Let's Move2EVM

Anonymous Authors

Abstract

The Move programming language, designed with strong safety guarantees such as linear resource semantics and borrow-checking, has emerged as a secure and reliable choice for writing smart contracts. However, these guarantees depend on the assumption that all interacting contracts are well-formed—a condition naturally met in Move's native execution environment but not in heterogeneous or untrusted platforms like the Ethereum Virtual Machine (EVM). This hinders the usage of Move as a source language for such platforms: for instance, we show in this paper that the existing Move-to-EVM compiler is not secure, meaning the compilation of secure Move contracts yields vulnerable EVM bytecode.

This work addresses the challenge of preserving Move's security guarantees when compiling to EVM. We introduce a novel compiler design extending the existing Move-to-EVM compiler with an Inlined-Reference-Monitor-(IRM)-based protection layer. Our approach enforces Move's linear semantics and borrow-checking rules at runtime in EVM, ensuring the correctness and safety of the compiled smart contracts, even in adversarial execution environments. We formally define the compilation process, establish correctness guarantees for the translation, and implement our protection mechanism in the original compiler. To evaluate our approach, we compare a standard ERC-20 implementation in Solidity and a contract in Move offering comparable features. Our experiments reveal that execution costs for transferring tokens increase by roughly 33% when using Move. Notably, more than half of this overhead is introduced by the original compiler, which was not focus of any optimization in this work.

1 Introduction

Despite numerous alternatives, Ethereum remains the most widely adopted smart contract platform to date, with a Total Value Locked of approximately $66 billion at the time of writing [5]. With its Ethereum Virtual Machine (EVM), it provides a quasi-Turing-complete execution environment, enabling the deployment of smart contracts, which are, broadly speaking, programs that execute *on the blockchain*.

However, writing secure smart contracts has proven to be far from trivial. In practice, many vulnerabilities stem from a misalignment between the programmer's assumptions about the EVM execution model and its actual behavior. This discrepancy has been the root cause of numerous bugs in the past, often leading to significant financial losses. The infamous DAO hack remains a prominent example of such failures [15], though more recent attacks [16] highlight the ongoing challenges in achieving robust security.

To mitigate these risks, developers rely on human-mandated security audits and automated static analysis tools [18, 25] to detect vulnerabilities. However, while valuable, audits are expensive and offer no formal security guarantees. Meanwhile, automated analysis remains inherently difficult due to properties of EVM bytecode, such as dynamic dispatch and dynamically determined jump destinations, which increase the complexity of program analysis [23, 32].

In response to these challenges, recent research has focused on designing domain-specific languages for writing smart contracts [17, 27, 33, 34]. Among these, the Move language [12], initially introduced by the *Diem* project (formerly *Libra*) and later on further developed by other blockchain projects such as Sui, Aptos, Iota, and Starcoin [1–4], is the most widely deployed. What makes Move unique in the smart contract language literature are its unique *security-by-design* features. In particular, Move introduces statically verifiable well-formedness properties that rule out whole classes of bugs in its smart contracts. For example, reentrancy attacks are rendered impossible by the type system [29]. Furthermore, Move's type system supports digital assets natively and prevents them from accidentally being destroyed while being passed between contracts.

These unique security features make the development and deployment of smart contracts in Move-enabled blockchains particularly appealing. In particular, with a Total Value Locked of roughly $1.8 billion and $1 billion [5], respectively, Sui and Aptos rank among the 11 largest blockchains

at the time of writing, and they are rapidly growing.

Ideally, one would like to carry over the security of the Move language to other blockchains, such as Ethereum, or, to put it differently, use the Move language to program secure Ethereum smart contracts. This is in line with one of Move's central goals, which is to be blockchain agnostic [13]. Indeed, the official Move distribution includes a proof-of-concept compiler from Move to EVM bytecode [8].

Unfortunately, as we show in this work, said compiler is not secure, *i.e.*, the compilation of secure Move code yields insecure EVM bytecode. The fundamental problem is that Move's guarantees rely on the implicit assumption that all deployed contracts are well-typed. This assumption is reasonable–and enforceable through the consensus mechanism–in blockchains natively supporting Move but does not hold in others. The original compiler [8] does not address interactions with non-well-formed contracts, which can occur in the adversarial or untrusted environment of Ethereum. This oversight results in the violation of key safety properties that are fundamental to the Move programming language, such as linear resource semantics, thereby failing to carry over the guarantees provided by the source language to the target blockchain.

Contributions This work presents the first sound compiler from Move to EVM bytecode. In particular, we show how to leverage *Inlined Reference Monitoring* in order to preserve core safety properties of Move when compiling to EVM bytecode, even if compiled code interacts with unknown, possibly adversarial, contracts. More specifically, our contributions are as follows:

- We identify and formalize safety properties intrinsic to Move that must be enforced dynamically in the presence of arbitrary contracts in the less restrictive Ethereum environment.
- We show that the existing Move-to-EVM compiler that is part of the original Move repository [8] falls short of preserving such safety properties. Specifically, we show that the existing compiler fails to preserve the linearity of resources passed to untrusted contracts.
- We devise an Inlined Reference Monitoring technique to enforce Move safety properties in Ethereum and integrate it into the Move-to-EVM compiler [8].
- We formalize and prove the soundness of our compiler against the Move and EVM bytecode semantics [6].
- We evaluate our approach by comparing a standard ERC-20 implementation in Solidity with a Move contract offering similar functionality. By comparing the gas costs of a token transfer, we show that using Move leads to an overhead of approximately 33% (*i.e.*, gas costs of roughly 47K compared to 64K), which we consider a reasonable price to pay to get security by design. Furthermore, as our experiments reveal, more than half of the overhead is caused by the original compiler and not by the security mechanism described in this work.

Related Work Despite the growing popularity of Move and the widespread deployment of EVM, this is the first paper that tackles the secure compilation of the former to the latter. There exist two open-source compilers from Move to EVM [8] and to Solana [7], developed by Diem and Solana Labs, respectively. None of them comes with a formal definition of the compilation process or a correctness proof. In fact, we demonstrate that [8] fails to preserve core properties of Move. Building on [8], we show how this issue can be addressed. Due to the lack of documentation, it is unclear whether [7] preserves core properties of Move. Benetollo et al. introduced the ALGOMOVE framework for adapting Move to the Algorand blockchain [11], showing the feasibility of using Move on a non-native platform. Their work primarily focuses on designing a conceptual framework and demonstrating its viability, omitting many of the detailed aspects of compilation and formal correctness proofs.

Patrignani et al. [29] showed how Move contracts can be proven safe in an adversarial context. Contrary to our work, they assume this context to be well-typed. A general overview of secure compilation techniques that aim to preserve properties of the source language is given in [28]. They, however, explicitly do not consider approaches based on IRMs.

To the best of our knowledge, the concept of Inline Reference Monitoring is first described by Erlingsson *et al*. [20–22]. In [22], the authors describe an approach of compensating missing security checks of older versions of the Java Virtual Machine by monitors, enforcing compliance with those checks dynamically. Backes *et al*. show how IRMs can be used to protect from untrusted applications on the Android Platform [10]. Despite considering different platforms, both [22] and [10] also conceptually differ from our approach as both assume that the untrusted code can be patched to inline dynamic checks, which does not hold for EVM contracts.

A field related to monitoring is the automated patching of vulnerable smart contracts [9, 26, 30, 35–37], as both take insecure smart contracts and turn them into secure ones. The main difference from the subject of this paper is twofold. On the one hand, approaches that take EVM bytecode as input [9, 30, 35–37] have to reconstruct control flow and data layout information in order to be able to modify the code of the contract effectively. As our input language, Move bytecode, is much more structured, we can ignore this concern. On the other hand, all approaches of these papers target generic properties that are immediately and locally enforceable, such as the absence of arithmetic overflows [9, 26, 30, 35, 37] or reentrancies [26, 35, 37]). Violations of such properties can be detected the moment they happen in the affected contract.

Contrary to that, we introduce a method of relaxing the immediate enforcement of properties by asserting them at the start and end of a transaction, thus allowing us to take the behavior of other (non-monitorable) contracts into account.

An outlier in the field of automated smart contract patching is SCREPAIR [36]. First, it works on a high-level representation of smart contracts (Solidity) and, second, it uses an evolutionary algorithm (instead of vulnerability-specific patches) together with a test suite to "mutate" a vulnerable contract into a safe one. Contrary to our approach, SCREPAIR offers no formal guarantee that the mutated contract is semantically equivalent (modulo vulnerabilities) to the original or safe for inputs outside the test suite. Furthermore, it is implausible that a mutation-based approach (intended to correct minor oversights by programmers) would consistently introduce linearity-preserving checks, such as our approach.

2 Overview

In this section, we shortly introduce the Move virtual machine and Ethereum virtual machine (EVM), emphasizing the key differences between them. Next, we give an intuition on the challenges of compiling Move bytecode into EVM bytecode. Finally, we overview our *Inlined Reference Monitors* (IRM) strategy for dynamically enforcing security properties in EVM bytecode and how to leverage that for the secure compilation of Move bytecode into EVM bytecode.

2.1 The Source-Language: Move

In the literature, the term *Move* is used ambiguously to sometimes refer to the source language and sometimes to the bytecode. Our compiler targets Move bytecode, although, for readability's sake, we often use the source language in code snippets. The core properties mentioned in this section apply to both source language and bytecode. Our formalization of the Move semantics extends the one from the literature [13,14,29] to capture halting and exception states of the Move virtual machine (*cf*. Figure 1). Furthermore, we slightly adapt the definition of *locals* to separate them from memory. A full definition of our adapted small-step semantics can be found in the technical report [6].

Modules In Move, programs are represented by modules. We will describe a Move module Ω as a tuple $(\Phi, \mathcal{T}_\Omega)$ consisting of a set of functions Φ and a set of types $\mathcal{T}_\Omega$ defined by the module. A function $\phi \in \Phi$ is defined by its input types, its output types, and its list of instructions. Modules are associated with accounts where each account is identified by its address and can hold multiple modules.

Type System Move is a statically typed language. Besides the *primitive* types typically supported by typed languages (*e.g*., Java, C and C++), Move supports *linear* resource types. These types characterize values that can only be moved but not copied and that can only be created and destroyed by the module that introduced them. Resources also hide their internals, making them only accessible for the module that introduced the respective resource type. Adherence to this discipline is guaranteed by the type system at compile time. For simplicity, we will assume that every type in $\mathcal{T}_\Omega$ is a linear resource type, although technically in Move a module can introduce types that are not linear: such types can trivially be handled by selectively omitting linearity-preserving policies while still enforcing protection against forgery and manipulation of such resources. By ensuring that all modules are well-typed, resources can be passed as parameters to other modules' functions without the risk of them being accidentally *dropped* (*e.g*. by popping them of the operand stack) or manipulated. This allows us to treat resources as first-class citizens, which fits well with the characteristics of digital assets. Besides that, Move's type system also provide a borrow mechanism similar to that of Rust, *i.e*., Move supports references and statically enforces a borrow discipline to avoid invalid references at runtime [12].

Move VM state	$State_{mv}$	:=	(M, G, st, C, A)
Memory	M	:	$Loc \mapsto \mathcal{R}$
Globals	G	:	$\mathcal{A} \times \mathcal{T} \mapsto Loc$
Operand stack	st	$\in$	$\mathcal{L}(\mathcal{V})$
Call Stacks	C	:=	$C_{plain} \mid ERR(c) :: C_{plain}$
			$\mid HALT :: C_{plain}$
Plain Call Stacks	C_{plain}	:=	$(\Omega, \phi, L, pc) :: C_{plain} \mid \varepsilon$

Figure 1: State of the Move Virtual Machine

The Move Virtual Machine The Move Virtual Machine is a stack machine whose state is described by the tuple (M, G, st, C, A) as shown by Figure 1. The memory M is a mapping from the set of locations Loc to the set of resources $\mathcal{R}$. G denotes the globals, i.e., a mapping from *resource identifiers* (a tuple (a, τ) of an account address a and a resource type τ) to a location ℓ inside the memory. The call stack C keeps track of the local states of the modules currently executed. An execution frame of C is a tuple (Ω, ϕ, L, pc) consisting of the function ϕ of the module Ω that is currently executed, the program counter pc denoting the next opcode to execute and the locals L defined as a mapping $\mathbb{N} \mapsto \mathcal{V}$ such that some index l, representing the $l-$th Local, is mapped to some value $v \in \mathcal{V}$. Finally, the operand stack st is a list of values in $\mathcal{V}$. Interestingly, the operand stack is shared across all executions, which allows passing parameters and return values via the operand stack. The type system prevents executions from accessing portions of the stack that belong to other executions.

$$
\begin{array}{rrcl}
\text{Call Stacks} & S & := & S_{plain} \mid EXC :: S_{plain} \\
 & & & \mid HALT(\sigma, d) :: S_{plain} \\
\text{Plain call stacks} & S_{plain} & := & (\mu, \sigma, \iota) :: S_{plain} \mid \varepsilon \\
\text{Machine states} & \mu & := & (pc, m, st, d) \\
\text{Execution environment} & \iota & := & (actor, in, sender, code) \\
\text{Accounts} & acc & := & (stor, code) \in \mathbb{A} \\
\text{Global states} & \sigma & : & \mathbb{N}_{160} \mapsto \mathbb{A}
\end{array}
$$

$$
\begin{array}{lll}
pc \in \mathbb{N}_{256} & d, code, in \in \mathcal{L}\left[\mathbb{B}^8\right] & st \in \mathcal{L}\left(\mathbb{B}^{256}\right) \\
stor\colon \mathbb{B}^{256} \mapsto \mathbb{B}^{256} & actor, sender \in \mathbb{N}_{160} & m\colon \mathbb{B}^{256} \mapsto \mathbb{B}^8
\end{array}
$$

Figure 2: State of the EVM Virtual Machine

2.2 The Target-Language: EVM bytecode

We follow the EVM semantic formalization from the literature [24], simplifying it slightly for presentation by omitting aspects irrelevant to our compilation. For example, we disregard gas costs or any information related to the underlying blockchain infrastructure, which cannot be mapped back to any aspect of the blockchain-agnostic model of Move.

Smart Contracts Moves's modules correspond to smart contracts in EVM, which are simply represented by a sequence of opcodes. The EVM bytecode does not support *functions*. Instead, internal function calls from higher-level languages like Solidity are translated to jumps. Furthermore, EVM bytecode is not typed: any finite sequence of bytes small enough to be deployed is a valid smart contract. All smart contracts are associated with a distinct account and are thus uniquely identified by their account's address.

The Ethereum Virtual Machine The state of the EVM can be described by the current state of its call stack S as described in Figure 9. The call stack keeps track of the different executions of (possibly) different smart contracts. An execution frame is a tuple (μ, σ, ι) consisting of the machine state μ, the global state σ, and the execution environment ι. The machine state μ is a tuple (pc, m, st, d) where pc is the program counter, m (the memory) is temporary storage addressed by words of size 256 bit each mapping to byte-sized cells, st is the local operand stack and d contains the return value of the last call. The global state σ maps account addresses (of size 20 bytes) to accounts represented as a tuple $(stor, code)$ where $stor$ is a word-addressed, persistent storage and $code$ is a (possibly empty) byte string representing the smart contract of the account if it exists. Finally, the execution environment ι is a tuple $(actor, in, sender, code)$ consisting of the address of the executed contract of the respective execution frame, the input data provided by the caller, the address of the caller and the code of the executing contract respectively.

Execution Traces in EVM An execution trace is a sequence of call stacks, each consisting of possibly multiple execution frames describing the execution of a contract. Consider a call stack S of height n. The execution frame at the top of S describes the execution of the currently executed contract. Let us refer to this contract as *Alice*. Upon execution, *Alice* itself can invoke another contract which we refer to as *Bob*. When Alice invokes Bob, the next call stack S' in the trace has a height of $n+1$ with an execution frame on top describing the execution of Bob. We say that the execution of Bob is *in the span* of the execution of *Alice*. The definition of the span of an execution has a recursive nature: If Bob itself invokes another contract - *Charlie* - it runs in the span of Bob and in the span of Alice. Executions in EVM can *revert*. That is, smart contracts can (typically upon encountering an error) abort the execution, reverting all the changes. However, when reverting, not only the changes of the aborting execution are reverted, but also all changes occurred in the span of the execution. Considering the example above, if the execution of Bob reverts, all changes of Charlie are reverted too. Alice can, however, catch the error and decide whether or not to revert.

Key Differences Between Move and EVM bytecode First, we want to address a potential source of confusion: in Move, *memory* refers to global, persistent storage, whereas in EVM, memory denotes temporary, local storage. Despite this difference, the term *memory* is consistently used in the respective contexts within the literature. Hence, we will adhere to this established terminology.

The storage in EVM translates to the globals and the memory in Move. In Move, a location in the memory can point to a value of arbitrary type (and size). However, in EVM, the storage is a mapping from addresses of size 256 bits to words of size 256 bits. Similarly, the operand stack in EVM stores elements of size 256 bits while, in Move, values of any type are pushed onto the stack.

In Move, modules share one global memory to store their data. The type system ensures that a module cannot directly access data belonging to other modules. References to memory locations can be shared between modules if they are crafted in a manner permitted by the type system. In EVM, the storage of different smart contracts is strictly isolated from each other. References to storage can be represented by storage addresses, but cannot be shared between two smart contracts in any meaningful way.

A key difference lies in Move's underlying type system, which inherently prevents certain sources of bugs. For instance, the type system disallows dynamically evaluated call targets. Instead, every external module referenced must be known at compile time, rendering re-entrancy attacks impossible by design [29].

2.3 Challenges

While Move modules are assumed to run in a well-typed environment, no such assumption can be made for smart contracts

```
1  module 0x1::Coin {
2    struct Coin has key {
3      value: u64,
4    }
5    fun getCoin(acc:&signer): Coin acquires Coin{
6      let coin = move_from<Coin>(addr_of(acc));
7      return coin;
8    }
9    fun setZero(coin:&Coin){
10     coin.value = 0;
11   }
12   fun putBack(acc:&signer, coin: Coin){
13     move_to<Coin>(&acc, coin);
14   }
15 }
```

Figure 3: Move code of the `Coin` module

```
1  module 0x2::UntypedAttacker {
2      use 0x1::Coin;
3      fun oups(acc:&signer) {
4          let coin = Coin::getCoin(acc);
5          // Oups! I dropped the coin
6      }
7      fun atm(acc:&signer) {
8          let coin = Coin::get(acc);
9          coin.value += 100;
10         putBack(acc, coin);
11     }
12     fun forgeRef() {
13         let ref:&Coin = ... ;
14         Coin::setZero(ref);
15     }
16     fun forgeCoin(acc:&signer) {
17         let coin:Coin = ... ;
18         putBack(acc, coin);
19     }
20 }
```

Figure 4: Move code showing an untyped *attacker* contract

in EVM. Thus, critical discrepancies arise when compiling a Move module to EVM bytecode. One of the most fundamental challenges lies in the handling of resources. Move's type system enforces linear semantics on resources, ensuring that they are uniquely owned, cannot be duplicated and must be consumed or returned. In contrast, data returned to an *untrusted* contract in EVM can be arbitrarily tampered with, duplicated, or deleted entirely.

Figure 3 and Figure 4 show two modules written in a simplified version of the Move source language. In Figure 4, we relaxed well-typedness restrictions normally enforced in Move, to illustrate the challenge of running Move modules inside an ill-typed environment.

In the example given, the module `Coin` exports the resource type called `Coin`. Along with this type, the module offers some basic functions: The function `getCoin` in line 5 returns the coin of a specific account. The function `setZero` takes a reference to an instance of type `Coin` and sets its value to zero. Finally, the function `putBack` takes a coin as argument and *puts it back* to the memory. While the module `Coin` obeys the rules of Move's type system, these are disregarded by module `UntypedAttacker` in Figure 4. Hence, when `UntypedAttacker` calls the module `Coin`, the following challenges arise:

C1: Dropping Resources First, consider the function `oups` in line 3. It calls the function `getCoin` of the `Coin` module which returns an instance of type `Coin`. However, the function `oups` never consumes the returned instance, breaking the linearity property of the type. Consequently, when leaving the function `oups`, the coin is implicitly *popped* from the stack. In actual Move, this would have been prevented by the type system: instances of linear types must be consumed.

C2: Manipulation of Resources The function `atm` in line 7 does consume the requested coin by calling the function `putBack` of the `Coin` module. However, before doing so, it increases the coin's value. This manipulation of the resource's internal state is prohibited in Move statically: Internals of a type can only be accessed by the module introducing it.

C3: Forging References Next, consider the function `forgeRef` in line 12. It generates a reference to a coin *out of thin air* which, again, would not be possible in Move. In order to create a reference to a resource, one must be in possession of the resource or have received the reference from someone in possession of the resource. However, in an untyped environment, special mechanisms have to be in place to avoid references being created arbitrarily.

C4: Forging Resources Similar to the function `forgeRef`, the function `forgeCoin` in line 16 *forges* a new instance of type `Coin`. However, only the module that introduced the type is allowed to create instances of it. In actual Move this code would therefore not compile.

We note that existing work on compiling Move to EVM [8] is vulnerable to C1, C2 and C4. Challenge C3 is only *prevented* by not supporting references being passed as parameters [6].

2.4 Inlined Reference Monitors

The key idea behind our translation from Move to EVM is to compensate the missing well-typedness guarantees of Move's environment by enforcing *well-behavedness* of the environment dynamically in EVM. For that purpose, we have to identify the semantic properties implied by well-typedness that we are interested in and supplement the compiled smart contracts by dynamic checks that enforce them. The *inlining* of safety checks is a well-known technique referred to in the literature as *Inlined Reference Monitor* (IRM) [20]. In Ethereum, however, we are restricted to dynamic checks included in the bytecode of our module. Hence, the violation of some property can only be detected or prevented when the smart contract obtained by compilation is currently executed. Especially for C1, this poses a problem as the violation seems to happen outside the scope of the compiled code.

A natural way of expressing property C1 is *"eventually, all resources must be consumed"*. Properties of this form

are known in the literature as *liveness properties*. Contrary, properties such as C2, C3 and C4 are of the form *"at no point in execution it may be that ..."* which corresponds to *safety properties*. More formally, a safety property is a property of an execution trace whose violation, contrary to liveness properties, can always be attributed to one particular step in execution. Thus, safety properties like C2-C4 can be verified dynamically by verifying the property at these critical points in execution. For example, while we may not be able to detect a violation of C2 in line 9 in Figure 4, as it happens in the context of `UntypedAttacker`, we can detect it when the manipulated resource is passed back to the `Coin` module.

On the contrary, liveness properties like C1 in general cannot be verified dynamically [19]. However, given that in EVM every execution trace is finite [31], liveness properties of the form *"eventually property ι holds"* can be translated to safety properties of the form *"property ι holds at the end of the execution"*. In a nutshell, this allows us to dynamically verify properties like C1 as follows: (i) ensure that a trusted party controls the end of the execution, (ii) find a way to detect the violation of the property and (iii) make sure to revert every change in case the property was violated. We will elaborate on these points in the remaining part of this section.

IRM for EVM Smart Contracts First, we formally introduce IRMs for EVM bytecode. Our monitor formalization is agnostic to the monitored property making it applicable in more general use cases. Typically, IRMs are described in combination with a *rewriter* that patches the code, *inlining* security checks induced by the monitor. In our case, IRMs get inlined during the compilation.

We formally describe an IRM in EVM as a tuple

$$\mathcal{M} := \left(\left(\iota^+, \psi^+\right), L, \left(\iota^-, \psi^-\right) \right)_F .$$

The set $F \subseteq \mathbb{N}_{32}$ represents a filter containing so-called function selectors. In EVM, when invoking a smart contract, the first four bytes conventionally determine which function of the smart contract is called. Hence, smart contracts usually start with some dispatcher code that reads out the first four bytes and jumps to the area of the bytecode representing the respective function. A monitor $\mathcal{M}$ adapts this dispatcher to ensure that upon invocation of the smart contract, the monitor's logic gets executed whenever the first four bytes of the passed data match one of the selectors in F. The aforementioned monitor logic is represented by the tuples (ι^+, ψ^+) and (ι^-, ψ^-). With ι^+ and ι^- we denote safety properties that are verified at the beginning and at the end of the smart contract execution, respectively. These safety properties are assumed to be *efficiently* decidable within the capabilities of the EVM bytecode language. If verification of the properties fail, the execution is reverted. The symbols $\psi^+, \psi^- \in \mathcal{L}\left(\mathbb{B}^8\right)$ represent sequences of EVM bytecode depicting pre- and post-execution scripts that are executed directly after the verification of ι^+ and ι^-, respectively. Last, we do allow a monitor

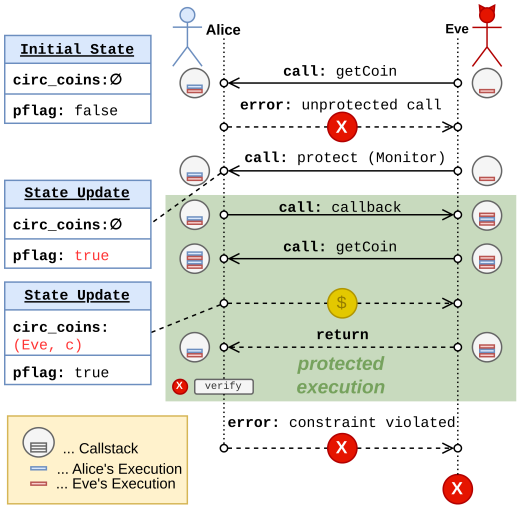

Figure 5: Protecting coins from getting lost

$\mathcal{M}$ to store housekeeping data. Usually, a monitor uses this data to detect violations of its properties. Hence, this data must not be manipulated by anything other than the monitor itself (through ψ^+ and ψ^-) . To define these protected storage areas, a monitor can define a set of storage locations $L \subseteq \mathbb{N}_{256}$ that must not be accessed by the actual smart contract's bytecode. Monitors might, however, be allowed to share data via protected storage locations.

Monitors can be used to naturally formulate the pre- and post-conditions of smart contract invocations. In particular, monitors can be used to tackle C2-C4. Assume, for instance, that our contract is able to keep track of the set of resources created by the contract. The contract can introduce a monitor to verify that each resource passed as a parameter (precondition) is part of the set of created resources, preventing manipulation or forgery (*cf*. C2 and C4).

Monitors can also be used to verify properties like C1. To illustrate that, consider a simple contract that provides a *coin service*: we will refer to this contract as *Alice*. Alice keeps track of a user's coin resource where each coin c has a value associated to it. A user may request their coin (by invoking `getCoin`) or give a previously requested coin back to Alice (by invoking `putBack`). Alice wants to enforce that users give back any requested coin before the end of the execution.

