}(\pi(i))$. For clarity, we sometimes write an execution π as $\pi = s_1 \Rightarrow s_2 \Rightarrow \cdots s_n$

We say that a transition τ *appears in* a π, denoted by $\tau \in \pi$, if $\pi = \_ \cdot \tau \cdot \_$. We say that a state s *appears in* a π, denoted by $s \in \pi$, if there is a transition $\tau \in \pi$ such that $s \in \{\mathrm{src}(\tau), \mathrm{trg}(\tau)\}$. We denote the sets of transitions and states that appear in an execution π by $\mathrm{States}(\pi)$ and $\mathrm{Transitions}(\pi)$, respectively. An *execution* π' is a *subexecution* of an execution π, denoted by $\pi' \sqsubseteq \pi$, if it is a subsequence of π.

We denote the *first* and *last* states of a non-empty execution π by $\mathrm{src}(\pi) = \mathrm{src}(\pi(1))$ and $\mathrm{trg}(\pi) = \mathrm{trg}(\pi(|\pi|))$. We say that $\pi = \tau\pi'\tau'$ is a *complete execution* if $\mathrm{src}(\pi)$ and $\mathrm{trg}(\pi)$ are quiescent states and π' contains only active states. A *run* is a concatenation of complete executions.

The *minimal* and *maximal depths* of a non-empty execution π, denoted by $\mathrm{minDepth}(\pi) = \min\{Depth(s) \mid s \in \mathrm{States}(\pi)\}$ and $\mathrm{maxDepth}(\pi) = \max\{Depth(s) \mid s \in \mathrm{States}(\pi)\}$ are the minimal, respectively, maximal depths of any of the states it contains.

A well formed execution π' is an *invocation* in an execution π if there exist transition τ and τ' such that $\pi = \_\cdot\tau\cdot\pi'\cdot\tau'\cdot\_$, where $\mathrm{Depth}(\mathrm{src}(\tau)) = \mathrm{Depth}(\mathrm{trg}(\tau'))$ and $\mathrm{minDepth}(\pi') = \mathrm{Depth}(\mathrm{src}(\tau))+1$. We refer to $\mathrm{Depth}(\mathrm{trg}(\tau))$ as the *depth* of the invocation π' and denote it by $Depth(\pi')$. Note that according to this definition, the depth of an invocation that results from calling a contract method on a quiescent state is one.

Traces. We define an *event* as a triple $e = (\iota, o, a)$, consisting of a transition identifier ι, an object o, and a primitive step a. Each transition τ can be transformed to an event by $e(\tau) = (\iota(\tau), o(\tau), c(\tau))$. A *trace* is a sequence of events, denoted by T. The trace matching an execution π is received by point-wise application of $e(\cdot)$ on all the transitions in π, denoted $T(\pi)$.

Conflicts. Two transitions τ and τ' *conflict*, denoted by $Conflict(\tau, \tau')$, if both have the same object ($o(\tau) = o(\tau')$), and their primitive steps $c(\tau)$ and $c(\tau')$ both access the same field of the current object, and at least one of these accesses is a write (i.e., a command such as $F := x$).

Execution equivalence. We define two notions of equivalence of executions: final-state equivalence and conflict equivalence.

Definition 3.1. Executions π_1 and π_2 are *final-state equivalent* if they start in the same state, and finish in the same state:

$$\pi_1 \simeq_{FS} \pi_2 \iff \mathrm{src}(\pi_1) = \mathrm{src}(\pi_2) \wedge \mathrm{trg}(\pi_1) = \mathrm{trg}(\pi_2)$$

Definition 3.2. Executions π_1 and π_2 are *conflict-equivalent* if:

(1) There is a permutation φ of the events of π_1 such that $T(\pi_2) = \varphi(T(\pi_1))$
(2) Conflict-ordering is retained. Namely, if the ith transition conflicts with the jth transition in π_1, then $i < j \iff \varphi(i) < \varphi(j)$.

Formally:

$$\pi_1 \simeq_C \pi_2 \iff \exists\varphi.T(\pi_2) = \varphi(T(\pi_1)) \wedge \forall i, j.(Conflict(\pi(i), \pi(j)) \implies (i < j \iff \varphi(i) < \varphi(j)))$$

4 CORRECTNESS CONDITIONS

In this section we give a formal definition for two notions of the ECF property. We start by formally defining callbacks and callback-freedom in executions.

A stack frame γ *is a callback frame* in a stack Γ if there exist stack frames γ' and γ'' such that $\Gamma = \_\gamma\_\gamma'\_\gamma''\_$ and $o(\gamma) = o(\gamma'')$, but $o(\gamma) \neq o(\gamma')$. A stack Γ contains a *callback*, denoted by $\widehat{\Gamma}$, if it contains a callback frame. A state s contains a *callback*, denoted by $\widehat{s}$, if its stack does, and an execution π contains a *callback*, denoted by $\widehat{\pi}$, if it contains a state s such that $\widehat{s}$. A stack resp. state resp. execution is *callback-free*, denoted by $\neg\widehat{\Gamma}$ resp. $\neg\widehat{s}$ resp. $\neg\widehat{\pi}$, if it does not contain a callback.

In this section, we present two definitions of the general ECF property (effective callback-freedom) for executions. We begin with $\mathrm{ECF}_{\mathrm{FS}}$, which is based on final-state equivalence:

Definition 4.1. An execution π is *equivalently effectively callback-free (DECF_{FS})* if there is a well-formed callback-free execution π' final-state equivalent to π:

$$\pi \vDash \mathrm{DECF}_{\mathrm{FS}} \iff \exists\pi'.\neg\widehat{\pi'} \wedge \pi \simeq_{FS} \pi'$$

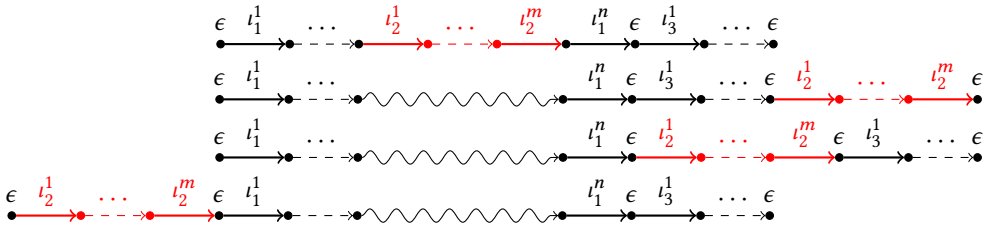

Fig. 8. The callback reorder process. The first graph represents the original execution, which contains a callback (in red). The three other graphs represent all possible callback free executions. Red marks the moved callback transitions. Wave edges indicate that a call was replaced with a havoc transition.

The execution π' is a *witness* for π being an DECF_{FS} execution.

Checking the DECF_{FS} property is difficult in practice, and undecidable in general for models with an infinite state. We describe a stronger definition of ECF, based on conflict-equivalence, called ECF_{C}, which permits an efficient algorithm for checking it. Interestingly, even though executions in our model do not allow for concurrency, callbacks can be thought of as allowing to express a limited subset of concurrent executions. In fact, the ECF property in our model is analogous to serializability in models that permit concurrency. Using this analogy, *invocations* are analogous to *transactions*. We show what this means to reorder invocations in the sequential semantics of **SMAC**.

In general terms, ECF_{C} requires to find a callback-free execution which is conflict-equivalent to the execution with the callbacks. Conflict-equivalence requires that the trace of the callback-free execution is a permutation of the trace of the original execution. It is thus useful to start with a characterization of the legal permutations of an execution. Firstly, the permutation may not break program order of contract code. That is, the permutation must retain the ordering of events whose transitions are part of the same invocation π, and their state have the same depth as $\mathit{Depth}(\pi)$. Secondly, we want to allow permutations that remove callback invocations from their original call location, and sets them to execute in a quiescent state, and still get a well-formed execution.

We extend the definition of well-formed executions to *modular* well-formed executions, which allow *havoc transitions*:

Definition 4.2. An execution π is *modular well-formed* if all its transitions come from Tr or are *havoc transitions*, i.e., transitions of the form $\langle (o,\ x := o'(e),\ \rho) \cdot \Gamma, \sigma \rangle \rightsquigarrow \langle (o,\ \mathsf{done},\ \rho[\mathit{res} \mapsto n]) \cdot \Gamma, \sigma \rangle$ for any values of o, o', e, ρ, Γ, σ and n.

