Robust Safety for Move

Marco Patrignani
CISPA Helmholtz Center for Information Security

Sam Blackshear
F2 Research

Abstract

A program that maintains key safety properties even when interacting with arbitrary untrusted code is said to enjoy *robust safety*. Proving that a program written in a mainstream language is robustly safe is typically challenging because it requires static verification tools that work precisely even in the presence of language features like dynamic dispatch and shared mutability. The emerging Move programming language was designed to support strong encapsulation and static verification in the service of secure smart contract programming. However, the language design has not been analyzed using a theoretical framework like robust safety.

In this paper, we define robust safety for the Move language and introduce a generic framework for static tools that wish to enforce it. Our framework consists of two abstract components: a program verifier that can prove an invariant holds in a closed-world setting (e.g., the Move Prover [43]), and a novel *encapsulator* that checks if the verifier's result generalizes to an open-world setting. We formalise an escape analysis as an instantiation of the encapsulator and prove that it attains the required security properties.

Finally, we implement our encapsulator as an extension to the Move Prover and use the combination to analyze a representative benchmark set of real-world Move programs. This toolchain certifies >99% of the Move modules we analyze, validating that automatic enforcement of strong security properties like robust safety is practical for Move.

1 Introduction

Writing correct code is difficult. Writing code that maintains key safety properties even when interacting with untrusted code is harder still. Programs that have this property are said to enjoy *robust safety*, which is important in a number of real-world settings such as: operating system kernels running correctly in the presence of buggy or malicious user-space apps; browsers isolating JavaScript programs running on different websites; smart contracts exchanging funds with authorized users while preventing theft from attackers.

There are many techniques for enforcing robust safety: sandboxing [36], process isolation, programming patterns such as object capabilites [32], and specialized hardware [41]. A promising, but less common approach to robust safety is enforcement at the language level. The vision of this approach is that first, the programmer writes code and specifies key safety invariants. Then, the language semantics, in concert with static tools (e.g., type systems or program analyses), ensure that these invariants will hold even when the code links against and interacts with untrusted code.

This approach is clearly appealing due to its lack of runtime overhead and its treatment of security as a first class citizen, but it is less frequently used than alternatives for two reasons. First, real-world languages typically have features that frustrate writing robustly safe code. For example, dynamic dispatch, shared mutability, and reflection are all common language features that provide a broad attack surface for violating safety invariants. Second, most practical languages cannot be easily extended with safety-relevant static tools (e.g., efficient program verifiers) or with expressive, integrated specification languages. Both of these extensions are critical because robust safety is only meaningful with respect to a set of programmer-defined safety invariants that can be verified by a practical tool.

Unlike many existing programming languages, the emerging Move language [8] was designed to support both writing programs that interact safely with untrusted code and static verification. This design is due to Move being used to program secure smart contracts on the Diem blockchain [3]. For example, Move has strong encapsulation primitives and omits unsafe features such as dynamic dispatch that have led to costly *re-entrancy* vulnerabilities in other smart contract languages (e.g., the infamous DAO attack [10]). The language is co-developed with the Move Prover, a verification tool that checks whether Move code complies with invariants written in Move's integrated specification language [43].

Unfortunately, the Prover and the design of Move are not sufficient to ensure robust safety on their own (as we exemplify in Listing 1). One important gap is the absence of a

principled characterisation of what it means for Move programs to be robustly safe. In addition, it is not clear what security properties must be satisfied by the tools used to enforce robust safety for Move programs.

Thus, in this paper, we formalise robust safety for the Move language, define the security properties required of tools that wish to enforce it, and implement some concrete instantiations of these tools, and evaluate them on a representative benchmark set of real-world Move programs. Our evaluation lets us conclude that writing robustly safe Move programs is *practical* and *achievable* for ordinary programmers. This conclusion comes from these contributions:

1. We formalise a parametric framework for defining robust safety on Move modules (i.e., partial programs) of interest, which we call *trusted code*. Defining robust safety on Move modules relies on two tools: a program verifier (such as the existing Move Prover) and an *encapsulator* (which is novel). Intuitively, the verifier checks whether safety invariants of the trusted code hold in a closed world containing only trusted code, while the encapsulator is a static analysis that detects whether the trusted code contains safety leaks that untrusted code can exploit to violate the invariants. We prove that the combination of these two tools is sufficient to enforce robust safety.

   By building on top of the formal semantics for Move [8], we give a precise characterisation of the security properties that verifiers and encapsulators must uphold in order to attain robust safety; we call such verifiers and encapsulators *valid*. Then, we prove that any trusted code verified with a valid verifier and approved by a valid encapsulator is robustly safe: its safety invariants cannot be violated by any Move code the trusted code interacts with.

2. We focus on the role of the encapsulator and formalise a simple intraprocedural escape analysis that we prove to be a valid encapsulator. The analysis overapproximates the set of references pointing to internal state of the trusted code trusted code and flags references that may leak to unstrusted code.

3. We implement the escape analysis and evaluate both its efficiency and precision on a representative set of Move benchmarks from a variety of sources. Our results show that $>99\%$ Move modules pass the analysis while the remaining $<1\%$ are easily identifiable false positives. From this, we conclude automatically enforced robust safety is a practially achievable goal for Move programmers.

The paper proceeds by describing the main features of the Move language and giving a high-level description of robust safety (Section 2). Then it recounts the semantics of Move and formally defines robust safety as well as valid verifiers and valid encapsulators (Section 3). The paper then presents the encapsulator implementation and evaluates it on the aforementioned benchmarks (Section 4). Finally, the paper discusses related work (Section 5) and concludes (Section 6).

For space constraints many formal details (e.g., some semantics rules), auxiliary lemmas and proofs are elided, the interested reader can find them in the appendix. Our implementation is open source as part of the Move Prover tool (see Section 4).

2 Overview

We begin by introducing Move through a running example, focussing on the language features that empower programmers to enforce safety invariants even in the presence of adversarial code (Section 2.1). We defer the reader interested in a general tour of the Move language to the work of Blackshear et al. [8]. We then describe how the example is insecure: it respects an invariant locally, but not in the presence of arbitrary code, i.e., it is not robustly safe (Section 2.2). To recover from this insecurity, we show how our encapsulator analysis flags the vulnerability in the example; addressing this issue would make the code robustly safe (Section 2.3).

2.1 Background: Move Language

This section describes the Move features that are relevant for this paper by relying on the running example in Listing 1. The example contains a Move *module* that implements a custom currency `NextCoin`. Note that we defer presenting the security-relevant details of Listing 1 until Section 2.2.

Modules Each Move module consists of a list of struct type and procedure definitions. A module can import type definitions (e.g., `use 0x1::Signer` on Line 2) and call procedures declared in other modules. The fully-qualified name of a module begins with a 16 byte *account address* where the code for the module is stored (here, we write an account address like `0x1` as shorthand for a 16 byte hexadecimal address padded out with leading 0s). The account address acts as a namespace that distinguishes modules with the same name; e.g., `0x1::NextCoin` and `0x2::NextCoin` are different modules with their own types and procedures.

Structs The module defines two data structures `Coin` and `Info`. A `Coin` represents the currency allocated to users of the module while `Info` record how much of that currency exists in total. Both of these structs can be stored in the persistent global key/value store since they define keys; this is indicated by the `has key` syntax on the declaration.

```
1  module 0x1::NextCoin {
2   use 0x1::Signer;
3
4   struct Coin has key { value: u64 }
5   struct Info has key { total_supply: u64 }
6
7   const ADMIN: address = 0xB055;
8
9   // below is the definition of an invariant
10  spec { invariant: forall c: Coin, global<Info>(ADMIN).total_supply
        = sum(c.value) }
11
12  public fun initialize(account: &signer) {
13   assert(Signer::address_of(account) == ADMIN, 0);
14   move_to<Info>(account, Info { total_supply: 0 })
15  }
16
17  // the next function temporarily violates
18  // and then restores the invariant
19  public fun mint(account: &signer, value: u64): Coin {
20   let addr = Signer::address_of(account);
21   assert(addr == ADMIN, 0);
22   let info = borrow_global_mut<Info>(addr);
23   info.total_supply = info.total_supply + value;
24   // invariant temporarily violated
25   Coin { value } // invariant restored
26  }
27
28  // this function violates the invariant
29  public fun value_mut(coin: &mut Coin): &mut u64 {
30   &mut coin.value // not safe!
31  }
32 }
```

Listing 1: Implementation of a coin asset in Move.

Procedures The code defines an initialization, a safe procedure, and an unsafe procedure, which we now describe.

The `initialize` procedure must be called before any `Coin` is created, and it initializes the `total_supply` of the singleton `Info` value to zero. Here, `signer` is a special type that represents a user authenticated by logic outside of Move (similar to e.g., a Unix UID). Asserting that the `signer`'s address is equal to `ADMIN` ensures that this procedure can only be called by a designated administrator account (`0xB055`, in this case).

Procedure `mint` lets the administrator create new coins of a desired amount (Line 25); this is done after the total amount of coins is updated (Line 23). Like `initialize`, this procedure has access control to ensure that it can only be called by the administrator account (Lines 20 and 21).

Persistent Global Store The global store allows Move programmers to store persistent data (e.g., `Coin` balances) that can only be programmatically read/written by the module that owns it, but is also stored by a public ledger that can be viewed by users running code in other modules.

Each key in the global store consists of a fully-qualified type name (e.g., `0x1::NextCoin::Coin`) and an account address where a value of that type is stored (account addresses store both module code and struct data). Although the global store is shared among all modules, each module has exclusive read/write access to keys that contain its declared types. Thus, only the module that declares a struct type such as `Coin` can:

- Publish a value to global storage via the `move_to<Coin>` instruction (e.g., Line 14);
- Remove a value from global storage via the `move_from<Coin>` instruction;
- Acquire a reference to a value in global storage via the `borrow_global_mut<Coin>` instruction (e.g., Line 22).

Since a module "owns" the global storage entries keyed by its types, it can enforce constraints on this memory. For example, the code in Listing 1 ensures that only the `ADMIN` account address can hold a struct of type `0x1::NextCoin::Info`. It does this by only defining one procedure (`initialize`) that uses `move_to` on an `Info` type and enforcing the precondition that that `move_to` is called on the `ADMIN` address (Line 13).[1] These constraints are unlike invariants (which we describe next) since they require runtime checking. In this case, since parameter `account` is supplied at runtime, the programmer cannot enforce statically that it will always be `ADMIN`, hence the check on Line 13.

Invariants The module contains an invariant to be checked statically on Line 10: the amount stored in `Info` correctly tracks how many `Coins` have been allocated. This is described more in depth in Section 2.2.

Procedure `value_mut` takes a mutable reference to a `Coin` as input (thus the type `&mut`) and returns a mutable reference that points to the `value` field of the coin.

Move **Bytecode Verifier: Safe Type Reuse and Linearity** Although other modules cannot access global storage cells keyed by `0x1::NextCoin::Coin`, they can use this type in their own procedure and struct declarations. For example, another module could expose a `pay` function that accepts a `0x1::NextCoin::Coin` as input or a `Bank` struct with a `balance` field whose type is `0x1::NextCoin::Coin`.

At first glance, allowing sensitive values like `Coins` to flow out of the module that created them might seem dangerous – what stops a malicious client module from creating counterfeit `Coins`, artificially increasing the `value` of a `Coin` it posseses, or copying/destroying existing `Coins`? Fortunately, Move has a bytecode verifier (a type system enforced at the bytecode level, as in the JVM [27] and CLR [30]) that allows module authors to prevent these undesired outcomes. In particular, only the module that declares a struct type `Coin` can:

- Create a value of type `Coin` (e.g., Line 25);
- "Unpack" a value of type `Coin` into its component field(s) (`value`, in this case);
- Acquire a reference to a field of `Coin` via a Rust-style [29] mutable or immutable borrow (e.g., `&mut coin.value` at Line 30).

[1] Note that Move has transactional semantics—any program that fails an assertion or encounters a runtime error (e.g., integer overflow/underflow, `move_to<T>(a)` on an account address `a` that already stores a `T`) will *abort* and have no effect on the global storage.

This allows the module author to enforce invariants on the creation and field values of the structs declared in the module.

The verifier also enforces structs to be *linear* by default [7, 18, 40]. Linearity prevents copying and destruction (e.g., via overwriting the variable that stores the struct or allowing it to go out of scope) outside of the module that declared the struct.[2] Although the bytecode verifier of Move enforces many useful properties such as type safety, memory safety, and resource safety [9], it is not powerful enough to enforce robust safety. We now explain why by describing Move code invariants.

2.2 Invariants and Vulnerability

A safety invariant of the module is described on Line 10: the sum of the `value` fields of all the `Coin` objects in the system must be equal to the `total_value` field of the `Info` object stored at the `ADMIN` address. We refer to this invariant as the "*conservation property*". We want the conservation property to hold for all possible clients of the module (including malicious ones): any violation undermines the integrity of the currency. We now show how the conservation property is established and maintained using the encapsulation features of Move before explaining how procedure `value_mut` allows the property to be violated (despite the module being well-typed according to the verifier).

Establishing the Conservation Property Calling `initialize` sets up the module with the invariant: no `Coin` exists and thus the `total_supply` in `Info` is set to `0`.

After initialization, the `mint` procedure can be invoked to create `Coins`. Note that this procedure temporarily violates the conservation property! The invariant is not required to hold at every program point (which would be overly strict [12]); only at the beginning (precondition) and at the end (postcondition) of every public procedure of the module. And indeed the final line of the procedure restores the invariant by creating and returning a `Coin` with the corresponding `value`.

Violating the Conservation Property For procedures `initialize` and `mint`, the conservation property always holds at the postcondition under the assumption that it holds at the precondition. However, an attacker (Listing 2) can violate the property by leveraging procedure `value_mut`. Note that this procedure does not violate the conservation property on its own, but an attacker can use it to break the property:

```
1 fun attacker(c: &mut Coin) {
2   let value_ref = Coin::value_mut(c);
3   *value_ref = *value_ref + 1000; // violates conservation!
4 }
```

Listing 2: An attacker to the code of Listing 1.

[2] The programmer can choose to override these defaults by declaring a struct with the `copy` (e.g., `struct S has copy`) *ability* to allow copying or the `drop` ability to allow unconditional destruction.

Although our `Coin` example is somewhat artificial (a realistic coin implementation would have no need for a procedure like `value_mut`), it illustrates the difficulty of writing robustly safe code. It is not enough for the module code to establish and maintain key safety invariants internally; it must also ensure that no possible client can violate the invariant.

The way to ensure no client of `Coin` can violate the conservation property is to show that it is robustly safe, so we now describe how to enforce this property in practice.

2.3 Detecting Robust Safety Violations With Encapsulator Analysis

At an intuitive level, leaking references to fields of declared structs is the only way Move programs can fail to be robustly safe. Stated differently: if a Move module establishes its key invariants locally *and* avoids leaking references to structs involved in these invariants, then these invariants also hold globally for all possible clients of the module. Making this statement precise (and true) is the goal of the formalisation of robust safety in Section 3.

We detect leaks of structs involved in programmer-specified safety invariants with an intraprocedural escape analysis. When the analysis begins analzing a procedure, it binds all mutable reference parameters to the abstract value OkRef. Borrowing an invariant-relevent field from OkRef produces the abstract value InternalRef, indicating a pointer into module-internal state. The analysis flags a leak if an InternalRef flows into the return value of the function; such a flag means that sensitive writes to module-internal state may occur outside of the module. Because Move structs cannot store references and the global storage only holds struct values, this is the only way such a leak can occur.

If we apply this analysis to the problematic function `value_mut`, the `coin` parameter is initialized to OkRef. Borrowing the invariant-relevant `value` field (Line 30) consumes the OkRef and produces an InternalRef. This value is subsequently returned by the function, which is flagged by the analysis. None of the other functions return references, so the analysis flags only this vulnerable function. Deleting the function makes the module robustly safe with respect to the conservation property, and the analysis will recognize this.

Is Robust Safety so Simple? Our running example may leave the reader with the impression that it is trival to enforce robust safety: just avoid leaking internal state! We emphasize that this principle is also sufficient to ensure robust safey in other languages (e.g., Solidity, C++). However, languages typically provide many ways to "leak" (e.g., returning references, references stored in data structures, re-entrancy, memory safety violations, . . .), and precisely identifying such leaks with an intraprocedural analysis (or even a much more sophisticated analysis) is not practical. The difference between existing languages and Move is that it possible to state sufficient

conditions for robust safety and design an efficient, local analysis that checks whether these conditions hold. This enables the development of a generic developer tool that checks robust safety, i.e., the escape analysis that we present in Section 4.

Despite this intuitive simplicity, formalising what robust safety means precisely for Move (and what security properties that tools such as the escape analysis must uphold) is non-trivial – that is what the next section discusses.

3 Formal Results: Robust Safety for Move

This section provides a formalisation of the key security property the Move language attains: robust safety. For this, it first provides a brief definition the semantics of the Move language (Section 3.1), as taken from the work of Blackshear et al. [9]. Then, it describes the threat model we consider (Section 3.2): this includes the attacker formalisation, the trace model used to capture security-relevant behaviour, and the invariants that define robust safety. As robust safety is attained by virtue of three tools (the bytecode verifier, the prover, and the encapsulator) this section describes the formal properties such tools must fulfill (Section 3.3). Finally, this section proves that any Move module certified by tools that satisfy these properties is robustly safe (Appendix B.8).

Due to space constraints, this section contains a subset of the formal rules, no auxiliary lemmas, and no proofs; the interested reader can find the full formalisation and proofs in the appendix.

3.1 Move Language Excerpts

Move programs are functions that execute on a stack machine whose peculiarity is the treatment of the storage. Formally, the global store mentioned in Section 2.1 is split into two parts: the memory and the globals [9]. The memory is a first-order store and as such its cells cannot be used to store pointers (which we call locations) to memory cells. Globals are instead used to store pointers to memory cells, but they are indexed differently from the memory. In order to access a global, the code provides an address (a literal) and a type bound to that address (this is akin to the type structs mentioned in Section 2.1). This division simplifies formalising the semantics of moving values on the operand stack. In the Move language, any value can be destructively *moved* (invalidating the storage location that formerly held the value), but only certain values (e.g., integers) can be copied.

Move programs are organised in modules (Ω), which contain lists of functions declarations (P), which in turn contain input and output types as well as their list of instructions ($[i]$). Move programs run on a stack machine whose state (σ) is a tuple $\langle C, M, G, S\rangle$ composed of: the call stack, the memory, the globals and the operand stack. The state of the stack machine also maintains a function table (the module Ω itself) to resolve the instructions comprising the bodies of functions.

The call stack C contains a stack of triples that record which function is executing. Each triple $\langle P, \mathrm{pc}, L\rangle$ contains the name of the function and the program counter (P and pc, which are used to find the current instruction in the lookup table), and a stack of locals (L), which are bindings ($x \mapsto v$) from local variables (x) to arbitrary values (v). Values (v) can be either be locations (ℓ) or storable values (r); the latter can either be ground values (z, which include addresses a) or records (whose id we indicate as s). The memory (M) is a map from locations to storeable values ($\ell \mapsto r$) while the globals (G) map resource identifiers to locations ($\langle a, \rho\rangle \mapsto \ell$) that only contain records. Resource identifiers ($\langle a, \rho\rangle$) are a pair of an address (a) and a type (ρ), the latter is used to identify the function that defined that global and the type of the record the global points to (and for this, technically, ρ contains a module id and a struct id s). The shared operand stack (S) contains all values consumed (and produced) by instructions as well as those passed to (and returned by) functions. Given the presence of records, Move uses paths (p) i.e., lists of field names (f) to traverse nested records and look up or update part of a record. For simplicity we assume all field names are distinct.