For that purpose, Alice keeps track of two variables inside its own storage: (i) a set `circ_coins` of *circulating coins* consisting of tuples (u, c) where u represents a user and c a coin and (ii) a boolean flag `pflag`. Initially, `circ_coins` is an empty set and `pflag` is set to **false**. Whenever Alice returns a coin to a user (`getCoin`), Alice adds an entry (u, c) to `circ_coins` where u is the caller and c is a copy of the returned resource. When a user u calls `putBack` passing the coin c, Alice first checks if $(u, c) \in$ `circ_coins` and if so, removes it from the set. Now Alice can enforce that every

coin is returned at the end of the execution as follows: First, a monitor $\mathcal{M}^{\phi}_{sec}$ is introduced that ensures that `pflag` is set to **true** when calling `putBack` and `getCoin` and reverts if this is not the case. The second monitor $\mathcal{M}_{prot\text{-}layer}$ introduces a new function to Alice. This can be achieved by setting the set F of functions selectors monitored by $\mathcal{M}_{prot\text{-}layer}$ to $\{f_{\texttt{protect}}\}$ where $f_{\texttt{protect}}$ is a function selector different from that of any other function provided by Alice. The monitor $\mathcal{M}_{prot\text{-}layer}$ thus adds a new function which we will refer to as `protect` and which implements the following monitor logic: First, it verifies that `pflag` is set to **false** and reverts otherwise (corresponds to the safety property ι^{+}). After that, the monitor sets the flag `pflag` to **true** and calls back the caller (corresponds to ψ^{+}). When the caller returns to the monitor logic, the monitor verifies that `circ_coins` is empty and reverts otherwise (corresponds to ι^{-}). Finally, the monitor logic sets back the flag `pflag` to **false** (corresponds to ψ^{-}).

Assuming Alice has the two monitors $\mathcal{M}_{prot\text{-}layer}$ and $\mathcal{M}^{\phi}_{sec}$ in place, we can see how these monitors work in the presence of a malicious contract in Figure 2.4. The malicious contract is denoted with *Eve*. First, Eve tries to invoke `getCoin` directly. However, since `pflag` is still set to **false**, $\mathcal{M}^{\phi}_{sec}$ will revert the invocation. To call `getCoin`, Eve first has to call the function `protect` provided by $\mathcal{M}_{prot\text{-}layer}$ which sets `pflag` to **true** and calls back Eve. Figure 2.4 emphasizes that Eve is now executed in a higher call level by showing a simplified illustration of the call stack next to the sequence diagram. Eve can now call the function `getCoin` which returns Eve's coin and additionally records that Eve owns a coin by updating `circ_coins`, adding the tuple (Eve, c). After that, Eve returns without giving back the coin to Alice. However, upon returning, the control of the execution is passed back from Eve to Alice. More precisely, Eve returns to the monitor logic of $\mathcal{M}_{prot\text{-}layer}$ which now detects that `circ_coins` is not empty. As this violates our constraint, the monitor reverts. Since Eve was executed in the span of the monitor $\mathcal{M}_{prot\text{-}layer}$, all changes made by Eve are reverted too.

Safety Properties for Move After introducing monitors, we conclude by outlining the safety properties we want to protect: With regards to C1, we will show how to keep track of the set of *circulating* resources and enforce the safety property that this set needs to be empty at the end of execution using similar techniques as shown in the example above. Regarding C2, we consider it a manipulation of a resource if a malicious contract passes a manipulated resource to our contract. Consequently, C2 safety properties boil down to input validation. This also applies to C3 and C4.

3 Compiler Model

This section details the key aspects of the compiler design. Our compiler achieves two fundamental tasks. First, it translates Move bytecode to EVM bytecode, mimicking the behavior of the Move virtual machine in EVM. This functionality is relatively standard and borrowed from the original compiler [8]. The second task, which is the novel ingredient introduced in this work, is to regulate the interaction with the *outside world*, *i.e.* other, possibly adversarial, smart contracts. In this section, we give a high-level description of the key aspects of our compiler: we refer to the technical report for the complete formalization [6].

3.1 Basic Strategies for Compilation

First, globals and the memory in Move naturally translate to the storage in EVM as both are persistent storage. In Move, the memory is shared across all modules. Isolation of different module's data is enforced by the type system of the language. In EVM, a contract's storage is inherently isolated from others, eliminating the need for additional measures to protect it from direct access by other contracts. In Move, the globals map users and types to a location in Move's memory containing the actual value. In EVM, we get rid of this indirection and introduce a mapping from users and types directly to the value. Mappings are naturally supported in EVM as the storage is a mapping from 256-bit words to 256-bit words itself.

Locals, as temporary storage available only in the scope of an execution, naturally map to the memory in EVM. For the operand stack, first consider that Move has a typed operand stack where each slot can hold a value of arbitrary type. In EVM, instead, each element of the operand stack has a fixed size of 256 bits. For the primitive types in Move (*e.g.* integers, boolean, *etc.*), this suffices so that we can map stack elements of that type directly to EVM. For more complex types (*e.g.* resource types), in EVM we store a memory location on the stack that points to the actual data stored in memory.

In Move, invoking internal and external functions is conceptually similar. In both cases, a new execution frame is pushed on the call stack of the Move virtual machine. In EVM bytecode, there exists no concept of functions. Thus, in EVM, internal calls have to be simulated by jumps to a region in the bytecode that corresponds to the called function.

Last, in Move, references can point to data in memory (*i.e.*, storage in EVM) and to locals (*i.e.*, memory in EVM). In EVM, inside the trusted contract, we can represent references simply by storage or memory address. To differentiate between references to memory and storage, we follow [8] by using *fat pointers*, *i.e.*, references carrying additional information whether they point to storage or memory.

3.2 Resources in Move

By carefully crafting compilation rules for single opcodes in Move, certain well-typedness properties carry over from Move to EVM. For example, assuming we only compile well-typed Move modules, the generated smart contract adheres

$$M|_{\mathcal{T}_\Omega} := \left\{ \left(\ell, \overline{f}\right) \mid \ell \in Loc, M(\ell).\overline{f} = v, \texttt{type\_of}(v) \in \mathcal{T}_\Omega \right\}$$

$$\begin{aligned} Int_\Omega(State_{mv}) := &\left\{ M(\ell) \mid (\ell, \varepsilon) \in M|_{\mathcal{T}_\Omega} \right\} \cup \\ &\left\{ M(\ell) \mid \left(\ell, \overline{f}.f\right) \in M|_{\mathcal{T}_\Omega}, M(\ell).\overline{f} \in Int_\Omega(State_{mv}) \right\} \end{aligned}$$

$$Ext_\Omega(State_{mv}) := \left\{ M(\ell).\overline{f} \mid \left(\ell, \overline{f}\right) \in M|_{\mathcal{T}_\Omega} \right\} \setminus Int_\Omega$$

$$Tr_{stack,\Omega}(State_{mv}) := \left\{ st[i].\overline{f} \mid 0 \le i < |st|, \texttt{type\_of}\left(st[i].\overline{f}\right) \in \mathcal{T}_\Omega \right\}$$

$$\begin{aligned} Tr_{loc,\Omega}(State_{mv}) := \{ &L(l).\overline{f} \mid (\Omega', \phi, L, pc) \in C, \\ &\texttt{type\_of}\left(L(l).\overline{f}\right) \in \mathcal{T}_\Omega \} \end{aligned}$$

$$Tr_\Omega(State_{mv}) := Tr_{stack,\Omega}(State_{mv}) \cup Tr_{loc,\Omega}(State_{mv})$$

$$State_{mv} := (M, G, st, C, A)$$

f ... field $\quad \overline{f}$ ... path/sequence of fields $\quad \varepsilon$ ... *empty path*

Figure 6: Definition of transients, externals, and internals

to the type system of Move insofar as it avoids dropping resources accidentally. When deployed in an untrusted environment, however, security properties no longer carry over directly from Move. The example in §2.3 shows that an untrusted callee can break linearity by dropping a resource. In this section, we therefore want to focus on how to leverage the techniques introduced in §2.4 to protect core properties of Move even in an *adversarial* context. We will denote the considered Move module with Ω and will refer to the contract resulting from its compilation as the *trusted contract* in EVM. All other deployed contracts we consider *untrusted*.

In Move, resources can reside in different storage areas during execution: First, consider a Move module Ω that defines a resource type `Coin` and consider a state of the Move virtual machine (M, G, st, C, A). Instances of type `Coin` can be located in memory, *i.e.*, there exists an address a such that $G(a, \texttt{Coin}) = \ell$ and $M(\ell) = v$, where v is of type `Coin` . The memory can, however, also store an instance in an *indirect way*: Consider, for example, another Move module Ω' that implements a type `Coin2` which simply wraps `Coin`. Hence, the memory may also contain locations ℓ' such that $M(\ell') = v$, where v is of type `Coin2` and $v.f$ is of type `Coin`, where f is some field name. In general, instances of `Coin` can be wrapped arbitrarily by types of other modules. We treat the memory as a *safe sink* for resources because returning them to memory maintains the linearity of the type. Whether a resource is stored directly or encapsulated within another resource is inconsequential in this regard. Besides memory, resources can, however, also be on the operand stack or in the locals. At these locations they are not allowed to reside, as upon termination, resources would be dropped implicitly.

This motivates the formal definition in Figure 6. Let $\Omega := (\Phi, \mathcal{T}_\Omega)$ be a Move module and $State_{mv} := (M, G, st, C, A)$ be a state of the Move virtual machine. We then formally introduce the set of internals (Int_Ω), externals (Ext_Ω), and transients (Tr_Ω). By considering the memory M, we can *filter out* all resources of type $\tau \in \mathcal{T}_\Omega$. The set $M|_{\mathcal{T}_\Omega}$ contains all *places* in M that contain a resource of type $\tau \in \mathcal{T}_\Omega$ where we define such places by a location ℓ and a path (*i.e.*, a sequence of field names) describing where resource r is stored within resource r' (the empty path ε indicates that r is not wrapped by another resource). The set of internals contains all resources directly accessible by Ω, *i.e.*, either not wrapped by another resource or wrapped by a resource, which itself is in the internals. All other resources in $M|_{\mathcal{T}_\Omega}$ are in the set of externals. The set of transients contains all resources of type $\tau \in \mathcal{T}_\Omega$ that are not yet stored in memory, but instead in the locals of execution frames in the call stack or on the operand stack.

3.2.1 Discussion

Let us now consider how resources are moved between the set of internals, externals, and transients. For that, consider a Move module Ω implementing the type `Coin` together with a function `foo(c: Coin): Coin` that takes a resource of type `Coin` as a parameter and returns an instance of the same type. Furthermore, consider a Move module Ω' invoking `foo`:

Before Invoking Let $State_{mv,inv}$ be the state of the Move virtual machine upon invoking `foo`. Furthermore, let `c` be the resource passed to `foo`. As parameters are passed via the operand stack, it must hold that $\texttt{c} \in Tr_\Omega(State_{mv,inv})$ when Ω' calls `foo`. However, the resource `c` might originate from different types of locations: i) The module Ω' might have wrapped `c` using a type introduce by Ω' itself in which case $\texttt{c} \in Ext_\Omega(State_{mv,init})$ at some initial state $State_{mv,init}$ at the start of the execution. ii) The module Ω' might have called a function `getCoin` also offered by Ω which returned `c`. In that case we have that $\texttt{c} \in Int_\Omega(State_{mv,init})$. iii) The module Ω' might have called a function `createCoin` also offered by Ω which returned a freshly created resource `c`. In this case it did not exist at the begin of the execution and thus $\texttt{c} \notin Int_\Omega(State_{mv,init}) \cup Ext_\Omega(State_{mv,init})$.

After Returning When `foo` is returning at state $State_{mv,ret}State$, the return value is passed by operand stack back to Ω' and consequently there must be a resource `r` (possible equivalent to `c`) such that $\texttt{r} \in Tr_\Omega(State_{mv,ret}State)$. Assuming that `r` was not destroyed (*e.g.* by Ω' calling another function of Ω, destructing `r`), upon termination of the execution it must have been consumed by storing it to memory. Hence, we have that either $\texttt{r} \in Int_\Omega(State_{mv,term}State)$ or $\texttt{r} \in Ext_\Omega(State_{mv,term}State)$ for the state $State_{mv,term}State$ at termination. In any case it has to hold that $\texttt{c}, \texttt{r} \notin Tr_\Omega(State_{mv,term}State)$. Even more, it holds that $Tr_\Omega(State_{mv,term}State) = \emptyset$.

Another interesting observation is that resources, before ending up in the set of internals or externals, are located in the set of transients. This is not surprising as resources have to be constructed at some point (which happens on the operand

stack), and for moving a resource between externals and internals (or the other way around), the resource has to be passed via the operand stack. Moving a resource from the transients to the internals and vice versa can only occur in module Ω (*i.e.*, the module that introduced the respective resource type). Likewise, moving a resource from the transients to the externals and vice versa can only occur in the module Ω' that introduced the type that wraps around our resource.

Considerations for EVM The previous observations indicate that one can represent the behavior of linear types in Move by enforcing rules that define how resources are allowed to move between the sets of internals, externals, and transients. While both the set and the rules for moving resources between those sets are defined implicitly in Move we will make them explicit in EVM.

As we have seen, internals refer to resources that are stored in our module's scope. Thus, these resources are simple to track in EVM as moving them between different sets happens within the trusted contract. Externals, however, present some challenges. There is no way in EVM to verify if an untrusted contract has stored our resource in its storage. This also affects the set of transients: as soon as a resource is passed to an untrusted contract, we do not know if it remains in the transients, is moved to the externals, or, even worse, simply dropped. Consequently, in EVM we store resources in the externals and transients inside the trusted contract. Untrusted contracts can then move resources from one set to the other by invoking dedicated functions of the trusted contract if this is allowed by an access policy described later on.

Keeping track of the set of transients in EVM ensures that resources do not get lost by returning them to untrusted contracts that do not expect to receive such a resource. Allowing untrusted contracts to move a resource from transients to externals corresponds to Move modules wrapping another module's resource and storing it in memory. In EVM, we do not assume that untrusted contracts actually store the resources we pass to them. Instead, we consider moving a resource from transients to externals as the untrusted contract merely claiming it, signaling that it *knows* about the concept of resources and obeys the protocol we enforce on them.

3.3 IRMs for Resource Safety in EVM

In this section, we show how the previously described enforcement strategy can be realized using IRMs as introduced in §2.4. For readability's sake, we chose a more abstract notation to define the monitor's logic. In particular, we assume proper serialization techniques in EVM and thus allow for storing Move's resources in the storage and the memory of the EVM.

First, let us reconsider the set $Int_\Omega(State_{mv})$. This set refers to the resources stored in the storage of the trusted contract in EVM and hence it is implicitly introduced by translating operations on the globals and memory in Move to storage operations in EVM. However, once resources are returned to an untrusted contract, they are no longer in $Int_\Omega(State_{mv})$ and hence would no longer be stored anywhere. We adapt to this by introducing a new structure inside the trusted contract's storage that keeps track of resources returned to untrusted contracts (*i.e.*, set of transients and externals in Move). First, however, we remind ourselves that in Move a module cannot access the internals of a resource of a type introduced by another module. We want to mimic this behavior in EVM by instead of returning the actual resource, returning an *identifier* that identifies the resource outside of our trusted contract. Thus, to keep track of resources outside the trusted contract, we introduce a (partial) mapping $R_\tau\colon \mathbb{N}_{256} \hookrightarrow \mathcal{R}$ (with $\mathcal{R}$ being the set of resources in Move) in the trusted contract's storage for each type $\tau \in \mathcal{T}_\Omega$ introduced by the module. Upon returning a resource r of type τ, the trusted contract generates a fresh identifier id and updates R_τ such that $R_\tau(id) = r$.

By returning an identifier of a resource instead of the resource itself, we have implicitly removed the possibility of resource manipulation by untrusted contracts (*cf.* C2). However, untrusted contracts can forge a resource id (*cf.* C4), allowing them to wrongfully claim a resource. Therefore, for resources in R_τ, we must ensure that access is granted exclusively to contracts that legitimately own the resource. Furthermore, we want to differentiate between resources in transients and externals as described before. Inside the trusted contract's storage, we, therefore, keep track of the two sets $Tr, Ext \subseteq (\mathbb{N}_{160} \times \mathbb{N}_{256})$ containing tuples of the form (a, id) with a being an account address and id a resource identifier as defined previously.

Definition of IRMs We proceed by defining monitors as introduced in §2.4. We apply monitors to verify the boundary points of the trusted contract, *i.e.*, whenever a function of our contract gets called or whenever we return back to an untrusted contract. Based on the function's signature, multiple monitors may be applied. We note that multiple monitors are executed sequentially. Whenever the order of monitor execution matters, it is mentioned explicitly.

First, consider a function ϕ of our Move module Ω that returns a resource of type $\tau \in \mathcal{T}_\Omega$. As mentioned previously, in EVM we instead return a resource identifier id and store the actual resource in R_τ inside the trusted contract's storage. We then apply the following monitor to each such function ϕ that returns a resource:

$$\mathcal{M}^{\phi}_{return} := ((\textbf{true}, \varepsilon), \{Tr\}, (\textbf{true}, Tr \leftarrow Tr \cup \{(caller, id)\}))_{\{\phi\}}$$

Intuitively, this monitor gives the caller access to the returned resource by adding a tuple consisting of the resource identifier and the caller's address to the set of transients.

Once in the set of transients, the caller can *claim* the resource similar to Move modules wrapping a resource of another module. For that, another monitor is introduced that effectively adds a new function to the trusted contract. As

mentioned in §2.4, monitors can add new functions by choosing a function selector that is not yet used inside the monitored contract. If `ST_EXT_SEL` is such a selector, then we introduce the following monitor:

$$\mathcal{M}^{\tau}_{\textit{store-ext}} := \left(\left(\iota_1^+, \psi_1^+\right), \{Tr, Ext\}, (\textbf{true}, \varepsilon)\right)_{\{\texttt{ST\_EXT\_SEL}\}}$$
$$\iota_1^+ := par \in dom\left(R_{\tau_{par}}\right) \wedge (caller, par) \in Tr$$
$$\psi_1^+ := Tr \leftarrow Tr \setminus (caller, par); \quad Ext \leftarrow Ext \cup (caller, par)$$

We assume that *par* refers to the parameter passed to the monitor's logic, *i.e.*, the resource identifier that the caller wants to claim. The monitor then checks if the passed resource identifier is valid by verifying that it is in the domain of R_τ (note that by doing so, we implicitly also have type-checked the resource). In case of a valid resource, the monitor additionally checks if the caller rightfully claims the resource ($(caller, par) \in Tr$) and, if so, moves the resource from the set of transients to the set of externals.

Untrusted contracts might, however, also put the resource back to the trusted contract. Each function ϕ of Ω that takes an input parameter of type $\tau \in \mathcal{T}_\Omega$ represents a possible way of external contracts putting back resources. Thus, for each parameter *par* of type $\tau \in \mathcal{T}_\Omega$ of every function ϕ, we introduce the monitor:

$$\mathcal{M}^{\phi, par}_{\textit{par-verif}} := \left(\left(\iota_2^+, \psi_2^+\right), \{Tr, Ext\}, (\textbf{true}, \varepsilon)\right)_{\{\phi\}}$$
$$\iota_2^+ := par \in dom\left(R_{\tau_{par}}\right) \wedge (caller, par) \in (Tr \cup Ext)$$
$$\psi_2^+ := Tr \leftarrow Tr \setminus (caller, par); \quad Ext \leftarrow Ext \setminus (caller, par)$$

Similar to before, we first verify the validity of the passed resource. A caller rightfully puts back a resource if $(caller, par)$ is either in the transients or externals. Note that by also considering externals here, we make the move from transients to externals implicit. After that, our monitor removes the resource from externals and transients. By our previous observation we would expect the resource to remain in the transients. The trusted contract will, however, not drop resources by construction, thus obviating tracking resources during its execution.

Additionally, we allow untrusted contracts to transfer resources to each other by adding a suitable function via a monitor. Thus, for every type $\tau \in \mathcal{T}_\Omega$ we introduce:

$$\mathcal{M}^{\tau}_{\textit{move-to}} := \left(\left(\iota_3^+, \psi_3^+\right), \{Tr, Ext, \texttt{pflag}\}, (\textbf{true}, \varepsilon)\right)_{\{\texttt{MV\_TO\_SEL}\}}$$
$$\iota_3^+ := par_1 \in dom\left(R_{\tau_{par_1}}\right) \wedge (caller, par_1) \in (Tr \cup Ext) \wedge \texttt{pflag} = \textbf{true}$$
$$\psi_3^+ := Tr \leftarrow (Tr \setminus \{(caller, par_1)\}) \cup \{(par_2, par_1)\}; \; Ext \leftarrow Ext \setminus \{(caller, par_1)\}$$

As before we assume `MV_TO_SEL` to be a fresh function selector. The monitor introduces a function that takes two input parameters: par_1 refers to the resource-to-be-moved and par_2 to the new owner. The monitor verifies the validity of the resource identifier and the ownership of the caller as explained before. After that, it removes the permission of the caller from the transients and externals and adds the tuple (par_2, par_1) to the set of transients effectively giving the account with address par_2 access to the resource with id par_1.

Last, we want to ensure that every resource in transient has either been put back or claimed (*i.e.*, moved to externals) at the end of execution. For that, we use the technique shown in §2.4, *i.e.*, the trusted contract keeps track of a boolean flag `pflag` inside its storage and we introduce the monitor $\mathcal{M}_{\textit{prot-layer}}$ defined as follows:

$$\mathcal{M}_{\textit{prot-layer}} := \left(\left(\iota_P^+, \psi_P^+\right), \{Tr, \texttt{pflag}\}, \left(\iota_P^-, \psi_P^-\right)\right)_{\{\texttt{PROT\_LAYER}\}}$$
$$\iota_P^+ := \texttt{pflag} = \textbf{false} \quad \psi_P^+ := \texttt{pflag} \leftarrow \textbf{true}; \; \texttt{call}(caller)$$
$$\iota_P^- := Tr = \emptyset \quad \psi_P^- := \texttt{pflag} \leftarrow \textbf{false};$$

Note that `PROT_LAYER` is again a fresh function selector. As already shown in §2.4, this monitor introduces a new function that sets `pflag` to true, calls back the caller, and upon the caller returning, verifies that the set of transients is empty. For each function ϕ we then introduce the monitor $\mathcal{M}^{\phi}_{sec}$ which simply verifies that `pflag` is set to true, *i.e.*, that the respective function was called in the span of the protection layer.

Now reconsider the code snippet depicted in Figure 3 and Figure 4, but this time we assume that our previously defined monitors are protecting us from `UntypedAttacker` dynamically. First, consider C1. Here the problem occurred when the function `oups` of `UntypedAttacker` was calling the function `getCoin` of the module `Coin`. Now with the monitors in place, $\mathcal{M}_{\textit{prot-layer}}$ together with $\mathcal{M}^{\phi}_{sec}$ will ensure that `getCoin` is always invoked in the span of $\mathcal{M}_{\textit{prot-layer}}$ (*cf*. Figure 2.4). Due to `getCoin` returning a resource, the monitor $\mathcal{M}^{\phi}_{return}$ ensures that the resource is put in the set of transients. Hence, when the function `oups` terminates, the set of transients is not empty. As `oups` must have been executed in the span of the monitor $\mathcal{M}_{\textit{prot-layer}}$, we will at some point reach its post-condition check which will fail as the set of transients is non-empty.

We have already seen that only returning a resource id instead of the actual resource resolved the manipulation of resources shown by C2. Furthermore, a forging of resources as described by C4 is prevented by our monitors: For each parameter of a resource type, the monitor $\mathcal{M}^{\phi, par}_{\textit{par-verif}}$ is verifying if the resource exists and if the caller possesses the resource. With regards to the previously defined challenges it remains to show how to overcome the problem described by C3. For that, however, we first have to consider how to handle references outside of the trusted contract in EVM.

3.4 IRMs for Reference Safety in EVM

In Move, references can be mutable or immutable, letting modules share storage with others for read-write or read-only access. We have already seen how references are represented in EVM inside the trusted contract. Due to the strict isolation of memory and storage in EVM, sharing these references between contracts is, however, not possible (see §3.6). Resource

references behave slightly differently: While leaking a reference to a primitive type (*e.g.*, an integer) allows external modules to write to the corresponding storage location, mutable resource references prevent, like resources themselves, accessing their internals. Instead, mutable resource references only allow the module that defines their type such access.

In Move, references obey a certain discipline enforced statically by Move's borrow-checker [14]. While the trusted contract inherits most of the properties internally, special care is required when passing references as parameters from an untrusted environment to the trusted contract. Without going into details of the borrow-checker's rules, we have to enforce the following properties dynamically: (i) the caller must rightfully *own* a reference, (ii) mutable references are not allowed to alias any other reference, (iii) reference parameters are not allowed to reference a resource passed as parameter.

Regarding (i), in Move, a resource reference can be created by the owner of the resource who can then share it. While for resources we explicitly allow untrusted contracts to transfer them to other untrusted contracts, we will not support a similar way of sharing for references. Rules enforced by the borrow-checker obey a strict discipline that is tricky to map correctly in an untrusted environment like EVM. Hence, for simplicity, we will only allow contracts that currently own a resource to reference it, *i.e.*, in order for a contract to rightfully own a resource reference, the referenced resource must be owned by the contract according to the set of transients and externals. Similar to resources themselves, we will represent references to them by their resource identifier.

To keep track of references, we introduce yet another structure inside the storage of the trusted contract, namely, the set $Ref \subseteq \mathbb{N}_{256}$ which initially is empty. Intuitively, a resource id is in the set *Ref* if it is currently mutably referenced by some of the parameters. Now first, consider a function ϕ with a mutable reference parameter *par* of type $\&\mathbf{mut}\,\tau_{par}$ with $\tau_{par} \in \mathcal{T}_\Omega$. For each such parameter and function we introduce the following monitor:

$$\mathcal{M}^{\phi,par}_{ref\text{-}mut} := \left(\left(\iota_4^+, \psi_4^+\right), \{Tr, Ext, Ref\}, \left(\mathbf{true}, \psi_4^-\right)\right)_{\{\phi\}}$$

$$\iota_4^+ := \iota_2^+ \wedge par \notin Ref$$

$$\psi_4^+ := Ref \leftarrow Ref \cup \{par\} \qquad \psi_4^- := Ref \leftarrow Ref \setminus \{par\}$$

Note that ψ_2^+ is the resource validation check we have already used for $\mathcal{M}^{\phi,par}_{par\text{-}verif}$ before. Additionally we also have to check if the referenced resource is already referenced by another parameter. For that, we check if $par \in Ref$. If this validation succeeds, the monitor marks the passed resource id as being mutably referenced by adding it to *Ref*. At the end of the function, we *release* the resource id by removing it from *Ref* again. For all immutable reference parameters, we apply the same pre-condition check but do not mark the passed resource id as being mutably referenced. Thus, for immutable reference parameters, the monitor is simply defined as

$$\mathcal{M}^{\phi,par}_{ref\text{-}imm} := \left(\left(\iota_4^+, \varepsilon\right), \{Tr, Ext, Ref\}, \left(\mathbf{true}, \varepsilon\right)\right)_{\{\phi\}}$$

Note that the order of monitor execution is crucial here: First $\mathcal{M}^{\phi,par}_{par\text{-}verif}$ has to be executed to remove resources being passed as parameters from *Tr*. After that, we execute $\mathcal{M}^{\phi,par}_{ref\text{-}mut}$ to mark all resources mutably referenced and last $\mathcal{M}^{\phi,par}_{ref\text{-}imm}$.