Modular well-formed executions, essentially, permit *havoc transitions* in addition to call transitions. Intuitively, a havoc transition allows to safely overapproximate the only effect that an object o may observe from the invocation of a method on an object o'; technically, a havoc transition allows to replace a call transition with a transition that justifies any return value for the call, without actually executing the call. It has the form $\langle (o,\ x := o'(e),\ \rho) \cdot \Gamma, \sigma \rangle \rightsquigarrow \langle (o,\ \mathsf{done},\ \rho[\mathit{res} \mapsto n]) \cdot \Gamma, \sigma \rangle$.

When we permute a trace such that a callback invocation is removed from its original place, we replace the call transition leading to the callback with a havoc transition. An example can be seen in Figure 8, showing all legal permutations of a trace that has a callback-free execution with havoc transitions.

Using modular well-formed executions, we can formally define the ECF_{C} property for executions, DECF_{C}:

Definition 4.3. An execution π is *conflict-equivalently effectively callback-free (DECF_C)* if there is a modular well-formed callback-free execution π' which is conflict-equivalent to π:

$$\pi \vDash \text{DECF}_C := \exists \pi'. \neg \widetilde{\pi'} \wedge \pi \simeq_C \pi'$$

Conflict equivalence implies final-state equivalence [Bernstein et al. 1987]. Thus, it can be concluded that ECF_C implies ECF_{FS} as we can use the same witness of DECF_C of π for proving π is DECF_{FS}.

Theorem 4.4. *Let π be a DECF_C execution. Then π is an DECF_{FS} execution.*

Finally, as we are also interested in ECF as a property of objects (sECF), we extend the definitions of ECF_{FS} and ECF_C to objects (sECF) instead of executions (DECF). To do this, we utilize the fact that objects are encapsulated to define *projected executions*, which include only transitions that pertain to a single object.

Definition 4.5. Let π be an execution and o be an object which is the current object of a state $s \in \text{States}(\pi)$. The *projected execution of o* is an execution π_o whose trace contains only events $e \in T(\pi)$ such that $o(e) = o$.

Definition 4.6. An object o is sECF_{FS} if every complete projected execution π_o of o is also DECF_{FS}. An object o is sECF_C if every complete projected execution π_o of o is also DECF_C.

5 DECIDABILITY

This section discusses the decidability of verifying ECF. Using Rice Theorem (see, e.g., Hopcroft et al. [2006]), it is easy to show that verifying sECF, namely, statically verifying whether all executions of an object are ECF_{FS} or ECF_C, is an undecidable problem. Interestingly, Rice Theorem can also be used to show checking ECF_{FS} for a single execution (DECF_{FS}) is clearly undecidable. In contrast, checking ECF_C for a single execution (DECF_C) is obviously decidable, as we can enumerate all of the permutations of a particular input trace.

Thus, we focus on verifying sECF, namely, statically verifying whether all executions of an object are ECF_{FS} or ECF_C, where the domains of the object variables are restricted to finite sets. Hence, such objects can be modeled with a pushdown-automaton (PDA). We begin with a rather simple lemma that shows ECF_{FS} of objects is indeed decidable in this model. We assume that the code of all objects is available and that their variables may take values coming from a finite domain.

Lemma 5.1. *Let o be an object, assuming a finite domain for variables. Then there is an algorithm that decides if o is sECF_{FS}.*

Proof. The lemma follows from the decidability result for reachability in deterministic pushdown automata with a finite state [Bouajjani et al. 1997].

Let A be the automaton realizing the code of o. We denote the finite state space of o by Σ and the finite set of arguments with which o can be called with by Arg. Each quiescent state will have a non-deterministic transition to one of the starting states of the object's contract method, determined by the argument choice of $arg \in Arg$. A transition from the quiescent state to a start state adds a frame to the stack, as well as a transition from a call to a start state. A transition from a finish state is removing an element from the stack.

We consider A', an automaton realizing the code of o that does not permit invocations to other objects. Such invocations are replaced with a non deterministic choice of a return value of the invocation (analogous to havoc transitions as mentioned in Section 4). The language recognized by A' is regular, as the stack is not used.

We denote by $R \subset \Sigma$ the set of all reachable states of A. R is finite since Σ is finite. To find R, we run A from its initial (quiescent) state s_0 to find a set of final (quiescent) states S_1. We then run A again from all the states in S_1 to another set of quiescent states S_2. We continue running A like this until we reach a fixpoint containing all reachable states: $R = \cup_{i=1}^{n} S_i$ The number of states of A is finite, hence R must be a finite union of sets of reachable states.

Each complete execution of A starts in a state $s_i \in R \cup \{s_0\}$ and finishes in a state $s_f \in R$. It suffices to find then if s_f is reachable from s_i in A' for each such pair of states s_i and s_f. As the language recognized by A' is regular, this is decidable. In case s_f is reachable from s_i in A' we conclude there is a final-state equivalent callback free execution in A for every complete execution π that starts in s_i and finishes in s_f. If we are able to prove reachability for each pair of states s_i and s_f, then o is sECF_{FS}.

As the number of choices of such pairs is finite, it can be concluded that it is decidable to check if an object is sECF_{FS} in a PDA. □

Showing the decidability of sECF_{C} of objects is not that easy, because it requires reasoning on permutations of events, which is not a regular property, even in the case of finite-state machines.

However, we show that we are able prove the decidability of sECF_{C} in the finite-state case by using induction,

Let $PDA_k(o)$ be a PDA realization of the code of o with a constraint on the maximal length of the stack, set to be k. Longer stacks are rejected. We omit standard details on the building of the PDA, as given in Hopcroft et al. [2006]. An external call in o is a non-deterministic transition to either a new invocation of o (enabling simulation of all possible callbacks), or either to no callback. In addition, we have a non-deterministic choice of the return value of the external call. All these non-deterministic choices are finite.

Our plan to prove the decidability of sECF_{C} is by starting with showing its decidability in PDA_2. We then show that if an object is sECF_{C} in PDA_n, then it is also sECF_{C} in PDA_{n+1}. From this inductive claim, we will be able to conclude that it is decidable for the general PDA as well.

We show a decision procedure for the sECF_{C} of $PDA_2(o)$:

Lemma 5.2. *Let o be an object with a finite state, realized by $PDA_2(o)$. There is a decision procedure that decides if o is sECF_{C}.*

Proof. We denote the finite state space of o by Σ and the finite set of arguments with which o can be called by Arg. o is sECF_{C} if all its *complete* executions are dECF_{C}. We consider complete executions (from a quiescent state to a quiescent state without any quiescent states in-between). The number of complete executions in $PDA_2(o)$ is finite: we have $|\Sigma||Arg|$ choices to begin an execution in depth 1, each such execution has a finite number of transitions in which it performs a call, and in each of these transitions, there are $|Arg|$ options to begin an execution of o in depth 2, and one additional option to not have a callback to o at all. For each external call, there is a finite number of options to choose the return value. Calls in higher depths are not allowed in $PDA_2(o)$.

Checking dECF_{C} for a finite number of complete executions is done as follows: For each complete execution π with maximal depth 2, we can write it as:

$$\pi = \pi_1 \pi_1' \pi_2 \pi_2' \cdots \pi_n' \pi_{n+1}$$

π contains a single invocation in depth 1, given as a series of subexecutions π_i such that $\forall i.Depth(\pi_i) = 1$; and multiple callback invocations $\pi_i' \sqsubseteq \pi$ such that $\forall i.Depth(\pi_i') = 2$. Finding an ECF_{C} equivalent execution involves finding a permutation φ on the events in $T(\pi)$ such that there is a modular well-formed execution π' such that $T(\pi') = \varphi(T(\pi))$. There is a finite number of φ permutations. Specifically, that number is equal to $n!(n+1)$ where $n!$ is the number of internal orderings of the callback invocations $\{\pi_i'\}_{i=1}^{n}$; and $(n+1)$ is equal to the number of options to

partition the callbacks to callbacks that will execute before the outer invocation $\pi_1 \cdot \ldots \cdot \pi_{n+1}$ and callbacks that will execute after it. All these permutations can be checked to be modular well-formed and for retaining the ordering of the conflicts.