3.1.1 Move Operational Semantics

The stack machine has a small-step semantics whose judgement is $\Omega, P, i \vdash \sigma \to \sigma'$ and it is read *"in module Ω, instruction i in function P modifies state σ into σ'"*. This semantics relies two additional kinds of reductions for global and local reductions. The first ones follow this judgement: $P, i \vdash \langle M, G, S\rangle \to_{glob} \langle M', G', S'\rangle$ and they describe the semantics of instructions i in function P that only modify globals (either via the operand stack S or via the memory M). The second ones follow this judgement: $i \vdash \langle M, L, S\rangle \to_{loc} \langle M', L', S'\rangle$ and they describe the semantics of instructions i that only modify locals (again, either via the operand stack or via the memory). The list of Move instructions is in Figure 1, they include calling, returning, branching conditionally and unconditionally, moving a value from memory to the stack (and back), borrowing a global, checking the existence of a global, packing and unpacking a record, moving a value to the local stack (and back), copying it, borrowing it, popping a value, loading a constant, binary operations, reading (and writing) to memory and accessing a record field.

instrs. **Call** $\langle P\rangle$ | **Ret** | **BranchCond** $\langle \mathrm{pc}\rangle$ | **Branch** $\langle \mathrm{pc}\rangle$
global instrs. **MoveTo** $\langle s\rangle$ | **MoveFrom** $\langle s\rangle$ | **BorrowGlobal** $\langle s\rangle$
| **Exists** $\langle s\rangle$ | **Pack** $\langle s\rangle$ | **Unpack** $\langle s\rangle$
local var instrs. **MvLoc** $\langle x\rangle$ | **StLoc** $\langle x\rangle$ | **CpLoc** $\langle \ell\rangle$
| **BorrowLoc** $\langle x\rangle$ | **Pop** | **LoadConst** $\langle a\rangle$ | **Op**
| **ReadRef** | **WriteRef** | **BorrowFld** $\langle f\rangle$

Figure 1: Instructions of the Move language.

Most of the rules are unsurprising and therefore omitted, we only provide the most interesting ones in Figure 2, i.e., those that deal with the moving of resources. Notation-wise, we indicate accessing a map (such as the memory) as $M(\ell)$ and updating its content as $M[\ell \mapsto v]$; we use the same notation for locals, globals and for looking up functions in a module. We indicate the domain of a map M as $\mathtt{dom}(M)$. A list of elements K is denoted with $[K]$, and its length as $\|[K]\|$. We use dot notation to access sub-parts of procedures P, namely $P.\mathtt{mid}$ is the module identifier of the procedure and $P.\mathtt{inty}$ and $P.\mathtt{rety}$ are the lists of inputs and return types of P respectively. Function $\mathtt{instr}(\Omega,\sigma)$ returns the current instruction by looking it up in the codebase Ω given the current function and the pc from the top of the call stack in σ.

$$\frac{L(x) = \ell \qquad \ell \in \mathtt{dom}(M)}{\textbf{MvLoc}\,\langle x\rangle \vdash \langle M,L,S\rangle \rightarrow_{loc} \langle M\setminus\ell, L\setminus x, M(\ell)::S\rangle}\ \text{([MoveLoc])}$$

$$\frac{v \in \mathsf{StorableValue} \qquad \ell \notin \mathtt{dom}(M) \qquad M' = M\setminus L(x) \text{ if } L(x)\in\mathtt{dom}(M) \text{ else } M}{\textbf{StLoc}\,\langle x\rangle \vdash \langle M,L,v::S\rangle \rightarrow_{loc} \left\langle M'_{C\setminus\ell}[\ell\mapsto v], L[x\mapsto\ell], S\right\rangle}\ \text{([StoreLoc])}$$

$$\frac{\rho = \langle P.\mathtt{mid}, s\rangle \qquad G(\langle a,\rho\rangle) = \ell \qquad M(\ell) = v}{P, \textbf{MoveFrom}\,\langle s\rangle \vdash \langle M,G,a::S\rangle \rightarrow_{glob} \langle M\setminus\ell, G\setminus\langle a,s\rangle, v::S\rangle}\ \text{([MoveFrom])}$$

$$\frac{\rho = \langle P.\mathtt{mid}, s\rangle \qquad \langle a,\rho\rangle \notin \mathtt{dom}(G) \qquad \ell\notin\mathtt{dom}(M) \qquad M' = M_{C\setminus\ell}[\ell\mapsto v] \qquad G' = G[\langle a,\rho\rangle\mapsto\ell]}{P, \textbf{MoveTo}\,\langle s\rangle \vdash \langle M,G,a::v::S\rangle \rightarrow_{glob} \langle M',G',S\rangle}\ \text{([MoveTo])}$$

$$\frac{\mathtt{instr}(\Omega,\sigma) = i \qquad i \vdash \langle M,L,S\rangle \rightarrow_{loc} \langle M',L',S'\rangle}{\Omega \vdash \langle\langle P,\mathrm{pc},L\rangle::C,M,G,S\rangle \rightarrow \langle\langle P,\mathrm{pc}+1,L'\rangle::C,M',G,S'\rangle}\ \text{([Step-Loc])}$$

$$\frac{\mathtt{instr}(\Omega,\sigma) = i \qquad P,i \vdash \langle M,G,S\rangle \rightarrow_{glob} \langle M',G',S'\rangle}{\Omega \vdash \langle\langle P,\mathrm{pc},L\rangle::C,M,G,S\rangle \rightarrow \langle\langle P,\mathrm{pc}+1,L\rangle::C,M',G',S'\rangle}\ \text{([Step-Glob])}$$

Figure 2: Semantics of the Move language (excerpts).

Rule [MoveLoc] performs a destructive read of local variable x by removing it from the domain of L and placing its value ($L(x)$) on the stack. Rule [StoreLoc] places the top of the stack in variable x, and that variable in a fresh memory location ℓ, deleting any location that x pointed to from memory. This rule also shows the memory allocator $\mathcal{C}$, which is a set of (fresh) locations that allocation can draw from, for simplicity we often omit $\mathcal{C}$ and report it only when necessary. Rule [MoveFrom] starts from an address (a) and the type ρ of the currently-executing function P to look up a memory location ℓ and then push its content v on the operand stack, removing the memory and global locations just read. Rule [MoveTo] publishes a value v to a fresh memory location ℓ that is itself published to a fresh global $\langle a,\rho\rangle$ whose type is defined by the currently-executing function P. The role of ρ is key here: note that it is not programmer-supplied, but it is computed by the semantics (i.e., by the Move abstract machine) which ensures that any resource being moved belongs to the code that is moving it. This rules out certain attacks (as resources defined in a module cannot be accessed outside it, as mentioned in Section 2.1), but it still leaves the door open for confused deputy attacks, where external code tricks trusted code into insecure behaviour (similar to Listing 2). Finally, Rules [Step-Loc] and [Step-Glob] show how the local and global steps affect the top-level reduction judgements.

3.1.2 Static Semantics

As we mentioned in Section 2 any Move code that is executed must pass through a bytecode verifier [9] to ensure that all Move code is well-typed, meaning that, e.g., operations that require a $\mathbb{N}$ are supplied a $\mathbb{N}$ and values that cannot be copied are not copied but only moved. We indicate a module Ω to be well-typed as: $\vdash \Omega : wt$. In order to typecheck instructions, the verifier uses a stack of local types ($\tilde{L}$) and of operand types ($\tilde{S}$), which are analogous to their semantics counterpart save that instead of tracking values they track types (τ). The typing of Move instructions follows the judgement $\Omega, P, i \vdash \tilde{L}, \tilde{S} \rightarrow \tilde{L}', \tilde{S}'$, which reads *"instruction i (in function P, in module Ω) requires locals typed $\tilde{L}$ and operands typed $\tilde{S}$ and returns locals typed $\tilde{L}'$ and operands typed $\tilde{S}'$"*. As for the semantics, typing is unsurprising and therefore omitted.

3.2 Threat Model

The start of our threat model is the element whose security we are interested in, and that is some Move module of interest that we call the trusted code and that we denote as $\Omega^\dagger$. We now describe what are the attackers to the trusted code (Section 3.2.1) and invariants, i.e., the specific formulation of security properties that must hold on trusted code (Section 3.2.2). We conclude this section by describing the trace model used to formalise the trace semantics capturing the security-relevant behaviour of the trusted code (Section 3.2.3).

3.2.1 Attacker

An attacker is code that is linked against the trusted code so that they call each other's function (and return to each other after said calls). We can thus identify a *boundary* that separates between attacker code and trusted code and that some of the notions described below rely on.

To clearly capture the power of such an attacker (A), we formalise them as pairs consisting of a code environment (Ω) and a main function (P_{main}). We impose minimal constrainst on attackers, namely that they are well-typed (i.e., $\vdash A : wt$) and that they define functions that do not overlap with those defined in the trusted code $\Omega^\dagger$. We call these attackers valid and denote this fact as: $\Omega^\dagger \vdash A : atk$.

Linking some trusted code $\Omega^\dagger$ against attacker A is denoted as $\Omega^\dagger + A$ and it returns a module comprising all functions

defined in both $\Omega^{\dagger}$ and in the module part of A. With a small abuse of notation we use metavariable A for both an attacker and for just its code environment to differentiate it from the code of interest. When an attacker is linked against the trusted code, we assume execution starts from the function P_{main} defined in A. We call that the starting state of the stack machine (i.e., memory, globals and stacks are all empty) and indicate it as $\Omega_0\left(\Omega^{\dagger}+A\right)$.

3.2.2 Invariants

Invariants contain the list of globals that point to memory locations with a logical invariant as well as the list of memory locations with a logical invariant. For each memory location, invariants define a logical condition that must hold for the content of that location (as in Listing 1).

We indicate invariants as ι and leave their formal details abstract to avoid binding our formalisation to a specific implementation. For this reason, we work with invariants axiomatically, via the functions described below. Function $\texttt{domG}(\iota)$ returns the globals for which invariants are defined, i.e., the pairs a,ρ that identify globals for which an invariant is defined. We indicate whether some field f belongs to a global with an invariant as $f \in \iota$. Function $\texttt{cond}(\iota,\ell,v)$ evaluates the condition for location ℓ on value v and returns true (if the condition is satisfied) or false (otherwise).

With these functions we can define whether a memory and a global satisfy some invariant ($M,G \vdash \iota$). This holds if restricting all memory locations to those mapped by a global yields a memory that contains values that satisfy the conditions the invariant imposes on them. We use notation $M|_{\ell}$ to restrict memory M (and similarly for globals and other elements) to the element ℓ, which is in the domain of M.

$$\frac{\begin{array}{c}G_i = G|_{\texttt{domG}(\iota)} \qquad M_i = M|_{G_i} \\ \forall \ell \in \texttt{dom}(M_i).\, \texttt{cond}(\iota,\ell,M_i(\ell)) = true\end{array}}{M,G \vdash \iota}\ \text{(Invariant Satisfaction)}$$

Invariants are defined for a code environment, which can be obtained from ι as follows: $\texttt{codeof}(\iota) = \Omega$. A code environment Ω and an invariant ι are in agreement if the former is the code of the latter. Formally: $\Omega \frown \iota \stackrel{\text{def}}{=} \texttt{codeof}(\iota) = \Omega$.

Dually, given a code environment, we can identify its subset wrt an invariant as follows: $\Omega|_{\iota}$. This operation returns the code environment Ω' that is contained in Ω and that only talks about the code mentioned in ι, without any other code that has no invariant on. This is used to identify the sub-part of a code environment that needs to be encapsulated, as we discuss in Section 4.2.2.

Invariants uphold a key property: none of the types mentioned in their globals (i.e., in $\texttt{domG}(\iota)$) are attacker types, i.e., all of those types are defined in the code of that the invariant refers to.

Property 1 (Invariants are not on Attacker-Typed Globals)**.**

$$\forall \langle a,\rho\rangle \in \texttt{domG}(\iota), \rho \in \texttt{declaredtypes}(\texttt{codeof}(\iota))$$

3.2.3 Trace Model and Trace Semantics

Having defined invariants, we need to collect all security-relevant events produced as computation progresses in order to check whether those invariants hold or not. Choosing when events are produced is crucial in order to assess safety of trusted code and in this work we record events any control is passed from trusted code to the attacker and back. This way the trusted code can internally violate the invariants, but so long as they are reinstated before control is passed to the attacker, no safety violation is detected. This is intuitively ok, as explained in Section 2.2. The only missing bit is that we need to ensure that the attacker does not tamper with our invariants – or that if she does, this will be recorded – and this is discussed later, in Section 3.4.

Formally, observable events (also called actions, α), follow this grammar and they are concatenated in traces ($\overline{\alpha}$).

$$\overline{\alpha} ::= [\,] \mid \overline{\alpha} :: \alpha$$
$$\alpha ::= \texttt{call}\ P\ M,G? \mid \texttt{call}\ P\ M,G! \mid \texttt{ret}\ M,G! \mid \texttt{ret}\ M,G?$$

Actions include calling function P into trusted code, calling function P into attacker code, returning to attacker code and returning to trusted code.[3] Crucially, all actions record elements of the stack machine state that are relevant from a security perspective: the globals G and the memory M. Given an action $\overline{\alpha}$, we indicate its M and G elements as $\texttt{mg}(\overline{\alpha})$. This lets us apply the invariant verification (i.e., Rule Invariant Satisfaction) to the globals and memory sub-part of an action and then to a trace as:

$$\frac{\texttt{mg}(\alpha) \vdash \iota}{\alpha \Vdash \iota}\ \text{(Action-check)} \qquad \frac{\forall \alpha \in \overline{\alpha}.\, \alpha \Vdash \iota}{\overline{\alpha} \Vdash \iota}\ \text{(Trace-check)}$$

Trace Semantics We now define a big-step trace semantics on top of the small-step operational semantics of Section 3.1.1 in order to obtain the traces of some trusted code of interest. The trace semantics is structured on three levels and selected rules are presented in Figure 3. First, there is a single-step, single-labelled semantics that is responsible of generating the single actions, its judgement is $\Omega^{\dagger} \triangleright \Omega,P,i \vdash \sigma \xrightarrow{\alpha} \sigma'$. The trace semantics is defined for whole programs, i.e., for trusted code that is linked against some attacker and then run. However, the trace semantics needs to remember the perspective from which the trace is being generated, i.e., which one is the trusted code of interest. This gets reflected in the

[3] We borrow decorators ? and ! from process calculi literature in order to indicate the "direction" of the action i.e., from attacker to trusted code (?) or back (!) [37].

judgement of the trace semantics which extends the one of the operational semantics with this information (the $\Omega^\dagger$ on the left). Second, there is a big-step, single-labelled semantics that is the reflexive-transitive closure of the previous one, its judgement is $\Omega^\dagger \triangleright \Omega, P \vdash \sigma \xRightarrow{\alpha} \sigma'$. Third, there is the big-step, trace-labelled semantics that concatenates all big-step single-labelled steps into a trace, its judgement is $\Omega^\dagger \triangleright \Omega, P \vdash \sigma \xRightarrow{\overline{\alpha}} \sigma'$. Lastly, in order to decorate the generated actions with ? or !, we rely on function $\Omega^\dagger \vdash C : ?/!/same$. This function analyses the top two elements of the call stack C and tells whether they belong to functions defined by trusted code $\Omega^\dagger$ and attacker (?), attacker and trusted code (!), or by the same entity (*same*).

$$\frac{\begin{array}{c}\texttt{instr}(\Omega,\sigma) = i \quad \Omega \vdash \sigma \to \sigma' \quad \sigma = \langle C,M,G,S\rangle \\ (i \neq \textbf{Call} \text{ and } i \neq \textbf{Ret}) \text{ or} \\ (i = \textbf{Call}\,\langle P_0\rangle \text{ and } \Omega^\dagger \vdash C :: \langle P_0, 0, \varnothing\rangle : same) \text{ or} \\ (i = \textbf{Ret} \text{ and } \Omega^\dagger \vdash C : same)\end{array}}{\Omega^\dagger \triangleright \Omega \vdash \sigma \xrightarrow{[]} \sigma'} \text{(Action-No)}$$

$$\frac{\begin{array}{c}\texttt{instr}(\Omega,\sigma) = \textbf{Call}\,\langle P_0\rangle \quad \Omega \vdash \sigma \to \sigma' \\ \sigma = \langle C,M,G,S\rangle \quad \Omega^\dagger \vdash C :: \langle P_0, 0, \varnothing\rangle : ?\end{array}}{\Omega^\dagger \triangleright \Omega \vdash \sigma \xrightarrow{\texttt{call}\ P_0\ M,G?} \sigma'} \text{(Action-Call)}$$

$$\frac{\begin{array}{c}\texttt{instr}(\Omega,\sigma) = \textbf{Ret} \quad \Omega \vdash \sigma \to \sigma' \\ \sigma = \langle C,M,G,S\rangle \quad \Omega^\dagger \vdash C : !\end{array}}{\Omega^\dagger \triangleright \Omega \vdash \sigma \xrightarrow{\texttt{ret}\ M,G!} \sigma'} \text{(Action-Return)}$$

$$\frac{\Omega^\dagger \triangleright \Omega \vdash \sigma \xrightarrow{[]} \sigma'' \quad \sigma'' = \langle P, \mathrm{pc}, L\rangle :: C,M,G,S \quad \Omega^\dagger \triangleright \Omega \vdash \sigma'' \xrightarrow{\alpha} \sigma'}{\Omega^\dagger \triangleright \Omega \vdash \sigma \xRightarrow{\alpha} \sigma'} \text{(Single)}$$

$$\frac{\Omega^\dagger \triangleright \Omega \vdash \sigma \xRightarrow{\overline{\alpha}} \sigma'' \quad \Omega^\dagger \triangleright \Omega \vdash \sigma'' \xRightarrow{\alpha?} \sigma''' \quad \Omega^\dagger \triangleright \Omega \vdash \sigma''' \xRightarrow{\alpha!} \sigma'}{\Omega^\dagger \triangleright \Omega \vdash \sigma \xRightarrow{\overline{\alpha}::\alpha?::\alpha!} \sigma'} \text{(Trace-Both)}$$

$$\frac{\Omega^\dagger \triangleright \Omega \vdash \sigma \xRightarrow{\overline{\alpha}} \sigma'' \quad \Omega^\dagger \triangleright \Omega \vdash \sigma'' \xRightarrow{\alpha?} \sigma''' \quad \neg(\Omega^\dagger \triangleright \Omega \vdash \sigma''' \xRightarrow{\alpha!} \sigma')}{\Omega^\dagger \triangleright \Omega \vdash \sigma \xRightarrow{\overline{\alpha}::\alpha?} \sigma'} \text{(Trace-Single)}$$

Figure 3: Trace semantics for Move programs (excerpts).

Rule Action-No says that no action is produced if the underlying small-step reduction is caused by an instruction that is not a **Call**, nor a **Ret**, or the jump caused by the **Call** or by the **Ret** does not cross the boundary between trusted code and attacker. Rule Action-Call generates a call action in case of the attacker calls a function defined by the trusted code while Rule Action-Return generates a return action when the trusted code returns to the attacker. Rule Single concatenates a series of empty steps followed by an action as a single action that is then used by Rules Trace-Both and Trace-Single to generate a trace.

Given a trusted code $\Omega^\dagger$ and an attacker A, we indicate the trace $\overline{\alpha}$ of $\Omega^\dagger$ generated according to the rules above, starting from the starting state as:

$$\Omega_0\left(\Omega^\dagger + A\right) \rightsquigarrow \overline{\alpha}$$

3.3 Tools to Attain Robust Safety

Security of the trusted code is attained via three tools: the bytecode verifier ensuring all code is well-typed (as presented in Section 3.1.2), a prover that checks whether invariants hold locally for trusted code (Section 3.3.1) and an encapsulator ensuring trusted code does not leak globals that have an invariant (Section 3.3.2). This section focusses on the two tools whose job is purely security-oriented, as the goal of the already-presented verifier pertains to more general functional correctness.