With these additional monitors in place, let us reconsider C3 showing an untyped attacker simply forging a reference. In the example, we do not specify if it is a mutable or immutable reference. In both cases, the behavior is, however, the same: due to the policy enforced by the monitors we require `UntypedAttacker` to be the owner of the resource in order to reference it, which does not hold here. Consequently, the monitor's precondition will fail which aborts execution.

A brief comment on reentrancy In Move reentrancy is prevented by the type system: dependencies between modules have to be known statically and must be acyclic. In EVM, we have to enforce that dynamically by introducing another monitor for every function ϕ:

$$\mathcal{M}^{\phi}_{reentr} := \left(\left(rf = \mathbf{false}, rf \leftarrow \mathbf{true}\right), \{rf\}, \left(\mathbf{true}, rf \leftarrow \mathbf{false}\right)\right)_{\{\phi\}}$$

This monitor implements a common approach for protecting against reentrancy dynamically: We introduce a new storage variable *rf* that is asserted to be false at the start of each method and then set to true. After leaving the function we set back the flag to false, effectively unlocking the contract again. This monitor is executed before any other monitor.

3.5 Formal Results

In this section, we give a rough overview of the formal results of our approach. For details of the proof, we refer to the technical report [6]. We consider a well-typed Move module Ω and its compilation to EVM denoted by $[\![\Omega]\!]^{\texttt{move}}_{\texttt{evm}}$. We will use the term *unit* to refer to the module Ω and the contract $[\![\Omega]\!]^{\texttt{move}}_{\texttt{evm}}$ in their respective context. In simple terms, we want to show that each behavior encountered in EVM can be mapped back to a behavior in Move. For that purpose we introduce a term of *similarity* between a state $State_{mv}$ in Move and a state $State_{evm}$ in EVM which we denote by $\simeq_\Omega : \subseteq States_{mv} \times States_{evm}$. While this relation cannot be maintained after every computational step (in particular because one step in Move may correspond to many steps in EVM), it does hold true at boundary points, that is, entry and exit points of the module and contract. Formally, we introduce a *trace semantics* for both Move and EVM that describes an execution as a sequence of actions α of the form

$$\alpha := \texttt{call}\ S? \mid \texttt{call}\ S! \mid \texttt{ret}\ S? \mid \texttt{ret}\ S! \mid \texttt{term}\ S$$

where we borrow parts of the notation from the work of Patrignani *et al.* [29]. Actions are X either calls to/from or returns to/from the considered unit. An action additionally describes the state before leaving/before entering the unit and its direction, *i.e.*, either from the environment to the unit (denoted by

?) or the other way around (denoted by !). `term` S describes the successful termination of the execution. We write α_{mv} and α_{evm} to denote an action in Move and EVM respectively and write $\alpha_{mv} \simeq_\Omega \alpha_{evm}$ if the action are of similar kind (*i.e.*, direction and type of action is the same) and for the two carried states $State_{mv}$ and $State_{evm}$ it holds that $State_{mv} \simeq_\Omega State_{evm}$.

A *trace* is a sequence of actions denoted by t. We can lift the similarity relation for traces in a natural way: Given a trace t_{mv} and a trace t_{evm} in EVM we write $t_{mv} \simeq_\Omega t_{evm}$ if $|t_{mv}| = |t_{evm}|$ and $t_{mv,i} \simeq_\Omega t_{evm,i}$ for each $0 \leq i < |t_{mv}|$. A trace is called *complete* if it ends with the action `term` S, *i.e.*, the execution terminates successfully. We then say that a module Ω in Move *admits* a trace and write $\Omega \vdash t_{mv}$ if the t_{mv} is valid for Ω with respect to the semantics of Move. Likewise we define the notion $[\![\Omega]\!]^{\texttt{move}}_{\texttt{evm}} \vdash t_{evm}$ for EVM.

Theorem 1. *For every Move module Ω it holds that if $[\![\Omega]\!]^{move}_{evm} \vdash t_{evm}$ in EVM such that t_{evm} is a complete trace, then there exists a complete trace t_{mv} in Move such that $t_{mv} \simeq_\Omega t_{evm}$ and $\Omega \vdash t_{mv}$.*

Roughly speaking, the theorem states that if an unintended behavior of the compilation $[\![\Omega]\!]^{\texttt{move}}_{\texttt{evm}}$ was encountered on EVM, a similar behavior can also be encountered in *Move*. Thus, the observed behavior was not introduced by the compiler but instead by the original Move code itself.

3.6 Limitations of the Compiler

First, we assume that our Move modules do not have dependencies to other Move modules (besides standard libraries). This effectively means that we assume that our modules are only called and do not make external calls themselves. While this might appear to be a significant restriction, we would like to offer some clarifying remarks on this point: In Move every module called has to be known at compile time. By arguing that one can verify the callee at compile time, it can be considered as trusted code [29]. Furthermore, when dealing with an untrusted callee, it is not straightforward to differentiate between the protection obligation of the caller (our module) and that of the callee. For instance, if a malicious callee returns a resource, they could later claim that the resource is invalid when the caller wants to return it. Here, it is unclear if this is an attack on the caller's integrity or not.

Second, we allow reference parameters and return values to reference only resources. Additionally, return values are not allowed to reference to internal fields of a resource. Regarding the first point, note that references to primitive types (*e.g.* integers) would allow external contracts to access specific parts of the memory and storage of a contract, which in EVM cannot be mapped directly. Leaking references to internals (even if it is a resource reference) would require a more complex representation of references which, for simplicity, we do not currently support. Also, as shown by Patrignani *et al.* leaking such references may break module-specific safety invariants [29] and hence could be considered bad practice.

Last, for the sake of simplicity, we disregard vectors that are natively supported in Move. There is no limitation in our approach that would require such simplification. However, it reduces the set of opcodes that have to be considered and simplifies memory and storage handling as the size of all values can be assumed to be statically known.

4 Implementation and Experiments

In this section, we first review the original Move-to-EVM compiler and then describe the implementation of our IRM-based protection layer. To differentiate, we will refer to the original compiler as *Move-to-EVM compiler*. To our adapted version we will refer to as *IRM-based compiler*. Finally, we present and discuss the experiments we conducted.

4.1 Compiler Implementation

The original compiler The Move-to-EVM compiler produces fully functional EVM-compatible smart contracts by compiling Move language to EVM bytecode. The compilation process consists of four main stages. The first encompasses the standard steps of a typical compilation pipeline. Next, Move code is converted into a stackless-bytecode intermediate representation, as the one outlined in [14]. This intermediate form retains semantic equivalence with the original Move bytecode while simplifying program analysis and transformation, acting as a bridge between Move's high-level semantics and lower-level representations. In the third stage, the Move stackless bytecode is converted into Yul.[1] This is where the actual translation from Move to EVM takes place. Finally, the Yul code is compiled into EVM bytecode using the Solidity compiler, producing smart contracts that can be deployed and executed on the Ethereum blockchain.

As we have shown in §2.3 and discussed on several occasions, the original compiler fails to preserve core properties of Move. In particular, the linear semantics of resource types is not preserved when resources are returned to untrusted contracts, leading to a potential loss of resources. Furthermore, modules compiled with the original compiler are vulnerable to reentrancy attacks which is highly problematic, as developers might not consider this attack vector when writing Move code, given that reentrancy is ruled out by design in Move [29].

IRM-based compiler The implementation of the IRM protection layer is integrated into the third stage of the compilation pipeline, where Move bytecode is translated into Yul. Incorporating the protection layer at this stage enables fine-grained control over the emitted code, ensuring that the additional logic aligns precisely with the target EVM bytecode and minimizes unnecessary overhead.

[1] Yul is a low-level IR designed for compiling Solidity to EVM bytecode.

Implementation	Compiler	Gas Cost
ERC-20	Solidity	47812
ERC-20MV	Move-to-EVM compiler	89343
ERC-Coin	Move-to-EVM compiler	57026
ERC-Coin	IRM-based compiler	64245

Figure 7: Comparison of gas costs for the `transfer` function across different contract implementations.

The IRM implementation involves embedding runtime checks at critical points within the function dispatcher. These checks are inserted both before and after invoking the core logic of each function to validate the integrity of incoming parameters, such as resources and references, and to ensure proper handling of returned resources. Additionally, we introduce specialized functions within the dispatcher that complete the functionality of the protection layer.

To optimize the gas costs of the protection layer, our implementation leverages the transient storage feature recently introduced in Solidity 0.8.26. This feature allows for temporary storage during contract execution without incurring the higher costs associated with persistent storage.

4.2 Experiments

In our experiments, we compare the results of four different contract-compiler configurations. First, we show the gas costs of a classical ERC-20 contract compiled with the Solidity compiler 0.8.26. Second, to show the overhead induced by the Move-to-EVM compiler when writing contracts in Move, we measured gas costs for an ERC-20 implementation written in Move (denoted by *ERC-20MV*) and compiled with the original Move-to-EVM compiler. The code for this contract is included in the repository of the original compiler [8]. Third, we developed in Move an ERC-20-like coin contract with features comparable to those of the ERC-20 standard. We denote this contract by *ERC-Coin*. Contrary to ERC-20MV, which does not leverage features of the Move language (like resource types) and uses only EVM-specific features supported by the Move-to-EVM-compiler, the implementation of ERC-Coin uses resources to represent ERC-20 tokens. This contract is compiled once with the Move-to-EVM compiler and once with the IRM-based compiler. We stress that compiling ERC-Coin with the Move-to-EVM compiler does not yield secure code, as discussed in §2.3. However, it serves as a baseline for evaluating the costs introduced by our approach.

To evaluate performance, we chose to compare the gas costs incurred during the execution of the `transfer` function across the different implementations. The results of this measurement are shown in Figure 7. First, consider the significant difference in gas costs between the ERC-20 in Solidity and ERC-20MV compiled with the Move-to-EVM compiler. By using Move in an unidiomatic way, gas costs increase by over 80%, thereby reducing incentives for switching to Move. However, consider the results for ERC-Coin compiled with the original Move-to-EVM compiler: compared to the ERC-20 version written in Solidity, gas costs have increased by about 19% making it significantly cheaper than the ERC-20MV contract and comparable to the ERC-20 contract written in Solidity. Still, with the original compiler, no security measures are in place, making our contract vulnerable as shown previously. The overhead induced by our protection mechanisms is shown in the last line of Figure 7. With our protection layer in place, gas costs have increased by roughly 33% compared to the ERC-20 contract written in Solidity. By comparing the gas costs of *ERC-Coin* compiled with the Move-to-EVM compiler, we see that our protection layer accounts for only 43% of the overhead while the rest is caused by the original Move-to-EVM compiler. Particularly, we want to emphasize that, while the Solidity compiler is actively maintained and continuously improved, the Move-to-EVM compiler remains in an experimental state. We thus conjecture that optimizing the current compiler could potentially reduce the compiler-induced gas overhead.

Conclusion

To sum it up, we have shown that the existing compiler for Move-to-EVM fails to preserve core properties of Move in an untrusted environment as it is given by Ethereum. We identified core challenges that arise when running Move code in such an environment and have demonstrated how IRMs can be used to tackle them at runtime. Specifically, we have shown how core safety properties of Move can be preserved even in an untrusted environment, enabling developers to switch to a language that is more natural to design digital assets. With our monitors, we effectively allow Move code to run on the EVM without any change in the consensus layer of Ethereum, making our approach feasible in practice. Furthermore, we stress that our definition of IRMs for EVM introduced in §2.4 is agnostic to the monitored properties, and thus, similar techniques can be used for protecting protocols involving multiple (potentially ill-behaved) actors on-chain, in general.

By formally analyzing our compiler, we were able to show the correctness of compilation in the sense that the behavior of contracts produced by our compiler can be mapped back to a *similar* behavior in Move.

Finally, we have seen that the gas overhead introduced by our approach is moderate, considering that our protection mechanisms are capable of enforcing callee-side behavior of arbitrary contracts at runtime. In this regard, we particularly want to note that the protection layer introduced is completely agnostic to the compiled Move module it protects. Hence, future work might investigate approaches where multiple contracts share a protection layer implemented by one dedicated contract (a trusted *Move protection provider*, so to speak). By additionally bundling transactions together, this approach also

seems promising for reducing the per-transaction overhead of our protection mechanism.

Ethical Considerations

The research conducted for this paper did neither involve human subjects nor live systems outside the possession of the authors' respective research institutions. Consequently, no human subjects were deceived in particular or harmed in general, and no terms of any services were breached.

The presented approach is purely defensive in nature and does thus not present a negative potential by itself. We, however, discuss that *not* using our approach (i.e., using the existing compiler) might expose users to vulnerabilities (without presenting concrete vulnerabilities found "in the wild" or trying to find out how prevalent these vulnerabilities are). From a technical perspective, we deem the question of prevalence, which by itself would be a considerable task, out of scope. From an ethical perspective, scanning the blockchain for vulnerable contracts would not help mitigate the potential negative impact of this paper, as, due to the anonymous nature of Ethereum, we would not be able to disclose any vulnerabilities found to the relevant stakeholders. Additionally, we want to stress that the vulnerable compiler is still marked as "experimental" and its application in security-critical domains is likely limited. We did, nevertheless, relay our findings to the current maintainers of the compiler (Aptos) who, in turn, informed us that they are not aware of any production deployments of the compiler.

The authors of this paper did, in all conscience, not break the laws of any relevant jurisdiction while conducting the research presented here.

Open Science

The implementation of the IRM-based compiler is publicly available on GitHub at [6]. The repository is a fork of the original Move-to-EVM compiler, and it contains the full source code, along with examples and instructions for building and using the compiler.

A Formal Background

Formally, the compiler from Move modules to EVM bytecode can be described as a function

$$[\![\cdot]\!]^{\texttt{move}}_{\texttt{evm}} : \mathfrak{Q} \mapsto \mathcal{L}\left[\mathbb{B}^8\right]$$

where $\mathfrak{Q}$ denotes the set of Move modules. In Move we describe the state of the virtual machine as depicted in Figure 8. The semantics of the Move bytecode language is then defined over a small-step semantics that can formally defined as a relation between move states:

$$(\circ \rightarrow_{mv} \circ) \subseteq States_{mv} \times States_{mv}$$

$$\begin{array}{rlll}
\text{Move VM state} & State_{mv} & := & (M, G, st, C, A) \\
\text{Memory} & M & : & Loc \mapsto \mathcal{R} \\
\text{Globals} & G & : & \mathcal{A} \times \mathcal{T} \mapsto Loc \\
\text{Operand stack} & st & \in & \mathcal{L}(\mathcal{V}) \\
\text{Call Stacks} & C & := & C_{plain} | ERR(c) :: C_{plain} \\
 & & & | \, HALT :: C_{plain} \\
\text{Plain Call Stacks} & C_{plain} & := & (\Omega, \phi, L, pc) :: C_{plain} \mid \varepsilon \\
\text{Account Map} & A & := & \mathcal{A} \mapsto 2^{\mathfrak{Q}}
\end{array}$$

Figure 8: State of the Move virtual machine

$$\begin{array}{rlll}
\text{Call Stacks} & S & := & S_{plain} \mid EXC :: S_{plain} \\
 & & & \mid HALT(\sigma, d) :: S_{plain} \\
\text{Plain call stacks} & S_{plain} & := & (\mu, \sigma, \iota) :: S_{plain} \mid \varepsilon \\
\text{Machine states} & \mu & := & (pc, m, st, d) \\
\text{Execution environment} & \iota & := & (actor, in, sender, code) \\
\text{Accounts} & acc & := & (stor, code) \in \mathbb{A} \\
\text{Global states} & \sigma & : & \mathbb{N}_{160} \mapsto \mathbb{A}
\end{array}$$

$$\begin{array}{lll}
pc \in \mathbb{N}_{256} & d, code, in \in \mathcal{L}\left[\mathbb{B}^8\right] & st \in \mathcal{L}\left(\mathbb{B}^{256}\right) \\
stor : \mathbb{B}^{256} \mapsto \mathbb{B}^{256} & actor, sender \in \mathbb{N}_{160} & m : \mathbb{B}^{256} \mapsto \mathbb{B}^8
\end{array}$$

Figure 9: State of the EVM

We thus write $State_{mv} \rightarrow_{mv} State'_{mv}$ if the semantics of Move allow a *small-step-transition* from state $State_{mv}$ to $State'_{mv}$. The reflexive and transitive closure is then denoted by $\circ \overset{*}{\rightarrow}_{mv} \circ$. A full definition of the small-step semantics can be found in section D.

Likewise, we define the state of the EVM as depicted in Figure 9. Similar as for Move, the semantics of the EVM bytecode language can be described as a relation

$$(\circ \rightarrow_{evm} \circ) \subseteq State_{evm} \times States_{evm}$$

The reflexive and transitive closure will then be denoted by $\overset{*}{\rightarrow}_{evm}$. We have omitted details of state description of the EVM irrelevant for us. A complete definition of the state of the EVM as well as a detailed definition of the EVM bytecode's small-step semantic can be found in [31].

A.1 Trace Semantics of Move Modules

Given a Move module Ω, we define its semantics as the set of traces that are *admitted* by Ω. However, defining such traces in a straight forward way over the small-step semantics, comes with two problems: first, this definition of a trace might be to fine grained making it cumbersome to work with. Second, such traces would also include execution traces of other modules making it complicated to compare traces in Move to traces in EVM.

$$\frac{\phi[pc] = \texttt{Call}\,\langle \Omega', \phi' \rangle \qquad \Omega^\dagger \neq \Omega' \qquad State_{mv} := \left(M, G, st, \left(\Omega^\dagger, \phi, L, pc\right) :: C, A\right) \rightarrow_{mv} State'_{mv}}{\Omega^\dagger \rhd State_{mv} \xrightarrow{\texttt{call}\, State_{mv}!}_{mv} State'_{mv}}$$

$$\frac{\phi'[pc'] = \texttt{Call}\,\left\langle \Omega^\dagger, \phi \right\rangle \qquad \Omega^\dagger \neq \Omega' \qquad State_{mv} := \left(M, G, st, \left(\Omega', \phi', L', pc'\right) :: C, A\right) \rightarrow_{mv} State'_{mv}}{\Omega^\dagger \rhd State_{mv} \xrightarrow{\texttt{call}\, State'_{mv}?}_{mv} State'_{mv}}$$

$$\frac{\phi[pc] = \texttt{Ret} \qquad \Omega^\dagger \neq \Omega' \qquad C = \left(\Omega', \phi', L', pc'\right) :: C' \qquad State_{mv} := \left(M, G, st, \left(\Omega^\dagger, \phi, L, pc\right) :: C, A\right) \rightarrow_{mv} State'_{mv}}{\Omega^\dagger \rhd State_{mv} \xrightarrow{\texttt{ret}\, !State_{mv}}_{mv} State'_{mv}}$$

$$\frac{\phi'[pc'] = \texttt{Ret} \qquad \Omega^\dagger \neq \Omega' \qquad C = \left(\Omega^\dagger, \phi, L, pc\right) :: C' \qquad State_{mv} := \left(M, G, st, \left(\Omega', \phi', L', pc'\right) :: C, A\right) \rightarrow_{mv} State'_{mv}}{\Omega^\dagger \rhd State_{mv} \xrightarrow{\texttt{ret}\, ?State'_{mv}}_{mv} State'_{mv}}$$

Figure 10: Action annotation for relevant opcodes.

We hence introduce a trace semantics, that only considers *boundary* points of the execution. That is, whenever we enter or leave the module under consideration. This semantics is thus defined from the point of view of a specific Move module. We therefore, fix an arbitrary Move module as the *module under consideration* and denote it with $\Omega^\dagger$.

We define a trace $t \in T$ as a sequence of action α of the form

$$\begin{array}{rcl} \alpha & := & \texttt{call}\, State_{mv}? \mid \texttt{call}\, State_{mv}! \mid \texttt{ret}\, State_{mv}? \\ & & \mid \texttt{ret}\, State_{mv}! \mid \texttt{term}\, State_{mv} \end{array}$$

With $\texttt{call}\, State_{mv}?$ we denote the action of calling our considered module $\Omega^\dagger$ were $State_{mv}$ denotes the state of the Move virtual machine upon calling $\Omega^\dagger$. Calling from $\Omega^\dagger$ outside to another module is denoted by $\texttt{call}\, State_{mv}!$ where $State_{mv}$ denotes the state upon calling the external module. Returning from $\Omega^\dagger$ to another module Ω and returning from another module Ω to our module $\Omega^\dagger$ is then denoted by $\texttt{ret}\, State_{mv}!$ and $\texttt{ret}\, State_{mv}?$ respectively where again $State_{mv}$ refers to the state upon returning. Finally, the action $\texttt{term}\, State_{mv}$ denotes a termination of the execution upon state $State_{mv}$.

Let $\rightarrow_{mv} \subset States_{mv} \times States_{mv}$ denote the small-step semantic of Move. We then introduce a small-step semantics with judgments of the form $\Omega^\dagger \rhd State_{mv} \xrightarrow{\ell}_{mv} State'_{mv}$ with the intended meaning that from point of view of $\Omega^\dagger$, the state $State_{mv}$ evolves to $State'_{mv}$ by emitting a label ℓ which is either empty (denoted by ε) or an action α as introduced before. For most of the opcodes of the Move language, these semantics simply behaves as the original small-step semantic by simply emitting the empty label ε. However, for certain opcodes we do emit an action. For these opcodes the rules are defined in Figure 10.

$$\frac{\Omega^\dagger \rhd State_{mv} \xrightarrow{\alpha}_{mv} State'_{mv} \qquad State'_{mv} \neq HALT}{\Omega^\dagger \rhd State_{mv} \xrightarrow{\alpha}\!\!\!\!\rightarrow_{mv} State'_{mv}}$$

$$\frac{\Omega^\dagger \rhd State_{mv} \xrightarrow{x}_{mv} State'_{mv} \qquad x = \alpha \vee x = \varepsilon \qquad State'_{mv} = HALT}{\Omega^\dagger \rhd State_{mv} \xrightarrow{x::\texttt{term}\, State'_{mv}}\!\!\!\!\rightarrow_{mv} State'_{mv}}$$

$$\frac{\Omega^\dagger \rhd State_{mv} \xrightarrow{\varepsilon}_{mv} State'_{mv} \qquad \Omega^\dagger \rhd State'_{mv} \xrightarrow{\alpha}\!\!\!\!\rightarrow_{mv} State''_{mv}}{\Omega^\dagger \rhd State_{mv} \xrightarrow{\alpha}\!\!\!\!\rightarrow_{mv} State''_{mv}}$$

Figure 11: Large-Step semantics for Move.

$$\frac{\Omega^\dagger \rhd State_{mv} \xRightarrow{\overline{\alpha}}_{mv} State'_{mv} \qquad \Omega^\dagger \rhd State'_{mv} \xrightarrow{\alpha?}\!\!\!\!\rightarrow_{mv} State''_{mv} \qquad \Omega^\dagger \rhd State''_{mv} \xrightarrow{\alpha!}\!\!\!\!\rightarrow_{mv} State'''_{mv}}{\Omega^\dagger \rhd State_{mv} \xRightarrow{\overline{\alpha}::\alpha?::\alpha!}_{mv} State'''_{mv}}$$

$$\frac{\Omega^\dagger \rhd State_{mv} \xRightarrow{\overline{\alpha}}_{mv} State'_{mv} \qquad \Omega^\dagger \rhd State'_{mv} \xrightarrow{\alpha?}\!\!\!\!\rightarrow_{mv} State''_{mv} \qquad \Omega^\dagger \rhd State''_{mv} \xrightarrow{\texttt{term}\, mvState'''}\!\!\!\!\rightarrow_{mv} State'''_{mv}}{\Omega^\dagger \rhd State_{mv} \xRightarrow{\overline{\alpha}::\alpha?::\texttt{term}\, State'''_{mv}}_{mv} State'''_{mv}}$$

Figure 12: Trace semantics for Move.

Figure 10.

Do get rid of the no-action transition, we introduce a *large-step* semantics based on the annotated small-step semantic. Furthermore, the large step semantic will end a termination action for each terminating trace. See Figure 11 for details.

Intuitively, the above rule allows us to skip no-action transitions. Based on that we can then proceed defining the trace semantic as depicted in Figure 12. Given a trace $\overline{\alpha}$ such that for two states $State_{mv}, State'_{mv} \in States_{mv}$ we have that

$$\Omega^\dagger \rhd State_{mv} \xRightarrow{\overline{\alpha}}_{mv} State'_{mv}$$

we say that $\Omega^\dagger$ *admits* $\overline{\alpha}$ and write $\Omega^\dagger \vdash \overline{\alpha}$. We say that $\overline{\alpha}$ is a *complete trace* if it cannot be extended and terminates, *i.e.* if there does not exist a pair of action $\alpha? :: \alpha!$ such that $\Omega^\dagger \vdash \alpha? :: \alpha! :: \overline{\alpha}$ and $\overline{\alpha}$ ends with a $\texttt{term}\, State_{mv}$ for some state $State_{mv}$.

A.2 Trace Semantics for EVM

Similar as for Move, we introduce a trace semantic for EVM where a trace is again a sequence of actions where an action α is defined as

$$\begin{array}{rcl} \alpha & := & \texttt{call}\, State_{evm}? \mid \texttt{call}\, State_{evm}! \mid \texttt{ret}\, State_{evm}? \\ & & \mid \texttt{ret}\, State_{evm}! \mid \texttt{term}\, State_{evm} \\ & & \mid \texttt{call}^*\, State_{evm}? \mid \texttt{call}^*\, State_{evm}! \mid \texttt{ret}^*\, State_{evm}? \\ & & \mid \texttt{ret}^*\, State_{evm}! \end{array}$$

$$\frac{\omega_{\mu,\iota} = \texttt{Call} \qquad \iota'.actor = a^\dagger \qquad State_{evm} \rightarrow_{evm} State'_{evm} := (\mu', \sigma', \iota') :: S}{a^\dagger \rhd State_{evm} \xrightarrow{\texttt{call}^*\ State'_{evm}?}_{evm} State'_{evm}}$$

$$\frac{\omega_{\mu,\iota} = \texttt{Call} \qquad \iota.actor = a^\dagger \qquad State_{evm} := (\mu, \sigma, \iota) :: S \rightarrow_{evm} State'_{evm}}{a^\dagger \rhd State_{evm} \xrightarrow{\texttt{call}^*\ State_{evm}!}_{evm} State'_{evm}}$$

$$\frac{\omega_{\mu,\iota} = \texttt{Return} \qquad \iota'.actor = a^\dagger \qquad State_{evm} \rightarrow_{evm} State'_{evm} := (\mu', \sigma', \iota') :: S}{a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}^*\ State'_{evm}?}_{evm} State'_{evm}}$$

$$\frac{\omega_{\mu,\iota} = \texttt{Return} \qquad \iota.actor = a^\dagger \qquad State_{evm} := (\mu, \sigma, \iota) :: S \rightarrow_{evm} State'_{evm}}{a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}^*\ State_{evm}!}_{evm} State'_{evm}}$$

Figure 13: Annotated small-step semantics for EVM

$$\frac{\omega_{\mu,\iota} = \texttt{SIG\_FUNC\_CALL} \qquad \iota.actor = a^\dagger \qquad State_{evm} := (\mu, \sigma, \iota) :: S \rightarrow_{evm} State'_{evm}}{a^\dagger \rhd State_{evm} \xrightarrow{\texttt{call}\ State'_{evm}?}_{evm} State'_{evm}}$$

$$\frac{\omega_{\mu,\iota} = \texttt{SIG\_FUNC\_RETURN} \qquad \iota.actor = a^\dagger \qquad State_{evm} := (\mu, \sigma, \iota) :: S \rightarrow_{evm} State'_{evm}}{a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}\ State_{evm}!}_{evm} State'_{evm}}$$

Figure 14: Annotated small-step semantics for EVM

with a similar meaning as before. However, note every action (besides the `term` $State_{mv}$) has a *sibling* marked with an $*$. For EVM we will differ between the call to (respectively return from) a contract and a call to (respectively return from) a code segment that represents the corresponding function in Move.