By repeating the process of checking ECF_C on all complete executions in $PDA_2(o)$ we can determine if o is sECF_C with maximal stack depth equal to 2. □

We continue with a lemma, from which we conclude that sECF_C is decidable in a a general PDA, if and only if it is decidable in PDA_2, which we already proved in Lemma 5.2. One direction is trivial: Clearly, if sECF_C is found to be decidable in a PDA, then it is decidable in a PDA_2. To prove the other direction, we continue with an inductive claim, that if $PDA_n(o)$ is sECF_C, then $PDA_{n+1}(o)$ is also sECF_C.

Lemma 5.3. *Let o be an object, and $n \geq 2$. We denote $P_n = PDA_n(o)$ be the realization of o up to a stack of depth n, and $P_{n+1} = PDA_{n+1}(o)$ be its realization up to a stack of depth $n+1$. Then, if P_n is $sECF_C$, P_{n+1} is also $sECF_C$.*

Proof. Let π be a complete execution of P_{n+1}. If $\text{maxDepth}(\pi) = 1$, namely, π has no callbacks, then there is nothing to check. Otherwise, π can be written as:

$$\pi = \pi_1 \pi_1' \pi_2 \pi_2' \cdots \pi_k' \pi_{k+1}$$

where π_i are subexecutions such that all transitions start in depth 1: $\forall j.\mathit{Depth}(\text{src}(\pi_i(j))) = 1$. and π_i' are all callbacks, or more formally, subexecutions such that $\forall i.\ \text{minDepth}(\pi_i') = 1$. We consider a specific $i \in \{1, \ldots, k\}$. Because $\text{minDepth}(\pi_i') = 1$, we may denote the bottom element of the stack in all states $\text{States}(\pi_i')$ by γ_i. By ignoring γ_i, we can find a complete execution π_i'' realizable in P_n with the exact same transitions, in which all states have the same stack as in π_i', only with the bottom element γ_i removed. Because P_n is sECF_C, then π_i'' is DECF_C. Therefore, there is a conflict equivalent execution of π_i'', denoted π_i^c, which has $\text{maxDepth}(\pi_i^c) = 1$ and is also conflict equivalent to π_i'. From π_i^c, we can find an execution $\pi_i^{c'}$ in P_{n+1}, which is conflict equivalent to π_i'. We get $\pi_i^{c'}$ by modifying the stack of all states in $\text{States}(\pi_i^c)$ to include γ_i, the bottom element removed from π_i', as the bottom element of each state's stack.

By replacing all π_i' with $\pi_i^{c'}$, we get an execution

$$\pi' = \pi_1 \pi_1^{c'} \pi_2 \pi_2^{c'} \cdots \pi_k^{c'} \pi_{k+1}$$

It is easy to see that π is conflict equivalent to π'. Each pair of conflicting transitions within a specific π_i has the same order in both π and π'. Each pair of conflicting transitions within a specific π_i' retained its order in $\pi_i^{c'}$ because $\pi_i' \simeq_C \pi_i^{c'}$. Finally, each pair of conflicting transitions between different subexecutions of which π are comprised must also appear in the same order in π' because each subexecution that was replaced, was replaced with a subexecution of the same length.

In addition, π' has $\text{maxDepth}(\pi') = 2$, because in each π_i we have $\text{maxDepth}(\pi_i) = 2$ by definition, and each $\pi_i^{c'}$ also has $\text{maxDepth}(\pi_i^{c'}) = 2$ as it is based upon a π_i^c which has $\text{maxDepth}(\pi_i^c) = 1$, but with an additional single stack element γ_i in all states.

We already know that o is sECF_C in P_n, and $n \geq 2$, thus π' is an execution in P_2 and thus must be DECF_C. By transitivity of the conflict equivalence relation, π is also DECF_C, as the same callback-free execution which is conflict equivalent to π' is also conflict-equivalent to π. □

We can immediately conclude for a general PDA, that if an object is sECF_C in a PDA_2, then it is sECF_C when we realize it in a PDA which has unbounded stack.

Corollary 5.4. *Let o be an object. Then, if $PDA_2(o)$ is $sECF_C$, $PDA(o)$ is also $sECF_C$.*

Proof. Assume towards a contradiction that $PDA(o)$ is not sECF_C. Then, there is a complete execution π which is not dECF_C. We denote $n = \text{maxDepth}(\pi)$. Thus, π is a path in $PDA_n(o)$. We saw in lemma 5.3, that if $PDA_n(o)$ is sECF_C, then so is $PDA_{n+1}(o)$. As it is given that $PDA_2(o)$ is sECF_C, then so is $PDA_n(o)$, by induction. Thus, π must be sECF_C, contradiction. □

sECF_C is decidable in a *PDA* as an immediate corollary of the combination of Lemma 5.2 and Corollary 5.4.

Corollary 5.5. *$sECF_C$ is decidable in a PDA.*

6 OBJECT-LEVEL ANALYSIS

While the ECF property is capable of detecting unwanted executions which do not satisfy it, it can be further used for modular analysis of objects. We will show that in environments in which objects are encapsulated, we can consider ECF for executions of a single object only, to help simplify object-level analysis.

We define the notion of a *most general client (MGC)* in our model. The most general client for an object o, MGC_o, is an external program that works on a system that includes a single object in the store. The store σ of MGC_o contains a single object $NoCB(o)$, which is built based on the original object o. Every invocation of an object of the form $x := o'(e)$ in o is replaced with a non-deterministic choice of the value of x as returned by the call, in correspondence with the definition of havoc transitions in Section 4. Furthermore, MGC_o is allowed to repeatedly call $NoCB(o)$ with any parameter and in any order. As such, the semantics of MGC_o soundly approximate all executions of the object o (see, e.g., Gotsman and Yang [2011]), while every execution in MGC_o is in fact a projected, callback-free execution of o.

We show that, if the object o is ECF in the general model, then any object-level assertion can be soundly verified on MGC_o. This is because, all reachable states of an object o in a given system with a store $\sigma \in \textbf{Store}$, are reachable in the system containing only $NoCB(o)$.

Theorem 6.1. *Let R be the set of all states of the object o in a quiescent state, and let R_0 be the set of all states of the object o in a run of MGC_o. If o is $sECF_{FS}$ then $R_0 \supset R$.*

Clearly, the theorem holds if the object is shown to be sECF_C, since this implies it is also sECF_FS (see Theorem 4.4).

This analysis is not overly imprecise, since in real environments, such as Ethereum, we could simulate such behaviors. This is particularly correct since in Ethereum the store of the objects is updateable, and new objects may be added to the system.

Importantly, the analysis simply assumes ECF, and does not require to prove it: An alternative formulation of Theorem 6.1 is assuming that the runtime system enforces dECF on all executions. In that case, the analysis is still sound. It is not unreasonable to assume such a dynamic analysis of dECF, because we found out an efficient method to verify it, presented in Section 7.

We illustrate Theorem 6.1 using the example shown in Figure 1. We implemented `Dao` as a class in Dafny [Leino 2010] with two methods: `deposit` and `withdrawAll`, whose pre- and post-conditions capture the object invariant which should be valid after every execution of the `Dao` object. Primarily, we wish to ensure that the data elements in the `credit` map are not negative, and that the sum of all these elements is equal to the balance of the `Dao`. We model the `pay` method without the recursive call to `withdrawAll`, but annotate it as possibly modifying (any field of) the `Dao` object. This annotation generalizes the possible behaviors of the `Dao` object without the ECF property. With such weak assumptions, and perhaps unsurprisingly, Dafny fails to verify the postconditions for both the original and the fixed versions of the `Dao` object. In contrast, when we assume that the ECF property holds, technically by adding a postcondition to `pay` which ensures

that the previously read fields of the Dao object, i.e., balance and credit[o], are not modified, Dafny is able to establish the post condition. Theorem 6.1 implies the fixed Dao contract respects the given specification when executed using the original runtime system and the original DAO object respects the specification if it is executed on a runtime system which enforces ECF.