As mentioned, the prover and the encapsulator verify two different properties on some trusted code $\Omega^\dagger$ to asses whether it respects invariants ι. Given an execution state σ, the prover checks that the globals G and the memory M respect ι, we call this the local property (Rule Weak Property - Inv).

$$\frac{\sigma = \langle C,M,G,S\rangle \quad M,G \vdash \iota}{\Omega^\dagger \models \sigma \propto \iota : local} \text{(Weak Property - Locality)}$$

On the other hand, the encapsulator takes a state and checks that the memory and globals that are reachable from the attacker do not intersect ($\not\frown$) with those with an invariant, we call this the unreachability property (Rule Weak Property - Atk Changes). We rely on judgement $\Omega^\dagger, \sigma \vdash M_a, G_a : attackerpart$ to traverse state σ and extract the parts of globals (G_a) and memory (M_a) that belong to the attacker, i.e., that do not belong to code defined in $\Omega^\dagger$.

$$\frac{\begin{array}{c}\sigma = \langle C,M,G,S\rangle \quad G_i = G|_{\texttt{domG}(\iota)} \quad M_i = M|_{G_i} \\ \Omega^\dagger, \sigma \vdash M_a, G_a : attackerpart \\ G_i \not\frown G_a \qquad \texttt{dom}(M_i) \not\frown \texttt{dom}(M_a)\end{array}}{\Omega^\dagger \models \sigma \propto \iota : unreachable} \text{(Weak Property - Unreachability)}$$

We call these properties weak because them alone are not sufficient to entail security of trusted code. However, a state σ that satisfies *both* properties is strong enough to be secure (Rule Strong Property).

$$\frac{\Omega^\dagger \models \sigma \propto \iota : local \quad \Omega^\dagger \models \sigma \propto \iota : unreachable}{\Omega^\dagger \models \sigma \propto \iota : strong} \text{(Strong Property)}$$

These properties are the key to the robust safety theorem (Theorem 3 later on), as well as to defining the properties that the prover and the encapsulator must uphold.

3.3.1 Prover

The prover is a tool that verifies that some trusted code $\Omega^\dagger$ satisfies invariants ι *locally*. That is, a programmer can run the prover on her code (and its dependencies) before deploying it and linking it against attacker code. In practice [43], the prover processes a module by assuming global invariants specified by the programmer hold at the entry of each public function and ensuring that they continue to hold at the exit. The prover translates both invariants and Move bytecode into Boogie [5], which uses Z3 [14] to prove that the invariants hold or find a counterexample.

The prover can show that invariants hold in a *closed* world containing a fixed set of modules, but we want to ensure that invariants will continue to hold in an *open* world with arbitrary modules that may be written by attackers. Informally, we want trusted code that has been gone through the prover to have this property: if the trusted code code starts executing in some state σ, then when control is given back to the attacker, the memory and globals there respect the invariant. Formally, this is denoted with $\Omega^\dagger \Vdash \iota : \mathit{local}$, as captured by Definition 3 below. Given an execution in trusted code that starts from a state satisfying the strong property and producing a visible action, the ending state satisfies the local property.

Definition 1 (Local Invariant Satisfaction)**.**

$$\Omega^\dagger \Vdash \iota : \mathit{local} \stackrel{\text{def}}{=} \text{ let } \sigma = \langle C, M, G, S\rangle \text{ if } \Omega^\dagger \vDash \sigma \propto \iota : \mathit{strong} \text{ and } \Omega^\dagger \triangleright \Omega, P \vdash \sigma \stackrel{\alpha!}{\Longrightarrow} \sigma' \text{ then } \alpha! \Vdash \iota \text{ and } \Omega^\dagger \vDash \sigma' \propto \iota : \mathit{local}$$

3.3.2 Encapsulator

The encapsulator is a static analysis that verifies that no mutable reference to a global is passed to attacker code.

We first reason about an encapsulator (Ξ) as an abstract entity in order to define what property it must uphold (Definition 4); we discuss a concrete encapsulator that satisfies this property later in this section. We denote the encapsulator analysing the trusted code $\Omega^\dagger$ as: $\Xi(\Omega^\dagger)$. Since we formulate the encapsulator as a static analysis, these are all the parameters it needs, if it were a dynamic analysis we would have to supply runtime states.

Informally, we want trusted code that is encapsulated to have this property: if the trusted code code starts executing in some state σ, then when control is given back to the attacker, she has no access to globals or memory with an invariant. Formally, this is denoted with $\iota \vdash \Omega^\dagger : \Xi$, as captured by Definition 4. Given an execution in trusted code that starts from a state satisfying the strong property, the state when control is passed to the attacker satisfies the unreachability property.

Definition 2 (Encapsulated Code Satisfaction)**.**

$$\iota \vdash \Omega^\dagger : \Xi \stackrel{\text{def}}{=} \text{ let } \sigma = \langle C, M, G, S\rangle \text{ if } \Omega^\dagger \vDash \sigma \propto \iota : \mathit{strong} \text{ and } \Omega^\dagger \triangleright \Omega, P \vdash \sigma \stackrel{\alpha!}{\Longrightarrow} \sigma' \text{ and } \Xi(\Omega^\dagger|_\iota) \text{ then } \Omega^\dagger \vDash \sigma \propto \iota : \mathit{unreachable}$$

Here we restrict the encapsulator to only run on the subset of $\Omega^\dagger$ that is invariant-defined (i.e., $\Omega^\dagger|_\iota$) since it is sometimes the case that only part of the codebase needs to be encapsulated, as we discuss in Section 4.2.2.

A Concrete Encapsulator for Blockchain Move **Code** We now describe a concrete encapsulator, denoted with Ξ_{ea}, that satisfies Definition 4 under the assumption that the Move code used to verify it is deployed on a public blockchain. This assumption is fulfilled in practice, since currently, most Move programs are smart contracts deployed on blockchains. This assumption further defines what attackers are considered in this setting and what security properties are of interest, since all code and data are public on the blockchain. All code being public means that whenever trusted code gets deployed on the blockchain, any existing code is known and therefore it is not attacker code, only code deployed after the trusted code is the attacker. This means that trusted code cannot call attacker code, but it can return to it. All data being public means that data confidentiality is not a security goal, but data integrity is (i.e., we are not interested in hiding how much money there is, but we are interested in nobody getting more money than they should).[4]

Ξ_{ea} is a static intraprocedural escape analysis that formalises the intuition presented in Section 2.3. The analysis abstracts the concrete values bound to local variables and stack locations using a lattice with three abstract values: NonRef, OkRef, InternalRef. We indicate abstract values as $\hat{v}$ and abstract locals (resp. globals) as $\hat{L}$ (resp. $\hat{S}$). The lattice ordering is NonRef $\sqsubseteq$ InternalRef and OkRef $\sqsubseteq$ InternalRef. Intuitively, NonRef represents any non-reference value, OkRef represents a reference that does not point to resource defined in trusted code, and InternalRef represents a reference that *may* point to a resource defined in trusted code. The goal of the analysis is to prevent an InternalRef from "leaking" to a caller of the trusted code via a **Ret**. Since Move records cannot store references, this is the only way such leaks occur.

Applying Ξ_{ea} to a module Ω (still denoted as $\Xi_{ea}(\Omega)$) makes Ξ_{ea} traverse all the functions in the module of interest, and in each function it verifies all instructions that make up their bodies (Figure 4). Formally, the analysis follows this judgement $\Omega, P, \iota, i \vdash \langle \hat{L}, \hat{S}\rangle \rightsquigarrow \langle \hat{L}', \hat{S}'\rangle$, which reads " *under*

[4] We leave considering a different attacker and thus devising an encapsulator that enforces confidentiality of data for future work.

invariant ι*, instruction i (in function P, in module* Ω*) consumes abstract locals* $\hat{L}$ *and abstract globals* $\hat{S}$ *and produces* $\hat{L}'$ *and* $\hat{S}'$*".*

$$\frac{f \in \iota}{\Omega, P, \iota, \textbf{BorrowFld}\,\langle f\rangle \vdash \langle \hat{L}, \hat{v}::\hat{S}\rangle \rightsquigarrow \langle \hat{L}, \mathsf{InternalRef}::\hat{S}\rangle}\ (\Xi_{ea}\text{-BorrowFld-Relevant})$$

$$\frac{f \notin \iota}{\Omega, P, \iota, \textbf{BorrowFld}\,\langle f\rangle \vdash \langle \hat{L}, \hat{v}::\hat{S}\rangle \rightsquigarrow \langle \hat{L}, \hat{v}::\hat{S}\rangle}\ (\Xi_{ea}\text{-BorrowFld-Irrelevant})$$

$$\frac{}{\Omega, P, \iota, \textbf{BorrowGlobal}\,\langle s\rangle \vdash \langle \hat{L}, \hat{v}::\hat{S}\rangle \rightsquigarrow \langle \hat{L}, \mathsf{InternalRef}::\hat{S}\rangle}\ (\Xi_{ea}\text{-BorrowGlobal})$$

$$\frac{}{\Omega, P, \iota, \textbf{BorrowLoc}\,\langle x\rangle \vdash \langle \hat{L}, \hat{S}\rangle \rightsquigarrow \langle \hat{L}, \mathsf{OkRef}::\hat{S}\rangle}\ (\Xi_{ea}\text{-BorrowLoc})$$

$$\frac{\|\Omega(P).\texttt{retY}\| = n \qquad \forall i \in 1..n.\, \hat{v_i} \neq \mathsf{InternalRef}}{\Omega, P, \iota, \textbf{Ret} \vdash \langle \hat{L}, \hat{v_1} :: \hat{v_n}::\hat{S}\rangle \rightsquigarrow \langle \hat{L}, \hat{v_1} :: \hat{v_n}::\hat{S}\rangle}\ (\Xi_{ea}\text{-Return})$$

Figure 4: Ξ_{ea} escape analysis (excerpts).

Rule BorrowFld-InvRelevant states that when borrowing a field that has an invariant on may point to a resource defined in trusted code and thus InternalRef. Rule BorrowFld-InvIrrelevant propagates the abstract values when the field has no invariant on. Rule BorrowGlobal applies to globals, since one such reference should never be leaked, the borrowed global is InternalRef. Rule BorrowLoc, on the other side, applies to locals, which cannot outlive the current function, so any value retrieved this way is OkRef. Crucially, **Ret** cannot return InternalRef (Rule Return). Finally, as mentioned, the analysis is intraprocedural, so we assume all calls return the join of their function inputs.

As mentioned, we have proven that Ξ_{ea} is a valid encapsulator, i.e., it satisfies Definition 4 (as captured by Theorem 4). We describe the implementation of Ξ_{ea} in Section 4.

Theorem 1 (Ξ_{ea} is a Valid Encapsulator)**.**

$$\text{if } \Omega^\dagger \vDash \sigma \propto \iota : \mathit{strong} \text{ and } \Omega^\dagger \triangleright \Omega, P \vdash \sigma \xRightarrow{\alpha!} \sigma'$$
$$\text{and } \Xi_{ea}(\Omega^\dagger|_\iota) \text{ then } \Omega^\dagger \vDash \sigma' \propto \iota : \mathit{unreachable}$$

Intuitively, this holds because code that is encapsulated with Ξ_{ea} cannot leak references to globals with invariants to attacker code. The reason is that the only way to leak those references is through returns, and Rule Return prevents that so long as the reference is InternalRef. To ensure that any reference to a global with invariants that we load in the stack is InternalRef, Ξ_{ea} ensures that any way to load those references tags them as InternalRef. This is exactly what Rules BorrowFld-InvRelevant and BorrowGlobal do.

Technically, the statement of Theorem 4 contains concrete states, yet applying Ξ_{ea} (i.e., $\Xi_{ea}(\Omega^\dagger|_\iota)$) operates on abstract ones. To connect the two, we rely on two functions: $\gamma\left(\hat{L}, \hat{G}, \Omega^\dagger\right)$ and $\texttt{absty}\,(v)$. The first is a concretisation function that returns all possible concrete states whose locals and globals match their abstract counterparts. The latter is an abstraction function used to generate abstract locals and globals by abstracting any value v contained in their concrete counterparts.

3.4 Robust Safety for Move

We now have all the technical setup to state and prove robust safety for trusted code (Theorem 3). After presenting and discussing the theorem, we analyse the robust aspect from a security perspective.

Any trusted code $\Omega^\dagger$ that is verified (in the sense of Section 3.1.2), proved (in the sense of Section 3.3.1) and encapsulated from Ξ (in the sense of Section 3.3.2), can interact with *any* attacker A and its invariants ι cannot be violated. This means that the trusted code is robustly safe, and it is indicated as $\triangleright_{RS}\ \Omega^\dagger : \iota, \Xi$.

Theorem 2 (Move Modules are Robustly-Safe)**.**

$$\begin{aligned}\triangleright_{RS}\ \Omega^\dagger : \iota, \Xi \stackrel{\text{def}}{=} \forall A.\ &\text{if } \Omega^\dagger \vdash A : atk \text{ and } \Omega^\dagger \frown \iota \\ &\text{and } \vdash \Omega^\dagger : wt \text{ and } \Omega^\dagger \Vdash \iota : local \text{ and } \iota \vdash \Omega^\dagger : \Xi \\ &\text{and } \Omega_0\left(\Omega^\dagger + A\right) \rightsquigarrow \overline{\alpha} \text{ then } \overline{\alpha} \Vdash \iota\end{aligned}$$

The first premise of the theorem ensures that only valid attackers are considered while the second ensures that the invariants are specified for the trusted code. The third, fourth and fifth premise ensure that the trusted code is verified, proved and encapsulated by Ξ. Note that the result is general, no matter what encapsulator is used, so long as that encapsulator is valid in the sense of Definition 4. The final premise introduces the trace yielded by the interaction of the trusted code with the attacker and the conclusion of the theorem confirms that the trace does not violate the invariants.

Robustness What makes Theorem 3 relevant for security is the universal quantification over attackers A, since that differentiates *robust* safety from closed-world safety. That universal quantification ensures that we are considering *arbitrary* code as attacker, so that includes e.g., the code of Listing 2. Crucially, that code needs not be present at verification time. This is particularly relevant for blockchain-deployed Move code, since attacker code is written (and published) *temporally after* the trusted code. Closed-world safety requires the entire codebase for verification, which is not possible for an evolving system such as a public blockchain.

4 Evaluation

In this section, we present an implementation of the escape analysis Ξ_{ea} described in Section 3.3.2 (Section 4.1). Then,

we measure its performance on a large set of Move benchmarks (Section 4.2). We evaluate the analysis according to two criteria:

Performance We claim the escape analysis is fast: it adds negligible overhead over the companion verifier (Section 4.2.1);

Precision We claim the escape analysis is precise: it rarely flags Move code that is robustly safe w.r.t its safety invariants (Section 4.2.2).

4.1 Implementation

We have implemented the escape analysis in approximately 300 lines of Rust code on top of the Move Prover analysis framework.[5] The framework has libraries for parsing Move bytecode, control-flow graph construction, and fixed point computation that are not included in the total above.

We use Move Prover specification language[6] as the invariant language ι. This specification language lets programmers write source code invariants similar to our example in Section 2.2. The invariants are converted to SMT and checked by the Move Prover against the compiled Move bytecode.

Unlike our minimalistic formalism in Section 3, the Move bytecode languages distinguishes between mutable (`&mut T`) and immutable (`&T`) references. A mutable reference can be either written or read, whereas an immutable reference can only be written. In the public blockchain Move code we consider in our evaluation, attacker-controlled reads are not concerning because the entire blockchain state is world-readable by external users for auditability. Thus, our implementation only flags functions that may return a *mutable* reference to a field involved in a prover `spec` for the module under analysis. If the module does not have any invariant, we conservatively flag all such functions.

4.2 Benchmarks

We ran our analysis on the ten benchmarks shown in Figure 5. The benchmarks fall roughly into three categories: blockchain management logic implemented in Move (starcoin, diem, bridge), utility libraries (taohe, stdlib), and applications (mai, blackhole, alma, starswap, meteor). All of the benchmarks contains some Move Prover specs, though we note that not all modules have specs and the density of specification varies across benchmarks. We feel that our benchmark set contains a substantial and representative sample of production Move code, particularly because Move is a young language that is only beginning to gain traction.

[5] Our escape analysis is available at: https://github.com/diem/diem/blob/03c30e1/language/move-prover/bytecode/src/escape_analysis.rs

[6] https://github.com/diem/diem/blob/03c30e1/language/move-prover/doc/user/spec-lang.md

4.2.1 Evaluating Performance

The results in Figure 5 support our claim that the analysis is fast; it takes well under a second on all benchmarks and under 10ms on most benchmarks. As we can see, this time is a tiny fraction of the time taken to run the prover (which is several orders of magnitude slower) on each benchmark.

Thus, we can use the escape analysis to strengthen the prover's closed-world guarantees to open-world ones with no user-visible performance degradation. In the future, we plan to do this by incorporating the escape analysis into the prover's pipeline of pre-analyses (e.g., liveness anaysis, invariant instrumentation). This will improve performance of the escape analysis even more by sharing the steps of parsing bytecodes and building control-flow graphs with the other analysies in the pipeline. Anecdotally, these steps take roughly half of the escape analysis running time.

4.2.2 Evaluating Precision

The results also support our claim that the analysis is precise. Only three functions (0.1% of the total analyzed) in three distict modules (0.9% of the total analyzed) were flagged as potentially containing robust safety violations.

We manually investigated each finding to determine whether it indicated a genuine robust safety issue. In all three cases, the function *does* leak a mutable reference to module-internal state, but the reference cannot point into memory used by the module's invariants (and thus, all three are false positives).

The following code captures the essence of two reports from `starcoin` and `stdlib` (which both contain variants of the `Option` module).

```
module 0x1::OptionVariant {
 struct Option<T> { v: vector<T> }

 // Typically, Options are defined as None | Some (x)
 // Move does not have sum types, so encode None as an
 // empty vector and Some(x) as a vector of length 1 containing x
 spec Option { invariant len(v) ≤ 1; }

 // False positive flagged by analysis as unsafe, but safe
 public fun get_mut<T>(t: &mut Option<T>): &mut T {
  Vector::borrow_mut(&mut t.v, 0)
 }
}
```

In this code, the `get_mut` function does indeed leak an internal reference, but this is intentional–`Option` is a collection intended to be instantiated by clients who need to mutate the contents of the `Option` in-place using this function. The analysis sees that the invariant contains the field `v` and conservatively reports leaks not only of `v`, but also of references that extend from `v` (`&v[0]`, in this case). We note that it *would* be a robust safety violation to leak a reference to `Option.v`, since an attacker could use this reference to violate the invariant `len(v)`$\leq$ `1` (e.g., by adding extra elements to the vector).

The third case (from `starcoin`) is somewhat similar: a module implementing a collection type leaks a reference to

Bench	Mod	Fun	Rec	Instr	Err	$\mathbf{T}_p$	$\mathbf{T}_e$
starcoin	60	431	88	8243	2	3178	10
diem	13	102	19	1830	0	1651	1
mai	45	411	77	7881	0	4209	12
bridge	36	352	85	8060	0	2428	8
blackhole	36	324	72	6030	0	2289	7
alma	35	333	67	6318	0	2102	8
starswap	33	335	67	6617	0	14993	7
meteor	32	323	69	5981	0	1641	7
taohe	11	40	7	305	0	1022	1
stdlib	9	66	5	933	1	1151	1
Total	310	2717	556	52198	3	34664	62

Figure 5: Checking for robust safety with the escape analysis encapsulator. The **Mod**, **Fun**, **Rec**, and **Instr** columns show the number of modules, declared functions, declared record types, and bytecode instructions in each project and its dependencies. The **Err** column shows the number of functions flagged by the escape analysis. The $\mathbf{T}_p$ and $\mathbf{T}_e$ columns show the time taken to run the ***p***rover and ***e***scape analysis in milliseconds on a 2.4 GHz Intel Core i9 laptop with 64GB RAM.

its internal vector, but does so intentionally to allow client modules to add elements to the vector.

```
module 0x1::OwnedVector {
 struct OwnedVec<T> { v: vector<T>, owner: address }

 // flagged by analysis
 public fun get_mut<T>(c: &mut OwnedVec<T>, i: u64): &mut T {
  &mut c.v
 }
}
```

This module does not contain any invariant, so the analysis conservatively flags all leaks of internal references.

Discussion These examples demonstrate an interesting and perhaps counterintuitive point–although encapsulation is generally a good idea, it is not desirable to fully encapsulate *all* modules. Modules like `OptionVariant` and `OwnedVector` are utility modules that are intended to be specialized by clients who need the flexibility to write the internal state of these modules. For example, clients of `OwnedVector` would not be able to add/remove elements from the vector without the `get_mut` function. Thus, although it is tempting to suggest integrating the escape analysis into Move's bytecode verifier (and thus make *all* Move code robustly safe by construction), there is evidence that this would remove expressivity used by real Move programmers.