Let $\rightarrow_{evm} \subseteq States_{evm} \times States_{evm}$ be the small-step semantic of our extended EVM bytecode. We then introduce again an annotated small-step semantic with judgments of the form $a^\dagger \rhd State_{evm} \xrightarrow{\ell}_{evm} State'_{evm}$ where $a^\dagger \in \mathbb{N}_{160}$ is an arbitrary but fixed account address and ℓ, again, is either an empty label or an action α. The intended meaning is that from the point of view of the account with address $a^\dagger$, the state $State_{evm}$ evolves to $State'_{evm}$ by emitting the label ℓ.

For our annotated small-step semantic we introduce two new macros `SIG_FUNC_CALL` and `SIG_FUNC_RETURN` on the EVM side that mark the begin and end of an invocation of a function respectively. Semantically, these macros are NOPs, *i.e.* they do not change the state of the EVM (besides increasing the program counter).

Most of the rules of the annotated small-step semantics are updated in an intuitive way: local operations will emit no annotation while operations that lead to a call or a return as well as the macros `SIG_FUNC_CALL` and `SIG_FUNC_RETURN` will emit a corresponding action. Note that we will not introduce rules that emit actions of the form `call` $State_{evm}!$ or `ret` $State_{evm}?$ as we do assume that our modules do not call outside.

The relevant rules of the annotated small-step are shown in Figure 13 and Figure 14. Note however, that for the actions marked with $*$ other opcodes exist that emit these actions (*e.g.* a `Delegatecall` or a return by the last opcode). However, these rules are defined in a similar fashion.

The annotated small-step semantic of EVM is then lifted to a large-step semantic and based on that to trace semantic as before. However, while we keep them in the large-step semantics, actions of the form `call`* $State_{evm}?$ and `ret`* $State_{evm}!$ are removed from the trace semantics. This is important to ensure that traces in Move and EVM are comparable.

We then end up with a trace semantic with judgments of the form $a^\dagger \rhd State_{evm} \overset{\overline{\alpha}}{\Longrightarrow}_{evm} State'_{evm}$. In a similar fashion than before, we want to write judgments of the form $[\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}} \vdash \overline{\alpha}$ for some trace $\overline{\alpha}$, expressing that the compiled module *admits* a trace. However, for this to make sense, we need some further notation:

Consider the set of states of the EVM $States_{evm}$. We then write $States_{evm}|_{a^\dagger \mapsto [\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}}}$ to denote the set of all valid EVM states that agree in their global states σ to have $\sigma\left(a^\dagger\right).code = [\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}}$ for some address $a^\dagger \in \mathbb{N}_{160}$.

We then write $[\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}} \vdash \overline{\alpha}$ if for an arbitrary $a^\dagger \in \mathbb{N}_{160}$ there exists two states

$$State_{evm}, State'_{evm} \in States_{evm}|_{a^\dagger \mapsto [\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}}}$$

such that $a^\dagger \rhd State_{evm} \overset{\overline{\alpha}}{\Longrightarrow}_{evm} State'_{evm}$. Again, we say that $\overline{\alpha}$ is a *complete trace* if it cannot be extended and it terminates (as defined before).

A.3 Storage and Memory Model in EVM

In EVM storage and memory are linear storages that are word-addressed or byte-addressed respectively, *i.e.* each cell is either a word (*i.e.* 256 bits in EVM) or a byte. However, in Move storage (like global memory or locals) can hold more complex data. Consequently, when compiling from Move to EVM a serialization strategy is needed. While such serialization is cumbersome to define, it does not give any interesting insights in the compilation itself. We thus introduce a model for storage and memory that allows us to reason about stored data without the need of defining a concrete serialization. Later on, we allow the compiler to use macros to access the

storage and memory whose correctness is defined over the model established in this section.

Storage as Separation Logical Model First, we consider EVM's storage. We formally denote (a partial) storage as a partial function $s\colon \mathbb{N}_{256} \hookrightarrow \mathbb{N}_{256}$. With $dom(s) \subseteq \mathbb{N}_{256}$ we denote the domain of s. Given two storages s,t, we write $s \perp t$ if $dom(s) \cap dom(t) = \emptyset$.

Furthermore, if $s \perp t$ we write $s \cdot t$ to denote the union of the two storages defined in the natural way. While reasoning about partial storages is useful to separate different sections of the storage from each other, the storage of the EVM is always a total function of the form $stor\colon \mathbb{N}_{256} \mapsto \mathbb{N}_{256}$. We therefore introduce the notation $cl(s)$ for a partial storage $stor$ to denote the closure s obtained by setting each undefined location to the value zero. Sometimes we want to *cut out* a specific part of the storage. Given a storage s, we denote with $s|_{intv}$ the part of s including all locations inside the interval $intv$. For example, with $s|_{[1,n]}$ we may denote the part of s that contains all locations between 1 and n.

We will define our logic over propositions p of the form $i \mapsto n$ for $i,n \in \mathbb{N}_{256}$. Given a storage s we write $s \vdash i \mapsto n$ if $dom(s) = \{i\}$ and $s(i) = n$. Furthermore, we assume typical connectives of standard logic as $\wedge$ and $\vee$ and assume their standard semantic. Given a storage s and two propositions p,q we write $s \vdash p * q$ if their exists some storages s_1 and s_2 such that $s_1 \vdash p$, $s_2 \vdash q$ and $s = s_1 \cdot s_2$. Furthermore, we write $s \vdash p \mathrel{-\!\!*} q$ if for every storage t such that $s \perp t$ and $t \vdash p$ it holds that $s \cdot t \vdash q$.

For every storage s it holds that $s \vdash \textbf{true}$ while $s \vdash \textbf{false}$ indicates that $dom(s) = \emptyset$.

Serialization in Move Consider an object v of type τ in Move. We then assume that there exists a formula $stores(v,\tau,i)$ of the form $x_1 \mapsto y_1 * \cdots * x_n \mapsto y_n$ such that for a (EVM) storage s we write $s \vdash stores(v,\tau,i)$ to denote that s stores the serialization of v at storage location i. Note that by the form of $stores(v,\tau,i)$ we can ensure thata (partial) storage s such that $s \vdash stores(v,\tau,i)$ only contains values defined by the serialization of v. If τ is a structure type v of the form $(v_1 : \tau_1, \ldots, v_n : \tau_n)$ we do assume that $stores(v,\tau,i)$ is defined such that for each storage s it holds that

$$s \vdash stores(\tau,v,i) \mathrel{-\!\!*} stores(v_1,\tau_1,i_1) * \cdots * stores(v_n,\tau_n,i_n)$$

for some storage locations i_j.

Memory in EVM In EVM, the memory is a mapping of the form $\mathbb{N}_{256} \mapsto \mathbb{B}^8$. Similar to storage we need a serialization strategy also for memory, as locals in Move (which correspond to memory in EVM), can also hold values of any type. Thus, we introduce the same notion for memory that we have already introduced for storage. In particular, we again assume a formula $stores(v,\tau,i)$ with a similar meaning than before. Note that we informally use the symbol $stores(\cdot,\cdot,\cdot)$ for both, memory and storage, although technically, the definitions might differ. Anyhow, it will always be clear from the context to which version we refer.

A.4 Relating EVM to Move States

Given a Move state $State_{mv}$ and an EVM state $State_{evm}$ we want to introduce a relation $\sim_{\Omega^\dagger,a^\dagger} \subseteq States_{mv} \times States_{evm}$ such that if $State_{mv} \sim_{\Omega^\dagger,a^\dagger} State_{evm}$, we say that $State_{mv}$ and $State_{evm}$ are *similar*. Note that the relation is parameterized by a Move module $\Omega^\dagger$ and an EVM address $a^\dagger$.

We will define the relation $\sim_{\Omega^\dagger,a^\dagger}$ in an unusual (but more convenient way): We assume for a Move state $State_{mv}$ and an EVM state $State_{evm}$ that $State_{mv} \sim_{\Omega^\dagger,a^\dagger} State_{evm}$ and show the different implication of the relation. At the end, we define $\sim_{\Omega^\dagger,a^\dagger}$ to be the *largest* relation such that all the implication are fulfilled.

First, we want to express that a Move state $State_{mv}$ is in the same point of execution as an EVM state $State_{evm}$. More formally, if $State_{mv}$ is describing a point of execution of the module $\Omega^\dagger$, $State_{evm}$ is only similar to $State_{mv}$ if it describes a similar point of execution of the compiled module $[\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}}$. For that, we assume that the compiler generates a source-map that relates any function ϕ and any program counter pc of the module $\Omega^\dagger$ to a program counter pc of the contract $[\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}}$ such that if ϕ at pc pc depicts the opcode c, the contract $[\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}}$ contains the script $[\![c]\!]^{\texttt{move}}_{\texttt{evm}}$ starting at pc pc.

Definition 1 (Point of Execution)**.**

$$\begin{aligned}\left(M,G,st,\left(\Omega^\dagger,\phi,L,pc\right)::C,A\right) \sim_{\Omega^\dagger,a^\dagger} (\mu,\sigma,\iota)::S \\ \implies \quad & src\_map_{\Omega^\dagger}(\phi,pc) = \mu.pc \\ & \wedge\, \sigma(\iota.actor).code = [\![\Omega^\dagger]\!]^{move}_{evm} \\ & \wedge\, \iota.actor = a^\dagger\end{aligned}$$

Now let us consider the operand stack. For that, consider a state $State_{mv}$ and a state $State_{evm}$ such that both describing a running execution. Intuitively, we want that the elements on the operand stack in Move to match those in EVM. However, in Move, all executions on the callstack share the same operand stack. On the other hand, the compiler may want to put some book-keeping-data on the bottom of the operand stack on the EVM side.

Therefore, in Move we introduce the function $loc\_stack\_of\colon \mathbb{N} \mapsto \mathbb{\Omega}$ that given a stack position, returns the corresponding module in whose execution this part of the stack lies. Since Move's stack is well typed, we can assume that such function exist. That is, given a specific state $State_{mv}$ of the Move virtual machine, for each position on the operand stack, it is always defined to which execution it belongs and therefore to which module.

Furthermore, due to the well-typed nature of the Move stack, we can also safely assume to have access to the type

information of any element on the stack (by using the method `type_of`). For primitive types $\tau \in \mathcal{T}_{prim}$ (*i.e.* integers, addresses, *etc.*) we assume the compiler to introduce a relation $\sim \subseteq (\mathcal{T}_{prim} \times \mathcal{V}) \times \mathbb{N}_{256}$ that relates a tuple consisting of a value v and its type τ to a value $n \in \mathbb{N}_{256}$. For most primitive types as integers this relation is defined intuitively. However, for addresses we assume an arbitrary but fixed relation. In case the type of the Move value is clear from the context, we, for simplicity, also might write $v \sim n$ instead of $(v,\tau) \sim n$.

For resource types, we require that in EVM only a handle is stored on stack that points to a memory region. Finally, if we encounter a reference on the Move side, this might relate to a reference to a memory object or a storage object in EVM. We require the compiler to enforce an encoding that let us differ between memory references and storage references at runtime on the EVM side (as we cannot get this information statically from the Move side). For this purpose, the compiler must introduce the function $\mathsf{mem\_ref} : \mathbb{N}_{256} \mapsto \{\mathbf{true}, \mathbf{false}\}$, such that $\mathsf{mem\_ref}(x)$ returns **true** if x is a memory reference and **false** if it is storage reference. Furthermore, the compiler must introduce a function $\mathsf{deref} : \mathbb{N}_{256} \mapsto \mathbb{N}_{256}$ that given a reference, returns the corresponding memory or storage address.

Definition 2 (Operand Stack)**.**

$$
\begin{array}{ll}
 & \left(M, G, st, \left(\Omega^\dagger, \phi, L, pc\right) :: C, A\right) \sim_{\Omega^\dagger, a^\dagger} (\mu, \sigma, \iota) :: S \\
\Longrightarrow & |st|, |\mu.st| \geq n \\
 & \wedge\ st := [v_1, \ldots, v_n, \ldots] \wedge \mu.st := [w_1, \ldots, w_n, \ldots] \\
 & \wedge\ |st| = n \vee loc\_stack\_of(n+1) \neq \Omega^\dagger \\
 & \wedge\ \tau_1 := type\_of\,(v_1) \wedge \cdots \wedge \tau_n := type\_of\,(v_n) \\
 & \wedge\ \forall 1 \leq i \leq n. \\
 & \quad \tau_i \in \mathcal{T}_{prim} \implies (\tau_i, v_i) \sim w_i \\
 & \quad \tau_i \in \mathcal{T}_\Omega \implies \mu.m \vdash stores\,(v_i, \tau_i, w_i) \\
 & \quad \&\tau' := \tau_i \in \mathcal{T}_{ref} \wedge mem\_ref(w_i) \\
 & \qquad \implies \mu.m \vdash stores\,(v_i, \tau', deref(w_i)) * \mathbf{true} \\
 & \quad \&\tau' := \tau_i \in \mathcal{T}_{ref} \wedge \neg mem\_ref(w_i) \\
 & \qquad \implies \sigma\,(actor)\,.stor \vdash stores\,(v_i, \tau', deref(w_i)) * \mathbf{true}
\end{array}
$$

Next we consider the globals and the global memory in Move. First, let us consider resources, that are *directly accessible* by the Move module $\Omega^\dagger$. That is, let $\mathcal{T}_{\Omega^\dagger}$ be the set of types defined by the module $\Omega^\dagger$, and let G, M be the globals and memory of a Move state respectively, then we first consider all resources r of type $\tau \in \mathcal{T}_{\Omega^\dagger}$ such that there is an address a with $G(a,\tau) = \ell$ and $M(\ell) = r$.

In EVM we expect a special storage region for such resources. More particularly, we expect the compiler to define a function $\mathsf{int\_stor}_\tau : \mathbb{N}_{256} \mapsto \mathbb{N}_{256}$ for every type $\tau \in \mathcal{T}_{\Omega^\dagger}$ that given some value $n \in \mathbb{N}_{256}$ computes a storage address:

Definition 3 (Storage of internals)**.**

$$
\begin{array}{ll}
 & \left(M, G, st, \left(\Omega^\dagger, \phi, L, pc\right) :: C, A\right) \sim_{\Omega^\dagger, a^\dagger} (\mu, \sigma, \iota) :: S \\
\Longrightarrow & \textcircled{1}\ \forall a_{evm} \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}. \\
 & \quad \sigma\,(actor)\,.stor \vdash stores\,(r, \tau, int\_stor_\tau\,(a_{evm})) * \mathbf{true} \\
 & \quad \implies \exists a_{mv} \in \mathcal{A}. \\
 & \qquad a_{mv} \sim a_{evm} \\
 & \qquad G(a_{mv}, \tau) = \ell \wedge M(\ell) = r
\end{array}
$$

Besides the just described resources in the globals, resources of a type defined by the module can also be wrapped in other modules types and stored in global memory. We also need to keep track of the on the EVM side. Consequently, similar than before, we assume a special storage region for resources represented by a compiler provided function $\mathsf{ext\_stor}_\tau : \mathbb{N}_{256} \mapsto \mathbb{N}_{256}$ with a similar meaning than before.

Also, we assume a function *wraps* that given a resource returns a set of all sub-resources wrapped by the passed resource. We will not define this function formally as it's behavior is rather intuitive. For clarity we however not that it always holds that $r \in wraps\,(r)$.

Definition 4 (Storage of externals)**.**

$$
\begin{array}{ll}
 & \left(M, G, st, \left(\Omega^\dagger, \phi, L, pc\right) :: C, A\right) \sim_{\Omega^\dagger, a^\dagger} (\mu, \sigma, \iota) :: S \\
\Longrightarrow & \textcircled{2}\ \forall a_{evm} \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}. \\
 & \quad n \in \mathbb{N}_{256}. \\
 & \qquad \sigma\,(actor)\,.stor \vdash stores\,(r, \tau, ext\_stor_\tau\,(a_{evm}, n)) * \mathbf{true} \\
 & \qquad \implies \exists a_{mv}, a'_{mv} \in \mathcal{A}, \\
 & \qquad\quad \tau' \in \mathcal{T} \\
 & \qquad\qquad \Omega \in A\,(a_{mv}) \wedge \tau' \in \mathcal{T}_\Omega \\
 & \qquad\qquad \wedge\ a_{mv} \sim a_{evm} \\
 & \qquad\qquad \wedge\ G(a'_{mv}, \tau') = \ell \wedge r \in wraps\,(M(\ell))
\end{array}
$$

Resources, besides being safely stored in global memory can however also be located in the execution environment of some other module invocation, *i.e.*, in the locals or on a portion of the operand stack not accessible to our model's invocation. Note that since in Move the stack is well typed, given a Move state (in particular the call stack) we can decide which portion of the operand stack belongs to which module. We express this by the function *loc_stack_of* that given a stack position returns the module that has direct access to stack portion.

We then expect our compiler to keep track of resources stored in some execution environment in order to avoid them from being dropped. Consequently, the compiler needs to provide a special storage area for these *transient* resources. In a similar fashion than before we therefore assume a compiler provided function tr_stor:

Definition 5 (Transient resources)**.**

$$\begin{array}{l}\left(M,G,st,\left(\Omega^{\dagger},\phi,L,pc\right)::C,A\right)\sim_{\Omega^{\dagger},a^{\dagger}}(\mu,\sigma,\iota)::S\\ \Longrightarrow\ \textcircled{3}\,\forall a_{evm}\in\mathbb{N}_{256},\tau\in\mathcal{T}_{\Omega^{\dagger}},\\ \quad n\in\mathbb{N}_{256}.\\ \quad\sigma(actor).stor\vdash stores(r,\tau,tr\_stor_{\tau}(a_{evm},n))*\textbf{true}\\ \quad\Longrightarrow\ (\ \exists a_{mv}\in\mathcal{A}\\ \quad\quad 0\le i\le|st|\\ \quad\quad loc\_stack\_of(i)=\Omega\neq\Omega^{\dagger}\\ \quad\quad \Omega\in A(a)\\ \quad\quad\wedge\ \texttt{address},a_{mv}\sim a_{evm}\\ \quad\quad\wedge\ r\in wraps(st[i])\ )\\ \quad\vee(\ \exists\ \ (\Omega,\phi',L',pc')\in C,\\ \quad\quad l\in dom(L).\\ \quad\quad r\in wraps(L(l))\end{array}$$

At the end of a transaction (*i.e.* on termination) we only care about persistent storage, *i.e.* the set of externals, internals and transient must match according to Definition 3-5. More formally, we thus define:

Definition 6 (Halting State)**.**

$$\begin{array}{l}(M,G,st,HALT::C,A)\sim_{\Omega^{\dagger},a^{\dagger}}HALT(\sigma,d)::S\\ \Longrightarrow\quad|C|=|S|=0\wedge\textcircled{1}\wedge\textcircled{2}\wedge\textcircled{3}\end{array}$$

Last, we consider Move's locals. We expect the compiler to store locals in the memory. The memory location for each local l is expressed by the compiler by defining the function local_stor similar than before for the storage. Note that the number of locals defined by each function ϕ in the module $\Omega^{\dagger}$ is statically known. However, the corresponding memory location in EVM cannot be assigned to a constant value. This is because internal function calls in Move have to be simulated in EVM by jumps. Consequently, different function invocation share the same memory. To avoid two functions overriding each others locals, the memory address of the locals might depend on some dynamically computed offset. Thus, we allow the function local_stor to depend on the current content of the memory as well:

Definition 7 (Locals)**.**

$$\begin{array}{l}\left(M,G,st,\left(\Omega^{\dagger},\phi,L,pc\right)::C,A\right)\sim_{\Omega^{\dagger},a^{\dagger}}(\mu,\sigma,\iota)::S\\ \Longrightarrow\quad\forall l\in dom(L).\\ \quad L(l)=v\wedge\texttt{type\_of}(v)=\tau\\ \quad\iff\mu.m\vdash stores(v,\tau,local\_stor(\mu.m,l))*\textbf{true}\end{array}$$

Definition 8 (Relating Move to EVM)**.** *For any Move module $\Omega^{\dagger}$ and any EVM address $a^{\dagger}$ the relation*

$$\circ\sim_{\Omega^{\dagger},a^{\dagger}}\circ\subseteq States_{mv}\times States_{evm}$$

is the largest relation (in a set-theoretic sense) such that Definition 1-7 hold.

We then lift the relation on states to relations on traces as follows:

Definition 9 (Lifting to traces)**.** *Given a Move trace t_{mv} and an EVM trace t_{evm}. Then for a Move module $\Omega^{\dagger}$ we write that $t_{mv}\sim_{\Omega^{\dagger}}t_{evm}$ if*

1. *it holds that $|t_{mv}|=|t_{evm}|$*
2. *and there exists an address $a^{\dagger}$ such that for all $0\le i<|t_{mv}|$ it holds that*

$$t_{mv}[i]=(a\ State_{mv}\ d)\iff t_{evm}[i]=(a\ State_{evm}\ d)$$

where $a\in\{call,return,term\}$ and $d\in\{?,!,\varepsilon\}$ and

$$State_{mv}\sim_{\Omega^{\dagger},a^{\dagger}}State_{evm}$$

A.5 Soundness Property

Based on the trace semantic for Move and EVM shown before, we then introduce our soundness property as follows:

Definition 10 (Sound Compilation)**.** *Given a compiler function $[\![\cdot]\!]^{move}_{evm}:\mathbb{Q}\mapsto\mathcal{L}\left[\mathbb{B}^8\right]$ compiling from Move modules to EVM bytecode, then we call the compiler sound if for each Move module $\Omega^{\dagger}$*

$$\begin{array}{l}\forall t_{evm}\in T_{evm}.\,[\![\Omega]\!]^{move}_{evm}\vdash t_{evm}\\ \quad\Longrightarrow\exists t_{mv}\in T_{mv}.\\ \quad\quad t_{mv}\sim_{\Omega^{\dagger}}t_{evm}\ \wedge\ \Omega\vdash t_{mv}\end{array}$$

B Soundness Proof

Instead of proving soundness of a concrete compiler, in this section we identify properties of the compiler that, if fulfilled, results in a sound compiler according to Definition 10. We refer to these properties as *compiler obligations*.

B.1 Local Preservation

First, we introduce the compiler obligation of *local preservation*. Intuitively, we want to express that when for any Move state and any EVM state such that we have established a similarity relation according to Definition 8, execution of local operations (*i.e.* operations that do not manipulate the callstack) preserve this relation:

Definition 11 (Local Preservation)**.** *Consider a compiler function $[\![\cdot]\!]^{move}_{evm}:\mathbb{Q}\mapsto\mathcal{L}\left[\mathbb{B}^8\right]$. We say that $[\![\cdot]\!]^{move}_{evm}$ is **locally preserving** if for each Move module $\Omega^{\dagger}$ and EVM address $a^{\dagger}$ it holds that*

$$\begin{array}{lll}\forall & State_{evm},State'_{evm},State''_{evm} & \in States_{evm}\\ & State_{mv} & \in States_{mv}.\\ & State_{mv}\sim_{\Omega^{\dagger},a^{\dagger}}State_{evm}\\ & \wedge\ a^{\dagger}\rhd State_{evm}\xrightarrow{ret\ State'_{evm}\ !}{}_{evm}\ State''_{evm}\\ & \Longrightarrow\ \exists & State'_{mv},State''_{mv}\in States_{mv}\\ & & \Omega^{\dagger}\rhd State_{mv}\xrightarrow{ret\ State'_{mv}\ !}{}_{mv}\ State''_{mv}\\ & & \wedge\ State'_{mv}\sim_{\Omega^{\dagger},a^{\dagger}}State'_{evm}\end{array}$$

The reader might wonder why in the definition of the *Local Preservation* property of a compiler, the compiler itself does not occur in the definition. However, note that the compiler is included implicitly due to the definition of the similarity relation $\sim_{\Omega^\dagger,a^\dagger}$. In particular, note that Definition 1 enforces the compiled Move module to be the code associated with the address $a^\dagger$.

Lemma 1 (Local Preservation for Small-Step)**.** *A compiler function* $[\![\cdot]\!]^{move}_{evm} : \mathbb{Q} \mapsto \mathcal{L}\left[\mathbb{B}^8\right]$ *is* ***locally preserving*** *if there exists a function* $\mathsf{script\_len}_{\Omega^\dagger} : \mathbb{N}_{256} \mapsto \mathbb{N}$ *such that for each Move module* $\Omega^\dagger$ *and EVM address* $a^\dagger$ *it holds that*

$$
\begin{array}{l}
\forall State_{evm}, State'_{evm}, State''_{evm} \in States_{evm} \\
\quad State_{mv} \qquad\qquad\qquad\qquad \in State_{mv}. \\
\quad\quad State_{mv} \sim_{\Omega^\dagger,a^\dagger} State_{evm} \\
\quad\quad \wedge\, a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}\ State'_{evm}\ !}\!\!\!\!\!\rightarrow_{evm} State''_{evm} \\
\quad\quad \Longrightarrow\ \exists State'_{mv}, State''_{mv}, State^*_{mv} \in States_{mv} \\
\qquad\qquad State^*_{evm} \qquad\qquad\qquad\qquad \in States_{evm} \\
\qquad\qquad\quad \mathsf{script\_len}_{\Omega^\dagger}(\mu.pc) = n \in \mathbb{N} \\
\qquad\qquad\quad \wedge\, a^\dagger \rhd \overbrace{State_{evm} \xrightarrow{\varepsilon}_{evm} \cdots \xrightarrow{\ell}_{evm}}^{n\ times} State^*_{evm} \\
\qquad\qquad\quad \wedge (\ \ell = \varepsilon \\
\qquad\qquad\qquad \wedge\, \Omega^\dagger \rhd State_{mv} \xrightarrow{\varepsilon}_{mv} State^*_{mv} \\
\qquad\qquad\qquad \wedge\, State^*_{mv} \sim_{\Omega^\dagger,a^\dagger} State^*_{evm} \\
\qquad\qquad\qquad \wedge\, a^\dagger \rhd State^*_{evm} \xrightarrow{\texttt{ret}\ State'_{evm}\ !}\!\!\!\!\!\rightarrow_{evm} State''_{evm} \\
\qquad\qquad\quad \vee\, \ell = \texttt{ret}\ State'_{evm}\ ! \\
\qquad\qquad\qquad \wedge\, State^*_{evm} = State''_{evm} \\
\qquad\qquad\qquad \wedge\, \Omega^\dagger \rhd State_{mv} \xrightarrow{\texttt{ret}\ State'_{mv}\ !}\!\!\!\!\!\rightarrow_{evm} State''_{mv} \\
\qquad\qquad\qquad \wedge\, State'_{mv} \sim_{\Omega^\dagger,a^\dagger} State'_{evm}\ )
\end{array}
$$

Proof. Follows directly by unfolding $\cdot \xrightarrow{\texttt{ret}\ State'_{evm}\ !}\!\!\!\!\!\rightarrow_{evm} \cdot$ and $\cdot \xrightarrow{\texttt{ret}\ State'_{mv}\ !}\!\!\!\!\!\rightarrow_{mv} \cdot$ according to its definition □

Intuitively, Lemma 1 expresses the intuition, that our Move-to-EVM compiler translates *local* opcodes in Move to sequences of opcodes in EVM. We expect the compiler to produce a sequence of EVM opcodesfor each such Move opcode that represents a small step on the Move virtual machine, *i.e.* after some finite number of small-steps on the EVM side, we must end up in a state that can be related to a state in Move after a single step.