7 DYNAMIC VERIFICATION

We describe a sound procedure for verifying the DECF_C property dynamically. More precisely, for each execution, it will check for every object that participates in the execution, if the subsequence of the transitions that pertain only to that object (the *projected execution*) is ECF_C. We assume the existence of an interpreter or virtual machine implementing the semantics defined in Section 3. Below is a description of the data structures used by the algorithm, as well as the instrumentation of the object code to maintain these data structures. We then present a higher-level description of the algorithm, followed with pseudo-code and a complexity analysis. We use the example presented in the overview section in Figure 1 to explain the procedure.

The general structure of the procedure is that the instrumentation step starts every time we exit a quiescent state, and ends when we reach the next quiescent state. Once instrumentation has completed, the algorithm runs on the instrumented structures and returns a result, of whether all projected executions derived from the execution were ECF. The procedure repeats each time we enter an active state.

7.1 Data Structures

A *segment* is a data structure that captures metadata about a portion of the execution's states. This portion consists of a sequence of adjacent transitions, whose top stack frames have the same active object. That is, an invocation of a different object marks the beginning of a new segment, as well as returning from an invocation to a caller invocation which is executed in the context of a different object. However, a call from one object to itself does not break the current segment (This is motivated by the definition of callbacks in Section 4). In simpler terms, a new segment is defined each time the active object changes, either when we push a stack frame with a different object, or pop a stack frame such that the new top frame has a different active object. We show how segments are determined in the instrumentation in Figure 9, using hooks on calls and returns.

Example 7.1. In the example DAO contract in Section 2, an attack execution consists of 6 segments: (1) the first invocation of withdrawAll, lines 1-3; (2) an invocation of pay, lines 3-5; (3) the second invocation of withdrawAll, lines 1-3; (4) a full invocation of pay, lines 3-4,7; (5) the second invocation of withdrawAll, line 5; (6) the first invocation of withdrawAll, line 5;

Definition 7.2 (Segments). A *segment* t is representative of a maximal sequence of adjacent transitions pertaining to the same object. A segment $t = (R, W, D, Idx)$ contains information about fields accessed in the segment, denoted $R(t)$ and $W(t)$ for the read- and write- sets, respectively. In addition, a segment contains information about the depth of the invocation (denoted $D(t)$), which is equal to the depth of the transitions' states. Last, the index in the execution (denoted $Idx(t)$), is strictly increasing according to order of creation of the segments.

The primary metadata saved in each segment is the read and write sets of the fields of the object that were accessed by commands executed in the transitions that pertain to the segment. Other metadata includes the depth of the invocations in the stack, and an index to maintain the order of the segments in the execution.

Example 7.3. We write down the segments that pertain to the DAO object in the overview example of the attack execution, in the same order as they appear in the execution:

$t_1 = (\{credit[\texttt{Attacker}], balance\}, \{balance\}, 0, 1)$ $t_2 = (\{credit[\texttt{Attacker}], balance\}, \{balance\}, 1, 2)$
$t_3 = (\{\}, \{credit[\texttt{Attacker}]\}, 1, 3)$ $t_4 = (\{\}, \{credit[\texttt{Attacker}]\}, 0, 4)$

An execution can be represented as a linear sequence of segments. Furthermore, from these segments we can determine the invocations that the execution contains.

Remark 7.1. *Segments can be used as an alternative representation of executions and invocations, that generalize data saved by a sequence of transitions. In this section only, we redefine the notions of executions and invocations to refer to segments instead of transitions.*

Definition 7.4 (Executions, Invocations, and Callbacks). An *execution* can be represented using a sequence of its instrumented segments $\pi = (t_1, \ldots, t_n)$. We can access the j'th segment of the execution using $\pi(j) = t_j$. We trivially have that $Idx(t_j) = j$. An *invocation* is a sequence of segments $I = (t^I_1, \ldots, t^I_k)$ such that there is a number d for which:

(1) $\forall t \in I.D(t) = d$ (all segments of the invocation are in the same depth).
(2) $d > D(\pi(Idx(t^I_1) - 1))$ (the first segment before the first segment in the invocation has lower depth, proving it is indeed the beginning of an invocation).
(3) $d > D(\pi(Idx(t^I_k) + 1))$ (the first segment after the last segment in the invocations has lower depth, proving it is indeed the end of an invocation).
(4) $\forall j. Idx(t^I_1) < j < Idx(t^I_k) \implies D(\pi(j)) \geq d$ (the invocation does not end before the last segment, that is all segments of depth d in the given range belong to the same invocation).

As all segments included in the invocation has the same depth d, we denote the *depth of an invocation* by $D(I) = d$. We say that an invocation I is a *callback* in another invocation I' (denoted $I \sqsubseteq I'$) if $Idx(I(1)) > Idx(I'(1)) \wedge Idx(I(1)) < Idx(I'(|I'|))$.

Remark 7.2. *Unlike the definition of invocations in Section 3, here invocations capture only the transitions in the same depth as the depth of its first transition, and not transitions in higher depth. This allows to define $D(I)$ for an invocation I.*

Example 7.5. In the attack execution presented in Section 2, the first invocation of withdrawAll is $I_{wd_1} = (t_1, t_4)$, and the second invocation is $I_{wd_2} = (t_2, t_3)$. I_{wd_2} is a callback of I_{wd_1}: $I_{wd_2} \sqsubseteq I_{wd_1}$.

We associate with each segment in depth > 1 a *prefix-set* and *suffix-set* of all segments in the caller that precede, or respectively, proceed it:

Definition 7.6 (Prefix and Suffix segments). Let a set of segments representing an invocation $I = (t^i_{caller})$, and a single segment t_{cb} with $D(t_{cb}) > D(I)$ and $Idx(t_{cb}) \in \{Idx(I(1)), \ldots, Idx(I(|I|))\}$. We define for t_{cb} its *prefix and suffix sets relatively to a caller I* by partitioning the segment in I to segments whose index in the execution is smaller than the index of the callback segment t_{cb} (prefix), and segments whose index in the execution is larger than it (suffix):

$$\begin{aligned} prefix(I, t_{cb}) &= \{t_{caller} \in I \mid Idx(t_{caller}) < Idx(t_{cb})\} \\ suffix(I, t_{cb}) &= \{t_{caller} \in I \mid Idx(t_{caller}) > Idx(t_{cb})\} \end{aligned}$$

Example 7.7. The prefix and suffix segments of t_2 and t_3 with respect to I_{wd_1} are:

$$\begin{aligned} prefix(I_{wd_1}, t_2) = prefix(I_{wd_1}, t_3) &= \{t_1\} \\ suffix(I_{wd_1}, t_2) = suffix(I_{wd_1}, t_3) &= \{t_4\} \end{aligned}$$

The instrumentation process creates the segments and the invocations. We show pseudo-code of the instrumentation procedure in Figure 9.

```
Segment { Obj, Caller, R, W, D, I }
Invocation { Caller, Obj }

Init():
 execution := ()
 curSegment := ⊥
 invocations := Map<Invocation -> Segment>

UponInvocation(object):
 if fromQuiescent // Procedure starts
  Init()

 if object != curSegment.Obj
  caller := fromQuiescent ? TopInvocation : curSegment.Caller
  inv := Invocation(caller, object)
  AddSegment(object, inv, curSegment.D+1, curSegment.I+1)

UponReturn(object):
 caller := toQuiescent ? TopInvocation : curSegment.Caller.Caller
 if caller.Obj != object
  AddSegment(object, caller, curSegment.D-1, curSegment.I+1)

 if caller == TopInvocation // End of instrumentation step
  CheckECFForAllObjects() // Run the algorithm, and finish procedure

AddSegment(object, caller, D, I):
  segment := Segment(object, caller, {}, {}, D, I)
  Append(execution, segment)
  Append(invocations[caller], segment)
  curSegment := segment

UponObjectVarRead(object, F):
 curSegment.R[F] := 1

UponObjectVarWrite(object, F):
 curSegment.W[F] := 1
```

Fig. 9. Instrumentation procedures, implemented as hooks called upon call commands, return commands, and object variable read/write access command. Generates the `execution`, which is a list of segments, and `invocations`, a map of invocation identifiers to an invocation object keeping the caller of an invocation and the list of segments that are part of the invocation in the same depth. The top-level invocation is identified as `TopInvocation`.

The basic check on segments is the commutativity check. We define segment commutativity using read and write sets. We will show that we actually check commutativity of a segment with either a prefix or suffix segment. As the prefix/suffix segments are sets of segments, the read and write sets of prefix/suffix segments are a union of the respective read and write sets of all the segments contained in the prefix or suffix segment.