We believe it would be possible to eliminate the false positive in the `OptionVariant` example by using a more sophisticated abstract domain that tracks the set of *access paths* [24] associated with each reference. The analysis could compare the leaked access paths to the access paths mentioned in specifications and only complain if there is an overlap between the paths *or* their possible suffixes. For example, the analysis could determine that the `get_mut` function leaks the path `Option.v[0]`, but the specification only mentions the incomparable path `Option.v.length`. The analysis would also need to flag a leak of a path like `Option.v[0]` if the specficiation mentions a prefix of the path (e.g., `invariant v == vec[1,2]`).

However, this analysis would be somewhat more complex than our straightforward three-value abstract domain; e.g., we would need to introduce a widening operator [13] because the access path domain is not finite height. Furthermore, our results suggest that the precision gain from this improvement would be fairly small because the existing analysis is already quite precise in practice.

4.3 Security Conclusions

Thus, *all* of the Move modules we looked are robustly safe w.r.t their specified invariants, and the Move Prover augmented with our escape analysis can automatically prove this for >99% of the modules. This indicates that language-supported robust safety is indeed a practically achievable goal for Move programmers.

5 Related Work

Smart Contract Languages Ensuring that key safety invariants hold even in the presence of attackers is a challenging and important task for all smart contract programmers. The Solidity [16] source language and its executable Ethereum Virtual Machine (EVM) [42] bytecode language are the most popular smart contract languages and have been studied the most extensively with security in mind .

The primary barrier to writing encapsulated code in these languages is dynamic dispatch. When the target of a callback is determined by the contract `C`'s caller (which is common, e.g., every payment operation fits this pattern), the contract author cannot know statically how it will change the global state. This is particularly pernicious when the callback supplied by the caller is *re-entrant*–that is, the callback invokes one or more functions from `C`. Attackers can leverage this to change the state of `C` in ways that the author did not anticipate. and/or to observe (and exploit, by injecting code via dynamic dispatch) the interval when a key safety invariant is violated. For example, in the DAO [10] attack, the vulnerable contract made a dynamic dispatch call while a key conservation invariant is violated, which the attacker leveraged to steal funds from the contract.

Many approaches to mitigating re-entrancy have been proposed, including design patterns [11], dynamic analysis [21],

and static analysis [2]. Although absence of re-entrancy facilitates proving robust safety, we are not aware of any work that attempts to define robust safety of EVM code, or any tools that can prove EVM code robustly safe.

The problem of ensuring robust safety is quite different in Move and the EVM. While Move does not have dynamic dispatch (and thus also does not have re-entrancy), it does have mutable references that can escape from the module that created them. In the EVM, references are represented as indexes into a sequential memory that is only accessible by a single contract, so they cannot escape. We note that precisely and efficiently verifying the absence of re-entrancy for EVM code is challenging, whereas our escape analysis for verifying the absence of leaked mutable references is precise, efficient, and relatively straightforward.

Scilla [38] is a newer smart contract language that shares some design goals with Move. Scilla restricts dynamic dispatch by requiring a dynamic function call to be the last instruction in a procedure, which largely mitigates the re-entrancy issues afflicting the EVM. Scilla was also designed to support automated static analysis; its toolchain includes an abstract interpretation framework that supports both built-in (e.g., determining where monetary values can flow) and user-defined analyses.

Language Design for Isolation Language design to support safe interaction with untrusted code is not unique to smart contract languages or Move. Typed assembly language [33] and capability machines like CHERI [41] are low-level approaches to isolating memory from untrusted code running in the same process. The Singularity OS project [26] used the type system of Sing# (a variant of C#) to enforce strong ownership of memory that crosses trust boundaries. The Joe-E [31] language defines a secure subset of Java to enable capability-based programming patterns.

WASM [23] is designed to isolate untrusted applications from the trusted host system. Recent work on WASM has studied a variety of mechanisms [15, 23] (e.g., static and dynamic checks) to enhance the system with the ability to isolate untrusted applications from each other as well as from the host.

Broadly speaking, an important difference between these previous systems and Move is that they do not satisfy key requirements for smart contract programming such as determinism, metered execution, and first-class currency. In addition, these systems are typically concerned with low-level isolation to ensure generic safety properties (such as memory safety) rather than enforcing application-specific, programmer-specified properties like the those specified/verified using the Move Prover toolchain.

Robust Safety The robust safety property originated in the context of modular model checking [22] and has then been widely applied to reason about security protocols that interact with adversaries [1, 4, 6, 17, 20, 28, 35]. In this setting, security protocols are written in concurrent languages (often process calculi) and given a type system that enforces robust safety and therefore ensures safe interaction with untyped adversaries. The type system of these works is analogous to the encapsulator of this paper: it is a static analysis whose goal is to prevent leaks. In order to model the security invariants, some of these languages have explicit assertions, which are proven to never fail because of robust safety.

Swasey et al. [39], instead, use robust safety to verify object capability patterns, a programming pattern that enables programmers to protect the private state of their objects from corruption by untrusted code [32]. Their language is richer than Move (and thus not amenable for safe smart contract programming) and lets programmers define custom assertions, which robust safety ensures to never fail. To ensure this, their code is verified with a powerful mechanism built on top of the Iris logic [25]. Their (static) verification step is analogous to our encapsulator but it relies on user-defined logical assertions to describe invariants that are more complex than Move assertions, as the underlying language is also more complex.

Sammler et al. [36] use robust safety to demonstrate the end-to-end security property of sandboxing. Sandboxing is a common technique that allows trusted and untrusted components to interact safely [19, 34]. This work defines invariants outside of the language, as a system call policy, and robust safety means that any program execution respects the policy. To enforce robust safety, they rely on a type system: any well-typed program can be linked with untyped code and the resulting program is robustly-safe.

6 Conclusion

We have formalised robust safety for the Move language and gave a precise characterisation of the security properties needed of the tools used to attain it in practice. One of these tools is an encapsulator, which ensures no sensitive references are leaked to attacker code. We have also implemented a valid encapsulator and evaluated its precision and performance on a representative set of Move benchmarks. Our evaluation confirms that the enscapsulator can augment existing tools like the Move prover to enable practical enforcement of robust safety for Move programmers.

Acknowledgements This work was partially supported: by the Office of Naval Research for support through grant N00014-18-1-2620, Accountable Protocol Customization, by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), by Novi with a research gift (number 70000077554).

A The Move Language

This section contains the formalisation we borrow (and extend) from Blackshear et al. [9].

A.1 Syntax

We rely on a series of notions:

- Code environments Ω map module ids m to modules
- Modules are pairs of maps: struct ids to struct definitions and procedure id to procedure definitions
- Memories M are maps from locations ℓ to storable values, which can be addresses, tagged records, bools or nats
- Operand stacks S are lists of values v
- Local stacks L are lists of mappings from variables x to locations or references
- Call stacks C are lists of call stack frames
- Global storages G map resource ids to locations ℓ
- Program states σ are tuples of a call stack, a memory, a global storage and an operand stack: $\langle C, M, G, S \rangle$

Instructions Set

local var instructions	**MvLoc** $\langle x \rangle$ | **CpLoc** $\langle \ell \rangle$ | **StLoc** $\langle x \rangle$ | **BorrowLoc** $\langle x \rangle$
ref instructions	**ReadRef** | **WriteRef** | **BorrowFld** $\langle f \rangle$
record instructions	**Pack** $\langle s \rangle$ | **Unpack** $\langle s \rangle$
global instructions	**MoveTo** $\langle s \rangle$ | **MoveFrom** $\langle s \rangle$ | **BorrowGlobal** $\langle s \rangle$ | **Exists** $\langle s \rangle$
stack instructions	**Pop** | **LoadConst** $\langle v \rangle$ | **Op**
procedure instructions	**Call** $\langle p \rangle$ | **Ret** | **Branch** $\langle \text{p}c \rangle$ | **BranchCond** $\langle \text{p}c \rangle$

A.2 Static Semantics

This section reworks the notation of the work by Sam to my own liking.

We have these notions:

- A stack of operand types $\widehat{S}$, which is a list of types τ
- A map of local types $\widehat{L}$, which maps variables x to types τ

We indicate that a code environment, a stack of operand types and a map of local types typecheck a state as follows:

$$\frac{\begin{array}{c}\text{(Well-typedness of states)} \\ \Omega, \widehat{L} \vdash \langle C, M \rangle : wtcs \\ \forall i \in 0..\texttt{len}\left(\widehat{S}\right). \Omega, M \vdash S(i) : \widehat{S(i)} \\ \forall \langle \alpha, \tau \rangle \in \texttt{dom}(G). \Omega, M \vdash M(G(\langle \alpha, \tau \rangle)) : \tau\end{array}}{\Omega, \widehat{S}, \widehat{L} \vdash \langle C, M, G, S \rangle : wt}$$

We indicate that a code environment and a map of local types typecheck a call stack and a memory as follows:

$$\frac{\text{(Well-typedness of callstacks - base)}}{\Omega, \widehat{[]} \vdash \langle [], M \rangle : wtvals}$$

$$\frac{\text{(Well-typedness of callstacks - ind)} \qquad \texttt{dom}\left(\widehat{L}\right) = \texttt{dom}(L) \qquad \forall x \in \texttt{dom}(L). \Omega, M \vdash L(x) : \widehat{L(x)}}{\Omega, \widehat{L} \vdash \langle \langle P, pc, L \rangle :: C, M \rangle : wtcs}$$

We indicate that a code environment and a memory typecheck a value at a type as follows:

$$\frac{\overset{()}{\text{Def 2.2 in Blackshear et al. [9]}}}{\Omega, M \vdash v : \tau}$$

A.2.1 Whole-Program Typing

We indicate that a procedure P is well-typed in a module environment Ω as:

$$\frac{\overset{()}{\text{see Blackshear et al. [9]}}}{\Omega \vdash P : wt}$$

The role of P is to indicate the `main` procedure that begins execution of a transaction. The procedure may call procedures of other modules in Ω and/or perform local computation.

A.3 Dynamic Semantics

This section reworks the notation of the work by Blackshear et al. [9] to our own liking and it makes two important changes:

- It adds canary values to stacks to help identify what stack is from the component (see later for what we mean) and what is attacker stack.

  Formally:

  $$S \in \text{OpStk} = S \uplus K$$
  $$K \in \text{Canary} = \text{startfun P}$$

  A canary K tells that the next values on the stack belong to function P.

- It changes the call rule to take a single step.

  The work by Blackshear et al. [9] has a big-step reduction for the call to simplify proofs.

- It changes the call and return rule to respectively push a canary on the stack and pop it.

  The operand stack is ensured to be split between portions accessible only to the function using that portion, canaries help us understand where the boundaries of each portion lie.

The semantic changes are in Appendix A.3.3.

A.3.1 Local Semantics

We have a notion of local state $\langle M, L, S\rangle$ that keeps track of a memory M, a list of local variables L and an operand stack S. The local semantics tells how an instruction i makes a local state evolve into the next local state and so it follows this judgement:

$$i \vdash \langle M, L, S\rangle \rightarrow_{loc} \langle M', L', S'\rangle$$

Memories Memory locations ℓ come from an infinite, denumerable set of abstract locations $\mathcal{C}$. The memory M is parametrised by the set $\mathcal{C}$ in order to know where a fresh location should come from, we write that as $M_{\mathcal{C}}$. For simplicity, we omit the $\mathcal{C}$ unless when it is necessary, and simply write M for $M_{\mathcal{C}}$.

$$\frac{\overset{\text{([MoveLoc])}}{L(x) = \ell \qquad \ell \in \texttt{dom}(M)}}{\textbf{MvLoc}\,\langle x\rangle \vdash \langle M, L, S\rangle \rightarrow_{loc} \langle M \setminus \ell, L \setminus x, M(\ell)::S\rangle} \qquad \frac{\overset{\text{([MoveLocRef])}}{L(x) = \langle \ell, p\rangle}}{\textbf{MvLoc}\,\langle x\rangle \vdash \langle M, L, S\rangle \rightarrow_{loc} \langle M, L \setminus x, L(x)::S\rangle}$$

$$\frac{\overset{\text{([CopyLoc])}}{L(x) = \ell \qquad \ell \in \texttt{dom}(M)}}{\textbf{CpLoc}\,\langle x\rangle \vdash \langle M, L, S\rangle \rightarrow_{loc} \langle M, L, M(\ell)::S\rangle} \qquad \frac{\overset{\text{([CopyLocRef])}}{v = L(x) = \langle \ell, p\rangle}}{\textbf{CpLoc}\,\langle x\rangle \vdash \langle M, L, S\rangle \rightarrow_{loc} \langle M, L, v::S\rangle}$$

$$\frac{v \in \mathsf{StorableValue} \qquad \ell \notin \mathtt{dom}(M) \qquad M' = M \setminus L(x) \text{ if } L(x) \in \mathtt{dom}(M) \text{ else } M}{\mathbf{StLoc}\,\langle x\rangle \vdash \langle M, L, v::S\rangle \to_{loc} \left\langle M'_{C\setminus\ell}\left[\ell \mapsto v\right], L\left[x \mapsto \ell\right], S\right\rangle}\ ([\text{StoreLoc}])$$

$$\frac{v \in \mathsf{Reference}}{\mathbf{StLoc}\,\langle x\rangle \vdash \langle M, L, v::S\rangle \to_{loc} \langle M, L\left[x \mapsto v\right], S\rangle}\ ([\text{StoreLoc-Ref}]) \qquad \frac{L(x) = \ell}{\mathbf{BorrowLoc}\,\langle x\rangle \vdash \langle M, L, S\rangle \to_{loc} \langle M, L, \langle \ell, []\rangle::S\rangle}\ ([\text{BorrowLoc}])$$

$$\frac{v = \langle \ell, p\rangle \qquad \ell \in \mathtt{dom}(M) \qquad M(\ell)[p] = \left\langle \left\{(f, v_f), \cdots\right\}, t\right\rangle}{\mathbf{BorrowFld}\,\langle f\rangle \vdash \langle M, L, v::S\rangle \to_{loc} \langle M, L, \langle \ell, p::f\rangle::S\rangle}\ ([\text{Borrowfld}])$$

$$\frac{v = \langle \ell, p\rangle \qquad \ell \in \mathtt{dom}(M) \qquad M(\ell)[p] = v_p}{\mathbf{ReadRef} \vdash \langle M, L, v::S\rangle \to_{loc} \langle M, L, v_p::S\rangle}\ ([\text{ReadRef}])$$

$$\frac{v_2 = \langle \ell, p\rangle \qquad v' = M(\ell)}{\mathbf{WriteRef} \vdash \langle M, L, v_1::v_2::S\rangle \to_{loc} \langle M\left[\ell \mapsto v'[p := v_1]\right], L, S\rangle}\ ([\text{Writeref}])$$

$$\frac{}{\mathbf{Pop} \vdash \langle M, L, v::S\rangle \to_{loc} \langle M, L, S\rangle}\ ([\text{Pop}])$$

$$\frac{v \in \mathsf{Addr} \uplus \mathbb{B} \uplus \mathbb{N}}{\mathbf{LoadConst}\,\langle v\rangle \vdash \langle M, L, S\rangle \to_{loc} \langle M, L, v::S\rangle}\ ([\text{LoadConst}]) \qquad \frac{v'' = [[v\ \mathbf{Op}\ v']]}{\mathbf{Op} \vdash \langle M, L, v::v'::S\rangle \to_{loc} \langle M, L, v''::S\rangle}\ ([\text{Op}])$$

For simplicity we only consider binary ops.

A.3.2 Global Semantics

We have a notion of global state $\langle M, G, S\rangle$ that keeps track of a memory M, a list of global variables G and an operand stack S. The global semantics tells how an instruction i in procedure P makes a global state evolve into the next global one according to a code environment Ω, so it has this judgement:

$$P, i \vdash \langle M, G, S\rangle \to_{glob} \left\langle M', G', S'\right\rangle$$

$$\frac{t = gen\_tag(\Omega(\langle P.mid, s\rangle).kind) \qquad v = \{(f_i, v_i) \mid 1 \le i \le n\}}{P, \mathbf{Pack}\,\langle s\rangle \vdash \langle M, G, v_1::\cdots::v_n::S\rangle \to_{glob} \langle M, G, \langle v, t\rangle::S\rangle}\ ([\text{Pack}])$$

$$\frac{v = \langle \{(f_i, v_i) \mid 1 \le i \le n\}, t\rangle}{P, \mathbf{Unpack}\,\langle s\rangle \vdash \langle M, G, v::S\rangle \to_{glob} \langle M, G, v_1::\cdots::v_n::S\rangle}\ ([\text{Unpack}])$$

$$\frac{\rho = \langle P.mid, s\rangle \qquad G(\langle a, \rho\rangle) = \ell \qquad M(\ell) = v}{P, \mathbf{MoveFrom}\,\langle s\rangle \vdash \langle M, G, a::S\rangle \to_{glob} \langle M \setminus \ell, G \setminus \langle a, s\rangle, v::S\rangle}\ ([\text{MoveFrom}])$$

$$\frac{\rho = \langle P.mid, s\rangle \qquad \langle a, \rho\rangle \rangle \notin \mathtt{dom}(G) \qquad \ell \notin \mathtt{dom}(M)}{P, \mathbf{MoveTo}\,\langle s\rangle \vdash \langle M, G, a::v::S\rangle \to_{glob} \left\langle M_{C\setminus\ell}\left[\ell \mapsto v\right], G\left[\langle a, \rho\rangle \mapsto \ell\right], S\right\rangle}\ ([\text{MoveTo}])$$

$$\frac{\rho = \langle P.mid, s\rangle \qquad G(\langle a, \rho\rangle) = \ell}{P, \mathbf{BorrowGlobal}\,\langle s\rangle \vdash \langle M, G, a::S\rangle \to_{glob} \langle M, G, \langle \ell, []\rangle::S\rangle}\ ([\text{BorrowGlobal}])$$

A.3.3 Inter-procedural Semantics

The inter-procedural semantics relies on the notion of inter-procedural state $\langle C, M, G, S\rangle$ that extends the global state with a call stack C.

$$\Omega \vdash \langle C, M, G, S\rangle \to \left\langle C', M', G', S'\right\rangle$$

We change the rule for call and edit the rule for return.

$$\frac{\Omega(P_1).\text{code}[\text{pc}_1] = i = \mathbf{Call}\,\langle P_0\rangle \qquad \mathtt{sizeof}\,(S_{args}) = \mathtt{sizeof}\,(\Omega(P_0).1)}{\begin{array}{l}\Omega, P_1, i \vdash \langle\langle P_1, \text{pc}_1, L_1\rangle::C, M, G, S_{args}::S\rangle \to \\ \quad \langle\langle P_0, 0, \varnothing\rangle::\langle P_1, \text{pc}_1, L_1\rangle::C, M, G, S_{args} :: \text{startfun } P_0::S\rangle\end{array}}\ ([\text{Call}])$$

$$\frac{\Omega(P_0).\text{code}[\text{pc}_0] = \textbf{Ret}}{\Omega, P_0, i \vdash \langle\langle P_0, \text{pc}_0, L_0\rangle :: \langle P_1, \text{pc}_1, L_1\rangle :: C, M, G, S_0 :: \text{startfun } P_0 :: S\rangle \rightarrow \langle\langle P_1, \text{pc}_1 + 1, L_1\rangle :: C, M, G, S_0 :: S\rangle} \text{([Return])}$$

$$\frac{\Omega(P).code[\text{pc}] = i \qquad i \vdash \langle M, L, S\rangle \rightarrow_{loc} \langle M', L', S'\rangle}{\Omega \vdash \langle\langle P, \text{pc}, L\rangle :: C, M, G, S\rangle \rightarrow \langle\langle P, \text{pc} + 1, L'\rangle :: C, M', G, S'\rangle} \text{([Step-Loc])}$$

$$\frac{\Omega(P).code[\text{pc}] = i \qquad P, i \vdash \langle M, G, S\rangle \rightarrow_{glob} \langle M', G', S'\rangle}{\Omega \vdash \langle\langle P, \text{pc}, L\rangle :: C, M, G, S\rangle \rightarrow \langle\langle P, \text{pc} + 1, L\rangle :: C, M', G', S'\rangle} \text{([Step-Glob])}$$

$$\frac{\Omega(P).code[\text{pc}] = i = \textbf{Branch}\,\langle \text{pc}'\rangle}{\Omega \vdash \langle\langle P, \text{pc}, L\rangle :: C, M, G, S\rangle \rightarrow \langle\langle P, \text{pc}', L\rangle :: C, M, G, S\rangle} \text{([Branch])}$$

$$\frac{\Omega(P).code[\text{pc}] = i = \textbf{BranchCond}\,\langle \text{pc}'\rangle \qquad v = \texttt{true}}{\Omega \vdash \langle\langle P, \text{pc}, L\rangle :: C, M, G, v :: S\rangle \rightarrow \langle\langle P, \text{pc}', L\rangle :: C, M, G, S\rangle} \text{([BranchTrue])}$$

$$\frac{\Omega(P).code[\text{pc}] = i = \textbf{BranchCond}\,\langle \text{pc}'\rangle \qquad v = \texttt{false}}{\Omega \vdash \langle\langle P, \text{pc}, L\rangle :: C, M, G, v :: S\rangle \rightarrow \langle\langle P, \text{pc} + 1, L\rangle :: C, M, G, S\rangle} \text{([BranchFalse])}$$

A.3.4 Reflexive-Transitive Closure of the Semantics

The inter-procedural semantics is the top-level one, we indicate its reflexive and transitive closure as:

$$\Omega, P, i \vdash \langle C, M, G, S\rangle \rightarrow^* \langle C', M', G', S'\rangle$$

B Robust Safety Additions

The overall goal of this section is provide measures to reason locally about a collection of Move modules (formally, a code environment Ω). The move modules of interest we call:

trusted code

We want to prove that given some invariants that hold for trusted codeΩ alone (i.e., locally), we can compose Ω with another module A, get a whole program Ω^w, and state that the same invariants now hold for Ω^w. To state this theorem (Appendix B.8) we need to define:

- code environment composition $+ : \Omega \times \Omega \rightarrow \Omega$ (Appendix B.1);
- traces $\overline{\alpha}$ of events that capture what is relevant to be monitored for robust safety (Appendix B.2);
- global invariants ι that indicate what are conditions that programmers specify on trusted code (Appendix B.4);
- a local static analysis that can prove that an invariant ι holds locally for trusted codeΩ (Appendix B.5);
- a semantics that produces traces according to the existing small-step semantics (Appendix B.3);
- a global static analysis that checks whether an invariant holds for all the actions of a trace (Appendix B.6);
- an ensapsulator Ξ that proves that trusted code respects some coding / structural impositions $\vdash \Omega : \Xi$ (Appendix B.7).