B.2 Global Preservation

Roughly speaking, local preservation ensures soundness while the contract itself is execute. However, the compiler also needs to ensure that certain properties are preserved between two invocations. Hence, in what follows we will define obligation that have to hold outside the contract's execution and between two executions.

We first restrict the behavior of the contract between the program counter that corresponds to a return of a function in Move and the actual return of the contract. In these part the compiler must ensure that the transient resources are updated by adding returned resources. This is captured by the following obligation:

Compiler Obligation 1 (Updating Transients)**.** *Given a compiler* $[\![\cdot]\!]^{move}_{evm} : \mathbb{Q} \mapsto \mathcal{L}\left[\mathbb{B}^8\right]$ *from Move modules to EVM bytecode. Then for every modules* $\Omega^\dagger$ *and address* $a^\dagger$ *it has to hold:*

$$
\begin{array}{l}
\forall State_{evm}, States'_{evm}, States^*_{evm} States^{**}_{evm} \in States_{evm}|_{a^\dagger \mapsto [\![\Omega^\dagger]\!]^{move}_{evm}} \\
\quad State_{mv} \qquad\qquad\qquad\qquad \in States_{mv} \\
\quad\quad a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}\ State_{evm}\ !}\!\!\!\!\!\rightarrow_{evm} State^*_{evm} \\
\quad\quad \wedge\, a^\dagger \rhd State^*_{evm} \xrightarrow{\texttt{ret}^*\ State'_{evm}\ !}\!\!\!\!\!\!\rightarrow_{evm} State^{**}_{evm} \\
\quad\quad \wedge\, State_{mv} \sim_{\Omega^\dagger,a^\dagger} State_{evm} \\
\quad\quad \wedge\, State_{evm} = (\mu, \sigma, \iota) :: S \\
\quad\quad \wedge\, State'_{evm} = (\mu', \sigma', \iota) :: S \\
\quad\quad \wedge\, State'_{mv} = (M, G, st, (\Omega, \phi, L, pc) :: C, A) \\
\quad\quad \Longrightarrow\ \forall n \in \mathbb{N}_{256}, x \in a, \tau \in \mathcal{T}_{\Omega^\dagger}, r. \\
\qquad\qquad \sigma'(stor) \vdash stores(r, \tau, \mathsf{int\_stor}_\tau(x)) * \mathbf{true} \\
\qquad\qquad\quad \Longleftrightarrow\ \sigma(stor) \vdash stores(r, \tau, \mathsf{int\_stor}_\tau(x)) * \mathbf{true} \\
\\
\qquad\qquad \wedge\, \sigma'(stor) \vdash stores(r, \tau, \mathsf{ext\_stor}_\tau(x, n)) * \mathbf{true} \\
\qquad\qquad\quad \Longleftrightarrow\ \sigma(stor) \vdash stores(r, \tau, \mathsf{ext\_stor}_\tau(x, n)) * \mathbf{true} \\
\\
\qquad\qquad \wedge\, \sigma'(stor) \vdash stores(r, \tau, \mathsf{tr\_stor}_\tau(x, n)) * \mathbf{true} \\
\qquad\qquad\quad \Longleftrightarrow \sigma(stor) \vdash stores(r, \tau, \mathsf{tr\_stor}_\tau(x, n)) * \mathbf{true} \\
\qquad\qquad\qquad \vee\, \exists 1 \le i \le \phi.num\_ret. \\
\qquad\qquad\qquad\quad st[|st| - i] = r \\
\qquad\qquad\qquad\quad \wedge\, \texttt{type\_of}(st[|st| - i]) = \tau \\
\qquad\qquad\qquad\quad \wedge\, x = \iota.sender
\end{array}
$$

Besides ensuring that resources are put correctly into the set of transients,Obligation 1 requires that the externals and internals remain unchanged compared to the state that corresponds to the return in Move.

Now between two invocations of our contract, we do allow changes in the storage. This seems counter intuitive on the EVM side, as storage of a contract can only change when the storage is executed. However, note that methods introduced by monitors as described do not yield an actual invocation in our trace semantic.

Storage changes between contract invocations are however not arbitrary. In what follows we will define allowed changes in storage that occur outside of the actual functions of the contract. For that purpose we introduce a special set of EVM state, $States_{evm,init}$, that denotes the set of all possible initial states at the beginning of a transaction.

Compiler Obligation 2 (Preservation of Transients and Externals)**.** *Given a compiler* $[\![\cdot]\!]^{move}_{evm} : \mathbb{Q} \mapsto \mathcal{L}\left[\mathbb{B}^8\right]$ *from Move modules to EVM bytecode. Then for every modules* $\Omega^\dagger$ *and*

address $a^\dagger$ it has to hold:

$$
\begin{aligned}
&\forall State_{evm}, State'_{evm}, State''_{evm} \in States_{evm} \\
&\quad a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}^*\ State_{evm}\ !}\!\!\!\!\rightarrow_{evm} State'_{evm} \\
&\qquad \vee State_{evm} = State'_{evm} \in States_{evm,init} \\
&\quad \wedge a^\dagger \rhd State'_{evm} \xrightarrow{\texttt{call}^*\ State''_{evm}\ ?}\!\!\!\!\rightarrow_{evm} State''_{evm} \\
&\quad \wedge State_{evm} = (\mu, \sigma, \iota) :: S' \\
&\quad \wedge State''_{evm} = (\mu'', \sigma'', \iota'') :: S'' \\
&\\
&\quad \Longrightarrow \forall a, n \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}, r, x \in \mathbb{N}_{160} \\
&\qquad a = \mathit{tr\_stor}_\tau(x,n) \vee \mathit{ext\_stor}_\tau(x,n) \\
&\qquad \wedge \sigma\left(a^\dagger\right).stor \vdash stores(r,\tau,a) * \mathbf{true} \\
&\\
&\qquad \Longleftrightarrow \exists x' \in \mathbb{N}_{160}, n', a' \in \mathbb{N}_{256}. \\
&\qquad\qquad a' = \mathit{tr\_stor}_\tau(x',n') \vee \mathit{ext\_stor}_\tau(x',n') \\
&\qquad\qquad \wedge \sigma''\left(a^\dagger\right).stor \vdash stores(r,\tau,a') * \mathbf{true} \\
&\qquad\qquad \left(\wedge \sigma''\left(a^\dagger\right).stor \not\vdash stores(r,\tau,a) * \mathbf{true}\right. \\
&\qquad\qquad\quad \left.\vee a = a'\right)
\end{aligned}
$$

Compiler Obligation 3 (Preservation of Transients and Externals upon Termination)**.** *Given a compiler $[\![\cdot]\!]^{move}_{evm} : \mathbb{Q} \mapsto \mathcal{L}\left[\mathbb{B}^8\right]$ from Move modules to EVM bytecode. Then for every modules $\Omega^\dagger$ and address $a^\dagger$ it has to hold:*

$$
\begin{aligned}
&\forall State_{evm}, State'_{evm}, State''_{evm} \in States_{evm} \\
&\quad a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}^*\ State_{evm}\ !}\!\!\!\!\rightarrow_{evm} State'_{evm} \\
&\qquad \vee State_{evm} = State'_{evm} \in States_{evm,init} \\
&\quad \wedge a^\dagger \rhd State'_{evm} \xrightarrow{\texttt{term}\ State''_{evm}}\!\!\!\!\rightarrow_{evm} State''_{evm} \\
&\quad \wedge State_{evm} = (\mu, \sigma, \iota) :: S' \\
&\quad \wedge State''_{evm} = HALT(\sigma'', d) :: S'' \\
&\\
&\quad \Longrightarrow \forall a, n \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}, r, x \in \mathbb{N}_{160} \\
&\qquad a = \mathit{tr\_stor}_\tau(x,n) \\
&\qquad \wedge \sigma\left(a^\dagger\right).stor \vdash stores(r,\tau,a) * \mathbf{true} \\
&\\
&\qquad \Longleftrightarrow \exists x' \in \mathbb{N}_{160}, n', a' \in \mathbb{N}_{256}. \\
&\qquad\qquad a' = \mathit{tr\_stor}_\tau(x',n') \vee \mathit{ext\_stor}(x',n') \\
&\qquad\qquad \wedge \sigma''\left(a^\dagger\right).stor \vdash stores(r,\tau,a') * \mathbf{true} \\
&\qquad\qquad \left(\wedge \sigma''\left(a^\dagger\right).stor \not\vdash stores(r,\tau,a) * \mathbf{true}\right. \\
&\qquad\qquad\quad \left.\vee a = a'\right)
\end{aligned}
$$

Obligation 3 handles the case where we terminate after the last return instead of calling back. Both obligation ensure that no externals or transient get lost between two invocation and further, that no additional resources are created (*i.e.* no creation outside of the contract). However, note that Obligation 2 and Obligation 3 allow for resources to move around the sets of transients and externals.

While Externals and Transient are allowed to be moved around, internals have to be preserved during invocations (in the very same location). This is captured by the following definition:

Compiler Obligation 4 (Strict Preservation of Internals)**.** *Given a compiler $[\![\cdot]\!]^{move}_{evm} : \mathbb{Q} \mapsto \mathcal{L}\left[\mathbb{B}^8\right]$ from Move modules to EVM bytecode. Then for every modules $\Omega^\dagger$ and address $a^\dagger$ it has to hold:*

$$
\begin{aligned}
&\forall State_{evm}, State'_{evm}, State''_{evm}, State^*_{evm} \in States_{evm} \\
&\quad a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}^*\ State_{evm}\ !}\!\!\!\!\rightarrow_{evm} State'_{evm} \\
&\qquad \vee State_{evm} = State'_{evm} \in States_{evm,init} \\
&\quad \wedge a^\dagger \rhd State'_{evm} \xrightarrow{\varepsilon}_{evm} \cdots \xrightarrow{\varepsilon}_{evm} State^*_{evm} \\
&\quad \wedge a^\dagger \rhd State^*_{evm} \xrightarrow{\texttt{call}^*\ State''_{evm}\ ?}\!\!\!\!\rightarrow_{evm} State''_{evm} \\
&\quad \wedge State_{evm} = (\mu, \sigma, \iota) :: S' \\
&\quad \wedge State^*_{evm} = (\mu^*, \sigma^*, \iota^*) :: S'' \\
&\quad \wedge \forall a \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}, r, x \in \mathbb{N}_{160} \\
&\qquad a = \mathit{int\_stor}_\tau(x) \\
&\qquad \Longrightarrow \sigma\left(a^\dagger\right).stor \vdash stores(r,\tau,a) * \mathbf{true} \\
&\qquad\quad \Longleftrightarrow \sigma^*[a^\dagger].stor \vdash stores(r,\tau,a) * \mathbf{true}
\end{aligned}
$$

Compiler Obligation 5 (Strict Preservation of Internals upon Termination)**.** *Given a compiler $[\![\cdot]\!]^{move}_{evm} : \mathbb{Q} \mapsto \mathcal{L}\left[\mathbb{B}^8\right]$ from Move modules to EVM bytecode. Then for every modules $\Omega^\dagger$ and address $a^\dagger$ it has to hold:*

$$
\begin{aligned}
&\forall State_{evm}, State'_{evm}, State''_{evm}, State^*_{evm} \in States_{evm} \\
&\quad a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}^*\ State_{evm}\ !}\!\!\!\!\rightarrow_{evm} State'_{evm} \\
&\qquad \vee State_{evm} = State'_{evm} \in States_{evm,init} \\
&\quad \wedge a^\dagger \rhd State'_{evm} \xrightarrow{\varepsilon}_{evm} \cdots \xrightarrow{\varepsilon}_{evm} State^*_{evm} \\
&\quad \wedge a^\dagger \rhd State^*_{evm} \xrightarrow{\texttt{term}\ State''_{evm}}\!\!\!\!\rightarrow_{evm} State''_{evm} \\
&\quad \wedge State_{evm} = (\mu, \sigma, \iota) :: S' \\
&\quad \wedge State^*_{evm} = (\mu^*, \sigma^*, \iota^*) :: S'' \\
&\quad \wedge \forall a \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}, r, x \in \mathbb{N}_{160} \\
&\qquad a = \mathit{int\_stor}_\tau(x) \\
&\qquad \Longrightarrow \sigma\left(a^\dagger\right).stor \vdash stores(r,\tau,a) * \mathbf{true} \\
&\qquad\quad \Longleftrightarrow \sigma^*[a^\dagger].stor \vdash stores(r,\tau,a) * \mathbf{true}
\end{aligned}
$$

Finally, upon calling we require that the compiler prepares the execution accordingly, i.e. after the call we must be at some pc, relatable to the initial pc of some of the Move module's function, the operand stack must be empty and all the parameters are passed correctly. All of these is handled by the next compiler obligation:

Compiler Obligation 6 (Sound Call)**.** *Given a compiler $[\![\cdot]\!]^{move}_{evm} : \mathbb{Q} \mapsto \mathcal{L}\left[\mathbb{B}^8\right]$ from Move modules to EVM bytecode.*

Then for every modules $\Omega^\dagger$ and address $a^\dagger$ it has to hold:

$$
\begin{aligned}
&\forall State_{evm}, State^*_{evm}, State'_{evm} \in States_{evm}.\\
&\quad a^\dagger \rhd State^*_{evm} \xrightarrow{\texttt{call}^*\ State_{evm}\texttt{?}} {}_{evm} State_{evm}\\
&\quad a^\dagger \rhd State_{evm} \xrightarrow{\texttt{call}\ State'_{evm}\texttt{?}} {}_{evm} State'_{evm}\\
&\quad \wedge State_{evm} = (\mu, \sigma, \iota) :: S\\
&\quad \wedge State'_{evm} = (\mu', \sigma', \iota) :: S\\
&\quad \implies \exists \phi \in \Phi_{\Omega^\dagger}.\mathit{src\_map}(\phi, 0) = \mu'.pc\\
&\qquad \wedge \forall 0 \leq i < |\phi.form\_par|\\
&\qquad\quad \phi.form\_par\_type[i] \in \mathcal{T}_{\Omega^\dagger}\\
&\qquad\quad \implies \exists a, a', n \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}, r\\
&\qquad\qquad\quad \tau = \phi.form\_par\_type[i]\\
&\qquad\qquad\quad \wedge a = \mathit{local\_stor}(\mu'.m, \phi.form\_par[i])\\
&\qquad\qquad\quad \wedge a' = \mathit{tr\_stor}(\iota.actor, n)\\
&\qquad\qquad\quad\quad \vee a' = \mathit{ext\_stor}(\iota.actor, n)\\
&\qquad\qquad\quad \wedge \mu'.m \vdash stores(r, \phi.form\_par\_type[i], a) * \mathbf{true}\\
&\qquad\qquad\quad \wedge \sigma(a^\dagger).stor \vdash stores(r, \tau, a') * \mathbf{true}\\
&\qquad\qquad\quad \wedge \sigma'(a^\dagger).stor \not\vdash stores(r, \tau, a') * \mathbf{true}\\
&\qquad \wedge \forall l \in dom(L) \setminus \{\phi.form\_par[i] \mid 0 \leq i < |\phi.form\_par|\}\\
&\qquad\quad \nexists a, v. a = \mathit{local\_stor}(\mu'.m, l)\\
&\qquad\qquad \wedge \mu'.m \vdash stores(v, \tau,) * \mathbf{true}
\end{aligned}
$$

Although, we only consider modules without dependencies (*i.e.* our modules do not call other modules) we still need to ensure that the compiler is not introducing calls that lead to reentrant execution (which we cannot map to Move behavior). Note however, that our notion of reentrancy is weaker than what usually is used, *i.e.*, we allow the contract to be reentered as long as no function relating to a function of the Move module is reentered.

Compiler Obligation 7 (No Reentrancy)**.** *Given a compiler $[\![\cdot]\!]^{move}_{evm} : \mathbb{Q} \mapsto \mathcal{L}[\mathbb{B}^8]$ from Move modules to EVM bytecode. Then for every modules $\Omega^\dagger$ the code $[\![\Omega^\dagger]\!]^{move}_{evm}$ is protected agains reentrancy of Move-related functions. That means, for every trace t_{evm} such that*

$$[\![\Omega^\dagger]\!]^{move}_{evm} \vdash t_{evm} :: \texttt{call}\ State_{evm}\texttt{?}$$

there does not exist a state $State'_{evm}$ such that

$$[\![\Omega^\dagger]\!]^{move}_{evm} \vdash t_{evm} :: \texttt{call}\ State_{evm}\texttt{?} :: \texttt{call}\ State'_{evm}\texttt{?}$$

Note that Observation 7 is formulated in the trace semantics instead of the large-step semantics used for the other obligations. Traces do not contain occurrences of actions of the form $\texttt{call}^*\ State_{evm}\texttt{?}$ and $\texttt{ret}^*\ State_{evm}\texttt{!}$.

Finally, our compiler must make sure that before the first invocation of our contract in a transaction and at termination of the transaction, there are no resources stored as transient. This captures the fact that no resources are allowed to be dropped. We capture this behavior with the following obligation:

Compiler Obligation 8 (Transient management)**.** *Given a compiler $[\![\cdot]\!]^{move}_{evm} : \mathbb{Q} \mapsto \mathcal{L}[\mathbb{B}^8]$ from Move modules to EVM bytecode. Then for every modules $\Omega^\dagger$ and any complete trace of the form*

$$t_{evm} := (\texttt{call}\ State_{evm}\texttt{?}) :: t'_{evm} :: (\texttt{term}\ State'_{evm})$$

such that $[\![\Omega^\dagger]\!]^{move}_{evm} \vdash t_{evm}$ and

$$State_{evm} = (\mu, \sigma, \iota) :: S$$
$$State'_{evm} = HALT(\sigma', d) :: S'$$

it has to hold that

$$
\begin{aligned}
\nexists \quad & n \in \mathbb{N}_{256}, x \in a, \tau \in \mathcal{T}_{\Omega^\dagger}, r\\
& a = \mathit{tr\_stor}_\tau(x, n)\\
& \wedge \sigma(a^\dagger).stor \vdash stores(r, \tau, a)\\
& \wedge \sigma'(a^\dagger).stor \vdash stores(r, \tau, a)
\end{aligned}
$$

With Obligation 1-8 we can then finally formulate the notion of *global preservation* as follows:

Definition 12 (Globally preserving)**.** *Consider a compiler function $[\![\cdot]\!]^{move}_{evm} : \mathbb{Q} \mapsto \mathcal{L}[\mathbb{B}^8]$. We say that $[\![\cdot]\!]^{move}_{evm}$ is **globally preserving** if it fulfills Obligation 1-8.*

For showing soundness, our compiler needs to fulfill one further property, namely *sound initialization* which forces the compiler to initialize the storage for internal, external and transient resources to being empty initially. This prevents our contract being deployed in a state that cannot be related to any state in the Move word:

Definition 13 (Sound initialization)**.** *Consider a compiler function $[\![\cdot]\!]^{move}_{evm} : \mathbb{Q} \mapsto \mathcal{L}[\mathbb{B}^8]$. We say that $[\![\cdot]\!]^{move}_{evm}$ is **soundly initializing** if for any module $\Omega^\dagger$ the storage of externals, internals and transients for the contract $[\![\Omega^\dagger]\!]^{move}_{evm}$ is empty after deployment.*

Theorem 2 (Soundness)**.** *A compiler function $[\![\cdot]\!]^{move}_{evm} : \mathbb{Q} \mapsto \mathcal{L}[\mathbb{B}^8]$ is sound as defined in Definition 10 if it fulfills Definition 11, Definition 12, and Definition 13.*

B.3 Proof

In our proof we consider a fixed module $\Omega^\dagger$ and its compilation $[\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}}$. We then have to show that for each trace t_{evm} such that $[\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}} \vdash t_{evm}$ there exists a trace t_{mv} such that $\Omega^\dagger \vdash t_{mv}$ and $t_{mv} \sim_{\Omega^\dagger} t_{evm}$.

To do so, it is sufficient, given a trace t_{evm}, to construct a Move state (M, G, st, C, A) that is admitting the same path. Our construction of such state will always have the same structure: we assume a module Ω_{main} consisting of a single function ϕ_{main}. This function is constructed through the proof and will ensure that we get a similar trace structure on the Move side. Besides that additional modules might be constructed through our proof the help reconstruct a state that can be matched to a state on the EVM side.

B.3.1 Initialization

Consider any trace t_{evm} such that $[\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}} \vdash t_{evm}$. Let us for now ignore the special case of a trace of the form $\texttt{term}\ State_{evm}$ (*i.e.* our modules was never called). Then

we consider the first action of the trace which must be of the form $\texttt{call}\ State_{evm}\texttt{?}$ with $State_{evm} := (\mu, \sigma, \iota) :: S$. Since $[\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}} \vdash t_{evm}$ we know, that for some address $a^\dagger$ we have that $\sigma(a^\dagger).code = [\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}}$.

Let $State_{evm,init}$ be the initial state of the EVM at the begin of the transaction described by t_{evm} and let $State_{evm,init} := [(\mu_{init}, \sigma_{init}, \iota_{init})]$. Given the storage $stor_{init} = \sigma_{init}(a^\dagger).stor$, we introduce the sets

$$\begin{aligned}
Tr(stor_{init}) &:= \{(r,a) \mid \exists\, n \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}, a \in \mathbb{N}_{160}. \\
&\qquad stor_{init} \vdash stores(r, \tau, \mathsf{tr_stor}_\tau(a,n)) * \mathbf{true} \\
Ext(stor_{init}) &:= \{(r,a) \mid \exists\, n \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}, a \in \mathbb{N}_{160}. \\
&\qquad stor_{init} \vdash stores(r, \tau, \mathsf{ext_stor}_\tau(a,n)) * \mathbf{true} \\
Int(stor_{init}) &:= \{(r,a) \mid \exists\, \tau \in \mathcal{T}_{\Omega^\dagger}, a \in \mathbb{N}_{160}. \\
&\qquad stor_{init} \vdash stores(r, \tau, \mathsf{int_stor}_\tau(a)) * \mathbf{true}
\end{aligned}$$

In a first step, we now want to construct a state $State_{mv,init} := (M_{init}, G_{init}, st_{init}, C_{init}, A_{init})$ such that

$$\Omega^\dagger \triangleright State_{mv,init} \xrightarrow{\texttt{call}\ State_{mv}\texttt{?}} \!\!\!\!\rightarrow_{mv} State_{mv}$$

and $State_{mv} \sim_{\Omega^\dagger, a^\dagger} State_{evm}$.

Address mapping First, let us assume a fixed relation $\cdot \sim \cdot$ that relates addresses a_{mv} in Move to addresses a_{evm} in EVM. The relation is a partial relation such that each Move address is at most related to one EVM address and vice versa. Note that the address space in Move and EVM differ. However, since our proof only works *in one direction* (namely from EVM to Move) and the address space of EVM(20 byte addresses) is smaller than that of move (32 byte addresses) we always can be sure to have enough addresses available to reconstruct a trace occurring in EVM also in a theoretical setting. The relation will be defined on the fly for the relevant parts. For all the other addresses, the mapping is arbitrary (*i.e.* it can also be undefined).

The Main Module We introduce a module Ω_{main} that acts as an entry point. This modules has only one function, namely ϕ_{main} which will be executed in the beginning. The function ϕ_{main} will be constructed through the proof to reconstruct the trace we have encountered in EVM. In Move, the signer of the transaction is usually passed as a parameter to the entry point. We will denote the signer with $a_{mv_{signer}}$ and assume that $a_{mv_{signer}} \sim \iota_{init}.sender$. The signer on the Move side is important as new data in the globals can only be created in the scope of the signer. Furthermore, we assume an address $a_{mv_{main}}$ that can be related to an arbitrary (not yet used) EVM address (or we let the relation undefined) and set $\Omega_{main} \in A_{init}(a_{mv_{main}})$.

Internals Consider the set $Int(stor_{init})$ of internals. We then introduce a Move memory M_{init} and the globals G_{init} such that for each tuple $(r, a_{evm}) \in Int(stor_{init})$, $M_{init}(\ell) = r$ for some fresh location ℓ and set $G_{init}(a_{mv}, \texttt{type\_of}(r)) = \ell$ for some Move address such that $a_{mv} \sim a_{evm}$.

Externals Next, consider the set $Ext(stor_{init})$ of externals. Then for each tuple $(r, a_{evm,i})$ with $\mathcal{T}_{\Omega^\dagger} \ni \tau = \texttt{type\_of}(r)$ we assume that Ω_{main} implements a type $\tau_{wrp,\tau} \in \mathcal{T}_{\Omega_{main}}$ such that $\tau_{wrp,\tau}$ wraps around τ (*i.e.*, $\tau_{wrp,\tau}$ has a field f of type τ).

We then update the globals G_{init} and the memory M_{init} such that $M_{init}(\ell) = r'$ with r' of type $\tau_{wrp,\tau}$ and $r'.f = r$ and $G_{init}(a_{mv}, \tau_{wrp,\tau}) = r'$ such that a_{mv} is an arbitrary address but unique for any tuple in $Ext(stor_{init})$.

Transients By Obligation 8 we know that it holds that $Tr(stor_{init}) = \emptyset$. Hence, for now we do not have to care about transients.

The Initial Move State We then construct our initial state $State_{mv,init}$ as follows:

$$State_{mv,init} := (M_{init}, G_{init}, [\,], [(\Omega_{main}, \phi_{main}, L, 0)], A_{init})$$

Lemma 2 (Initial Contstruction)**.** *Let $State_{evm,init}$ be the initial state of a trace $t_{evm} := \texttt{call}\ State_{evm}\texttt{?} :: t'_{evm}$ and let $State_{mv,init}$ be the Move state constructed as described. Then it holds that*

$$\begin{aligned}
&\forall a_{evm} \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}. \\
&\quad stor_{init} \vdash stores(r, \tau, \mathit{int_stor}_\tau(a_{evm})) * \mathbf{true} \\
&\quad \Longrightarrow \exists a_{mv} \in \mathcal{A}. \\
&\qquad\qquad a_{mv} \sim a_{evm} \\
&\qquad\qquad \wedge\, G_{init}(a_{mv}, \tau) = \ell \wedge M_{init}(\ell) = r
\end{aligned}$$

$$\begin{aligned}
&\forall a_{evm} \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}. \\
&\ n \in \mathbb{N}_{256}. \\
&\quad stor_{init} \vdash stores(r, \tau, \mathit{ext_stor}_\tau(a_{evm}, n)) * \mathbf{true} \\
&\quad \Longrightarrow \exists a_{mv} \in \mathcal{A}, \tau' \in \mathcal{T}_{\Omega_{main}}. \\
&\qquad\qquad G_{init}(a'_{mv}, \tau') = \ell \wedge r \in wraps(M_{init}(\ell))
\end{aligned}$$

Proof. Both follow directly from the definition of $Ext(stor_{init})$ and $Int(stor_{init})$ and the construction of the state $State_{mv,init}$. □

Note that Lemma 2 refers to Definition 3 and Definition 4 introducing our similarity relation for internals and externals respectively. However, while $State_{mv,init}$ and $State_{evm,init}$ would fulfill the requirements introduced by Definition 3, this does not apply for Definition 4, as externals are currently all stored in the context of Ω_{main} on the Move side.