Definition 7.8 (Commutative Segments). Segments t_1 and t_2 commute, denoted by $t_1 \rightleftarrows t_2$, if:

$$t_1 \rightleftarrows t_2 \stackrel{\text{def}}{=} R(t_1) \cap W(t_2) = \emptyset \wedge R(t_2) \cap W(t_1) = \emptyset$$

If segments t_1 and t_2 do not commute, we denote this by $t_1 \not\rightleftarrows t_2$.

Example 7.9. In the attack execution presented in Section 2, indeed we have that $t_2 \not\rightleftarrows t_1$, as $R(t_1) \cap W(t_2) = \{balance\}$. Similarly, $t_3 \not\rightleftarrows t_1$ because of $credit[o]$, as $R(t_1) \cap W(t_3) = \{credit[o]\}$, and therefore also $t_2 \not\rightleftarrows t_4$. However, t_3 does commute with t_4: $t_3 \rightleftarrows t_4$.

7.2 Algorithm

We start with a high-level description of the algorithm. The algorithm is called every time the system reaches a quiescent state, working on the last complete execution. The algorithm generates a relation of invocations that defines constraints on the ordering of invocations in different stack depths, similar to a 'happens-before' [Lamport 1978] relation. We name this relation the *invocation order constraint (IOC) graph*. For example, if a segment t of a callback invocation I_{cb} is not commuting with its prefix with respect to one of its calling invocations t_{caller} (i.e., $t \not\rightleftarrows \mathit{prefix}(I_{caller}, t)$), then we add the constraint that the invocation of the caller has to occur before the callback: $I_{caller} \prec_{Inv} I_{cb}$. The IOC relation of invocations is thus defined as:

$$\begin{aligned} I \prec_{Inv} I' \quad \stackrel{\text{def}}{=} \quad & (I' \sqsubseteq I \wedge \exists t \in I'.t \not\rightleftarrows \mathit{prefix}(I, t)) \\ & \vee (I \sqsubseteq I' \wedge \exists t \in I.t \not\rightleftarrows \mathit{suffix}(I', t)) \end{aligned}$$

Example 7.10. The IOC relation of the attack execution in Section 2 can be easily calculated with the previous metadata given in examples 7.7 and 7.9. We have that $I_{wd_1} \prec_{Inv} I_{wd_2}$ as $I_{wd_2} \sqsubseteq I_{wd_1}$ and for $t_2 \in I_{wd_2}$, $t_2 \not\rightleftarrows \mathit{prefix}(I_{wd_1}, t_2)$. Similarly, $I_{wd_2} \prec_{Inv} I_{wd_1}$ as for $t_2 \in I_{wd_2}$, $t_2 \not\rightleftarrows \mathit{suffix}(I_{wd_1}, t_2)$.

After the IOC relation is defined, the algorithm considers the graph induced by this relation, and checks it has no cycles. A cycle in the graph could appear if, for example, there is a callback invocation and some caller invocation that contains it, for which there is both (1) a segment that does not commute with its prefix with respect to the caller; and (2) a segment that does not commute with its suffix with respect to the caller. As each vertex in this graph represents an invocation, the topological sorting returns an ordering of the invocations, which is $\mathrm{ECF_C}$. We are merely interested if there is such a topological sorting, that is, if the IOC relation does not contain a cycle.

Theorem 7.11. *Let π be an execution and let Inv be a map of the instrumented invocations to their segments. We denote by $\prec_{Inv}$ the IOC on Inv. If $\prec_{Inv}$ has no cycle, then π is $\mathrm{ECF_C}$.*

Proof. We assume $\prec_{Inv}$ has no cycle. We take a total order $\prec^t_{Inv}$ of Inv induced by the transitive closure on $\prec_{Inv}$. From $\prec^t_{Inv}$ we build a run π' such that every invocation in Inv starts in a quiescent state in the order determined by $\prec^t_{Inv}$. π' is conflict-equivalent to π. To show this, we consider two transitions τ_1 and τ_2 which conflict in π. If τ_1 and τ_2 are both captured in the same segment during instrumentation, then their ordering is kept in π' which only reorders invocations. In particular, the program order of invocations is kept. The same argument applies when τ_1 and τ_2 are not captured by the same segment, but their respective segments are both part of the same invocation. In the general case, τ_1 and τ_2 each belong to different segments, pertaining to different invocations. In that case, their ordering in π' is kept as $\prec^t_{Inv}$ respects that conflict. □

Example 7.12. In continuation to our running example, it is immediate that the IOC relation of the attack execution on the DAO object has a cycle: $I_{wd_1} \prec_{Inv} I_{wd_2} \prec_{Inv} I_{wd_1}$. Therefore, the algorithm cannot determine the attack execution is $\mathrm{ECF_C}$. But indeed, the attack execution is not ECF, thus it cannot be $\mathrm{ECF_C}$.

We already saw in Section 6 that due to state encapsulation, ECF is a modular property. Therefore, the procedure may either check ECF for the entire execution, by searching for a cycle in the full IOC relation, or to check ECF for one object at a time. A modular ECF check can be done by projecting the relation only on invocations of the object under examination. To align with the actual implementation of the algorithm, we chose to present it in its modular version here as well.

We give the complete pseudo-code of the algorithm in Figure 10. It begins with an additional step of preprocessing which is calculating the commutativity matrix of all segments against all prefix and suffix segments of all their enclosing invocations (invocations that directly or indirectly call the

```
CheckECFForAllObjects():
 commute_matrix := CalculateCommutativityMatrix()
 hbRelation := CalculateIOCRelation(commute_matrix)
 for each unique object in execution:
  if not CheckECF(object):
   Print "Object " object " is not ECF"

CheckECF(object):
 // It is guaranteed that IOC applies only to invocations of the same object
 hbRelationO := project hbRelation on invocations of object only
 return isDAG(hbRelation)

CalculateCommutativityMatrix():
 matrix := new Map<Invocation, Segment -> Bool, Bool>
 for each inv in invocations, segment in execution
  if encloses(inv, segment)
   prefix := (s for s in inv where s.I < segment.I)
   suffix := (s for s in inv where s.I > segment.I)
   prefixRS, prefixWS := Union(s.R for s in prefix), Union(s.W for s in prefix)
   suffixRS, suffixWS := Union(s.R for s in suffix), Union(s.W for s in suffix)
   prefixCommute := isCommutative(prefixRS, prefixWS, segment.R, segment.W)
   suffixCommute := isCommutative(suffixRS, suffixWS, segment.R, segment.W)

   if prefixCommute == False && suffixCommute == False
    Abort("Not ECF")
   matrix[inv, segment] := prefixCommute, suffixCommute

 return matrix

encloses(inv, segment):
  return inv.Obj = segment.Obj
        && segment.I between first segment and last segment in inv

CalculateIOCRelation(commute_matrix):
  rel := new Map<Invocation, Invocation -> Bool>
  for each inv1, inv2 in invocations
        if encloses(inv1, first segment in inv2)
          for each segment in inv2
                if commute_matrix[inv1, segment] == False, True
                  rel[inv1, inv2] := True

        if encloses(inv2, first segment in inv1)
          for each segment in inv1
                if commute_matrix[inv2, segment] == True, False
                  rel[inv1, inv2] := True
```

Fig. 10. Algorithm for verifying ECF of an execution. The code of `isDAG` and `isCommutative` is not given. The definition of `isCommutative` is given according to Definition 7.8.

invocation in which the segment is included). The commutativity matrix assists in calculating the IOC relation. We then iterate over all objects encountered in the execution, project the IOC relation on a single object in each iteration, and check if it has a cycle. If the check returns that it is a DAG, then we verified the projected execution is ECF. Otherwise, the cycle describes the invocations which cannot be moved, and helps identify the callbacks that cause the violation of ECF.