B.0.1 Blockchain and trusted code

Dealing with robustness in a Blockchain scenario is unlike a standard setting due to a number of assumptions. All code is on public on the blockchain, so if you deploy some code, you know all that is there, and cannot consider it an attacker. An attacker is therefore code that is deployed after your code. Because there is no dynamic dispatch, the control flow that can happen with attackers has a clear structure: attackers can only call you and you never call attackers.

Property 2 (Attacker code cannot be called)**.**

$$\begin{aligned} &\text{if } \ \Omega \vdash A : atk \\ &\text{then } \ \nexists \textbf{Call}\,\langle P\rangle \in \Omega \text{ where } P \in A \end{aligned}$$

B.1 Module Composition

We indicate the rest of the program that the trusted code links against as attackers. From the formalisation standpoint, attackers are pairs consisting of a code environment and a main function. With a small abuse of notation we use metavariable A for both an attacker and for just its code environment to differentiate it from the code of interest.

$$\text{Attackers } A ::= \Omega, P$$

An attacker code environment must be valid with respect to the trusted codeΩ. Specifically, all attacker modules must be certified by the Move bytecode verifier and link successfully against Ω.

$$\Omega \vdash A : atk \stackrel{\text{def}}{=} \vdash A : wt \text{ and } \texttt{funs}(A) \not\cap \texttt{funs}(\Omega)$$

B.1.1 Linking and Starting

We indicate the code environment resulting from the linking of two code environments as follows:

$$\frac{\begin{array}{c} SD = \Omega.1 \qquad SD' = \Omega'.1 \qquad PD = \Omega.2 \qquad PD' = \Omega'.2 \\ \texttt{dom}(SD) \cap \texttt{dom}(SD') = \varnothing \qquad \texttt{dom}(PD) \cap \texttt{dom}(PD') = \varnothing \\ \texttt{freenames}(SD) \in \texttt{dom}(PD) \cup \texttt{dom}(PD') \cup \texttt{dom}(SD') \\ \texttt{freenames}(SD') \in \texttt{dom}(PD) \cup \texttt{dom}(PD') \cup \texttt{dom}(SD) \\ \texttt{freenames}(PD) \in \texttt{dom}(SD) \cup \texttt{dom}(PD') \cup \texttt{dom}(SD') \\ \texttt{freenames}(PD') \in \texttt{dom}(SD') \cup \texttt{dom}(PD') \cup \texttt{dom}(SD) \end{array}}{\Omega + \Omega' = \Omega :: \Omega'} \text{(Link)}$$

The domain ($\texttt{dom}(\cdot)$) of a partial function is the list of *RecName* / *ProcName* defined for that function.

The $\texttt{freeenames}(\cdot)$ of a partial function are the free *RecName*s / *ProcName*s mentioned in the codomain in that function that are not defined in the domain of the function.

When linking we check that no function nor data structure is defined twice (second line). Then we check that all free names are defined (last four lines), for a free name to be defined, it needs to be defined in the domain of any of the other definitions.

Note that this linking is purely syntactical, any constraint on the well-typedness of the elements being linked is tested in the generation of the initial state. The initial state of a code environment and a main procedure is calculated as follows:

$$\frac{}{\Omega_0(\Omega, P) = \Omega, P} \text{(Initial State)}$$

Additionally, we need to calculate the initial configuration, which is simply an empty memory, empty globals, the call stack initialised to the `main` and the default parameter (0) for main on the operand stack, followed by the canary indicating that `main` was called.

$$\frac{\langle a, m\rangle \mapsto \langle \_, \texttt{main} \mapsto \_\rangle \in \Omega \qquad C = \langle\langle\langle a, m\rangle, \texttt{main}\rangle, 0, [\,]\rangle}{\sigma_0(\Omega) = \langle C, [\,], [\,], 0 :: \text{startfun } \texttt{main}\rangle} \text{(Initial Configuration)}$$

B.2 Traces

We collect traces of events as computation progresses. Events simply record the safety-relevant bit of the program state. The goal is to check that at any point in time, events respect invariants. The notion of 'any point in time' in this case is whenever control is passed from trusted code to attacker and vice versa. We choose this granularity in order to allow invariants to be broken while the attacker is not observing and then reinstated before the attacker starts observing again. This flexibility is critical for enabling relational invariants over mutable data. For example, a defender may want to enforce the invariant `x == y` in code that updates both `x` and `y`. The invariant will be temporarily violated in the time between (e.g.) an update to `x` and `y`, but this as acceptable as long as the invariant is restored before control is passed to the attacker. The quantification over all attackers ensures that if invariant-relevant resources are shared with the attacker, they will change on the trace, breaking any invariant.

Traces follow this grammar:

$$\overline{\alpha} ::= [\,] \mid \overline{\alpha} :: \alpha$$

$$\alpha ::= \texttt{call}\ P\ M,G? \mid \texttt{call}\ P\ M,G! \mid \texttt{ret}\ M,G! \mid \texttt{ret}\ M,G?$$

The key bit that traces record from a security perspective is the globals G and the memory M at each boundary crossing. For proof reasons, the actions indicate more: they tell what kind of instruction generated the action (**Call** or **Ret**) and in case of calls, where was the call directed to. Finally, actions are decorated with ? or ! depending on whether the action originated in the attacker or in the trusted code respectively

B.3 Trace Semantics

B.3.1 Cross-Boundary Helpers

We rely on a judgement that, knowing which functions are defined by the trusted code, takes a call stack and tells whether the jump between the head and the second element crosses the boundary, and if so, in which direction.

$$\frac{C = C' :: \langle P_2,\_,\_\rangle :: \langle P_1,\_,\_\rangle \qquad \Omega(P_2) = undefined \qquad \Omega(P_1) = \langle\_,\_,\_\rangle}{\Omega \vdash C : ?}\ \text{(Cross-?)}$$

$$\frac{C = C' :: \langle P_2,\_,\_\rangle :: \langle P_1,\_,\_\rangle \qquad \Omega(P_1) = undefined \qquad \Omega(P_2) = \langle\_,\_,\_\rangle}{\Omega \vdash C : !}\ \text{(Cross-!)} \qquad \frac{\Omega \nvdash C : ? \qquad \Omega \nvdash C : !}{\Omega \vdash C : same}\ \text{(Cross-no)}$$

B.3.2 Trace Semantics Rules

The trace semantics uses on the operational semantics, and depending on the instruction being executed, it produces an action that is concatenated on a trace. Most instructions generate a silent action, the only instructions that generate an action are **Call** $\langle P\,\langle\vec{\rho}\rangle\rangle$ and **Ret**. In both cases the generated action is the same, the current globals. Then, the trace semantics relies on a state that keeps track of an additional element, the trusted code, in order to decorate the actions with ? and !.

$$\frac{\begin{array}{c}\Omega \vdash \sigma \to \sigma' \qquad \sigma = \langle C,M,G,S\rangle \\ (i \neq \textbf{Call} \text{ and } i \neq \textbf{Ret}) \text{ or} \\ (i = \textbf{Call}\,\langle P_0\rangle \text{ and } \Omega' \vdash C :: \langle P_0,0,\varnothing\rangle : same) \text{ or} \\ (i = \textbf{Ret} \text{ and } \Omega' \vdash P_0 : same)\end{array}}{\Omega' \triangleright \Omega,P,i \vdash \sigma \xrightarrow{[]} \sigma'}\ \text{(Action-No)}$$

$$\frac{i = \textbf{Call}\,\langle P_0\rangle \qquad \Omega \vdash \sigma \to \sigma' \qquad \sigma = \langle C,M,G,S\rangle \qquad \Omega' \vdash C :: \langle P_0,0,\varnothing\rangle : ?}{\Omega' \triangleright \Omega,P,i \vdash \sigma \xrightarrow{\texttt{call}\ P_0\ M,G?} \sigma'}\ \text{(Action-Call)}$$

$$\frac{i = \textbf{Call}\,\langle P_0\rangle \qquad \Omega \vdash \sigma \to \sigma' \qquad \sigma = \langle C,M,G,S\rangle \qquad \Omega' \vdash C :: \langle P_0,0,\varnothing\rangle : !}{\Omega' \triangleright \Omega,P,i \vdash \sigma \xrightarrow{\texttt{call}\ P_0\ M,G!} \sigma'}\ \text{(Action-Callback)}$$

$$\frac{i = \textbf{Ret} \qquad \Omega \vdash \sigma \to \sigma' \qquad \sigma = \langle C,M,G,S\rangle \qquad \Omega' \vdash C : !}{\Omega' \triangleright \Omega,P,i \vdash \sigma \xrightarrow{\texttt{ret}\ M,G!} \sigma'}\ \text{(Action-Return)}$$

$$\frac{i = \textbf{Ret} \qquad \Omega \vdash \sigma \to \sigma' \qquad \sigma = \langle C,M,G,S\rangle \qquad \Omega' \vdash C : ?}{\Omega' \triangleright \Omega,P,i \vdash \sigma \xrightarrow{\texttt{ret}\ M,G?} \sigma'}\ \text{(Action-Returnback)}$$

$$\frac{\Omega' \triangleright \Omega,P \vdash \sigma \xRightarrow{[]} \sigma'' \qquad \sigma'' = \langle P,pc,L\rangle :: C,M,G,S \qquad \Omega' \triangleright \Omega,P,P(pc) \vdash \sigma'' \xrightarrow{\alpha} \sigma'}{\Omega' \triangleright \Omega,P \vdash \sigma \xRightarrow{\alpha} \sigma'}\ \text{(Single)}$$

$$\frac{}{\Omega' \triangleright \Omega,P \vdash \sigma \xRightarrow{[]} \sigma}\ \text{(Refl)}$$

$$\frac{\Omega' \triangleright \Omega, P \vdash \sigma \overset{\overline{\alpha}}{\Longrightarrow\!\!\!\!\!\!\rightarrow} \sigma'' \quad \Omega' \triangleright \Omega, P \vdash \sigma'' \overset{\alpha?}{\Longrightarrow} \sigma''' \quad \Omega' \triangleright \Omega, P \vdash \sigma''' \overset{\alpha!}{\Longrightarrow} \sigma'}{\Omega' \triangleright \Omega, P \vdash \sigma \overset{\overline{\alpha}::\alpha?::\alpha!}{\Longrightarrow\!\!\!\!\!\!\rightarrow} \sigma'} \text{(Trace-Both)}$$

$$\frac{\Omega' \triangleright \Omega, P \vdash \sigma \overset{\overline{\alpha}}{\Longrightarrow\!\!\!\!\!\!\rightarrow} \sigma'' \quad \Omega' \triangleright \Omega, P \vdash \sigma'' \overset{\alpha?}{\Longrightarrow} \sigma''' \quad \neg(\Omega' \triangleright \Omega, P \vdash \sigma''' \overset{\alpha!}{\Longrightarrow} \sigma')}{\Omega' \triangleright \Omega, P \vdash \sigma \overset{\overline{\alpha}::\alpha?}{\Longrightarrow\!\!\!\!\!\!\rightarrow} \sigma'} \text{(Trace-Single)}$$

$$\frac{}{\Omega^{\dagger} \triangleright \Omega, P \vdash \sigma \overset{[]}{\Longrightarrow\!\!\!\!\!\!\rightarrow} \sigma} \text{(Trace-Refl)}$$

Other RS Approaches: Assertions An alternative is to enrich the language with precise assertions. We could both use code-based assertions [39] or logical ones that are collected in assume statements [17, 20]. Our approach is similar to the former, but we choose to not introduce an instruction that checks invariants for several reasons. First, we do not want to modify our language. Second, we want to show that checks would never fail, so it is safe to skip them.

Thus, including these checks in the operational semantics is unwanted and unnecessary.

B.4 Invariants

We indicate invariants with ι. Invariants contain the list of globals that point to memory locations with a logical invariant. Then it contains the list of memory locations with a logical invariant, i.e., a map from locations ℓ to conditions that hold on the content of those locations in memory. We leave the conditions abstract and give an intuition of what they may look like via examples only.

We rely on these functions to manipulate invariants:

- invariants are defined for a code environment, which can be extracted from ι as follows: $\texttt{codeof}\,(\iota) = \Omega$.

  A code environment Ω and an invariant ι are in agreement if the former is the code of the latter. Formally $\Omega \frown \iota \overset{\text{def}}{=} \texttt{codeof}\,(\iota) = \Omega$.

- Invariants can refer to fields of records, so $f \in \iota / f \notin \iota$ indicates that the invariant ι does/does not refer to the field f.

- $\texttt{domG}\,(\iota)$ returns the indices of globals for which invariants are defined. That is, this returns the pairs a, ρ that identify globals for which an invariant is defined.

- $\texttt{domM}\,(\iota)$ returns the memory locations reachable from $\texttt{domG}\,(\iota)$, i.e., the memory locations for which invariants are defined.

- $\texttt{cond}\,(\iota, \ell, v)$ evaluates the condition for location ℓ on value v and returns true (if the condition is satisfied) or false (otherwise).

- We can restrict a code environment wrt an invariant as follows $\Omega|_{\iota}$ in order to carve out the code environment Ω' that is contained in Ω and that only talks about the code mentioned in ι, without any other code that has no invariant on. This is used to identify the sub-part of a code environment that needs to be encapsulated (Appendix B.7).

We rely on this property for invariants: none of the types mentioned in $\texttt{domG}\,(\iota)$ are attacker types.

Property 3 (Invariants are not on Attacker Typed Globals)**.**

$$\forall \langle a, \rho \rangle \in \texttt{domG}\,(\iota), \rho \in \texttt{declaredtypes}\,(\texttt{codeof}\,(\iota))$$

B.5 Local Invariant-Checking

We introduce a judgement for a static analysis that proves a trusted codeΩ satisfies an invariant ι *locally*: This relies on the invariants stated in Appendix B.8.2.

Definition 3 (Local Invariant)**.**

$$\Omega' \Vdash \iota : local \overset{\text{def}}{=} \forall \ldots\ldots$$
$$\text{let } \sigma = \langle C, M, G, S \rangle$$

$$
\begin{array}{l}
\text{if } \Omega' \vDash \sigma \propto \iota : \mathit{strong} \\
\text{and } \Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{\alpha!} \sigma' \\
\text{then } \alpha! \Vdash \iota \\
\text{and } \Omega' \vDash \sigma \propto \iota : \mathit{local}
\end{array}
$$

Local Checking in Practice The verifier checks invariants locally on some trusted code using a suitable static analysis tool (e.g., the Move Prover [43]).

B.6 Global Invariant-Checking

A trace respects an invariant *globally* if all of its actions respect the invariants.

$$
\frac{}{[] \Vdash \iota : \mathit{global}}\ \text{(Global-check-base)} \qquad \frac{\alpha \Vdash \iota \qquad \overline{\alpha} \Vdash \iota : \mathit{global}}{\overline{\alpha} :: \alpha \Vdash \iota : \mathit{global}}\ \text{(Global-check-ind)}
$$

An action respects an invariant if all the invariants applied to the action are true.

$$
\frac{}{\alpha \Vdash []}\ \text{(Event-check-base)} \qquad \frac{\alpha = \_M, G \qquad M, G \vdash \iota}{\alpha \Vdash \iota}\ \text{(Event-check-ind)}
$$

Intuitively, $M, G \vdash \iota$ holds if the part of the memory M restricted to just the addresses mentioned in G respects ι.

$$
\frac{G_i = G|_{\mathtt{domG}(\iota)} \qquad M_i = M|_{G_i} \qquad \forall \ell \in \mathtt{dom}(M_i).\, \mathtt{cond}(\iota, \ell, M_i(\ell)) = \mathit{true}}{M, G \vdash \iota}\ \text{(Invariant Satisfaction)}
$$

Almost-Everywhere Having invariants that hold at every line of code is impractical and meaningless: such invariants do not let us express interesting programming patterns. Instead, invariants should be broken temporarily, so long as they are reinstated before an attacker can notice. The noticing point is whenever the attacker is executing, so trusted code can violate an invariant so long as it is executing, but it will reinstate it before moving control to the attacker.

However, since invariants are only on trusted code relevant globals (and memory), we can enforce that they always hold at any step of attacker computation. The way to do this is the universal quantification over attackers in the classical RS setup (which we follow).

In fact, suppose there is an attacker that breaks an invariant of the trusted code. Since attackers are universally quantified, there also exists the attacker that right after breaking the invariant returns control to the trusted code. The second attacker generates a trace action with its globals and memory that violate the invariant. However, by definition of RS, that trace does violate the invariant. Thus, we reached a contradiction and the premise that the attacker could violate the invariant temporarily in its code is therefore wrong.

B.7 Encapsulator

We indicate that the code Ω follows the indications of the encapsulator Ξ as follows. Right now the encapsulator is an abstract entity, whose behaviour we abstract away as $\Xi(\Omega')$. We only pass these parameters because this is a static analysis, for a dynamic analysis we could also pass σ and σ'.

Definition 4 (Encapsulated Code)**.**

$$
\begin{array}{l}
\iota \vdash \Omega' : \Xi \stackrel{\text{def}}{=} \forall ..... \\
\quad \text{let } \sigma = \langle C, M, G, S \rangle \\
\quad \text{if } \Omega' \vDash \sigma \propto \iota : \mathit{strong} \\
\quad \text{and } \Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{\alpha!} \sigma' \\
\quad \text{and } \Xi(\Omega'|_\iota) \\
\quad \text{then } \Omega' \vDash \sigma \propto \iota : \mathit{unreachable}
\end{array}
$$

Intuitively, the encapsulator ensures that all writes to fields of the declared types of a given module can only occur inside the proceures of that module. If this condition holds, the internal state of a module is encapsulated and cannot be mutated by any other module. Similarly, the internal state of a code environment is encapsulated when no modules outside of the environment can mutate it. The local invariant checking performed by the Appendix B.5 is necessary, but not sufficient, to ensure that an invariant holds when a code environment is composed with attacker code. The encapsulator captures some generic restrictions that ensure the local invariant will continue to hold even in an adversarial setting.