Currently, the behavior of our function ϕ_{main} is not specified. In what follows, we will tweak ϕ_{main} in a way to produce the desired traces.

B.3.2 The First Call

Let's reconsider the first action $\texttt{call}\ State_{evm}\texttt{?}$ of the trace t_{evm} with $[\![\Omega^\dagger]\!]^{\texttt{move}}_{\texttt{evm}} \vdash t_{evm}$ where $State_{evm} := (\mu, \sigma, \iota) :: S$. Furthermore, let $State_{evm,init}$ be the initial state on *EVM* side as described before and $State_{mv,init}$ the initial on the Move side as

constructed previously. Let $stor := \sigma\left(a^{\dagger}\right).stor$ and consider the sets

$$\begin{aligned}
Tr(stor) &:= \{(r,a) \mid \exists\, n \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^{\dagger}}, a \in \mathbb{N}_{160}. \\
&\qquad stor \vdash stores(r, \tau, \mathsf{tr_stor}_{\tau}(a,n)) * \mathbf{true} \\
Ext(stor) &:= \{(r,a) \mid \exists\, n \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^{\dagger}}, a \in \mathbb{N}_{160}. \\
&\qquad stor \vdash stores(r, \tau, \mathsf{ext_stor}_{\tau}(a,n)) * \mathbf{true} \\
Int(stor) &:= \{(r,a) \mid \exists\, \tau \in \mathcal{T}_{\Omega^{\dagger}}, a \in \mathbb{N}_{160}. \\
&\qquad stor \vdash stores(r, \tau, \mathsf{int_stor}_{\tau}(a)) * \mathbf{true}
\end{aligned}$$

Furthermore, let $L := \{r \mid \mu.m \vdash stores(r, \tau', i)\}$ with $i := \mathsf{local\_stor}(\mu.m, \phi.form\_par[j])$ and $\tau' := \phi.form\_par\_type[j]$ such that $\tau' \in \mathcal{T}_{\Omega^{\dagger}}$ for any $0 \leq j < |\phi.form\_par|$, be the set of locals stored in the memory at $State_{evm}$ and let $Tr(stor_{init})$, $Int(stor_{init})$, and $Ext(stor_{init})$ be the sets as defined previously. Then it holds that

- the internals are preserved, *i.e.*,

  $$Int(stor_{init}) = Int(stor)$$

  due to Obligation 4.

- the set of externals is preserved although they might have been moved around to transients or are now used as parameters and therefore are in L, *i.e.*,

  $$\begin{aligned}
  \{r \mid (r,a) \in Ext(stor_{init})\} = &\{r \mid (r,a) \in Ext(stor)\} \\
  &\cup \{r \mid (r,a) \in Tr(stor)\} \\
  &\cup L
  \end{aligned}$$

  due to Obligation 8, Obligation 2 and Obligation 6.

Auxiliarly Modules From the previous observation we know that every resource in $Ext(stor)$ and $Tr(stor)$ is in the set $Ext(stor_{init})$. By construction of $State_{mv,init}$ we therefore know that for each such resource, their is a corresponding resource stored *in the scope* of the module Ω_{main}. Hence, to create a Move state that matches $stor$ we can adapt the main function ϕ_{main} and *distribute* the resources in $Ext(stor_{init})$ to other accounts.

For that purpose we will introduce auxiliary modules as needed. All of these auxiliary modules Ω_i have however the same structure:

- For each type $\tau \in \mathcal{T}_{\Omega^{\dagger}}$, Ω_i offers a function $\phi_{store,\tau}$ that takes a signer object, a resource of type τ and a parameter $n \in \mathbb{N}$, wraps the resource using $\tau_{wrp,n} \in \mathcal{T}_{\Omega_i}$ and stores it. We use the parameter n in case there is already a resource stored at that location. Hence, we assume that Ω_i provides a sufficient number of wrapper types. This is safe, as we consider only a specific trace and the number of resources that can be created within a trace if finite by the nature of blockchain transactions.

- For each type $\mathcal{T} \in \mathcal{T}_{\Omega^{\dagger}}$, Ω_i provides a function $\phi_{get,\tau}$ that takes an address, a resource of type τ and a parameter $n \in \mathbb{N}$, loads the resource of type $\tau_{wrp,n} \in \mathcal{T}_{\Omega_i}$ from memory, unwraps the resource r of type τ (and by doing so destroys the wrapper resource) and returns r.

- For each function $\phi \in \Phi_{\Omega^{\dagger}}$, Ω_i offers a proxy function that has the same signature as ϕ. The behavior of this proxy function differs however depending on where we use it and will be defined separately where needed.

Reconstructing the set of Externals Now consider the set $Ext(stor)$ and let

$$S := \{a_{evm} \mid (r, a_{evm}) \in Ext(stor)\}$$

be the set of distinct address occurring in $Ext(stor)$. Then for such $a_{evm} \in S$ we create a Move module $\Omega_{a_{mv}}$ as described before and update the account map such that $\Omega_{a_{mv}} \in A_{init}(a_{mv})$ with $a_{mv} \sim a_{evm}$.

Then for each tuple $(r, a_{evm}) \in Ext(stor)$ we know that by construction, there is some type $\tau \in \mathcal{T}_{\Omega_{main}}$ and some address $a_{mv} \in \mathcal{A}$ such that $G_{init}(a_{mv}, \tau) = \ell$ and $M(\ell).f = r$ for some field name f. Hence, for each such (r, a_{evm}) we adapt the function ϕ_{main} by adding the following lines of code:

```
let { f } = move_from<τ>(addr_of(a));
Ω_{a_mv}::φ_{store,type_of(r)}(signer, f, n);
```

By default we set parameter n to 0 for each call to $\phi_{store,\texttt{type\_of}(r)}$. Only if a module has multiple resources of the same type, we adapt n accordingly.

Lemma 3 (Reconstruction of Externals)**.** *Let $State'_{mv}$ after the execution of the previously defined construction of ϕ_{main} (and therefore $State_{mv,init} \xrightarrow{\varepsilon}{}^{*}_{mv} State'_{mv}$). And let $State'_{mv} := (M, G, st, C, A)$. Then it holds that*

$$\begin{aligned}
&\forall a_{evm} \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^{\dagger}}, n \in \mathbb{N}_{256}. \\
&\quad stor \vdash stores(r, \tau, \mathsf{ext_stor}_{\tau}(a_{evm}, n)) * \mathbf{true} \\
&\quad \implies \exists a_{mv}, a'_{mv} \in \mathcal{A}, \tau' \in \mathcal{T}_{\Omega_{a_{mv}}}. \\
&\qquad\quad a_{mv} \sim a_{evm} \\
&\qquad\quad \wedge\, G(a'_{mv}, \tau') = \ell \wedge r \in wraps(M(\ell))
\end{aligned}$$

Proof. By construction of ϕ_{main} and definition of $\phi_{store,\tau}$. □

Reconstructing the set of Transients Reconsider that with $State_{evm} := (\mu, \sigma, \iota) :: S$ we denote the state of the action `call` $State_{evm}$`?`. Then let

$$S := \{a_{evm} \mid (r, a_{evm}) \in Tr(stor)\} \cup \{\iota.sender\}$$

then for each address $a_{evm} \in S$ we first check if we have already created a module in the previous step. If not, we do so similarly then before. Also, in the same way as before, we invoke the respective functions $\phi_{store,\tau}$. Hence, for each tuple $(r, a_{evm}) \in Tr(stor)$ we have that there is some type $\tau \in \mathcal{T}_{\Omega_{main}}$ and some address $a_{mv} \in \mathcal{A}$ such that $G_{init}(a_{mv}, \tau) = \ell$ and $M(\ell).f = r$ for some field name f. We therefore add the lines

```
let { f } = move_from<τ>(addr_of(a));
Ω_{a_mv}::φ_{store,type_of(r)}(signer, f, n);
```

to the function Ω_{main} for each such tuple. Again, n has to be adapted accordingly. Now let $D := (\Omega_{a_{mv,0}}, \ldots, \Omega_{a_{mv,n}})$ be a sequence of all the modules that correspond to addresses in S such that $\Omega_{a_{mv,n}}$ corresponds to $\iota.sender$.

Consider again the state $State_{evm}$ of the first action `call` $State_{evm}$`?`. Then by Obligation 6 we know that at $State_{evm}$ we are at an execution state that can be mapped to a function $\phi \in \Phi_{\Omega^\dagger}$. So let ϕ be that function. Then we know that for ϕ each modules $\Omega_{a_{mv,i}}$ of D has a proxy function $\Omega_{a_{mv,i}} :: \phi$ that has the same signature as ϕ. Then for all $0 \leq i < n$ we define the behavior of $\Omega_{a_{mv,i}} :: \phi$ as follows:

```
let ( f_1 ) = move_from<τ_wrp,0>;
 ...
let ( f_k ) = move_from<τ_wrp,k>;

let r = Ω_{a_mv,i+1} ::φ(...);

move_to<τ_wrp,k>(&signer, τ_wrp,k {f: f_k});
 ...
move_to<τ_wrp,1>(&signer, τ_wrp,1 {f: f_1});
return r;
```

where for move-from and move-to correspond to each tuple $(r, a_{evm,i}) \in Tr(stor)$. So intuitively, we load every resource marked as transient in EVM from memory to the locals, call the next proxy function and afterwards store them back to memory again. For the module $\Omega_{a_{mv,n}}$ (which corresponds to the caller in EVM) we adapt above construction to call the actual function $\Omega^\dagger :: \phi$ instead of another proxy function.

The Actual Call With the previous construction, we are finally able to reconstruct the call on the Move side. For that we adapt the ϕ_{main} such that it calls $\Omega_{a_{mv0}} :: \phi$ from the previously defined sequence D.

For that to work, however, ϕ_{main} must provide parameters to $\Omega_{a_{mv0}} :: \phi$. We are here mostly interested in resource parameters as parameters to primitive types can be trivially created by ϕ_{main}. First note, that the resource parameters that needed to be passed correspond to the set L of resources on EVM side. We have already observed that

$$\begin{aligned}\{r \mid (r,a) \in Ext(stor_{init})\} = &\{r \mid (r,a) \in Ext(stor)\} \\ &\cup \{r \mid (r,a) \in Tr(stor)\} \\ &\cup L\end{aligned}$$

and by applying Lemma 2 we know that for some wrapper type $\tau \in \mathcal{T}_{\Omega_{main}}$ and some address a_{mv} we have that

$$G_{init}(a_{mv}, \tau) = \ell \wedge r \in wraps(M_{init}(\ell))$$

for all $r \in L$. Since by our construction, resources in L have not been touched and since Ω_{main} is the only module with access to these resources, this must also hold for memory and globals G and M after the execution of our current construction.

Now let $r_i \in L$, $a_{mv,i} \in \mathcal{A}$ and $\tau_i \in \mathcal{T}_{\Omega_{main}}$ be a resource, an address, and a type respectively such that $G_{init}(a_{mv}, \tau) = \ell$, $r_i \in wraps(M_{init}(\ell))$, and `type_of`$(r_i) =$ $\phi.form\_par\_type[i]$. We then can add the following lines to the function ϕ_{main}:

```
let ( r_i ) = move_from<τ_i>(a_mv,i);
```

Consider then the parameter signature $(\tau_0, \ldots, \tau_{|\phi.form\_par|-1})$ of the function ϕ. Then we construct the sequence $(v_0, \ldots, v_{|\phi.form\_par|-1})$ as follows: For any $0 \leq i < |\phi.form\_par|$, if τ_i is a primitive type (*e.g.*, an integer) then we set v_i to the value such that $v_i \sim n \in \mathbb{N}_{256}$ where n is the value stored in memory on the EVM side at the memory location of the local, *i.e.*,

$$\mu.m \vdash stores(n, \tau, \mathsf{local\_stor}(\mu.m, \phi.form\_par[i]))$$

In case τ_i is a resource type, we set it to the corresponding r_i constructed previously. Then finally, we adapt ϕ_{main} by adding the call $\Omega_{a_{mv,0}} :: \phi(v_0, \ldots, v_{|\phi.form\_par|-1})$. Given the construction, we then know that there is some state $State_{mv}$ such that

$$\Omega^\dagger \rhd State_{mv,init} \xrightarrow{\texttt{call}\ State_{mv}?} _{mv}\!\!\gg State_{mv}$$

Lemma 4. *For the state $State_{evm}$ and $State_{mv}$ as defined before it holds that $State_{mv} \sim_{\Omega^\dagger, a^\dagger} State_{evm}$.*

Proof. To show that indeed $State_{mv} \sim_{\Omega^\dagger, a^\dagger} State_{evm}$ we show that by assuming the relation, all the implications introduced by Definition 1-7 hold.

So let $State_{mv}$ and $State_{evm}$ be defined as

$$\begin{aligned}State_{mv} &:= \left(M, G, st, \left(\Omega^\dagger, \phi, L, pc\right) :: C, A\right), \\ State_{evm} &:= (\mu, \sigma, \iota) :: S.\end{aligned}$$

RE Definition 1 (Point of Execution) Then first, consider Definition 1, which requires that $State_{mv}$ and $State_{evm}$ are both in the *same point of execution*. By Obligation 6 we know that for some $\phi' \in \Phi_{\Omega^\dagger}$ it holds that $\mathsf{src\_map}_{\Omega^\dagger}(\phi', 0) = \mu.pc$. By construction we have $\phi = \phi'$ and by the definition of our trace semantic for Move we know that $State_{mv}$ is at the beginning of execution of ϕ and therefore $pc = 0$. Therefore, Definition 1 holds.

RE Definition 2 (Operand Stack) Definition 2 is trivially fulfilled as at this point of execution, $\Omega^\dagger$ has not pushed any value on the operand stack. Therefore, the content of the operand stack $\mu.st$ can be arbitrary.

RE Definition 3 (Internals) Note that by Lemma 2, the construction of G_{init} and M_{init} *would fulfill* Definition 3 (in the sense that if we replace globals and memory by the respective initial construction, we see that Definition 3 matches the second statement of Lemma 2).

By construction of G_{init} and M_{init}, for each $(r, a_{evm}) \in Tr(stor)$ we know that for an address $a_{mv} \sim a_{evm}$ we have

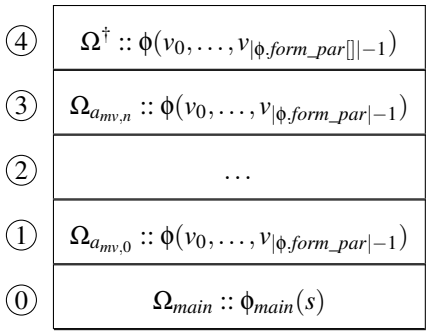

Figure 15: Callstack in Move upon calling $\Omega^\dagger :: \phi$.

that $G_{init}(a_{mv}, \texttt{type\_of}(r)) = \ell$ and $M(\ell) = r$. Furthermore, since for each resource r, $\texttt{type\_of}(r) \in \mathcal{T}_{\Omega^\dagger}$ these resources can only be accessed by $\Omega^\dagger$ and in particular not be manipulated by Ω_{main}. Therefore, our construction could not possible have moved them around and thus at $State_{mv}$ the globals G and the memory M must look the same with respect to $Int(stor)$. It is then easy to see that Definition 3 is fulfilled.

RE Definition 4 (Externals) Similarly then before, we have obtained already a similar result as Definition 4 by Lemma 3. Also, since the construction for reconstructing transients later on, only moves resources of the set $Tr(stor)$, we know that M and G look the same with respect to externals as in the state defined by Lemma 3. Therefore, Definition 4 is fulfilled.

RE Definition 5 (Transients) Consider again the sequence of modules $D := (\Omega_{a_{mv},0}, \ldots, \Omega_{a_{mv},n})$ defined previously. Then at $State_{mv}$ the callstack is of the form shown in Figure 15. Furthermore, by construction, for each tuple $(r, a_{evm}) \in Tr(stor)$ the module $\Omega_{a_{mv}}$ with $a_{mv} \sim_\Omega a_{evm}$ stores r in one of its locals. Consequently, at state $State_{mv}$ we have

$$\begin{aligned}&\forall a_{evm} \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}, n \in \mathbb{N}_{256}.\\ &\quad \sigma(actor).stor \vdash stores(r, \tau, \text{tr\_stor}_\tau(a_{evm}, n)) * \textbf{true}\\ &\quad \exists (\Omega_{a_{mv}}, \phi, L', pc') \in C,\\ &\quad\quad l \in dom(L').r \in wraps(L'(l))\end{aligned}$$

and consequently Definition 5 is fulfilled.

RE Definition 7 (Locals) First note, that the only locals set upon invoking ϕ in $State_{mv}$ are the function parameters. The same holds for $State_{evm}$ due to Obligation 6. By the construction of teh parameter values $(v_0, \ldots, v_{|\phi.form\_par|-1})$ as described before it is then easy to see that Definition 7 is fulfilled.

Concluding Since Definition 1-7 hold, it must hold that $State_{mv} \sim_{\Omega^\dagger, a^\dagger} State_{evm}$ by Definition 8: Since the relation $\sim_{\Omega^\dagger, a^\dagger} \subseteq States_{mv} \times States_{evm}$ is defined to be the largest relation such that Definition 1-7 hold, $State_{mv} \not\sim_{\Omega^\dagger, a^\dagger} State_{evm}$ would be a contradiction. □

Lemma 5 (First Call)**.** *There exists a partial trace t_{mv} of the form $t_{mv} := \texttt{call}\ State_{mv}\texttt{?}$ such that $State_{mv} \sim_{\Omega^\dagger, a^\dagger} State_{evm}$.*

Proof. Direct consequence of the above construction and Lemma 4. □

B.3.3 Local execution

Let us reconsider our trace t_{evm}. As for now, we have assumed that t_{evm} starts with an action $\texttt{call}\ State_{evm}\texttt{?}$. Due to Obligation 7 we know that t_{evm} is not of the form

$$\texttt{call}\ State_{evm}\texttt{?} :: \texttt{call}\ t_{evm}\texttt{?} :: t'_{evm}$$

i.e., t_{evm} is not reentrant. Furthermore, t_{evm} is a complete trace. By the definition of our trace semantics, we know that before halting, the execution of our contract must return. Thus, the the trace is of the form

$$\texttt{call}\ State_{evm}\texttt{?} :: \texttt{ret}\ State'_{evm}\texttt{!} :: t'_{evm}$$

Lemma 6. *Let $State_{evm}$ and $State_{mv}$ the two states of the EVM and Move virtual machine respectively, as defined by our previous construction,* i.e.*, it holds that $[\![\Omega^\dagger]\!]^{move}_{evm} \vdash \texttt{call}\ State_{evm}\texttt{?}$, $\Omega^\dagger \vdash \texttt{call}\ State_{mv}\texttt{?}$ and $State_{mv} \sim_{\Omega^\dagger, a^\dagger} State_{evm}$. Furthermore, let $State'_{evm}$ be the state such that*

$$[\![\Omega^\dagger]\!]^{move}_{evm} \vdash \texttt{call}\ State_{evm}\texttt{?} :: \texttt{ret}\ State'_{evm}\texttt{!}$$

Then there exists a state $State'_{mv}$ such that

$$\Omega^\dagger \vdash \texttt{call}\ State_{mv}\texttt{?} :: \texttt{ret}\ State'_{mv}\texttt{!},$$

and $State'_{mv} \sim_{\Omega^\dagger, a^\dagger} State'_{evm}$.

Proof. This follows directly from our compiler being locally preserving (*cf*. Definition 11). □

B.3.4 Returning

Let $State'_{mv}$ and $State'_{evm}$ be the states of the Move virtual machine and the EVM upon returning as defined by Lemma 6, *i.e.*, it holds that $State'_{mv} \sim_{\Omega^\dagger, a^\dagger} State'_{evm}$.

By definition of the trace semantics, there must exist states $State^*_{evm}, State^{**}_{evm}, State''_{evm}$ such that

$$a^\dagger \triangleright State'_{evm} \xrightarrow{\texttt{ret}\ State'_{evm}\texttt{!}}_{evm} State^*_{evm}$$

and

$$a^\dagger \triangleright State^*_{evm} \xrightarrow{\texttt{ret}^*\ State''_{evm}\texttt{!}}_{evm} State^{**}_{evm}.$$

Furthermore, by construction we have the state $State''_{mv}$ with $\Omega^\dagger \triangleright State'_{mv} \xrightarrow{\texttt{ret}\ State'_{mv}\texttt{!}}_{mv} State''_{mv}$ which describes the execution of the module $\Omega_{a_{mv},n}$ after the call of $\Omega^\dagger :: \phi$ (*cf*. Figure 15).

Lemma 7 (Internals at the End of Invocation)**.** *Let $stor''$ be the state of the storage at $State''_{evm}$ and let G'' and M'' be the globals and memory respectively at state $State''_{mv}$. Then it holds that*

$$\begin{aligned}&\forall a_{evm} \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}.\\ &\quad stor'' \vdash stores(r, \tau, \mathit{int\_stor}_\tau(a_{evm})) * \mathbf{true}\\ &\quad \implies \exists a_{mv} \in \mathcal{A}.\\ &\qquad\qquad a_{mv} \sim a_{evm}\\ &\qquad\qquad \wedge G''(a_{mv}, \tau) = \ell \wedge M''(\ell) = r.\end{aligned}$$

Proof. First, note that we have that $State'_{mv} \sim_{\Omega^\dagger, a^\dagger} State'_{evm}$ where $State'_{mv}$ and $State'_{evm}$ are the states upon returning from the function call to $\Omega^\dagger :: \phi$ as defined before. Furthermore, from $State_{mv}$ to $State'_{mv}$, by Move's semantics, it has to hold that

$$\begin{aligned}\forall a_{mv} \in \mathcal{A}, \mathcal{T} \in \mathcal{T}_{\Omega^\dagger}. G'(a_{mv}, \tau) = G''(a_{mv}, \tau) = \ell\\ \wedge M'(\ell) = M''(\ell)\end{aligned}$$

where G' and M' is the globals and the memory at state $State'_{mv}$, respectively. Note that this trivially holds by the fact that between $State'_{mv}$ and $State''_{mv}$ there is only a single smalls step executing the `Ret` opcode which does not access the storage nor the memory.

From the definition of the similarity relation and by Obligation 1, which states that the set of internals is preserved between $State'_{evm}$ and $State''_{evm}$ we then directly get Lemma 7. □

Lemma 8 (Externals at the End of Invocation)**.** *Let $stor''$ be the state of the storage at $State''_{evm}$ and let G'' and M'' be the globals and memory respectively at state $State''_{mv}$. Then it holds that*

$$\begin{aligned}&\forall a_{evm} \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}.\\ &\quad n \in \mathbb{N}_{256}.\\ &\qquad stor'' \vdash stores(r, \tau, \mathit{ext\_stor}_\tau(a_{evm}, n)) * \mathbf{true}\\ &\qquad \implies \exists a_{mv} \in \mathcal{A}, \tau' \in \mathcal{T}_{\Omega_{main}}.\\ &\qquad\qquad G''(a'_{mv}, \tau') = \ell \wedge r \in wraps(M''(\ell)).\end{aligned}$$

Proof. Analogously to Lemma 7. □

Lemma 9 (Transients at the End of Invocation)**.** *Let $stor''$ be the state of the storage at $State''_{evm}$ and let C'' denote the callstack at $State''_{mv}$,* i.e.*, $State''_{mv} := (M'', G'', st'', C'', A)$. Then it holds that*

$$\begin{aligned}&\forall a_{evm} \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}, n \in \mathbb{N}_{256}.\\ &\quad stor'' \vdash stores(r, \tau, \mathit{tr\_stor}_\tau(a_{evm}, n)) * \mathbf{true}\\ &\quad \exists (\Omega', \phi', L', pc') \in C'',\\ &\qquad l \in dom(L').r \in wraps(L'(l)).\\ &\quad \vee r \in st''\end{aligned}$$

Proof. Transients in Move refer to resources in the locals of other executions frames or at the operand stack at a position not reachable from our module's execution. In our construction, however, we know that all the transient resources are stored in the locals and not on the operand stack. Now let us consider the callstack C' at state $State'_{mv}$ and the storage of the EVM $stor'$ at state $stor'$. By the definition of the similarity relation we know that

$$\begin{aligned}&\forall a_{evm} \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}, n \in \mathbb{N}_{256}.\\ &\quad stor' \vdash stores(r, \tau, \mathsf{tr\_stor}_\tau(a_{evm}, n)) * \mathbf{true}\\ &\quad \exists (\Omega', \phi', L', pc') \in C',\\ &\qquad l \in dom(L').r \in wraps(L'(l)).\end{aligned}$$

Furthermore, note that $C = (\Omega^\dagger, \phi, L, pc) :: C''$. However, as $(\Omega^\dagger, \phi, L, pc)$ describes the state upon returning, we know that by linearity no resources can be in the locals (as otherwise they would be dropped). Consequently it has to hold that

$$\begin{aligned}&\{r \mid r \in wraps(L(l)), (\Omega', \phi', L', pc') \in C'\}\\ &\quad \subseteq \{r \mid r \in wraps(L(l)), (\Omega', \phi', L', pc') \in C''\}.\end{aligned}$$

By Obligation 1 we know that the set of transients stored at $stor''$ equals the set of transient stored at $stor'$ plus all the resources that are on top of stack at $State'_{mv}$ (*i.e.* all returned resources). By similarity of $State'_{evm}$ and $State'_{mv}$ we know that the returned resources are also on top of the operand stack at $State'_{mv}$ and by the small-step semantics of Move (where return values are returned via the operand stack) they are still on top of stack at $State''_{mv}$. It is then easy to see that Lemma 9 indeed holds. □

Intuitively, from the Lemmas 7, 8, and 9 we know that all the resources of our compiled module in EVM are still *somewhere around* at state $State''_{mv}$ after returning from the function call. By construction, all the introduced modules $\Omega_{a_{mv,i}}$ will now secure their respective transient resources by storing them to storage and propagate the return value from the call to $\Omega^\dagger :: \phi$ back to ϕ_{main} of the module Ω_{main}. Consequently, upon returning to ϕ_{main}, the only transient resources are the return values from calling $\Omega^\dagger :: \phi$.

For simplifying the reasoning for further calls, we will now undo our construction by moving back all relevant resources into the scope of the Ω_{main} module. For this, we simply call the respective $\Omega_{a_{mv,i}} :: \phi_{get,\tau}$ offered by each auxiliary module $\Omega_{a_{mv,i}}$ and store it back to memory by again assuming a sufficient amount of wrapper types in $\mathcal{T}_{\Omega^\dagger_{main}}$. Similarly, we also store the return values to storage. After this construction, we have a similar setting as for the initial construction.