7.3 Complexity

7.3.1 Time. The instrumentation step adds a constant factor of work to the runtime. To analyze the algorithm, we begin by looking at the preprocessing steps first. Let n denote the number

of invocations and m the number of segments ($n < m$). In addition, let k denote the maximal number of object variables accessed in an object participating in the execution ($k < m$). The `CalculateCommutativityMatrix` procedure loops on all invocations and all segments. For each pair of an invocation and a segment, the `encloses` predicate can be implemented to take constant time. The calculation of the prefix set and suffix set is taking time linear in the number of segments in an invocation, bounded by m. The time to calculate the read and write sets of the prefix and the suffix set is linear in k. Commutativity check, which involves checking set intersection, where our sets are implemented as associative arrays, is linear in k. Thus the time of `CalculateCommutativityMatrix` is $O(nm(m+k)) = O(nm^2)$. For `CalculateIOCRelation`, we have a loop over pairs of invocations, and another pair of non-nested loops over segments in an invocation, giving $O(mn^2)$. Projecting the IOC relation is linear in its size which is $O(n^2)$. The `isDAG` check is linear in the size of the projected relation, which is bounded by $O(n^2)$. (The graph it represents has $O(n)$ vertices and $O(n^2)$ edges, and checking for a graph to be a DAG is $O(|V| + |E|)$). In total, we have $O(nm^2)$.

7.3.2 Space. The instrumentation adds $O(m)$ space for keeping the segments, and $O(nm)$ for keeping the invocations. The commutativity matrix takes $O(nm)$ space, and the IOC relation takes $O(n^2)$ space. Therefore, the space complexity of the algorithm is $O(nm)$.

8 EVALUATION

We developed a prototype implementation for a dynamic monitor verifying ECF for *Ethereum*.[6] For each execution, it checks if any of the participating contracts has a non-ECF (projected) execution, and outputs all detections of non-ECF executions[7]. We ran our experiments by importing the entire blockchain from its inception in July 30, 2015 until March 30, 2017 [8]. The host we used is a 64-bit Ubuntu 16.04 with two 2.2 GHz Intel Xeon E5-2699 processors (22 cores each with 2 threads per core) and 256 GB of RAM. Both the instrumentation and the algorithm were integrated directly into the *Ethereum Virtual Machine (EVM)* module using hooks, as described in Section 7.

Our monitor worked in 'detect-mode' as to not affect the results, and for statistics gathering only. However, it is trivial to make it work in 'prevent-mode', that actively invalidates and reverts complete executions which are not ECF. Had all the Ethereum clients used such a monitor by design, the DAO incident would have been avoided, along with the controversial hard-fork. As our experiments described below prove, the false-positive rate of the monitor is miniscule: only 10 executions out of about 100 million were legitimately non-ECF. There is also no concern of performance impact, as the measured overhead of running the monitor was less than 3.5%. Furthermore, the benchmarks of the monitor were performed in an ideal environment that actually makes the overhead larger than it is in a normal environment. The reason behind it is that normal environments have additional overheads such as networking and disk accesses, which we disabled in order to scale our experiments.

Experiments. In Figure 11, we show a short list of experiments conducted. We also included the number of contracts created as an additional metric of the blockchain. The primary experiment was checking for ECF in all executions since the creation of blockchain until March 30, 2017. Of note, is that less than 0.01% of the executions were non-ECF. In the second experiment, we processed all executions starting from March 30, 2017 until June 23, 2017 [9].

[6]The source code is available at https://github.com/shellygr/ECFChecker.

[7]The monitor was implemented on top of the Go [Pike 2012] client for Ethereum, called *geth* [Ethereum Foundation 2017a], version 1.5.9.

[8]Without delving into the specifics of the blockchain paradigm, executions are organized in a structure called *blocks*. Our primary experiment was to import the first 3,444,354 blocks of the main Ethereum blockchain.

[9]The second experiment processed all blocks from block no. 3,444,355 to block no. 3,918,380.

Blockchain	Date	Contracts	Executions	Callbacks	Non-ECF (%)
Ethereum	30.VII.2015-30.III.2017	138,457	81,097,421	128,670	3,315 (0.004%)
Ethereum	30.III.2017-23.VI.2017	203,859	15,311,650	155,662	6 (<0.001%)
Eth. Classic	30.VII.2015-29.VI.2017	91,191	32,494,464	81,731	2,288 (0.007%)

Fig. 11. Experimental results. We use dates to mark the portion of the blockchain checked in the experiment. The Contracts column shows how many contracts were created (but not necessarily executed) in the relevant time period. The Executions column records the number of method invocations and the Callbacks column shows how many of these invocations were callbacks. Non-ECF column counts how many non-ECF executions were detected, and their percentage out of the total number of executions.

It is interesting to compare the results of the first experiment, conducted on a snapshot of the Ethereum blockchain taken in Mar. 30th, 2017, which we used for benchmarks, to the second experiment, in which we let our modified client to process all newer executions, until June 23rd, 2017. The number of non-ECF contracts decreased in both absolute quantity and in percentage of total executions. The newer executions expose the maturity of the network, expressed in both the number of total contracts created (almost 150% more contracts created in less than 3 months than in the entire existence of the blockchain, from August 2015 till the snapshot date), and the number of executions[10]. Moreover, the number of executions with callbacks increased significantly, indicating more complex contracts. In the first experiment there were 128,670 executions that contained callbacks, and in the second experiment there were 155,668 executions with callbacks. This amounts to a 641% increase in the number of callbacks in the later period compared with the earlier period. While the percentage of executions with callbacks is still only 1% of all executions, the absolute number of executions with callbacks is large enough to indicate that callbacks are inevitable, either because they are useful, or necessary. This means contracts show an increasing use of callbacks, and thus more complex code, that may be prone to bugs resulting from unintended interaction between contracts. In both experiments, the overall percentage of non-ECF executions out of the executions with callbacks, is 1.17%, and less than 0.01% out of all executions.

Discussion of non-ECF examples. We present a list of all contracts that demonstrated non-ECF executions in Figure 13. Contracts C2, C4 are related to the DAO. C2 is the original DAO [Buterin 2016]. C4 is known as 'The Dark DAO' [Pfeffer 2016], an object containing a copy of the DAO's code, as created by the attack (The mechanism of the DAO was such that, every withdrawal of funds, manifested in the form of a new object whose code is a copy, or 'split', of the DAO code). Contract C1 is an unrelated contract which suffered a vulnerability very similar to the DAO's. The vulnerability, also stemming from non-ECF behavior, was discovered during a security audit and disclosed a short time before the attack on the DAO [Ethereum Reddit 2016; Vessenes 2016]. Contract C5 is an exercise published on the blockchain to demonstrate the DAO attack [B9Lab 2017], and indeed a non-ECF execution was detected. In some contracts it is difficult to pinpoint the exact cause of the existence of non-ECF executions, as the only available code is EVM bytecode, which is not trivial to analyze and reverse. We were trying to connect these incognito contracts with their creators or users. We found evidence that C3 is also related to the DAO [p-s-dev 2016].

The contract at C7 [Etherscan 2017] was traced back to Validity Labs [Val 2017]. We contacted the authors and they provided us with the Solidity source-code of the contract [Validity Labs 2017].

[10]Assuming a new block is generated at an almost constant rate, there were 32 executions per block on average in the second experiment, compared with 23 executions in the first.

```
Object C
  Object Sender
  Method call(data, sender)
    if (Sender != nil) throw
    Sender = sender; ret = this.do(data); Sender = nil

  Method do(data)
    ... // read Sender
```

Fig. 12. Pattern used by contracts C6, C8 and C9. `Sender` is initialized to `nil`. `call` is a method that throws when `Sender` is not `nil`, and otherwise sets it, calls method `do`, and nullifies `Sender` afterwards.