Crucially, we only call the encapsulator on the part of the trusted code that is mentioned by the invariants, thus the restriction $\Omega'|_\iota$.

B.8 RS Theorem

The RS theorem below (Theorem 3 (Well-typed modules are Robustly-Safe)) is stated in a classical way. It tells us that some trusted codeΩ of interest, with a starting program P, is *RS* wrt an invariant ι and an encapsulator Ξ if no matter what valid attacker that trusted code is linked against, so long as the trusted code and the attacker are well-typed and the trusted code respects the invariants locally and it is programmed respecting the encapsulator then no matter what traces it generates when running alongside the attacker the traces never break the invariants.

In the statement below, we have a slight abuse of notation to make the statement simpler, without projections: A is both Ω and P.

The goal to prove is the following:

Theorem 3 (Well-typed modules are Robustly-Safe)**.**

$$\begin{aligned}
\triangleright_{RS}\ \Omega : \iota, \Xi \stackrel{\text{def}}{=} \forall A, P, \overline{\alpha}, \sigma'.\ & \\
& \text{if } \Omega \vdash A : atk \\
& \text{and } \Omega \frown \iota \\
& \text{and } \vdash \Omega : wt \\
& \text{and } \Omega \Vdash \iota : local \\
& \text{and } \iota \vdash \Omega : \Xi \\
& \text{and } \Omega \triangleright \Omega_0\,(\Omega + A) \vdash \sigma_0\,(\Omega + A) \stackrel{\overline{\alpha}}{\Longrightarrow\!\!\!\!\!\rightarrow} \sigma' \\
& \text{then } \overline{\alpha} \Vdash \iota : global
\end{aligned}$$

Proof of Theorem 3 (Well-typed modules are Robustly-Safe).

This holds by Lemma 12 (Generalised RS) and Lemma 2 (The Initial State Respects the Strong Property). □

♠

B.8.1 Auxiliary Functions

State Executing Attacker Code

$$\frac{\sigma = \langle C, M, G, S\rangle \quad C = \langle P, pc, L\rangle :: C' \quad \Omega(P) = \text{undefined}}{\Omega \vdash \sigma : atkcode}\ \text{(Attacker Code Running)}$$

$$\frac{\sigma = \langle C, M, G, S\rangle \quad C = \langle P, pc, L\rangle :: C' \quad \Omega(P) = \_}{\Omega \vdash \sigma : modcode}\ \text{(Module Code Running)}$$

We can tell whether attacker code is running by looking at the top of the stack: if the procedure that executes is not defined in the trusted code, then it is the attacker. Conversely, the trusted code is running if its code is on the top of the stack.

Call Stack of the Attacker

$$\frac{\texttt{atkstk}(\Omega, C) = C' \qquad \Omega(P) = \_}{\texttt{atkstk}(\Omega, \langle P, pc, L\rangle :: C) = C'} \text{ (Attacker Stack - No)}$$

$$\frac{\texttt{atkstk}(\Omega, C) = C' \qquad \Omega(P) = \text{undefined}}{\texttt{atkstk}(\Omega, \langle P, pc, L\rangle :: C) = \langle P, pc, L\rangle :: C'} \text{ (Attacker Stack - Yes)}$$

$$\frac{}{\texttt{atkstk}(\Omega, \varnothing) = \varnothing} \text{ (Attacker Stack - Base)}$$

According to the trusted code, given a global call stack C, C' is the part of the stack that only talks about attacker functions.

Locations in a Call Stack We indicate locations as ℓ and lists of locations as ℓ. The function below traverses a call stack and extracts all locations in its locals. We structure this function to eventually work on lists of values, so we can apply that both to lists of call stacks, to locals and to lists of values.

$$\frac{}{\texttt{locsof}([]) = []} \text{ (Locations-base)} \qquad \frac{\texttt{locsof}(L) = \ell \qquad \texttt{locsof}(C) = \ell'}{\texttt{locsof}(\langle L\rangle :: C) = \ell :: \ell'} \text{ (Locations-ind)}$$

$$\frac{}{\texttt{locsof}(x \mapsto v :: L) = \texttt{locsof}(v :: \texttt{locsof}(L))} \text{ (Locations-locals-ind)}$$

$$\frac{\texttt{locsof}(V) = \ell}{\texttt{locsof}(c :: V) = \ell :: \ell} \text{ (Locations-values-ind yes)} \qquad \frac{\texttt{locsof}(V) = \ell \qquad v \neq \ell}{\texttt{locsof}(v :: V) = \ell} \text{ (Locations-locals-ind - no)}$$

OpStack of the Attacker The function below traverses an operand stack and filters out all the sections that do not belong to attacker code (as defined from the viewpoint of trusted code).

$$\frac{}{\texttt{atkops}(\Omega, S :: \text{startfun } \texttt{main}) = \texttt{atkops}(\Omega, S :: \text{startfun } \texttt{main}, \top)} \text{ (atkops-main)}$$

$$\frac{\Omega P = \text{undefined}}{\texttt{atkops}(\Omega, S :: \text{startfun } P, \top) = \texttt{atkops}(\Omega, S, \top) :: \text{startfun } P} \text{ (atkops-atk)} \qquad \frac{\Omega P = \text{undefined}}{\texttt{atkops}(\Omega, S :: V, \top) = \texttt{atkops}(\Omega, S, \top) :: V} \text{ (atkops-atk-val)}$$

$$\frac{\Omega P = \_}{\texttt{atkops}(\Omega, S :: \text{startfun } P, \top) = \texttt{atkops}(\Omega, S, \bot)} \text{ (atkops-code)} \qquad \frac{\Omega P = \text{undefined}}{\texttt{atkops}(\Omega, S :: V, \bot) = \texttt{atkops}(\Omega, S, \bot)} \text{ (atkops-code-val)}$$

Types of Attackers Indicate with T a list of types. Assume given a function $\texttt{declaredTypes}(\Omega)$ that returns all the types declared in Ω. This function is used to retrieve all types defined by the trusted code.

We can overapproximate the types defined by the attacker by collecting all types that are valid and not mentioned by the trusted code.

$$\frac{T = \{\tau \mid \tau \in \text{Type and } \tau \notin \texttt{declaredTypes}(\Omega)\}}{\texttt{atktypes}(\Omega) = T} \text{ (Attacker Types)}$$

This definition has this form because we do not carry around the attacker definition, so we do not have its type definitions in the statement of the weak property where this function is used. Instead we just have the trusted code, so we rely on that information to know what types may the attacker declare.. An alternative (for where this function is used. i.e., for the weak property) is to take the globals, take the types defined by the module and remove them. This alternative would make this definition of $\texttt{atktypes}(\cdot)$ not necessary – we still need the $\texttt{declaredtypes}(\cdot)$ though.

Memory and Global Restriction Given a memory M and a set of locations ℓ, we indicate the sub-memory restricted to just the locations of ℓ as:

$$M|_\ell$$

Given globals G and a set of addresses A or of types T (such as those derivable by $\texttt{dom}(\iota)$), we indicate the sub-globals restricted to just the domain of A or T as:

$$G|_A \qquad G|_T$$

To perform a restriction according to both parameters, we write $G|_A|_T$ (the order of A and T is irrelevant here).

B.8.2 State Properties for Proofs

The RS proof is based upon the operational state maintaining a strong property, which is that if we take the globals, we can partition them in 2 parts: invariant related and not. The invariant related are never alterable from attacker code and at boundaries crossing, applying the invariants is always true. This last part is a bit strong, it is exactly what we need. The strong property is preserved if the attacker is executing, because the attacker cannot vary the invaraint-related globals.

Then the strong property is preserved if the trusted code is reducing because trusted code is verified and encspaulated. Intuitively, verification and encapsulation each give us a weaker property that, once combined, yield the strong one. If some code is locally verified, then starting from a state with the strong property, we do a !-action and at the end we have a state with a weaker property: for the globals that are invariant related, applying the invariant is always true. Then we need to apply the encapsulator: if some code is encapsulated, then starting from a state with the invariant, we do a !-action and at the end we have a state with a weaker property: for the globals that are invariant related, they are not alterable from attacker code.

$$\frac{\Omega \vDash \sigma \propto \iota : \mathit{local} \qquad \Omega \vDash \sigma \propto \iota : \mathit{unreachable}}{\Omega \vDash \sigma \propto \iota : \mathit{strong}} \text{(Strong Property)}$$

According to the module code Ω, the program state σ respects invariants ι strongly. That is, the program state respects the invariant weakly and makes a part of the global unreachable from attacker code.

For the weak properties we rely on knowledge of what code belongs to the module to understand which stack frame are not its own, and therefore they belong to the attacker.

$$\frac{\begin{array}{c}\sigma = \langle C, M, G, S\rangle \\ C_a = \mathtt{atkstk}(\Omega, C) \qquad L_a = C_a.\mathrm{locals} \qquad S_a = \mathtt{atkops}(\Omega, S) \qquad T = \mathtt{atktypes}(\Omega) \\ G_a = G|_T \qquad \mathtt{locsof}(L_a) \cup \mathtt{locsof}(G_a) \cup \mathtt{locsof}(S_a) = \ell \qquad M_a = M|_\ell\end{array}}{\Omega, \sigma \vdash M_a, G_a : \mathit{attackerpart}} \text{(Attacker Part of State)}$$

The relevant parts of a state for weak properties are attacker memory and attacker globals. Attacker memory M_a is the memory whose locations are found: in attacker locals (L_a), in attacker globals (G_a) and in attacker operand stack frames (S_a). Attacker globals are those whose type is attacker-defined (T). Attacker locals are the locals extracted from attacker call stacks frames (C_a). Attacker operand stack frames are those that belong to the attacker as extracted from the stack.

In the following, given two sets A and B, we write that they are disjoint as $A \not\cap B$, so $A \not\cap B \stackrel{\text{def}}{=} A \cap B = \varnothing$.

$$\frac{\sigma = \langle C, M, G, S\rangle \qquad M, G \vdash \iota}{\Omega \vDash \sigma \propto \iota : \mathit{local}} \text{(Weak Property - Inv)}$$

$$\frac{\begin{array}{c}\sigma = \langle C, M, G, S\rangle \qquad G_i = G|_{\mathtt{domG}(\iota)} \qquad M_i = M|_{\mathtt{domM}(\iota)} \\ \Omega, \sigma \vdash M_a, G_a : \mathit{attackerpart} \\ G_i \not\cap G_a \qquad \mathtt{dom}(M_i) \not\cap \mathtt{dom}(M_a)\end{array}}{\Omega \vDash \sigma \propto \iota : \mathit{unreachable}} \text{(Weak Property - Atk Changes)}$$

As stated, Rule Weak Property - Inv means that the memory and globals part of a state respect the invariant, as per Rule Invariant Satisfaction.

Instead, Rule Weak Property - Atk Changes first extracts the attacker memory and globals as per Rule Attacker Part of State. Then it checks that no attacker global matches one with an invariant on, and it checks that no attacker memory location matches a location with an invariant on. It is ok for the attacker to temporarily have a global address which has an invariant on, because when that address is used by attacker code, the type part will be auto-filled by the semantics with an attacker-supplied type, and this ensures the global accessed with that address does not have an invariant on.

Lemma 1 (The Two Weak Properties Imply the Strong One)**.**

$$\begin{array}{rl}\text{if} & \Omega' \vDash \sigma \propto \iota : \mathit{local} \\ \text{and} & \Omega' \vDash \sigma \propto \iota : \mathit{unreachable} \\ \text{then} & \Omega' \vDash \sigma \propto \iota : \mathit{strong}\end{array}$$

Proof of Lemma 1 (The Two Weak Properties Imply the Strong One).

By definition in Rule Strong Property. □

♠

B.8.3 Auxiliary Lemmas

We rely on the following auxiliary lemmas in order to prove the generalised RS statement (Lemma 12 (Generalised RS)).

First, initial states respect the strong property.

Lemma 2 (The Initial State Respects the Strong Property)**.**

$$\begin{aligned} &\forall \ldots \\ &\text{if} \quad \vdash \Omega + A : ok \\ &\text{then} \quad \Omega \vDash \Omega_0\,(\Omega + A) \propto \iota : strong \end{aligned}$$

Proof of Lemma 2 (The Initial State Respects the Strong Property).

Trivial. □

♠

The next lemma tells that reductions in the attacker (Lemma 3 (Attacker Reductions Respect the Strong Property)) respect the property.

Lemma 3 (Attacker Reductions Respect the Strong Property)**.**

$$\begin{aligned} &\text{if} \quad \Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{\alpha?} \sigma' \\ &\text{and} \quad \Omega' \frown \iota \\ &\text{and} \quad \Omega' \vdash \sigma : atkcode \\ &\text{and} \quad \Omega \vDash \sigma \propto \iota : strong \\ &\text{then} \quad \Omega' \vDash \sigma' \propto \iota : strong \\ &\text{and} \quad \alpha? \Vdash \iota \end{aligned}$$

Proof of Lemma 3 (Attacker Reductions Respect the Strong Property).

By Rule Single we need to prove:

1. $\Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{[]} \sigma''$

   This follows from Lemma 9 (Attacker Silent Reductions Respect the Strong Property) and we also get that $\Omega' \vDash \sigma'' \propto \iota : strong$ (HS2).

2. $\Omega' \triangleright \Omega, P, i \vdash \sigma'' \xrightarrow{\alpha} \sigma'$

   There are two cases here:

   Rule Action-Call: By HS2, we have $\Omega \vDash \sigma'' \propto \iota : local$ (HW1) and $\Omega \vDash \sigma'' \propto \iota : unreachable$ (HW2).
   We need to prove $\Omega \vDash \sigma' \propto \iota : local$ (TW1) and $\Omega \vDash \sigma' \propto \iota : unreachable$ (TW2).
   We have $\sigma'' = \langle C, M, G, S \rangle$ and $\sigma' = \langle C', M, G, S' \rangle$.
   Thus, since M and G are not changed, TW1 follows from HW1.
   Regarding TW2: C contains one more binding than C', but it is not in attacker functions and thus not considered by the weak property.
   Similarly and S also contains more locals than S' but they are not in attacker functions.
   Thus, the attacker-related entries in σ'' and in σ' are the same and TW2 holds by HW2.s

   Rule Action-Returnback: By Lemma 6 (Monotonicity for Strong Property).

   In both cases, by HW1, the second conjunct of the thesis holds, now we focus on the first one.

□

♠

This lemma tells us that if we know the weak invariant property on a state, and that state takes a step decreasing its call stack and/or its operand stack, the new state also upholds the weak invariant property.

Lemma 4 (Monotonicity for Weak Property INV)**.**

$$\begin{aligned} \text{if}\ & \Omega' \vDash \langle C, M, G, S\rangle \propto \iota : \mathit{local} \\ \text{and}\ & \Omega \vdash \langle C, M, G, S\rangle \to \langle C', M, G, S'\rangle \\ \text{and}\ & C' \subseteq C \\ \text{and}\ & S' \subseteq S \\ \text{then}\ & \Omega' \vDash \langle C, M, G, S\rangle \propto \iota : \mathit{local} \end{aligned}$$

Proof. By Rule Weak Property - Inv. □

♠

This lemma tells us that if we know the weak unreachable property on a state, and that state takes a step decreasing its call stack and/or its operand stack, the new state also upholds the weak unreachable property.

Lemma 5 (Monotonicity for Weak Property Unreachable)**.**

$$\begin{aligned} \text{if}\ & \Omega' \vDash \langle C, M, G, S\rangle \propto \iota : \mathit{unreachable} \\ \text{and}\ & \Omega \vdash \langle C, M, G, S\rangle \to \langle C', M, G, S'\rangle \\ \text{and}\ & C' \subseteq C \\ \text{and}\ & S' \subseteq S \\ \text{then}\ & \Omega' \vDash \langle C', M, G, S'\rangle \propto \iota : \mathit{unreachable} \end{aligned}$$

Proof. C contains less attacker-related bindings than C', and S also contains less attacker frames than S'.
Let C_a and S_a be the call stack and operand stack parts of Rule Attacker Part of State applied to the state with C and S.
Let C'_a and S'_a be the call stack and operand stack parts of Rule Attacker Part of State applied to the state with C' and S'.
By assumptions HP3 and HP4 we thus have HPC $C'_a \subseteq C_a$ and HPS $S'_a \subseteq S_a$
By assumption HP1 we have HPNS $G_i \notin S_a$ and HPNC $G_i \notin C_a$.
By HPC with HPNC and HPS with HPNS we can conclude that $G_i \notin S'_a$ and $G_i \notin C'_a$, so the thesis holds. □

♠

This lemma tells us that if we know the strong property on a state, and that state takes a step decreasing its call stack and/or its operand stack, the new state also upholds the strong property.

Lemma 6 (Monotonicity for Strong Property)**.**

$$\begin{aligned} \text{if}\ & \Omega' \vDash \langle C, M, G, S\rangle \propto \iota : \mathit{strong} \\ \text{and}\ & \Omega \vdash \langle C, M, G, S\rangle \to \langle C', M, G, S'\rangle \\ \text{and}\ & C' \subseteq C \\ \text{and}\ & S' \subseteq S \\ \text{then}\ & \Omega' \vDash \langle C, M, G, S\rangle \propto \iota : \mathit{strong} \end{aligned}$$

Proof. Indicate $\langle C, M, G, S\rangle$ as σ and $\langle C', M, G, S'\rangle$ as σ'.
By HP1, we have $\Omega \vDash \sigma \propto \iota : \mathit{local}$ (HW1) and $\Omega \vDash \sigma \propto \iota : \mathit{unreachable}$ (HW2).
By Rule Strong Property we need to prove

1. $\Omega \vDash \sigma' \propto \iota : \mathit{local}$ (TW1)

   Since M and G are not changed we apply Lemma 4 (Monotonicity for Weak Property INV) with HW1 and conclude TW1.

2. $\Omega \vDash \sigma' \propto \iota : \mathit{unreachable}$ (TW2)

   This follows from Lemma 5 (Monotonicity for Weak Property Unreachable).

□

♠

This lemma tells us that in order to derive the strong property on a state, we can split all its components in 2 parts and check the strong property on the 2 sub-parts individiually, so long as the second part has a very specific form: all new additions to globals and locals point to a memory location whose content was part of the state with the strong property.

Lemma 7 (Compositionality of Strong Property)**.**

$$\begin{aligned}
&\text{if } \sigma = \left\langle \left\langle P, \mathrm{pc}, L' :: L \right\rangle :: C, M' :: M, G' :: G, S \right\rangle \\
&\text{and } \Omega \vDash \langle \langle P, \mathrm{pc}, L \rangle :: C, M, G, S \rangle \propto \iota : \mathit{strong} \\
&\text{and } \Omega \vDash \left\langle \left\langle P, \mathrm{pc}, L' \right\rangle :: C, M', G', [\,] \right\rangle \propto \iota : \mathit{strong} \\
&\text{and } \forall a \in \mathit{img}(L').\, a \in \mathtt{dom}\left(M'\right) \\
&\text{and } \forall \langle a, \rho \rangle \in \mathtt{dom}\left(G'\right).\, G'(\langle a, \rho \rangle) \in \mathtt{dom}\left(M'\right) \\
&\text{and } \forall v \in \mathit{img}(M').\, v \in S \\
&\text{and } \mathtt{dom}\left(M'\right) \cap \mathtt{dom}\left(M\right) = \varnothing \\
&\text{then } \Omega \vDash \sigma \propto \iota : \mathit{strong}
\end{aligned}$$

Proof. By Rule Strong Property we have to prove:

1. $\Omega \vDash \sigma \propto \iota : \mathit{local}$

   From HP7 and HP 5 we have that M and M' as well as G and G' have disjoint domains.

   From HP2 we have the thesis for the M and G subparts.

   From HP3 we have the thesis for the M' and G' subparts.

2. $\Omega \vDash \sigma \propto \iota : \mathit{unreachable}$

   We need to show that the G_a part of G' is not intersecting G_i, which holds from HP3.

   Then we need to show that locations in L' do not intersect with M_i.

   This follows from HP6 since from HP4 those addresses and locations are in M' and from HP6 we have that they were in S, which from HP2 we know to satisfy the definition of Rule Weak Property - Atk Changes.