Further Function Invocation: For any further function invocation, we use the same construction as for the first call. One subtlety that is noteworthy is however the definition of the sequence of auxiliary modules that were used to define the invocation order for reconstructing transient resources. As in Move no circular dependencies are allowed, we may have to use a fresh sequence for each such call. However, since Move allows for multiple modules being defined for each account, this will not result in a conflict with the similarity definition for addresses introduced for preceding calls.

B.3.5 Termination

Consider a complete trace $t_{evm} := t'_{evm} :: \texttt{term}\ State^{\downarrow}_{evm}$ such that $[\![\Omega^{\dagger}]\!]^{\texttt{move}}_{\texttt{evm}} \vdash t_{evm}$. For any non-empty (partial) trace t'_{evm} we have shown that we can construct an environment in Move such that $\Omega^{\dagger} \vdash t'_{mv}$ with $t'_{mv} \sim_{\Omega^{\dagger},a^{\dagger}} t'_{evm}$.

Now consider an arbitrary trace $t_{evm} := t'_{evm} :: \texttt{term}\ State^{\downarrow}_{evm}$ such that $[\![\Omega^{\dagger}]\!]^{\texttt{move}}_{\texttt{evm}} \vdash t_{evm}$. Then there are to cases we have to differ: either $|t'_{evm}| = 0$ or $|t'_{evm}| > 0$. The first case refers to the case where non of our Move related function got called. In the second case, we have that t'_{evm} is of the form $t'_{evm} = t''_{evm} :: \texttt{ret}\ State'_{evm}\,!$.

For the first case ($|t'_{evm}| = 0$), we again consider the initial state $State_{evm,init}$ at the beginning of the transaction such that $a^{\dagger} \rhd State_{evm,init} \xrightarrow{\texttt{term}\ State^{\downarrow}_{evm}}\!\!\!\!\rightarrow_{evm} State^{\downarrow}_{evm}$ and construct an initial state $State_{mv,init}$ as described in B.3.1. As stated by Lemma 2 we then know that every external resource at $State_{evm,init}$ is stored in the scope of the main module Ω_{main} at state $State_{mv,init}$. The set of transients is empty at this point (*cf*. Obligation 8).

In the second case, where we pre-termination returned in a state $State'_{evm}$ (*i.e.*, $\texttt{ret}\ State'_{evm}\,!$) we know that there exists states $State^{*}_{evm}, State^{**}_{evm}, State''_{evm}$ such that

$$a^{\dagger} \rhd State'_{evm} \xrightarrow{\texttt{ret}\ State'_{evm}\,!}\!\!\!\!\rightarrow_{evm} State^{*}_{evm}$$

and

$$a^{\dagger} \rhd State^{*}_{evm} \xrightarrow{\texttt{ret}^{*}\ State''_{evm}\,!}\!\!\!\!\rightarrow_{evm} State^{**}_{evm}$$

By our construction on the Move side, at the state $State_{mv}$ of returning back to the main function, and after our pre-processing for the call as described in the previous section, we have that every resource in the transient or externals is stored in memory at $State_{mv}$ in the scope of Ω_{main}. Thus, at state $State_{mv,init}$ in the first case as well as at $State_{mv}$ in the second case, the module Ω_{main} has access to all transients (empty in the first case) and externals. We thus will apply the following construction in both cases:

Let $State^{\downarrow}_{evm}$ be the state at termination. By definition, $State^{\downarrow}_{evm}$ is of the form $HALT(\sigma, d)$ and let $stor^{\downarrow}$ be the storage at $State^{\downarrow}_{evm}$. Furthermore, let $State_{mv}$ be the state at the begin of the execution of ϕ_{main} (first case) or the state after returning to ϕ_{main} from the last call (second case). Then consider again the sets of transients, externals and internals denoted by $Tr\left(stor^{\downarrow}\right)$, $Ext\left(stor^{\downarrow}\right)$, and $Int\left(stor^{\downarrow}\right)$, respectively.

First of all, by Obligation 8, we have that $Tr\left(stor^{\downarrow}\right) = \emptyset$. Furthermore, by construction and by Obligation 5 we already know that

$$\begin{aligned}
&\forall a_{evm} \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^{\dagger}}.\\
&\quad stor^{\downarrow} \vdash stores(r, \tau, \mathsf{int_stor}_{\tau}(a_{evm})) * \mathbf{true}\\
&\quad \Longrightarrow \exists a_{mv} \in \mathcal{A}.\\
&\qquad\qquad a_{mv} \sim a_{evm}\\
&\qquad\qquad \wedge\, G(a_{mv}, \tau) = \ell \wedge M(\ell) = r.
\end{aligned}$$

where G and M is the state of the globals an memory at $State_{mv}$ respectively. Hence, by considering the similarity relation on halting states we see that it suffices to reconstruct the set of externals on the Move side. However, for this we can borrow the technique we already used for the construction of the call actions before:

For every tuple $(r_i, a_{evm,i}) \in Ext\left(stor^{\downarrow}\right)$ we first introduce a module $\Omega_{a_{mv,i}}$ with $a_{mv,i} \sim a_{evm,i}$. Then, since every resource r_i in $Ext\left(stor^{\downarrow}\right)$ is stored in the scope of Ω_{main} we can adapt ϕ_{main} to first retrieve the resource from memory and then pass it to the respective module by invoking $\Omega_{a_{mvmi}} :: \phi_{store,type\_of(r_i)}$ as already shown before. By reapplying the considerations from before it is then easy to see that this constructions reproduces the set $Ext\left(stor^{\downarrow}\right)$ on the Move side. After the that, we return from the ϕ_{main} producing the action $\texttt{term}\ State^{\downarrow}_{mv}$ also on the Move side for some state $State^{\downarrow}_{mv}$. Given that similarity of halting states is only defined over the sets of externals, internals and transient it is then straight forward to show that $State^{\downarrow}_{mv} \sim_{\Omega^{\dagger},a^{\dagger}} State^{\downarrow}_{evm}$.

B.3.6 Comment on Initial Construction

In our initial construction we assume that we can reconstruct any initial state of the EVM. More precisely, we assume that on the Move site, we can reconstruct the set of externals and internals. However, this is not valid in general. Consider for example a module $\Omega^{\dagger}$ and a type $\tau \in \mathcal{T}_{\Omega^{\dagger}}$. If $\Omega^{\dagger}$ does not provide a function with a return value type τ, then any state that has a resource of type τ in the externals cannot be reconstructed in Move. However, we argue that such states are then also not valid in EVM.

We will omit a formal proof of this claim and instead provide a brief outline of the argument: First note that we assume that our compiler fulfills Definition 13 which says that every contract resulting from our compiler must be initialized with empty transients, externals and internals. Hence for the very first invocation there are particularly no resources in the internals or externals. We claim that such state can be safely reconstructed in Move and matches the initial state of the module. From there it follows then directly by soundness, that every terminating state we can reach in EVM must also be reachable in Move.

C Compiler Model

In this section, we aim to demonstrate that the monitors introduced earlier are sufficient to satisfy Obligations 1–8. For this purpose we assume a locally preserving compiler $[\![\cdot]\!]^{\texttt{move}}_{\texttt{evm}} : \mathbb{Q} \mapsto \mathcal{L}\left[\mathbb{B}^{8}\right]$. However, for our monitors to work, we have to be a bit stricter than that: we assume that $[\![\cdot]\!]^{\texttt{move}}_{\texttt{evm}}$ is written in a way for each modules $\Omega^{\dagger}$ all write accesses to the storage happens either in the monitor logic (as shown later)

or between states $State_{evm}$ and $State'_{evm}$ such that

$$\left[\!\left[\Omega^{\dagger}\right]\!\right]^{\texttt{move}}_{\texttt{evm}} \vdash \texttt{call}\ State_{evm}\texttt{?} :: \texttt{call}\ State'_{evm}\texttt{!}$$

i.e., inside a code segment that corresponds to a function in *Move*.

In what follows we will consider a specific compiler $[\![\cdot]\!]^{\texttt{move}}_{\texttt{evm}}$ that fulfills these properties.

C.1 Monitors

For clarity and convenience, we maintain a high-level description of the monitors' logic and present their semantics within the framework of our previously introduced trace semantics, rather than reasoning at the EVM bytecode level.

C.1.1 Monitor Semantic

We formally describe a monitor $\mathcal{M}$ as a tuple

$$\left(\left(\iota^{+},\psi^{+}\right),L,\left(\iota^{-},\psi^{-}\right)\right)_{F}.$$

Intuitively, monitors restrict the set of valid traces further. Consequently, monitors have a natural formulation in the scope of our trace semantics: The safety properties ι^{+} and ι^{-} denote safety properties that have to hold upon invocation of the contract (*i.e.*, an action of the form $\texttt{call}^{*}\ State_{evm}\texttt{?}$) and upon termination of the contract's execution (*i.e.*, an action of the form $\texttt{ret}^{*}\ State_{evm}\texttt{!}$). As these safety properties reason about the state $State_{evm}$ it is only natural to view there validity in the scope of $State_{evm}$. We will write $State_{evm} \vdash \iota$ to denote that the property ι holds at state $State_{evm}$ without introducing an exact language and semantics for formulating such property.

The pre- and post-execution scripts can be seen as transformation functions $States_{evm} \mapsto States_{evm}$ that map a state $State_{evm}$ to another $State'_{evm}$ which is the result of executing the script. Given a script ψ, we will denote with $img(\psi) \subseteq States_{evm}$ the image set of the function induced by ψ, *i.e.* the set of all states that result in running the script ψ on some state $State_{evm} \in States_{evm}$.

In what follows we will extend our annotated small-step semantics introduced for EVM to also support reasoning over monitor's behavior. Note that monitors are used in parallel, *i.e.* one call to a contract might invoke multiple monitors. However, monitors are not isolated from each other (*i.e.* they share data with each other). It is therefore necessary to describe their behavior in the context of possible other monitors. Also, the order of the monitor application is relevant for the scripts. Thus, our small-step semantic will consist of judgements of the form

$$\mathcal{M}_1,\ldots,\mathcal{M}_n;a^{\dagger} \rhd State_{evm} \xrightarrow{\alpha}_{evm} State'_{evm}$$

with the following intended meaning: *if monitors $\mathcal{M}_1$ to $\mathcal{M}_n$ are in place at the contract stored at $a^{\dagger}$, then the step from $State_{evm}$ to $State'_{evm}$ is possible by emitting action α.*

Our monitors only care about specific points in execution, namely when entering and when leaving the contract. Conveniently, this matches with the points of execution where actions are emitted in our previous annotated small-step semantics. Thus we will base the definition of the extended small-step semantics on the original one.

The fact that monitors do not care about non-action-emitting steps, is covered by the first rule of our small-step semantic:

$$\frac{a^{\dagger} \rhd State_{evm} \xrightarrow{\varepsilon}_{evm} State'_{evm} \qquad State_{evm} = (\mu,\sigma,\iota) :: S \qquad \omega_{\mu,\iota} \neq \texttt{SStore}}{\mathcal{M}_1,\ldots,\mathcal{M}_n;a^{\dagger} \rhd State_{evm} \xrightarrow{\varepsilon}_{evm} State'_{evm}}$$

where $\omega_{\mu,\iota}$ denotes the current operation. As we can see, the opcode `SStore` is excluded from the rule above. This is because for `SStore` a special rule is introduced that prevents the contract from writing to storage location in L that denote protected storage locations. As monitors are allowed to write to that storage location, we assume a function *protected* that given a machine states μ (to detect where we are) and execution environment ι (to detect which contract is executed) returns **true** if we are currently executing monitor logic and **false** otherwise. We then introduce the following rule

$$\frac{a^{\dagger} \rhd State_{evm} \xrightarrow{\varepsilon}_{evm} State'_{evm} \qquad State_{evm} = (\mu,\sigma,\iota) :: S \qquad \omega_{\mu,\iota} = \texttt{SStore} \qquad \mu.st.0 \notin L_1 \cup \cdots \cup L_n \vee protected(\mu,\iota)}{\mathcal{M}_1,\ldots,\mathcal{M}_n;a^{\dagger} \rhd State_{evm} \xrightarrow{\varepsilon}_{evm} State'_{evm}}$$

For each monitor $\mathcal{M}_i$, the safety property ι_i^{+} has to hold when entering the contract, *i.e.* for every action of the form $\texttt{call}^{*}\ State_{evm}\texttt{?}$. We capture this by the following rule:

$$\frac{State'_{evm} := \left(\mu',\sigma',\iota'\right) :: S \qquad a^{\dagger} \rhd State_{evm} \xrightarrow{\texttt{call}^{*}\ State'_{evm}\texttt{?}}_{evm} State'_{evm} \qquad \iota'.d[0:4] \notin F_1 \vee State'_{evm} \vdash \iota_1^{+} \ldots \iota'.d[0:4] \notin F_n \vee State'_{evm} \vdash \iota_n^{+}}{\mathcal{M}_1,\ldots,\mathcal{M}_n;a^{\dagger} \rhd State_{evm} \xrightarrow{\texttt{call}^{*}\ State'_{evm}\texttt{?}}_{evm} State'_{evm}}$$

Now before any actual function is invoked (*i.e.* a piece of code that correspond to a function in Move) the script ψ_i^{+} has to be executed. Conveniently, our trace semantic captures the point when entering a function: actions of the form $\texttt{call}\ State_{evm}\texttt{?}$. Consequently we introduce the following rule:

$$\frac{State'_{evm} := \left(\mu',\sigma',\iota'\right) :: S \qquad a^{\dagger} \rhd State_{evm} \xrightarrow{\texttt{call}\ State'_{evm}\texttt{?}}_{evm} State'_{evm} \qquad State'_{evm} \in img\left(select(1) :: \cdots :: select(n)\right)}{\mathcal{M}_1,\ldots,\mathcal{M}_n;a^{\dagger} \rhd State_{evm} \xrightarrow{\texttt{call}\ State'_{evm}\texttt{?}}_{evm} State'_{evm}}$$

where

$$select(i) := \begin{cases} \varepsilon & \iota'.d[0:4] \notin F_i \\ \psi^{+} & otherwise \end{cases}$$

Whenever we return from a function, the property ι_i^{-} has to hold. In our trace semantic this corresponds to actions of

the form ret $State_{evm}$!. Consequently:

$$\frac{State_{evm} := (\mu,\sigma,\iota) :: S \qquad a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}\ State_{evm}!}_{evm} State'_{evm} \qquad \iota.d[0:4] \notin F_1 \vee State_{evm} \vdash \iota_1^- \ldots \iota.d[0:4] \notin F_1 \vee State_{evm} \vdash \iota_n^-}{\mathcal{M}_1,\ldots,\mathcal{M}_n; a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}\ State_{evm}!}_{evm} State'_{evm}}$$

Finally, before returning from the contract's execution, we must ensure that the script ψ_i^- was executed. However, there is a subtlety in the next rule that is however crucial for correctness of our monitors: Instead of demanding that the ψ_i^- has been executed we demand that the concatenation of ψ_i^+ and ψ_i^- has been executed. This is because some monitors introduce new functions that do not correspond to any Move function. However, in our trace semantic calling such function would not result in an action of the form call $State_{evm}$?. Consequently, the rule for pre-script execution introduced before would never be applied and we hence, cannot enforce execution of the pre-script. Thus our final rule is defined as follows:

$$\frac{\begin{array}{c} State_{evm} := (\mu,\sigma,\iota) :: S \\ a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}^*\ State_{evm}!}_{evm} State'_{evm} \\ \psi^+ = select\left(\psi_1^+\right) :: \cdots :: select\left(\psi_n^+\right) \\ \psi^- = select\left(\psi_n^-\right) :: \cdots :: select\left(\psi_1^-\right) \\ State_{evm} \in img\left(\psi^+ :: \psi^-\right) \end{array}}{\mathcal{M}_1,\ldots,\mathcal{M}_n; a^\dagger \rhd State_{evm} \xrightarrow{\texttt{ret}^*\ State_{evm}!}_{evm} State'_{evm}}$$

We then lift our extended small-step semantics to a large-step semantic and a trace semantic similarly than before.

C.1.2 The Protection Layer

While most Obligations introduced before can be enforced by runtime monitors in a straight-forward way, this is not true for Obligation 8 which enforces that no resource remains transient upon termination. Hence, we will mainly focus on this Obligation and only briefly discuss the others.

First consider the following monitor:

$$\mathcal{M}_{prot\text{-}layer} := \left(\left(\iota_P^+, \psi_P^+\right), \{Tr_{loc}, \texttt{pflag}\}, \left(\iota_P^-, \psi_P^-\right)\right)_{\{\texttt{PROT\_LAYER}\}}$$
$$\iota_P^+ := \texttt{pflag} = \textbf{false} \quad \psi_P^+ := \texttt{pflag} \leftarrow \textbf{true};\ \texttt{call}(\iota.sender)$$
$$\iota_P^- := Tr = \emptyset \quad \psi_P^- := \texttt{pflag} \leftarrow \textbf{false};$$

where Tr is defined as

$$Tr := \{\,(r,a) \mid \exists n \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}, a \in \mathbb{N}_{160}.\ stor_{init} \vdash stores\,(r,\tau,\mathsf{tr\_stor}_\tau\,(a,n)) * \textbf{true}\}$$

the storage variable pflag is an arbitrary storage location and Tr_{loc} is defined as

$$Tr_{loc} := \{\mathsf{tr\_stor}_\tau\,(a,n) \mid \exists \tau \in \mathcal{T}_{\Omega^\dagger}.\ stor \vdash stores\,(r,\tau,\mathsf{tr\_stor}_\tau\,(a,n)) * \textbf{true}\}.$$

With PROT_LAYER we denote a fresh function identifier as described previously. Thus $\mathcal{M}_{prot\text{-}layer}$ implements its own function. Intuitively, the logic introduced by $\mathcal{M}_{prot\text{-}layer}$ first verifies that the storage variable pflag is set to **false**. After that, it sets the variable to **true** and performs a callback to the caller. At the end we execute the check if the set of transients is empty and set pflag back to **false**.

Given a Move function $\phi \in \Phi_{\Omega^\dagger}$ we introduce the following monitor:

$$\mathcal{M}^\phi_{sec} := \left(\left(\texttt{pflag} = \textbf{true}, \varepsilon\right), \{\texttt{pflag}\}, \left(\textbf{true}, \varepsilon\right)\right)_{\{\phi\}}$$

Note that for the filter F we (informally) use the function ϕ itself to denote that $\mathcal{M}^\phi_{sec}$ is executed for each invocation in EVM that corresponds to a call of the function ϕ in Move.

The intuitive behavior of $\mathcal{M}^\phi_{sec}$ is to simply verify that at the beginning the storage variable pflag is set to **true**.

Finally, we additionally introduce the monitors $\mathcal{M}^\phi_{return}$ defined as follows:

$$\mathcal{M}^\phi_{return} := \left(\left(\textbf{true}, \varepsilon\right), \{Tr_{loc}\}, \left(\textbf{true}, Tr \leftarrow Tr \cup R\right)\right)_{\{\phi\}}$$
$$R := \{\,(r_1, sender), \ldots, (r_n, sender)\}$$

where with r_i we denote the $i-$th returned resource. At the point of script execution those are stored on top of stack. Note that by updating the set Tr in this high-level representation of the monitor, we implicitly update the storage in EVM adding the returned resources.

Lemma 10. *In the scope of the monitors $\mathcal{M}_{prot\text{-}layer}$, $\mathcal{M}^\phi_{sec}$ and $\mathcal{M}^\phi_{return}$, Obligation 8 is fulfilled,* i.e.*, for each module Ω and for each trace t such that*

$$\mathcal{M}_{prot\text{-}layer}, \mathcal{M}^\phi_{sec}, \mathcal{M}^\phi_{return}; [\![\Omega]\!]^{move}_{evm} \rhd t$$

the trace t fulfills the trace property induced by Obligation 8.

Proof. First, note that due to the monitors, storage access to the set of transient is restricted. In particular, only the monitor logic of $\mathcal{M}_{prot\text{-}layer}$, $\mathcal{M}^\phi_{sec}$, and $\mathcal{M}^\phi_{return}$ can access this storage area.

Thus the only way of adding resources to Tr is by making a call that corresponds to a function call of $\phi \in \Phi_{\Omega^\dagger}$ in Move. However, such calls first need to pass the pre-condition introduced by $\mathcal{M}^\phi_{sec}$, namely that the storage variable pflag is set to **true**.

Again, pflag is protected and can thus only be manipulated by monitor logic. The only monitor that sets pflag to **true** is $\mathcal{M}_{prot\text{-}layer}$. However, $\mathcal{M}_{prot\text{-}layer}$ sets the protection flag back to **false** upon contract return. Thus, every such call must happen *in the span of* the function induced by $\mathcal{M}_{prot\text{-}layer}$.

More precisely, for each (partial) t of the form

$$t := \texttt{call}\ State_{evm,\phi}? :: \texttt{ret}\ State'_{evm,\phi}!$$

such that

$$\mathcal{M}_{prot\text{-}layer}, \mathcal{M}^\phi_{sec}, \mathcal{M}^\phi_{return}; [\![\Omega]\!]^{\texttt{move}}_{\texttt{evm}} \rhd t$$

there exists a sequence $\left(\alpha_i, State_{evm,i}, State'_{evm,i}\right)_{1\leq i\leq n}$ such that for some i we have

$$\alpha_i\ State_{evm,i} = \texttt{call}\ State_{evm,\phi}?$$
$$\alpha_{i+1}\ State_{evm,i} = \texttt{call}\ State_{evm,\phi}!$$

and it holds that

$$a^\dagger \triangleright State^+_{evm} \xrightarrow{\texttt{call}^*\ State^+_{evm}!} _{evm}\!\!\twoheadrightarrow State'_{evm,1}$$
$$\ldots$$
$$a^\dagger \triangleright State_{evm,n-1} \xrightarrow{\alpha_n\ State_{evm,n}} _{evm}\!\!\twoheadrightarrow State'_{evm,n}$$
$$a^\dagger \triangleright State'_{evm,n} \xrightarrow{\texttt{ret}^*\ State^{+'}_{evm}?} _{evm}\!\!\twoheadrightarrow State^{+'}_{evm}$$

form some $a^\dagger$ where the states $State^+_{evm}$ and $State^{+'}_{evm}$ denote the same invocation of the monitor $\mathcal{M}_{prot\text{-}layer}$, *i.e.*,

$$State^+_{evm} := (\mu, \sigma, \iota) :: S$$
$$State^{+'}_{evm} := (\mu', \sigma', \iota) :: S$$
$$\iota.d[0:4] = \texttt{PROT\_LAYER}$$

By the monitor semantic of $\mathcal{M}_{prot\text{-}layer}$ we know that at $State^{+'}_{evm}$ the post-condition of $\mathcal{M}_{prot\text{-}layer}$ must hold, *i.e.* the set of transients is empty. Since each invocation of a move-related function (which are the only points in execution where transients are added) is wrapped by an execution of $\mathcal{M}_{prot\text{-}layer}$ this property is in particular preserved when (successfully) terminating. □

C.1.3 Preservation Properties

So far, Obligation 2, 3, 4, and 5 (preservation of internals, externals and transient between invocation) trivially hold: storage can only be accessed by the contract itself and by our assumption all storage writes are either inside a function block or inside monitor logic.

Currently, the only non-tracked call (*i.e.* a call that would not show up in the trace) is introduced by the monitor $\mathcal{M}_{prot\text{-}layer}$. However, this one does not allow for moving internals, externals or transient around.

However, in Move resources that correspond to transient or externals can be moved around. To allow this in EVM, we introduce additional monitor function while being careful not to violate any obligation.

First consider the following monitor:

$$\mathcal{M}_{mv} := \left(\left(\iota_3^+, \psi_3^+\right), \{Tr_{loc}, Ext_{loc}, \texttt{pflag}\}, (\textbf{true}, \varepsilon)\right)_{\{\texttt{MV\_TO\_SEL}\}}$$
$$\begin{array}{rcl} \iota_3^+ & := & (caller, par_1) \in (Tr \cup Ext) \wedge \texttt{pflag} = \textbf{true} \\ \psi_3^+ & := & Tr \leftarrow (Tr \setminus \{(caller, par_1)\}) \cup \{(par_2, par_1)\}; \\ & & Ext \leftarrow Ext \setminus \{(caller, par_1)\} \end{array}$$

such that Tr, Tr_{loc}, and pflag is defined as before, Ext is defined as

$$Ext := \{(r,a) \mid \exists n \in \mathbb{N}_{256}, \tau \in \mathcal{T}_{\Omega^\dagger}, a \in \mathbb{N}_{160}.$$
$$stor_{init} \vdash stores(r, \tau, \mathsf{ext\_stor}_\tau(a,n)) * \textbf{true}\}$$

and for Ext_{loc} we define

$$Ext := \{\mathsf{ext\_stor}_\tau(a,n) \mid \exists \tau \in \mathcal{T}_{\Omega^\dagger}.$$
$$stor \vdash stores(r, \tau, \mathsf{ext\_stor}_\tau(a,n)) * \textbf{true}\}.$$

Note that with par_i we denote the parameter passed to the function. Intuitively, $\mathcal{M}_{mv}$ allows the owner of a resource, to move it to another account. While the source of the resource can either be the set of transients or the set of externals it is important to note that the target of the move operation is always the set of transient.

This raises the question if we violate Obligation 8 by introducing $\mathcal{M}_{mv}$. However, note that the newly introduced monitor required pflag to be set to **true**. Consequently, we can show that $\mathcal{M}_{mv}$ can only be called in the span of $\mathcal{M}_{prot\text{-}layer}$ by using the same argument as before.

We can also see that $\mathcal{M}_{mv}$ clearly does not violate Obligation 2 and Obligation 3 as transients and externals can move around but clearly cannot be removed by $\mathcal{M}_{mv}$. Also, the storage locations for transient and externals are protected and hence cannot be accessed elsewhere.

This raises the question of how resources can be moved inside the set of externals. For that we introduce yet another monitor:

$$\mathcal{M}_{store} := \left(\left(\iota_1^+, \psi_1^+\right), \{Tr_{loc}, Ext_{loc}\}, (\textbf{true}, \varepsilon)\right)_{\{\texttt{ST\_EXT\_SEL}\}}$$
$$\begin{array}{rcl} \iota_1^+ & := & (caller, par) \in Tr \\ \psi_1^+ & := & Tr \leftarrow Tr \setminus (caller, par); \quad Ext \leftarrow Ext \cup (caller, par) \end{array}$$

Intuitively, this monitor allows an actor to move a resources from transient to externals. However only the owner of the resource is allowed to call that function.

Obligation 8 is unaffected from adding this rule as we only remove transients. It is also not necessary to enforce the span of the monitor $\mathcal{M}_{prot\text{-}layer}$ here. Also Obligation 2 and Obligation 3 are again not violated as we clearly only move around resources but do not drop them.

By assuming that our compiler only allows storage access within a function block or monitor logic and since non of our monitors tamper with the set of internals, Obligation 4 and Obligation 5 trivially holds.

Note that together with $\mathcal{M}^\phi_{return}$, also Obligation 1 is fulfilled: as monitors do not tamper with internals, and externals are only manipulated in the monitor function but not after returning from a function we only need to make sure that return values are but to the set of transients which is done by $\mathcal{M}^\phi_{return}$.