Name	Contract address	Execs.	Execs. w. cbs.	Non-ECF	Stack depth
	Ethereum Network (ETH)				
C1	0xd654bdd32fc99471455e...	924	143	10	3
C2	0xbb9bc244d798123fde78...	274,820	103,064	3,296	2-146
C3	0x34a5451ef61a567ee088...	91	8	1	46
C4	0x304a554a310c7e546dfe...	13,223	2,812	1	3
C5	0x59752433dbe28f5aa59b...	15	6	1	3
C6	0x97361ea911d6348cf2af...	44	42	6	2
C7	0xbf78025535c98f4c605f...	25	22	3	3-9
C8	0x232f3a7723137ced12bc...	144	142	1	2
C9	0x7c525c4e3b273a3afc4b...	35	33	2	2
	Ethereum Classic Network (ETC)				
C1	0xd654bdd32fc99471455e...	850	143	10	3
C2	0xbb9bc244d798123fde78...	195,428	86,573	805	2-146
C3	0x34a5451ef61a567ee088...	18	9	1	46
C4	0x304a554a310c7e546dfe...	14,150	3,064	1	3
C10	0xf4c64518ea10f995918a...	428	177	11	42-122
C11	0xb136707642a4ea12fb4b...	2,582	305	201	17-20
C12	0x0e0da70933f4c7849fc0...	5,330	3,992	1,259	12-57

Fig. 13. A sample of interesting Non-ECF contracts in Ethereum. Contracts are given a name C1, . . . , C12, and are ordered chronologically, by the date of the first non-ECF execution. The Executions and Executions with callbacks columns show statistics on usage style. The Non-ECF column shows how many executions were detected as non-ECF. Stack depth column indicates the range of the depths of the non-ECF subexecutions.

	Time	Memory (max)	Memory (end)
Monitor off	16h 17m	5.5GB	803MB
Monitor on	16h 50m (3.38% overhead)	5.5GB (0%)	940MB (17% overhead)

Fig. 14. Performance statistics. Benchmark experiment was importing the Ethereum main network blockchain, from its creation in July 30, 2015 until March 30, 2017. Compares the import with monitor on or off.

It was deliberately designed to have a DAO-style callback exploit, and was used in their training workshops to demonstrate its dangers.

The same high-level Solidity code of contracts C6, C8, and C9, were provided to us by their creators at Ambisafe [Ambisafe 2017]. The pattern used by these contracts gives rise to behaviors

that are purposefully non-ECF. We show a snippet illustrating the pattern in Figure 12. This pattern is inherently non-ECF.[11] The method `do` assumes the value of `Sender` is not `nil`, but this only occurs in the context of an invocation of `call`. The purpose behind this behavior, is to have `Sender` act as a lock, protecting against unexpected callbacks. Such a design may be avoided in presence of a monitor that allows only ECF executions.

The bottom part of the table in Figure 13 shows non-ECF contracts found in Ethereum Classic [2017]. Ethereum Classic (or ETC) is the continuation of the original Ethereum blockchain following the controversy of the hard-fork due to the DAO bug. Until July 20, 2016, both blockchains, Ethereum and Ethereum Classic, contain the same executions, and thus the same non-ECF executions. Our result and investigation show that all non-ECF executions discovered in the Ethereum Classic network are of copies of the DAO [Bok Consulting 2016a].

Generally, it is important to stress that: (1) there may be other non-ECF contracts, as crafting and deploying contracts that exploit non-ECF entails investment of real money, thus requires a strong incentive to do so; (2) attacking is harder as Ethereum employs (not bullet-proof) heuristics to limit callbacks; (3) a better playground may be the Ethereum TestNet on which we did not run the experiment, but may provide insight as a future work.

The actual overhead measured by enabling the ECF monitor is given in Figure 14. We used the first experiment, where the blocks were imported, as a benchmark. Normally, there is an additional overhead of network download times, which can vary significantly. The measured overhead is about 3.5%, when calculating the difference in time of importing the blockchain with the monitor off, and importing it with the monitor on. We believe the actual overhead is even smaller in most realistic scenarios. First, most clients import the blockchain using the network, which may cause unexpected latencies, unrelated to the monitor. Additionally, the process was pointed to a directory created on a 200 GB RAM disk to improve the scalability of the experiment.[12] Most clients use a physical disk and not a RAM disk. Even if the physical disk is an SSD drive, the experiment slows down significantly, and takes about 20h (18% more than with a RAM disk).

The additional memory footprint measured in the end of the import is about $140MB$, or 17%. It should be noted, that as the implementation is written in Go, which includes automatic garbage collection, the memory consumption varies between tests. The relative difference with the monitor on or off was consistent across repeated tests. The maximal memory used by the process is $5.5GB$ and is not related to the monitor. High memory consumption occurred during the processing of one of the DoS attacks on the blockchain.

9 RELATED WORK

9.1 Modular Reasoning

Modular reasoning is a topic which has been studied extensively with the seminal works of Hoare [1972] and Dijkstra [1976]. For more recent studies on modularity we refer the readers to Banerjee and Naumann [2005]; Leino and Müller [2005].

Averroes [Ali and Lhoták 2013] is a tool for generalizing call-graphs of applications by leaning on a *separate compilation assumption* to generate a general stub library for applications. This allows analysis tools to be modular, as generating full call-graphs is both expensive and imprecise. They

[11] In the formal definition, it actually is ECF, because a call of a contract to itself is not a callback. The contracts under examination were discovered due to a deviation of our monitor's implementation from the full definition of ECF. However, this example can be fitted into a slightly modified pattern which is not-ECF even according to the full definition, by adding an intermediary contract between `call` and `do`.

[12] The Ethereum blockchain suffered a DoS attack [Bok Consulting 2016b; StackExchange 2017] affecting the blockchain in the range of block numbers 2.2M-2.7M, causing all peers participating in the blockchain to make frequent accesses to disk. Running on a RAM disk was necessary to minimize the runtime of experiments.

show how encapsulation assumptions and proofs can be leveraged to improve the feasibility and the precision of analyses. In our work, we give a sufficient condition, ECF, for the ability to soundly reason about a single object in isolation from any other object.

The work of Leino and Nelson [2002] presents an idiom for verifying if an object behaves as expected in the presence of callbacks, called *Valid/state specification idiom*. Every object o maintains a 'valid' bit that indicates if its state is valid, i.e., satisfies its object invariants. The bit should be true in every first invocation of o in an execution. When o calls a method of another object o', o turns off its 'valid' bit. This way, if the execution of o' leads to another method call of o, before the original call to o completed, the code of o can take into account the fact that its object invariants do not necessarily hold. The existence of such a 'valid' bit is helpful to achieve modular soundness, that is the ability to reason about an object in isolation. This paper achieves modular soundness by relying on the encapsulation of the object's state. Essentially, an ECF object is an object for which the 'valid' bit is always turned on, as it is guaranteed that the object state changes only from within the object's methods, and that those methods too are only executed where originally the 'valid' bit would be turned on. Thus, with the assumption on all executions being ECF, there is no need to define a separate behavior of the code for when the 'valid' bit is turned on or off. To enable sound modular reasoning, we simply ignore external calls and assume any return value returned from any such external call. We note that the absence of shared state drastically simplify our life.

Logozzo [2009] presents a method for *modular inference of class invariants*. Specifically, it is shown that the trace semantics of an isolated class are sound and complete with respect to the trace semantics of a whole program. The goal is to find the strongest state-based sound class invariant, that holds in both the isolated and non isolated cases. Abstraction is used in order to compute such an invariant. If it the class invariant matches the specification of the class, then it is ensured that the class itself matches the specification even in the context of a whole program. The mentioned work enables modular reasoning by using abstraction. Our work does not attempt to find such a sound class invariant, but rather to satisfy the necessary conditions for being able to statically verify any specification of an object in isolation of other objects. The benefit here is that we do not depend on the precision of an abstraction, which may output an invariant that overapproximates the specification, and thus does not meet it.

9.2 Verification of Smart Contracts

Even before the events surrounding the bug in the DAO, there were discussions in the Ethereum community about formal verification of smart contracts. Following the extreme measures taken to avert the effects of the attack on the DAO by hard-forking the blockchain and effectively rewrite its history of executions, the discussion became more wide-spread.

Luu et al. [2016] characterized a class of security bugs in smart contracts called *Transaction-Ordering Dependence (TOD)*. A contract inflicted with TOD bugs may behave unexpectedly when there is more than one client using the system and the effect of the execution of one client depends on whether the other client already executed or not. In both TOD and ECF the bugs arise from the fact that the execution is performed in an unexpected state of the contract. However, TOD bugs arise when there is more than one execution (since smart contracts are executed in a distributed environment), whilst non-ECF arises even in a single execution which contains callbacks. One of the solutions suggested for TOD bugs is *guarded transactions*. The idea is to allow contract writers to define guard conditions which are verified by the virtual machine executing the contract code. The execution must satisfy the guard condition, otherwise it fails without any effect. However, by enabling modular reasoning on contracts by proving or asserting at runtime that the executions are ECF, we can verify similar conditions statically. The only addition that may be required for the virtual machine is the online ECF check, which we found to be inexpensive in practice. Checking

arbitrary conditions at runtime may either be inefficient, or not expressible enough to specify fully correct contract behavior. In addition, by verifying at runtime that executions are ECF, we are already able to detect and prevent executions which are, with high probability, unwanted or unexpected.