□

♠

This lemma tells that in a state whose call stack and operand stack can pe decomposed in 2 parts, but where M and G remain the same it is sufficient to know the strong property on part 1 of C and S and just the weak unreachable property on part 2 to derive the strong property on the full state.

Lemma 8 (Compositionality of Strong Property and Weak Property Conditions)**.**

$$\begin{aligned}
&\text{if } \sigma = \left\langle \left\langle P, \mathrm{pc}, L' :: L \right\rangle :: C, M, G, S' :: S \right\rangle \\
&\text{and } \Omega \vDash \langle C, M, G, S \rangle \propto \iota : \mathit{strong} \\
&\text{and } \forall \ell \in \mathtt{locsof}\left(L'\right) \cup \mathtt{locsof}\left(S'\right).\, \mathtt{dom}\left(M_i\right) \not\parallel c \\
&\text{then } \Omega \vDash \sigma \propto \iota : \mathit{strong}
\end{aligned}$$

Proof. From Rule Strong Property we have to prove

1. $\Omega \vDash \sigma \propto \iota : \mathit{local}$

   Since there is no change to M nor G, this holds from HP2.

2. $\Omega \vDash \sigma \propto \iota : \mathit{unreachable}$

   From Rule Attacker Part of State we have that $L_a = L'_a \cup L''_a$ where L'_a come from L and L''_a come from L'.

   Similarly $S_a = S'_a \cup S''_a$ where S'_a come from S and S''_a come from S'.

   Also, $M_a = M'_a \cup M''_a$ where M'_a come from $\texttt{locsof}\,(L'_a) \cup \texttt{locsof}\,(S'_a)$ and M''_a come from $\texttt{locsof}\,(L''_a) \cup \texttt{locsof}\,(S''_a)$.

   Thus, $A_{mem} = A'_{mem} \cup A''_{mem}$ where A'_{mem} come from M'_a and A''_{mem} come from M''_a.

   We need to prove that all the elements respect Rule Weak Property - Atk Changes.

   For all $'$ elements, this follows from HP2.

   For all $''$ elements, we have the following:

   (a) we need to show that $G_i \not\upharpoonright G_a$ this is trivial since G_a is empty

   (b) we need to show that $\texttt{dom}\,(M_i) \not\upharpoonright \texttt{dom}\,(M''_a)$, this follows from HP3

□

♠

Lemma 9 (Attacker Silent Reductions Respect the Strong Property)**.**

$$\begin{aligned} &\text{if } \Omega' \triangleright \Omega, P \vdash \sigma \overset{[]}{\Longrightarrow} \sigma' \\ &\text{and } \Omega' \frown \iota \\ &\text{and } \Omega' \vdash \sigma : \mathit{atkcode} \\ &\text{and } \Omega \vDash \sigma \propto \iota : \mathit{strong} \\ &\text{then } \Omega' \vDash \sigma' \propto \iota : \mathit{strong} \end{aligned}$$

Proof of Lemma 9 (Attacker Silent Reductions Respect the Strong Property).
This proof proceeds by induction on $\Longrightarrow$:

Base, Rule Refl: This trivially holds.

Inductive, Rule Single: so we have:

$\Omega' \triangleright \Omega, P \vdash \sigma \overset{[]}{\Longrightarrow} \sigma''$ By IH we have $\Omega' \vDash \sigma'' \propto \iota : \mathit{strong}$ (HS2).

$\Omega' \triangleright \Omega, P, i \vdash \sigma'' \xrightarrow{\alpha} \sigma'$

Since $\alpha = []$ we can only apply Rule Action-No, so the proof now proceeds by case analysis on $\Omega \vdash \sigma'' \rightarrow \sigma'$:

Rule [BranchTrue] By Lemma 6 (Monotonicity for Strong Property).

Rule [BranchFalse] By Lemma 6 (Monotonicity for Strong Property).

Rule [Return] Since no label is created this is a return to an attacker function (Rule Cross-no).
By Lemma 6 (Monotonicity for Strong Property).

Rule [Call] Since no label is created this is a call to an attacker function (Rule Cross-no).
The only change to the program state is a new element on C which contains no bindings.
So, this holds by a reasoning similar to Lemma 6 (Monotonicity for Strong Property).

Rule [Step-Glob] We perform case analysis on global reductions:

Rule [BorrowGlobal] We have $i = \textbf{BorrowGlobal}\,\langle s\rangle$ in

$$\frac{\rho = \langle P.mid, s\rangle \quad G(\langle a,\rho\rangle) = \ell}{\dfrac{P, \textbf{BorrowGlobal}\,\langle s\rangle \vdash \langle M,G,a::S\rangle \rightarrow_{glob} \langle M,G,\langle \ell,[]\rangle::S\rangle}{\Omega \vdash \langle\langle P,\mathrm{pc},L\rangle::C,M,G,a::S\rangle \rightarrow \langle\langle P,\mathrm{pc}+1,L\rangle::C,M,G,\langle \ell,[]\rangle::S\rangle}}$$

We need to prove:

- $\Omega' \vDash \langle\langle P,\mathrm{pc}+1,L\rangle::C,M,G,\langle \ell,[]\rangle::S\rangle \propto \iota : \mathit{strong}$

By Lemma 8 (Compositionality of Strong Property and Weak Property Conditions) it suffices to prove for $L' = []$ and $S' = \langle \ell,[]\rangle$:

1. $\Omega' \vDash \langle\langle P,\mathrm{pc}+1,L\rangle C,M,G,S\rangle \propto \iota : \mathit{strong}$, which follows from Lemma 6 (Monotonicity for Strong Property) with HS2 since pc+1 does not play a role in the properties.
2. $\forall \ell \in \texttt{locsof}\,(L') \cup \texttt{locsof}\,(S').\,\texttt{dom}\,(M_i) \not\frown c$
   By HS2 we have that $G_i.1 \not\frown a, \rho.1$, so ℓ is collected by $\texttt{locsof}\,(\cdot)$ and put in M_a, so we have HPG $G_i.1 \not\frown M(\ell)$.
   By Rule Attacker Part of State we have that ℓ is collected by $\texttt{locsof}\,()$, so it ends up in S_a.
   By Rule Weak Property - Atk Changes we need to prove that $M_i \not\frown \ell$ which holds by HPC.

Rule [MoveTo] We have $i = \textbf{MoveTo}\,\langle s\rangle$ in

$$\frac{\rho = \langle P.mid, s\rangle \quad \langle a,\rho\rangle \notin \texttt{dom}\,(G) \quad \ell \notin \texttt{dom}\,(M)}{\dfrac{P, \textbf{MoveTo}\,\langle s\rangle \vdash \langle M,G,a::v::S\rangle \rightarrow_{glob} \left\langle M_{C\setminus \ell}\left[\ell \mapsto v\right], G\left[\langle a,\rho\rangle \mapsto \ell\right], S\right\rangle}{\Omega \vdash \langle\langle P,\mathrm{pc},L\rangle::C,M,G,a::v::S\rangle \rightarrow \left\langle\langle P,\mathrm{pc}+1,L\rangle::C,M_{C\setminus \ell}\left[\ell \mapsto v\right], G\left[\langle a,\rho\rangle \mapsto \ell\right], S\right\rangle}}$$

We need to prove:

- $\Omega' \vDash \langle\langle P,\mathrm{pc}+1,L\rangle::C,M\left[\ell \mapsto v\right], G\left[\langle a,\rho\rangle \mapsto \ell\right], S\rangle \propto \iota : \mathit{strong}$

By Lemma 6 (Monotonicity for Strong Property) it suffices to prove:

- $\Omega' \vDash \langle\langle P,\mathrm{pc}+1,L\rangle::C,M\left[\ell \mapsto v\right], G\left[\langle a,\rho\rangle \mapsto \ell\right], v::S\rangle \propto \iota : \mathit{strong}$

By Lemma 7 (Compositionality of Strong Property) with HPL $L' = []$, HPM $M' = \ell \mapsto v$, HPG $G' = \langle a,\rho\rangle \mapsto \ell$ it suffices to prove:

1. $\Omega' \vDash \langle\langle P,\mathrm{pc}+1,L\rangle C,M,G,v::S\rangle \propto \iota : \mathit{strong}$, which follows from Lemma 6 (Monotonicity for Strong Property) with HS2 since pc+1 does not play a role in the properties.
2. $\Omega' \vDash \langle\langle P,\mathrm{pc}+1,[]\rangle,\ell \mapsto v,\langle a,\rho\rangle \mapsto \ell,[]\rangle \propto \iota : \mathit{strong}$
   By Rule Strong Property we have to prove:
   (a) $\Omega' \vDash \langle\langle P,\mathrm{pc}+1,[]\rangle,\ell \mapsto v,\langle a,\rho\rangle \mapsto \ell,[]\rangle \propto \iota : \mathit{local}$
   Since the reduction is in attacker code, (Pis attacker-defined), we have HPA: ρ is an attacker-defined type. Since $\langle a,\rho\rangle$ is fresh by HPA, $\langle a,\rho\rangle$ cannot be in $\texttt{domG}\,(\iota)$. Since ℓ is fresh, no existing $\langle a',\rho'\rangle$ can ever have pointed to it, so ℓ cannot be in $\texttt{domM}\,(\iota)$.
   Thus this case holds.
   (b) $\Omega' \vDash \langle\langle P,\mathrm{pc}+1,[]\rangle,\ell \mapsto v,\langle a,\rho\rangle \mapsto \ell,[]\rangle \propto \iota : \mathit{unreachable}$
   By Rule Weak Property - Atk Changes we need to prove
   i. $G_i \not\frown \langle a,\rho\rangle$
   Since ρ is an attacker-defined type, it matches $\texttt{atktypes}\,(\Omega)$.
   From Property 3 we know no element in G_i can mention ρ, so this case holds.
   ii. $M_i \not\frown \ell$
   This follows from the freshess of ℓ.
3. $\forall a \in \mathit{img}(L').\,a \in \texttt{dom}\,(M')$
   this is trivial from HPL
4. $\forall \langle a,\rho\rangle \in \texttt{dom}\,(G').\,G'(\langle a,\rho\rangle) \in \texttt{dom}\,(M')$
   this is trivial from HPG
5. $\forall v \in \mathit{img}(M').\,v \in S$
   this is true from HPM and definition of S.
6. $\texttt{dom}\,(M) \cap c = \varnothing$
   This is trivially true.

Rule [MoveFrom] We have $i = \textbf{MoveFrom}\,\langle s\rangle$ in

$$\frac{\rho = \langle P.mid, s\rangle \quad G(\langle a, \rho\rangle) = \ell \quad M(\ell) = v}{\frac{P, \textbf{MoveFrom}\,\langle s\rangle \vdash \langle M, G, a::S\rangle \rightarrow_{glob} \langle M \setminus \ell, G \setminus \langle a, s\rangle, v::S\rangle}{\Omega \vdash \langle\langle P, \mathrm{pc}, L\rangle::C, M, G, a::S\rangle \rightarrow \langle\langle P, \mathrm{pc}+1, L\rangle::C, M \setminus \ell, G \setminus \langle a, s\rangle, v::S\rangle}}$$

This follows the same structure as the case for BorrowGlobal except for the details that we describe below.
We need to prove Item 2 for $L' = []$ and $S' = v$.
Item 2 is trivial since v cannot be a location.

Rule [Unpack] We have $i = \textbf{Unpack}\,\langle s\rangle$ in

$$\frac{v = \langle \{(f_i, v_i) \mid 1 \leq i \leq n\}, t\rangle}{\frac{P, \textbf{Unpack}\,\langle s\rangle \vdash \langle M, G, v::S\rangle \rightarrow_{glob} \langle M, G, v_1::\cdots::v_n::S\rangle}{\Omega \vdash \langle\langle P, \mathrm{pc}, L\rangle::C, M, G, v::S\rangle \rightarrow \langle\langle P, \mathrm{pc}+1, L\rangle::C, M, G, v_1::\cdots::v_n::S\rangle}}$$

This follows the same structure as the case for BorrowGlobal except for the details that we describe below.
We need to prove Item 2 for $L' = []$ and $S' = v_1::\cdots::v_n$.
Item 2 is simple since all locations were already in attacker locations from HS2.

Rule [Pack] We have $i = \textbf{Unpack}\,\langle s\rangle$ in

$$\frac{t = gen\_tag(\Omega(\langle P.mid, s\rangle).kind) \quad v = \{(f_i, v_i) \mid 1 \leq i \leq n\}}{\frac{P, \textbf{Pack}\,\langle s\rangle \vdash \langle M, G, v_1::\cdots::v_n::S\rangle \rightarrow_{glob} \langle M, G, \langle v, t\rangle::S\rangle}{\Omega \vdash \langle\langle P, \mathrm{pc}, L\rangle::C, M, G, v_1::\cdots::v_n::S\rangle \rightarrow \langle\langle P, \mathrm{pc}+1, L\rangle::C, M, G, \langle v, t\rangle::S\rangle}}$$

This follows the inverse reasoning of the case above, the tag is irrelevant to our proof.

Rule [Step-Loc] We perform case analysis on local reductions:

Rule [Op] We have $i = \textbf{Op}$ in

$$\frac{v'' = [[v\ \textbf{Op}\ v']]}{\frac{\textbf{Op} \vdash \langle M, L, v::v'::S\rangle \rightarrow_{loc} \langle M, L, v''::S\rangle}{\Omega \vdash \langle\langle P, \mathrm{pc}, L\rangle::C, M, G, v::v'::S\rangle \rightarrow \langle\langle P, \mathrm{pc}+1, L\rangle::C, M, G, v''::S\rangle}}$$

We need to prove:

- $\Omega' \vDash \langle\langle P, \mathrm{pc}+1, L\rangle::C, M, G, v''::S\rangle \propto \iota : strong$

By Lemma 8 (Compositionality of Strong Property and Weak Property Conditions) it suffices to prove for $L' = []$ and $S' = v''$:

1. $\Omega' \vDash \langle\langle P, \mathrm{pc}+1, L\rangle C, M, G, S\rangle \propto \iota : strong$, which follows from Lemma 6 (Monotonicity for Strong Property) with HS2 since pc+1 does not play a role in the properties.
2. $\forall \ell \in \texttt{locsof}\,(L') \cup \texttt{locsof}\,(S').\texttt{dom}\,(M_i)\ \not\upharpoonright c$
   This is not possible since the return type is not a location.

Rule [LoadConst] This follows the same structure as the case for Op except for the details that we describe below.
We need to prove Item 2 for $L' = []$ and $S' = v$.
This is trivial since v can be a Nat, a Bool or a global address only.
Item 2 is trivial since v cannot be a location.

Rule [Pop] We have $i = \textbf{Pop}$ in

$$\frac{\textbf{Pop} \vdash \langle M, L, v::S\rangle \rightarrow_{loc} \langle M, L, S\rangle}{\Omega \vdash \langle\langle P, \mathrm{pc}, L\rangle::C, M, G, v::S\rangle \rightarrow \langle\langle P, \mathrm{pc}+1, L\rangle::C, M, G, S\rangle}$$

We need to prove:

- $\Omega' \vDash \langle\langle P, \mathrm{pc}+1, L\rangle::C, M, G, S\rangle \propto \iota : strong$

This follows from Lemma 6 (Monotonicity for Strong Property) with HS2.

Rule [Writeref] We have $i = \textbf{WriteRef}$ in

$$\frac{v_2 = \langle \ell, p\rangle \quad v' = M(\ell)}{\frac{\textbf{WriteRef} \vdash \langle M, L, v_1::v_2::S\rangle \rightarrow_{loc} \langle M[\ell \mapsto v'[p := v_1]], L, S\rangle}{\Omega \vdash \langle\langle P, \mathrm{pc}, L\rangle::C, M, G, v_1::v_2::S\rangle \rightarrow \langle\langle P, \mathrm{pc}+1, L\rangle::C, M[\ell \mapsto v'[p := v_1]], G, S\rangle}}$$

We need to prove:

- $\Omega' \vDash \langle\langle P, \mathrm{pc}+1, L\rangle::C, M[\ell \mapsto v'[p := v_1]], G, S\rangle \propto \iota : strong$

By Lemma 6 (Monotonicity for Strong Property) it suffices to prove:

- $\Omega' \vDash \langle\langle P, \mathrm{pc}+1, L\rangle::C, M[\ell \mapsto v'[p := v_1]], G, v_1::S\rangle \propto \iota : strong$

By Lemma 7 (Compositionality of Strong Property) with HPL $L' = []$, HPM $M' = \ell \mapsto v'[p := v_1]$, HPG $G' = []$ it suffices to prove

1. $\Omega' \vDash \langle\langle P, \mathrm{pc}+1, L\rangle C, M, G, v::S\rangle \propto \mathfrak{l} : \mathit{strong}$, which follows from Lemma 6 (Monotonicity for Strong Property) with HS2 since pc+1 does not play a role in the properties.
2. $\Omega' \vDash \langle\langle P, \mathrm{pc}+1, []\rangle, \ell \mapsto v'[p := v_1], [], []\rangle \propto \mathfrak{l} : \mathit{strong}$
   By Rule Strong Property we have to prove:
   (a) $\Omega' \vDash \langle\langle P, \mathrm{pc}+1, []\rangle, \ell \mapsto v'[p := v_1], [], []\rangle \propto \mathfrak{l} : \mathit{local}$
   By Rule Weak Property - Inv we have to prove that $\ell \mapsto v'[p := v_1] \vdash \mathfrak{l}$.
   From HS2 we get that ℓ is not an address with an invariant, so this point holds since Rule Invariant Satisfaction only considers locations with an invariant.
   (b) $\Omega' \vDash \langle\langle P, \mathrm{pc}+1, []\rangle, \ell \mapsto v'[p := v_1], [], []\rangle \propto \mathfrak{l} : \mathit{unreachable}$
   By Rule Weak Property - Atk Changes we need to prove
   i. $G_i \not\parallel G_a$
   This is trivial since G_a did not change from HS2.
   ii. $M_i \not\parallel c$.
   From HS2 we have $M_i \not\parallel M_a$.
   Since ℓ was already in M, HS2 tells us that $\ell \in M_a$, so this case holds.
3. $\forall a \in \mathit{img}(L').\, a \in \mathtt{dom}(M')$
   this is trivial from HPL
4. $\forall \langle a, \rho\rangle \in \mathtt{dom}(G').\, G'(\langle a, \rho\rangle) \in \mathtt{dom}(M')$
   this is trivial from HPG
5. $\forall v \in \mathit{img}(M').\, v \in S$
   this is true from HPM and definition of S.
6. $\mathtt{dom}(M) \cap c = \varnothing$
   This is trivially true.

Rule [ReadRef] We have $i =$ **ReadRef** in

$$\frac{\dfrac{v = \langle \ell, p\rangle \quad \ell \in \mathtt{dom}(M) \quad M(\ell)[p] = v_p}{\textbf{ReadRef} \vdash \langle M, L, v::S\rangle \rightarrow_{loc} \langle M, L, v_p::S\rangle}}{\Omega \vdash \langle\langle P, \mathrm{pc}, L\rangle::C, M, G, v::S\rangle \rightarrow \langle\langle P, \mathrm{pc}+1, L\rangle::C, M, G, v_p::S\rangle}$$

We need to prove:

- $\Omega' \vDash \langle\langle P, \mathrm{pc}+1, L\rangle::C, M, G, v_p::S\rangle \propto \mathfrak{l} : \mathit{strong}$

By Lemma 8 (Compositionality of Strong Property and Weak Property Conditions) it suffices to prove for $L' = []$ and $S' = v_p$:

1. $\Omega' \vDash \langle\langle P, \mathrm{pc}+1, L\rangle C, M, G, S\rangle \propto \mathfrak{l} : \mathit{strong}$, which follows from Lemma 6 (Monotonicity for Strong Property) with HS2 since pc+1 does not play a role in the properties.
2. $\forall \ell \in \mathtt{locsof}(L') \cup \mathtt{locsof}(S').\, \mathtt{dom}(M_i) \not\parallel c$
   This is not possible since v_p is stored in memory.