C.1.4 Concluding

We will omit a deeper analysis for Obligation 7 and Obligation 6. Regarding the first one (reentrancy protection) we note that this can be solved by another monitor in a straight forward way. We thus omit this obligation.

Obligation 6 basically states the correct behavior of a function dispatcher. While this can also be expressed via monitors,

it is not particularly interesting to do so as it basically boils down to an input verification that every passed resource is actually owned by the caller according to the set of transients and externals.

D Move Small Step Semantics

In this section we introduce the Move small-step semantics of our adapted Move model. While existing formalization of Move's small-step semantics express the semantics more *locally* by defining the behavior of each opcodes, our semantics also strives for describing the execution of a sequence of opcodes. The semantics described here is based on the original Move virtual machine and is extracted from the original repository's source code [**?**].

D.1 Basic Notions

We start by introducing basic notions used for describing Move' small-step semantics:

Module We first by start introducing Move modules formally. A modules Ω is a tuple $(id, \Phi, \mathcal{T})$ consisting of an identifier *id* a set of functions Φ and a set of module-defined types $\mathcal{T}$. We leave the exact form of the identifier unspecified. Given a module Ω we often use $\mathcal{T}_\Omega$ and Φ_Ω to denote $\Omega.\tau$ and $\Omega.\Phi$ respectively. The set of all modules is denoted by $\mathbb{\Omega}$.

Functions A function is described by a tuple $(id, code)$ where *id* is the function's identifier and *code* is a sequence of opcodes. We do assume that all of the code is well-formed according to Move's bytecode verifier [1]. For sake of readability, we use $\phi[i]$ as an abbreviation for $\phi.code[i]$ and to express the $i-$th opcode of ϕ. We call *i* the *program counter* in this context. Note that for each function the program counter starts at 0. Regarding the identifier, we again leave the exact form of *id* unspecified. However, in the scope of a modules Ω, the identifier is assumed to be unique.

Account Map The account map A is a mapping $\mathcal{A} \mapsto 2^{\mathbb{\Omega}}$ from an account identifier $a \in \mathcal{A}$ to a set of modules, which we refer to as the *account*. We do require that the modules' identifiers within an single account are unique. The exact form of and account identifier *a* is left unspecified.

Memory The memory is described as a mapping $Loc \mapsto \mathcal{R}$ from a set of location *Loc* to a set of storable resources $\mathcal{R}$. Again we leave the form of elements in *Loc* and $\mathcal{R}$ unspecified.

Move Virtual Machine States Based on that, we introduce the callstack-grammar for Move's virtual machine as follows:

$$
\begin{array}{rlcl}
\text{Move VM state} & State_{mv} & := & (M, G, st, C, A) \\
\text{Memory} & M & : & Loc \mapsto \mathcal{R} \\
\text{Globals} & G & : & \mathcal{A} \times \mathcal{T} \mapsto Loc \\
\text{Operand stack} & st & \in & \mathcal{L}(\mathcal{V}) \\
\text{Call Stacks} & C & := & C_{plain} | ERR(c) :: C_{plain} \\
 & & & \mid HALT :: C_{plain} \\
\text{Plain Call Stacks} & C_{plain} & := & (\Omega, \phi, L, pc) :: C_{plain} \mid \varepsilon \\
\text{Account Map} & A & := & \mathcal{A} \mapsto 2^{\mathbb{\Omega}}
\end{array}
$$

D.2 Small-Step Semantic

Simple Binary Operations We collect the semantics of binary operations (*e.g.* addition) with the following rule. We assume a function *binop* that carries out the operation and a function $overflow$ that returns **true** in case the operation overflows.

$$
\frac{\begin{array}{cc} \phi[pc] \in Op_{bin} & type\_of(a) = type\_of(b) = \tau \\ c = binop_{op,\tau}(a,b) & overflow_{op,\tau}(a,b) = \textbf{false} \end{array}}{\begin{array}{c}(M, G, a :: b :: st, (\Omega, \phi, L, pc) :: C, A) \rightarrow \\ (M, G, c :: st, (\Omega, \phi, L, pc+1) :: C, A)\end{array}} \textbf{ Bin. Op. I}
$$

$$
\frac{\begin{array}{cc} \phi[pc] \in Op_{bin} & type\_of(a) = type\_of(b) = \tau \\ c = binop_{op,\tau}(a,b) & overflow_{op,\tau}(a,b) = \textbf{true} \end{array}}{\begin{array}{c}(M, G, a :: b :: st, (\Omega, \phi, L, pc) :: C, A) \rightarrow \\ (M, G, c :: st, ERR(\texttt{MATH\_ERR}) :: C, A)\end{array}} \textbf{ Bin. Op. II}
$$

Push and Pop

$$
\frac{\phi[pc] = \texttt{Pop}}{\begin{array}{c}(M, G, v :: st, (\Omega, \phi, L, pc) :: C, A) \rightarrow \\ (M, G, st, (\Omega, \phi, L, pc+1) :: C, A)\end{array}} \textbf{ Pop}
$$

For pushing elements on the operand stack, Move offers `LdX` opcodes for

$$X \in \{\texttt{U8}, \texttt{U16}, \texttt{U32}, \texttt{U64}, \texttt{U128}, \texttt{U256}, \textbf{true}, \textbf{false}\}.$$

The semantic of all of these opcodes is collected by the following rule:

$$
\frac{\phi[pc] = \texttt{LdX}\,\langle v \rangle}{\begin{array}{c}(M, G, st, (\Omega, \phi, L, pc) :: C, A) \rightarrow \\ (M, G, i :: st, (\Omega, \phi, L, pc+1) :: C, A)\end{array}} \textbf{ LdX}
$$

Casting For cast operations Move offers `CastUN` opcodes for

$$N \in \{8, 16, 32, 64, 128, 256\}.$$

The semantic of all of these opcodes is collected by the following rules:

$$
\frac{\phi[pc] = \texttt{CastUN} \qquad 0 \leq i < 2^N}{\begin{array}{c}(M, G, i :: st, (\Omega, \phi, L, pc) :: C, A) \rightarrow \\ (M, G, i :: st, (\Omega, \phi, L, pc+1) :: C, A)\end{array}} \textbf{ CastUN I}
$$

$$\frac{\phi[pc] = \texttt{CastUN} \qquad i \geq 2^N}{(M,G,i::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M,G,st,ERR(\texttt{MATH\_ERR})::C,A)} \textbf{ CastUN II}$$

Reference Handling We represent references by a tuple $\mathbf{ref}(k,\overline{f},m)$, where k is either a location ℓ or a local $l \in dom(L)$, $\overline{f}$ is a sequence of *fields* f_i representing an identifier to access a specific field of a resource and m is the mutability of the reference and hence either **mut** or **imm**.

We note that semantically, the mutability of a reference is not relevant as correct usage of mutable an immutable references is checked by the type system. However, we still keep the mutability of references in the semantics.

$$\frac{\phi[pc] = \texttt{MutBorrowField}\langle f\rangle \quad ref = \mathbf{ref}(k,\overline{f},\mathbf{mut}) \qquad ref' = \mathbf{ref}(k,\overline{f}.f,\mathbf{mut})}{(M,G,ref::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M,G,ref'::st,(\Omega,\phi,L,pc+1)::C,A)} \textbf{ Borr. mut}$$

$$\frac{\phi[pc] = \texttt{ImmBorrowField}\langle f\rangle \quad ref = \mathbf{ref}(k,\overline{f},\mathbf{imm}) \qquad ref' = \mathbf{ref}(k,\overline{f}.f,\mathbf{imm})}{(M,G,ref::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M,G,ref'::st,(\Omega,\phi,L,pc+1)::C,A)} \textbf{ Borr. imm}$$

For reading and writing from references, we make use of the following helper functions:

$$isLocal(ref) := \begin{cases} \mathbf{false} & ref = \mathbf{ref}(\ell,\overline{f},m,,,\ell) \in Loc \\ \mathbf{true} & otherwise \end{cases}$$

$$readRef(ref,L,M) := \begin{cases} L(ref.k).\overline{f} & isLocal(ref), \\ M(ref.k).\overline{f} & otherwise. \end{cases}$$

$$writeRef(ref,v,L,M) := \begin{cases} (L(ref.k)[\overline{f} \mapsto v],M) & isLocal(ref), \\ (L,M(ref.k)[\overline{f} \mapsto v]) & otherwise. \end{cases}$$

The semantics of `ReadRef` and `WriteRef` is then defined as follows:

$$\frac{\phi[pc] = \texttt{ReadRef} \qquad v = readRef(ref,L,M)}{(M,G,ref::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M,G,v::st,(\Omega,\phi,L,pc+1)::C,A)} \textbf{ ReadRef}$$

$$\frac{\phi[pc] = \texttt{WriteRef} \quad (L',M') = writeRef(ref,v,L,M)}{(M,G,ref::v::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M',G,st,(\Omega,\phi,L',pc+1)::C,A)} \textbf{ WriteRef}$$

$$\frac{\phi[pc] = \texttt{FreezeRef} \quad ref = \mathbf{ref}(k,\overline{f},\mathbf{mut}) \qquad ref' = \mathbf{ref}(k,\overline{f},\mathbf{imm})}{(M,G,ref::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M,G,ref::st,(\Omega,\phi,L,pc)::C,A)} \textbf{ FreezeRef}$$

Vectors We represent vectors by the tuple $\mathbf{vec}(v_1,\ldots,v_n;n)$ where v_i represents the $i-$th element of the vector and n represents the length of the vector.

$$\frac{\phi[pc] = \texttt{VecPack}\langle\tau,n\rangle \qquad v = \mathbf{vec}((v_0,\ldots,v_{n-1});n)}{(M,G,v_1::\cdots::v_n::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M,G,v::st,(\Omega,\phi,L,pc+1)::C,A)} \textbf{ VecPack}$$

$$\frac{\phi[pc] = \texttt{VecUnpack}\langle\tau,n\rangle \qquad v = \mathbf{vec}((v_0,\ldots,v_{n-1});n)}{(M,G,v::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M,G,v_1::\cdots::v_n::st,(\Omega,\phi,L,pc+1)::C,A)} \textbf{ VecUnp.}$$

$$\frac{\phi[pc] = \texttt{VecLen} \qquad \mathbf{vec}(vs;n) = readRef(ref,L,M)}{(M,G,ref::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M,G,n::st,(\Omega,\phi,L,pc+1)::C,A)} \textbf{ VecLen}$$

$$\frac{\phi[pc] = \texttt{VecPushBack}\langle\tau\rangle \quad \mathbf{vec}(vs;n) = readRef(ref,L,M) \qquad n' = n+1 \quad u = \mathbf{vec}(vs::v;n') \qquad (L',M') = writeRef(ref,u,L,M)}{(M,G,v::ref::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M',G,st,(\Omega,\phi,L',pc+1)::C,A)} \textbf{ V.Push}$$

$$\frac{\phi[pc] = \texttt{VecPopBack}\langle\tau\rangle \quad \mathbf{vec}(vs::v;n) = readRef(ref,L,M) \qquad n>0 \quad u = \mathbf{vec}(vs;n-1) \qquad (L',M') = writeRef(ref,u,L,M)}{(M,G,ref::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M',G,v::st,(\Omega,\phi,L',pc+1)::C,A)} \textbf{ V.Pop}$$

$$\frac{\phi[pc] = \texttt{VecImmBorrow}\langle\tau\rangle \quad ref = \mathbf{ref}(k,\overline{f},m) \qquad \mathbf{vec}(vs;n) = readRef(ref,L,M) \quad i<n \qquad ref' = \mathbf{ref}(k,\overline{f}.i,\mathbf{imm})}{(M,G,i::ref::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M,G,ref'::st,(\Omega,\phi,L,pc+1)::C,A)} \textbf{ V.IBor.}$$

$$\frac{\phi[pc] = \texttt{VecMutBorrow}\langle\tau\rangle \quad i<n \qquad \mathbf{vec}(vs;n) = readRef(ref,L,M) \quad ref = \mathbf{ref}(k,\overline{f},\mathbf{mut}) \qquad ref' = \mathbf{ref}(k,\overline{f}.i,\mathbf{mut})}{(M,G,i::ref::st,(\Omega,\phi,L,pc)::C,A) \rightarrow (M,G,ref'::st,(\Omega,\phi,L,pc+1)::C,A)} \textbf{ V.MBor.}$$

Move allows for swapping two elements inside a vector. For this purpose, we introduce the helper function *vecSwap* that, given a vector an index i and an index j returns a vector were the $i-$th and $j-$th element are swapped. More formally, given a vector $v := \mathbf{vec}((v_0,\ldots,v_{n-1});n)$ and two indices i and j we define *vecSwap* as follows:

$$vecSwap(v,i,j) := \mathbf{vec}\left(x_l::v_{\max(i,j)}::x_m::v_{\min(i,j)}::x_u;n\right)$$

such that

$$x_l := (v_i)_{0\leq i<\min(i,j)} \qquad x_m := (v_i)_{\min(i,j)<i<\max(i,j)} \qquad x_u := (v_i)_{\max(i,j)<i<n}$$

We then define the semantic rule for `VecSwap` as follows:

$$\frac{\begin{array}{c}\phi[pc] = \texttt{VecSwap}\,\langle\tau\rangle \qquad 0 \le i, j < n \\ v = \mathbf{vec}\,((v_0,\ldots,v_n);n) = readRef(ref,L,M) \\ u = vecSwap(v,i,j) \qquad (L',M') = writeRef(ref,u,L,M)\end{array}}{\begin{array}{c}(M,G,i::j::ref::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M',G,st,\,(\Omega,\phi,L',pc+1)::C,A)\end{array}}\ \textbf{V.Swap}$$

Control Flow

$$\frac{\phi[pc] = \texttt{Branch}\,\langle off\rangle}{\begin{array}{c}(M,G,st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,st,\,(\Omega,\phi,L,off)::C,A)\end{array}}\ \textbf{Branch}$$

$$\frac{\phi[pc] = \texttt{BrTrue}\,\langle off\rangle \quad b = \mathbf{true}}{\begin{array}{c}(M,G,b::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,st,\,(\Omega,\phi,L,off)::C,A)\end{array}}\ \textbf{BrTrue I}$$

$$\frac{\phi[pc] = \texttt{BrTrue}\,\langle off\rangle \quad b = \mathbf{false}}{\begin{array}{c}(M,G,b::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,st,\,(\Omega,\phi,L,pc+1)::C,A)\end{array}}\ \textbf{BrTrue II}$$

$$\frac{\phi[pc] = \texttt{BrFalse}\,\langle off\rangle \quad b = \mathbf{false}}{\begin{array}{c}(M,G,b::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,st,\,(\Omega,\phi,L,off)::C,A)\end{array}}\ \textbf{BrFalse I}$$

$$\frac{\phi[pc] = \texttt{BrFalse}\,\langle off\rangle \quad b = \mathbf{true}}{\begin{array}{c}(M,G,b::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,st,\,(\Omega,\phi,L,pc+1)::C,A)\end{array}}\ \textbf{BrFalse II}$$

Global Operations

$$\frac{\begin{array}{c}\phi[pc] = \texttt{MoveTo}\,\langle\tau\rangle \qquad \ell \in Loc \\ \ell \notin dom(M) \qquad a = addr\_of(s) \qquad (a,\tau) \notin dom(G) \\ M' = M[\ell \mapsto r] \qquad G' = G[(a,\tau) \mapsto \ell]\end{array}}{\begin{array}{c}(M,G,r::s::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M',G',st,\,(\Omega,\phi,L,pc+1)::C,A)\end{array}}\ \textbf{MoveTo I}$$

$$\frac{\begin{array}{c}\phi[pc] = \texttt{MoveTo}\,\langle\tau\rangle \qquad \ell \in Loc \\ \ell \notin dom(M) \qquad a = addr\_of(s) \qquad (a,\tau) \in dom(G)\end{array}}{\begin{array}{c}(M,G,r::s::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,st,ERR\,(\texttt{CANNOT\_MOV\_RES})::C,A)\end{array}}\ \textbf{MoveTo II}$$

$$\frac{\begin{array}{c}\phi[pc] = \texttt{MoveFrom}\,\langle\tau\rangle \qquad G(a,\tau) = \ell \\ M(\ell) = r \qquad M' = M \setminus \ell \qquad G' = G \setminus (a,\tau)\end{array}}{\begin{array}{c}(M,G,a::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M',G',r::st,\,(\Omega,\phi,L,pc+1)::C,A)\end{array}}\ \textbf{MoveFrom I}$$

$$\frac{\phi[pc] = \texttt{MoveFrom}\,\langle\tau\rangle \qquad (a,\tau) \notin G}{\begin{array}{c}(M,G,a::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,st,ERR\,(\texttt{CANNOT\_MOV\_RES})::C,A)\end{array}}\ \textbf{MoveFrom II}$$

$$\frac{\begin{array}{c}\phi[pc] = \texttt{MutBorrowGlobal}\,\langle\tau\rangle \\ G(a,\tau) = \ell \qquad M(\ell) = rref = \mathbf{ref}_{global}(\ell,\varepsilon,\mathbf{mut})\end{array}}{\begin{array}{c}(M,G,a::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,ref::st,\,(\Omega,\phi,L,pc+1)::C,A)\end{array}}\ \textbf{MBo.Glob I}$$

$$\frac{\phi[pc] = \texttt{MutBorrowGlobal}\,\langle\tau\rangle \qquad (a,\tau) \notin G}{\begin{array}{c}(M,G,a::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,st,ERR\,(\texttt{CANNOT\_BORROW\_RES})::C,A)\end{array}}\ \textbf{MBo.Glob II}$$

$$\frac{\begin{array}{c}\phi[pc] = \texttt{ImmBorrowGlobal}\,\langle\tau\rangle \\ G(a,\tau) = \ell \qquad M(\ell) = r \qquad ref = \mathbf{ref}_{global}(\ell,\varepsilon,\mathbf{imm})\end{array}}{\begin{array}{c}(M,G,a::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,ref::st,\,(\Omega,\phi,L,pc+1)::C,A)\end{array}}\ \textbf{IBo.Glob I}$$

$$\frac{\phi[pc] = \texttt{MutBorrowGlobal}\,\langle\tau\rangle \qquad (a,\tau) \notin G}{\begin{array}{c}(M,G,a::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,ref::st,\,(\Omega,\phi,L,pc+1)::C,A)\end{array}}\ \textbf{IBo.Glob II}$$

$$\frac{\phi[pc] = \texttt{Exists}\,\langle\tau\rangle \quad (a,\tau) \in dom(G)}{\begin{array}{c}(M,G,a::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,\mathbf{true}::st,\,(\Omega,\phi,L,pc+1)::C,A)\end{array}}\ \textbf{Exists I}$$

$$\frac{\phi[pc] = \texttt{Exists}\,\langle\tau\rangle \quad (a,\tau) \notin dom(G)}{\begin{array}{c}(M,G,a::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,\mathbf{false}::st,\,(\Omega,\phi,L,pc+1)::C,A)\end{array}}\ \textbf{Exists II}$$

Local Operation

$$\frac{\phi[pc] = \texttt{CopyLoc}\,\langle l\rangle \qquad L(l) = v}{\begin{array}{c}(M,G,st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,v::st,\,(\Omega,\phi,L,pc+1)::C,A)\end{array}}\ \textbf{CopyLoc}$$

$$\frac{\phi[pc] = \texttt{MoveLoc}\,\langle l\rangle \qquad L(l) = r \qquad L' = L[l \mapsto \bot]}{\begin{array}{c}(M,G,st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,r::st,\,(\Omega,\phi,L',pc+1)::C,A)\end{array}}\ \textbf{MoveLoc}$$

$$\frac{\phi[pc] = \texttt{StLoc}\,\langle l\rangle \qquad L' = L[l \mapsto r]}{\begin{array}{c}(M,G,r::st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,st,\,(\Omega,\phi,L',pc+1)::C,A)\end{array}}\ \textbf{StLoc}$$

$$\frac{\phi[pc] = \texttt{MutBorrowLoc}\,\langle l\rangle \qquad ref = \mathbf{ref}_{local}(l,\varepsilon,\mathbf{mut})}{\begin{array}{c}(M,G,st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,ref::st,\,(\Omega,\phi,L,pc+1)::C,A)\end{array}}\ \textbf{MBo.Loc}$$

$$\frac{\phi[pc] = \texttt{ImmBorrowLoc}\,\langle l\rangle \qquad ref = \mathbf{ref}_{local}(l,\varepsilon,\mathbf{imm})}{\begin{array}{c}(M,G,st,\,(\Omega,\phi,L,pc)::C,A) \to \\ (M,G,ref::st,\,(\Omega,\phi,L,pc+1)::C,A)\end{array}}\ \textbf{IBo.Loc}$$

Operations on the Callstack

$$\frac{\begin{array}{c}\phi[pc] = \texttt{Call}\left\langle a, id_{\Omega'}, id_{\phi'}\right\rangle \\ \Omega' \in A(a) \qquad \Omega'.id = id_{\Omega'} \qquad \phi' \in \Phi_{\Omega'} \qquad \phi'.id = id_{\phi'} \\ L' = [\phi.form\_par[0] \mapsto st[n-1]] \ldots [\phi.form\_par[n-1] \mapsto st[0]]\end{array}}{\begin{array}{c}(M, G, st, (\Omega, \phi, L, pc) :: C, A) \rightarrow \\ \left(M, G, st, \left(\Omega', \phi', L', 0\right) :: (\Omega, \phi, L, pc+1) :: C, A\right)\end{array}} \textbf{ Call}$$

$$\frac{\phi[pc] = \texttt{Ret}}{(M, G, st, (\Omega, \phi, L, pc) :: C, A) \rightarrow (M, G, st, HALT :: C, A)} \textbf{ Ret I}$$

$$\frac{}{(M, G, st, HALT :: C, A) \rightarrow (M, G, st, C, A)} \textbf{ Ret II}$$

References

[1] Aptos. https://aptosfoundation.org/. [Online; Accessed on January 21, 2025].

[2] Iota. https://www.iota.org/. [Online; Accessed on January 21, 2025].

[3] Starcoin. https://starcoin.org/en/. [Online; Accessed on January 21, 2025].

[4] Sui. https://sui.io/. [Online; Accessed on January 21, 2025].

[5] Top blockchains by total value locked. https://www.coingecko.com/en/chains. [Online; Accessed on January 21, 2025].

[6] Anonymous Authors. Let's move2evm. https://lets-move-to-evm.github.io, January 2025. [Online; Accessed on January 16, 2025].

[7] Various Authors. GitHub - anza-xyz/move: Move compiler targeting llvm supported backends. https://github.com/anza-xyz/move, June 2022. [Online; Accessed on January 7, 2025].

[8] Various Authors. Move-on-evm. https://github.com/move-language/move/tree/main/language/evm, June 2022. [Online; Accessed on January 7, 2025].

[9] Gbadebo Ayoade, Erick Bauman, Latifur Khan, and Kevin W. Hamlen. Smart contract defense through bytecode rewriting. In *IEEE International Conference on Blockchain, Blockchain 2019, Atlanta, GA, USA, July 14-17, 2019*, pages 384–389. IEEE, 2019.

[10] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard–enforcing user requirements on android apps. In *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 19*, pages 543–548. Springer, 2013.

[11] Lorenzo Benetollo, Michele Bugliesi, Silvia Crafa, Sabina Rossi, and Alvise Spanò. Algomove–a move embedding for algorand. In *2023 IEEE International Conference on Blockchain (Blockchain)*, pages 62–67. IEEE, 2023.

[12] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. Move: A language with programmable resources. *Libra Assoc*, page 1, 2019.

[13] Sam Blackshear, David L Dill, Shaz Qadeer, Clark W Barrett, John C Mitchell, Oded Padon, and Yoni Zohar. Resources: A safe language abstraction for money. *arXiv preprint arXiv:2004.05106*, 2020.

[14] Sam Blackshear, John Mitchell, Todd Nowacki, and Shaz Qadeer. The move borrow checker. *arXiv preprint arXiv:2205.05181*, 2022.

[15] Vitalik Buterin. Critical update re: Dao vulnerability. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability, June 2016. [Online; Accessed on December 18, 2024].

[16] Pascal Caversaccio. A historical collection of reentrancy attacks. https://github.com/pcaversaccio/reentrancy-attacks, June 2022. [Online; Accessed on December 18, 2024].

[17] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and assets for safer blockchain programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(3):1–82, 2020.

[18] Monika Di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE international conference on decentralized applications and infrastructures (DAPPCON)*, pages 69–78. IEEE, 2019.

[19] Marcelo d'Amorim and Grigore Roşu. Efficient monitoring of ω-languages. In *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17*, pages 364–378. Springer, 2005.

[20] Ulfar Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, 2004.

[21] Ulfar Erlingsson and Fred B Schneider. Sasi enforcement of security policies: A retrospective. In *Proceedings of the 1999 workshop on New security paradigms*, pages 87–95, 1999.

[22] Ulfar Erlingsson and Fred B Schneider. Irm enforcement of java stack inspection. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 246–255. IEEE, 2000.

[23] Asem Ghaleb and Karthik Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–427, 2020.

[24] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *Principles of Security and Trust: 7th International Conference, POST 2018, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7*, pages 243–269. Springer, 2018.

[25] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. Ethereum smart contract analysis tools: A systematic review. *Ieee Access*, 10:57037–57062, 2022.

[26] Tai D. Nguyen, Long H. Pham, and Jun Sun. SGUARD: towards fixing vulnerable smart contracts automatically. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1215–1229. IEEE, 2021. https://doi.org/10.1109/SP40001.2021.00057.

[27] Russell O'Connor. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 107–120, 2017.

[28] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys (CSUR)*, 51(6):1–36, 2019.

[29] Marco Patrignani and Sam Blackshear. Robust safety for move. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*, pages 308–323. IEEE, 2023.

[30] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Evmpatch: Timely and automated patching of ethereum smart contracts. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1289–1306. USENIX Association, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/rodler.

[31] Clara Schneidewind. *Foundations for the security analysis of distributed blockchain applications*. Wien, 2021.

[32] Clara Schneidewind, Markus Scherer, and Matteo Maffei. The good, the bad and the ugly: Pitfalls and best practices in automated sound static analysis of ethereum smart contracts. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part III 9*, pages 212–231. Springer, 2020.

[33] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. Writing safe smart contracts in flint. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, pages 218–219, 2018.

[34] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with scilla. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[35] Christof Ferreira Torres, Hugo Jonker, and Radu State. Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts. In *25th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2022, Limassol, Cyprus, October 26-28, 2022*, pages 115–128. ACM, 2022. https://doi.org/10.1145/3545948.3545975.

[36] Xiao Liang Yu, Omar I. Al-Bataineh, David Lo, and Abhik Roychoudhury. Smart contract repair. *ACM Trans. Softw. Eng. Methodol.*, 29(4):27:1–27:32, 2020. https://doi.org/10.1145/3402450.

[37] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. SMARTSHIELD: automatic smart contract protection made easy. In Kostas Kontogiannis, Foutse Khomh, Alexander Chatzigeorgiou, Marios-Eleftherios Fokaefs, and Minghui Zhou, editors, *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, pages 23–34. IEEE, 2020. https://doi.org/10.1109/SANER48275.2020.9054825.