Luu et al. [2016] presents a tool called *Oyente* [Melonport 2017], based on symbolic execution of contracts. The tool's web version reports on the existence of 'reentrancy bugs', which is how the family of bugs such as the bug in the DAO were dubbed by the Ethereum community. We attempted to verify both ECF and non-ECF contract variations based on the DAO object presented in Figure 1. We received a report on a 'reentrancy bug', even on ECF contracts. We reported the false positives to the web Oyente team, and will submit the issue request by the camera-ready deadline.

The Why3 [Filliâtre and Paskevich 2013] tool was also applied to verify smart contracts written in Solidity. This requires whole code analysis and user supplied loop invariants.

Bhargavan et al. [2016] translate a subset of the high-level Solidity language for Smart Contract development to F*, enabling using F*'s verification framework on Smart Contracts. They also presented a decompiler for EVM bytecode to F*. Similarly to the Why3 approach, the authors faced the issue of translating peculiar syntactic features of the smart contract language Solidity to F*.

It should be noted that both F* and Oyente are successful in detecting other bugs, such as mishandled exceptions. For technical clarity, we omit discussion of the semantics of exceptions and rollbacks in Ethereum. Primarily, to arrive at general results that can be applied in domains other than Ethereum, and secondly, to not overbear the reader with technical details on the myriad ways Ethereum contracts may be invoked, and how exceptions may be handled in each of these ways.

Delmolino et al. [2015] discuss their insights from an educational smart contracts lab they held, and published example contracts used in the lab. We manually analyzed one such contract, implementing a *rock, paper, scissors* game [Delmolino et al. 2016]. We identified several control paths in which a non-ECF execution might manifest. Specifically, there are two control paths in registration to the game (in which players provide a sum as bounty), and three additional paths in the collection of the prize. However, the authors put a constraint on the ability to execute callbacks by limiting to a minimum the amount of *gas* available to the execution. *gas* is a novel concept in Ethereum that effectively bounds the runtime by associating with each low-level opcode a cost. If an execution is not provided with enough *gas* when called, it throws a special *out-of-gas* exception.

Sergey and Hobor [2017] offer an analogy between the nomenclature of Smart Contracts and that of concurrent objects. Specifically, the scenario of a contract calling another contract is compared to cooperative multitasking, in which contract invocation is analogous to the case where the caller yields control. One of the main challenges mentioned is that of being able to verify contracts in isolation of other contracts. The ECF property brings us closer to that goal, by allowing to check properties that can be specified as 'contract invariants' in a modular way.

10 CONCLUSION

In this paper we have presented a simple generic correctness condition for callbacks called Effective Callback Freedom and studied its usefulness. We have shown that it enables modular reasoning in environments with local-only mutable states like Ethereum. We have also shown that in Ethereum it can be used to prevent bugs without drastically limiting programming style, and that it can be checked dynamically with low runtime overhead. In the future, we expect to apply the concept of ECF and prove its usefulness in other environments such as Microservices and Amazon λ.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their helpful comments. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n° [321174], Len Blavatnik and the Blavatnik Family foundation, Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University, and the Pazy Foundation.

REFERENCES

2017. Validity Labs. https://validitylabs.org. [Online].

Karim Ali and Ondřej Lhoták. 2013. Averroes: Whole-program Analysis Without the Whole Program. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 378–400.

Ambisafe. 2017. Ethereum Asset Platform. https://www.ambisafe.co/. [Online; accessed 1-July-2017].

B9Lab. 2017. ING hack challenge. [Online].

Anindya Banerjee and David A. Naumann. 2005. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM* 52, 6 (2005), 894–960.

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

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. ACM, New York, NY, USA, 91–96.

Bok Consulting. 2016a. The DAO ETC Drains. https://github.com/bokkypoobah/TheDAOETCDrains. [Online; accessed 2-July-2017].

Bok Consulting. 2016b. Ethereum Network Attacker's IP Address Is Traceable. https://www.bokconsulting.com.au/blog/ethereum-network-attackers-ip-address-is-traceable/. [Online; accessed 30-June-2017].

Ahmed Bouajjani, Javier Esparza, and Oded Maler. 1997. Reachability analysis of pushdown automata: Application to model-checking. *CONCUR'97: Concurrency Theory* (1997), 135–150.

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

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

Kevin Delmolino, Mitchell Arnett, Ahmed E. Kosba, Andrew Miller, and Elaine Shi. 2015. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. *IACR Cryptology ePrint Archive* 2015 (2015), 460.

Kevin Delmolino, Mitchell Arnett, Ahmed E. Kosba, Andrew Miller, and Elaine Shi. 2016. Ethereum - Serpent Tutorial and Smart Contract Lab. https://github.com/mc2-umd/ethereumlab/blob/master/Examples/RPS_v2_new.py. [Online; accessed 2-July-2017].

Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall.

Ethereum Classic. 2017. Ethereum Classic. https://etcchain.com/. [Online; accessed 2-July-2017].

Ethereum Foundation. 2017a. Geth. https://github.com/ethereum/go-ethereum/wiki/geth. [Online; accessed 30-June-2017].

Ethereum Foundation. 2017b. Solidity. https://solidity.readthedocs.io/en/develop/. [Online; accessed 5-July-2017].

Ethereum Reddit. 2016. From the MAKER DAO slack: "Today we discovered a vulnerability in the ETH token wrapper which would let anyone drain it.". https://www.reddit.com/r/ethereum/comments/4nmohu/from_the_maker_dao_slack_today_we_discovered_a/. [Online; accessed 2-July-2017].

Etherscan. 2017. 0xbfBeA57d87E15529a30B6634C1C13F1A12Fa4d09. https://etherscan.io/address/0xbfbea57d87e15529a30b6634c1c13f1a12fa4d09. [Online; accessed 1-July-2017].

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Proceedings of the 22nd European Symposium on Programming (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 125–128.

Alexey Gotsman and Hongseok Yang. 2011. Liveness-Preserving Atomicity Abstraction. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*. 453–465.

C. A. R. Hoare. 1972. Proof of Correctness of Data Representations. *Acta Inf.* 1 (1972), 271–281.

J.E. Hopcroft, R. Motwani, and J.D. Ullman. 2006. *Introduction to automata theory, languages, and computation*. Addison-wesley.

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, Berlin, Heidelberg, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

K. Rustan M. Leino and Peter Müller. 2005. Modular Verification of Static Class Invariants. In *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*. 26–42.

K. Rustan M. Leino and Greg Nelson. 2002. Data Abstraction and Information Hiding. *ACM Trans. Program. Lang. Syst.* 24, 5 (Sept. 2002), 491–553.

Francesco Logozzo. 2009. Class Invariants As Abstract Interpretation of Trace Semantics. *Comput. Lang. Syst. Struct.* 35, 2 (July 2009), 100–142.

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

Melonport. 2017. Oyente. https://oyente.melonport.com. [Online; accessed 6-July-2017].

Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf.

p-s-dev. 2016. Split DAOs. https://gist.github.com/p-s-dev/e940788a3472fefa1539b327b8628943. [Online; accessed 1-July-2017].

Johannes Pfeffer. 2016. The rise of the Dark DAO. https://medium.com/@oaeee/the-rise-of-the-dark-dao-72b21a2212e3. [Online; accessed 2-July-2017].

Rob Pike. 2012. Go at Google. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. ACM, New York, NY, USA, 5–6.

Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. In *1st Workshop on Trusted Smart Contracts*.

Ethereum StackExchange. 2017. Why is my node synchronization stuck/extremely slow at block 2,306,843? https://ethereum.stackexchange.com/questions/9883/why-is-my-node-synchronization-stuck-extremely-slow-at-block-2-306-843. [Online; accessed 30-June-2017].

Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First Monday* 2, 9 (1997).

Validity Labs. 2017. Recursive call exploit. [Online].

Peter Vessenes. 2016. More Ethereum Attacks: Race-To-Empty is the Real Deal. http://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal/. [Online; accessed 2-July-2017].

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