Rule [Borrowfld] We have $i =$ **BorrowFld** in

$$\frac{\dfrac{v = \langle \ell, p\rangle \quad \ell \in \mathtt{dom}(M) \quad M(\ell)[p] = \langle \{(f, v_f), \cdots\}, t\rangle}{\textbf{BorrowFld}\,\langle f\rangle \vdash \langle M, L, v::S\rangle \rightarrow_{loc} \langle M, L, \langle \ell, p::f\rangle::S\rangle}}{\Omega \vdash \langle\langle P, \mathrm{pc}, L\rangle::C, M, G, v::S\rangle \rightarrow \langle\langle P, \mathrm{pc}+1, L\rangle::C, M, G, \langle \ell, p::f\rangle::S\rangle}$$

This follows the same structure as the case for ReadRef except for the details that we describe below.
We need to prove Item 2 for $L' = []$ and $S' = \langle \ell, p::f\rangle$.
From HP on v and from HS2 and Rule Weak Property - Atk Changes we know that all attacker memory locations (M_a) do not have an invariant on ($M_a \cap M_i = \varnothing$).
So c is not a memory address with an invariant on and Item 2 holds.

Rule [BorrowLoc] We have $i =$ **BorrowLoc** in

$$\frac{\dfrac{L(x) = \ell}{\textbf{BorrowLoc}\,\langle x\rangle \vdash \langle M, L, S\rangle \rightarrow_{loc} \langle M, L, \langle \ell, []\rangle::S\rangle}}{\Omega \vdash \langle\langle P, \mathrm{pc}, L\rangle::C, M, G, S\rangle \rightarrow \langle\langle P, \mathrm{pc}+1, L\rangle::C, M, G, \langle \ell, []\rangle::S\rangle}$$

This follows the same structure as the case for ReadRef except for the details that we describe below.
We need to prove Item 2 for $L' = [\,]$ and $S' = \langle \ell, [] \rangle$.
From HP, we have $L(x) = \ell$ and from HS2 and Rule Weak Property - Atk Changes we know that all attacker memory locations (M_a) do not have an invariant on ($M_a \cap M_i = \varnothing$).
So c is not a memory address with an invariant on and Item 2 holds.

Rule [StoreLoc-Ref] We have $i = \textbf{StLoc}$ in

$$\frac{v \in \mathsf{Reference}}{\textbf{StLoc}\,\langle x \rangle \vdash \langle M, L, v{::}S \rangle \rightarrow_{loc} \langle M, L[x \mapsto v], S \rangle} \over \Omega \vdash \langle \langle P, \mathrm{pc}, L \rangle {::} C, M, G, v{::}S \rangle \rightarrow \langle \langle P, \mathrm{pc}+1, L[x \mapsto v] \rangle {::} C, M, G, S \rangle$$

This follows the same structure as the case for ReadRef except for the details that we describe below.
We need to prove Item 2 for $L' = x \mapsto v$ and $S' = [\,]$.
Item 2 is simple since v is a location that was considered in HS2, so it is disjoint from M_i.

Rule [StoreLoc] We have $i = \textbf{StLoc}$ in

$$\frac{v \in \mathsf{StorableValue} \quad \ell \notin \mathtt{dom}(M) \quad M' = M \setminus L(x) \text{ if } L(x) \in \mathtt{dom}(M) \text{ else } M}{\textbf{StLoc}\,\langle x \rangle \vdash \langle M, L, v{::}S \rangle \rightarrow_{loc} \left\langle M'_{C \setminus \ell}[\ell \mapsto v], L[x \mapsto \ell], S \right\rangle} \over \Omega \vdash \langle \langle P, \mathrm{pc}, L \rangle {::} C, M, G, v{::}S \rangle \rightarrow \left\langle \langle P, \mathrm{pc}+1, L[x \mapsto \ell] \rangle {::} C, M'_{C \setminus \ell}[\ell \mapsto v], G, S \right\rangle$$

We need to prove:

- $\Omega' \vDash \left\langle \langle P, \mathrm{pc}+1, L[x \mapsto \ell] \rangle {::} C, M'_{C \setminus \ell}[\ell \mapsto v], G, S \right\rangle \propto \mathfrak{l} : \mathit{strong}$

By Lemma 6 (Monotonicity for Strong Property) it suffices to prove:

- $\Omega' \vDash \left\langle \langle P, \mathrm{pc}+1, L[x \mapsto \ell] \rangle {::} C, M'_{C \setminus \ell}[\ell \mapsto v], G, v{::}S \right\rangle \propto \mathfrak{l} : \mathit{strong}$

By Lemma 7 (Compositionality of Strong Property) with HPL $L' = x \mapsto \ell$, HPM $M' = \ell \mapsto v$, HPG $G' = [\,]$ it suffices to prove

1. $\Omega' \vDash \langle \langle P, \mathrm{pc}+1, L \setminus x \mapsto \ell \rangle C, M \setminus \ell \mapsto v, G, v{::}S \rangle \propto \mathfrak{l} : \mathit{strong}$, which follows from Lemma 6 (Monotonicity for Strong Property) with HS2 since pc+1 does not play a role in the properties.
2. $\Omega' \vDash \langle \langle P, \mathrm{pc}+1, x \mapsto \ell \rangle, [], \ell \mapsto v, [] \rangle \propto \mathfrak{l} : \mathit{strong}$
   By Rule Strong Property we have to prove:
   (a) $\Omega' \vDash \langle \langle P, \mathrm{pc}+1, x \mapsto \ell \rangle, [], \ell \mapsto v, [] \rangle \propto \mathfrak{l} : \mathit{local}$
   By Rule Weak Property - Inv we have to prove that $\ell \mapsto v \vdash \mathfrak{l}$.
   By Rule Invariant Satisfaction we have that $\ell \notin M_i$ because there are no globals pointing to ℓ since ℓ is fresh, so this case holds.
   (b) $\Omega' \vDash \langle \langle P, \mathrm{pc}+1, x \mapsto \ell \rangle, [], \ell \mapsto v, [] \rangle \propto \mathfrak{l} : \mathit{unreachable}$
   By Rule Weak Property - Atk Changes we need to prove
   i. $G_i \not\upharpoonright G_a$ this is trivial since G_a does not change from HS2.
   ii. $M_i \not\upharpoonright \ell$
   This follows from the freshess of ℓ.
3. $\forall a \in \mathit{img}(L').\, a \in \mathtt{dom}(M')$
   this is trivial from HPL
4. $\forall \langle a, \rho \rangle \in \mathtt{dom}(G').\, G'(\langle a, \rho \rangle) \in \mathtt{dom}(M')$
   this is trivial from HPG
5. $\forall v \in \mathit{img}(M').\, v \in S$
   this is true from HPM and definition of S.
6. $\mathtt{dom}(M) \cap c = \varnothing$
   This is trivially true.

Rule [CopyLocRef] We have $i = \textbf{CpLoc}$ in

$$\frac{v = L(x) = \langle \ell, p \rangle}{\textbf{CpLoc}\,\langle x \rangle \vdash \langle M, L, S \rangle \rightarrow_{loc} \langle M, L, v{::}S \rangle} \over \Omega \vdash \langle \langle P, \mathrm{pc}, L \rangle {::} C, M, G, S \rangle \rightarrow \langle \langle P, \mathrm{pc}+1, L \rangle {::} C, M, G, v{::}S \rangle$$

This follows the same structure as the case for ReadRef except for the details that we describe below.
We need to prove Item 2 for $L' = []$ and $S' = v$.
Item 2 is simple because its locations were considered in HS2.

Rule [CopyLoc] We have $i = \mathbf{CpLoc}$ in

$$\frac{\dfrac{L(x) = \ell \quad \ell \in \mathtt{dom}(M)}{\mathbf{CpLoc}\,\langle x\rangle \vdash \langle M, L, S\rangle \rightarrow_{loc} \langle M, L, M(\ell)::S\rangle}}{\Omega \vdash \langle\langle P, \mathrm{pc}, L\rangle::C, M, G, S\rangle \rightarrow \langle\langle P, \mathrm{pc}+1, L\rangle::C, M, G, M(\ell)::S\rangle}$$

This follows the same structure as the case for ReadRef except for the details that we describe below.
We need to prove Item 2 for $L' = []$ and $S' = M(\ell)$.
Item 2 is trivial since there are no locations.

Rule [MoveLocRef] We have $i = \mathbf{MvLoc}$ in

$$\frac{\dfrac{L(x) = \langle \ell, p\rangle}{\mathbf{MvLoc}\,\langle x\rangle \vdash \langle M, L, S\rangle \rightarrow_{loc} \langle M, L \setminus x, L(x)::S\rangle}}{\Omega \vdash \langle\langle P, \mathrm{pc}, L\rangle::C, M, G, S\rangle \rightarrow \langle\langle P, \mathrm{pc}+1, L \setminus x\rangle::C, M, G, L(x)::S\rangle}$$

This follows the same structure as the case for ReadRef except for the details that we describe below.
We need to prove Item 2 for $L' = []$ and $S' = L(x)$.
Item 2 is simple because its locations were considered in HS2.

Rule [MoveLoc] We have $i = \mathbf{MvLoc}$ in

$$\frac{\dfrac{L(x) = \ell \quad \ell \in \mathtt{dom}(M)}{\mathbf{MvLoc}\,\langle x\rangle \vdash \langle M, L, S\rangle \rightarrow_{loc} \langle M \setminus \ell, L \setminus x, M(\ell)::S\rangle}}{\Omega \vdash \langle\langle P, \mathrm{pc}, L\rangle::C, M, G, S\rangle \rightarrow \langle\langle P, \mathrm{pc}+1, L \setminus x\rangle::C, M \setminus \ell, G, M(\ell)::S\rangle}$$

This follows the same structure as the case for ReadRef except for the details that we describe below.
We need to prove Item 2 for $L' = []$ and $S' = M(\ell)$.
Item 2 is trivial since there are no locations.

□

High-level intuition: when the attacker creates new memory locations (Moveto, Storeloc), they are fresh and therefore there cannot be any existing invariant on them. When the attacker creates a new global (Moveto), it can only be with an attacker type, and thus it cannot be a global location with an invariant on. Any literal that the attacker creates (Op, LoadConst) can be the address part of a global, and that is ok. When that literal will be used, its type part makes it so that the global accessed with that address does not have invariants on. All other rules move existing values around, package them in structures and access the fields of those structures, so they do not alter the attacker capabilities.

The attacker also cannot craft locations nor globals that the trusted code will set an invariant on: the globals created by trusted code will not have attacker types but trusted code types, and the only way to pass a location is via globals, but the contents of globals are never read direcly, globals are just used as a proxy to those memory locations they point to, so the attacker-created location cannot reach trusted code because it is not stored in the memory.

One concern may be: what if an attacker crafts an address, and then passes it as a parameter to the trusted code, which then uses it naively and violates its invariants? This is addressed by the local verification and the encapsulation.

♠

The next lemma tells that reductions in the trusted code (Lemma 10 (trusted code Reductions Respect the Strong Property)) respect the strong property.

Lemma 10 (trusted code Reductions Respect the Strong Property)**.**

$$\begin{aligned}
&\text{if } \Omega' \triangleright \Omega, P \vdash \sigma \overset{\alpha!}{\Longrightarrow} \sigma' \\
&\text{and } \Omega' \frown \iota \\
&\text{and } \Omega' \vdash \sigma : \mathit{modcode} \\
&\text{and } \Omega \vDash \sigma \propto \iota : \mathit{strong} \\
&\text{then } \Omega' \vDash \sigma' \propto \iota : \mathit{strong} \\
&\text{and } \alpha! \Vdash \iota
\end{aligned}$$

Proof of Lemma 10 (trusted code Reductions Respect the Strong Property).

By Definition 3 we have $\Omega' \vDash \sigma_1 \propto \iota : \mathit{local}$ (HW1) and $\alpha! \Vdash \iota$, the second conjunct of the thesis, which is now proven and we need to prove just the first one.

By Definition 4 we have $\Omega' \vDash \sigma_2 \propto \iota : \mathit{unreachable}$ (HW2).

By Lemma 11 (Determinism of the Semantics) we conclude that the final states are the same so $\sigma_1 = \sigma_2 = \sigma'$.

By Lemma 1 (The Two Weak Properties Imply the Strong One) with HW1 and HW2 we have $\Omega' \vDash \sigma' \propto \iota : \mathit{strong}$. □

♠

An additional result we need is that semantics are deterministic.

Lemma 11 (Determinism of the Semantics)**.**

$$\begin{aligned} &\text{if } \Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{\alpha!} \sigma' \\ &\text{and } \Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{\alpha!} \sigma'' \\ &\text{then } \sigma' = \sigma'' \end{aligned}$$

Proof of Lemma 11.

Follows directly from the semantics being deterministic. □

♠

B.8.4 Main Result

Lemma 12 (Generalised RS)**.**

$$\begin{aligned} &\forall A, P, \overline{\alpha}, \sigma'. \\ &\text{if } \Omega \vdash A : \mathit{atk} \\ &\text{and } \Omega' \frown \iota \\ &\text{and } \Omega \vDash \sigma \propto \iota : \mathit{strong} \\ &\text{and } \Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{\overline{\alpha}} \sigma' \\ &\text{then } \Omega \vDash \sigma' \propto \iota : \mathit{strong} \\ &\text{and } \overline{\alpha} \Vdash \iota : \mathit{global} \end{aligned}$$

The conditions on P, i and Ω' are implicit: if the configuration steps accroring to $\Rightarrow\!\!\!\!\Rightarrow$, then P is a valid procedure in Ω' and i is a valid pc location in P (i.e., it is either the start of a function or the address right after a call).

Proof of Lemma 12 (Generalised RS).

This proof proceeds by induction on $\Rightarrow\!\!\!\!\Rightarrow$.

Base, Rule Trace-Refl: This case trivially holds by Rule Global-check-base since the generated action is $[]$.

Inductive: Depending on the rule, we have two cases:

Rule Trace-Both so we have:

$\Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{\overline{\alpha'}} \sigma''$ By IH we have $\Omega' \vDash \sigma'' \propto \iota : \mathit{strong}$ (HB) and $\overline{\alpha'} \Vdash \iota : \mathit{global}$ (HT).

$\Omega' \triangleright \Omega, P \vdash \sigma'' \xRightarrow{\alpha?} \sigma'''$ By Lemma 3 (Attacker Reductions Respect the Strong Property) with HB we have $\Omega' \vDash \sigma''' \propto \iota : \mathit{strong}$ (HA) and $\alpha? \Vdash \iota$ (HT1).

$\Omega' \triangleright \Omega, P \vdash \sigma''' \xRightarrow{\alpha!} \sigma'$ By Lemma 10 (trusted code Reductions Respect the Strong Property) with HA we have $\Omega' \vDash \sigma' \propto \iota : \mathit{strong}$ and $\alpha! \Vdash \iota$ (HT2).
By two applications of Rule Global-check-ind with HT and HT1 first, and then with HT2, the thesis holds.

Rule Trace-Single This case is analogous to half of the previous one.

□

♠

B.9 Examples

```
module M {
 // invariant: f > 0
 resource struct Counter { f: u64 }

 public fun create(): Counter {
  Counter { f: 1 }
 }

 public fun add(account: &signer, c: Counter) {
  move_to(account, c)
 }

 public fun remove(a: address): Counter {
  move_from(a)
 }

 public fun read(c: &Counter): &u64 {
  &c.f
 }

 public fun increment(c: &mut Counter) {
  c.f = *c.f + 1
 }

 // Violates the encapsulator!
 public fun read_mut(c: &mut Counter): &mut u64 {
  &mut c.f
 }
}
```

In the code above, a modular static analysis can prove that the invariant `f > 1` holds locally for all `Counter` instances, but the invariant does *not* hold in the presence of attacker code because of the `read_mut` function. Specifically, the attacker can do

```
let c = M::create();
let x = M::read_mut(&mut c);
*x = 0;
```

The encapsulator must reject `read_mut` because it "leaks" the internal state of the `Counter` resource. If this function is removed, the `f > 1` invariant will hold globally.

C Encapsulator Instances as Intraprocedural Escape Analyses

In this section, we define a simple intraprocedural escape analysis that satisfies Definition 4. The analysis abstracts the concrete values bound to local variables and stack locations using a lattice with three abstract values: NonRef, OkRef, InternalRef. We define NonRef $\sqsubseteq$ InternalRef and OkRef $\sqsubseteq$ InternalRef. Inuitively, NonRef represents any non-reference value, OkRef represents a reference that does not point inside of a resource defined in the current module, and InternalRef represents a reference that *may* point inside a resource defined in the current module. The purpose of the analysis is to prevent an InternalRef from "leaking" to a caller of the module via a **Ret**. Because Move structs cannot store references, this is the only way such a leak can occur.

High-Level Description

The analysis does the following:

- Initialize each reference parameter to OkRef and each non-reference parameter to NonRef
- Applying **BorrowFld** to a OkRef value produces an InternalRef value
- A **BorrowGlobal** produces an InternalRef value
- A **BorrowLoc** produces an OkRef value
- Applying a **Ret** instruction to an InternalRef is an error
- A **Call** marks all non-reference return values as NonRef. If the call accepts any InternalRef arguments, all reference return values are marked as InternalRef. Otherwise, all reference return values are marked as OkRef.
- All other instructions either propagate their input values or introduce NonRef values

We note that returning references to locals (e.g., `let x = param; return &x` and globals (e.g., `return borrow_global<T>(a)`) is already ruled out by the Move bytecode verifier and thus cannot leak an internal reference. However, we define our escape analysis as if these behaviors were possible in order to minimize dependence on the bytecode verifier invariants in our proof.

C.1 Encapsulator Ξ_{ea}: an Escape Analysis for Integrity

We use the symbol Ξ_{ea} to indicate an escape analysis that can be used as an encapsulator. In the rules below, we omit the Ω, P, ι on the left of the $\vdash$ when they are not necessary. This allows us to focus the escape analysis only on fields that are relevant to the key safety invariants.

$$\text{(BorrowFld-InvRelevant)}\ \frac{f \in \iota}{\iota, \mathbf{BorrowFld}\,\langle f\rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \mathsf{InternalRef}::\hat{S}\right\rangle} \qquad \text{(BorrowFld-InvIrrelevant)}\ \frac{f \notin \iota}{\iota, \mathbf{BorrowFld}\,\langle f\rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle}$$

$$\text{(BorrowGlobal)}\ \frac{}{\mathbf{BorrowGlobal}\,\langle s\rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \mathsf{InternalRef}::\hat{S}\right\rangle} \qquad \text{(BorrowLoc)}\ \frac{}{\mathbf{BorrowLoc}\,\langle x\rangle \vdash \left\langle \hat{L}, \hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \mathsf{OkRef}::\hat{S}\right\rangle}$$

$$\text{(Return)}\ \frac{\|\Omega(P).3\| = n \qquad \forall i \in 1..n.\, \hat{v}_i \neq \mathsf{InternalRef}}{\mathbf{Ret} \vdash \left\langle \hat{L}, \hat{v_1} :: \hat{v_n}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{v_1} :: \hat{v_n}::\hat{S}\right\rangle}$$

$$\text{(Call)}\ \frac{\begin{array}{c}\|\Omega(P_0).type\| = n \qquad \|\Omega(P_0).3\| = j \\ \forall i \in 1..j.\, \hat{v_i^r} = (\mathsf{NonRef}\ \text{if}\ \Omega(P).type(i) \neq \texttt{ref}) \\ \vee \hat{v_i^r} = (\mathsf{OkRef}\ \text{if}\ \Omega(P).type(i) = \texttt{ref} \wedge \forall m \in 1..n, v_m^a \neq \mathsf{InternalRef}) \\ \vee \hat{v_i^r} = (\mathsf{InternalRef}\ \text{if}\ \Omega(P).type(i) = \texttt{ref} \wedge \exists m \in 1..n, v_m^a = \mathsf{InternalRef})\end{array}}{\Omega, P, \iota, \mathbf{Call}\,\langle P_0\rangle \vdash \left\langle \hat{L}, \hat{v_1^a} \cdots \hat{v_n^a}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{v_1^r} \cdots \hat{v_j^r}::\hat{S}\right\rangle}$$

$$\text{(MoveLoc)}\ \frac{\hat{L}(x) = \hat{v}}{\mathbf{MvLoc}\,\langle x\rangle \vdash \left\langle \hat{L}, \hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L} \setminus x, \hat{v}::\hat{S}\right\rangle}$$

$$\text{(CopyLoc)}\ \frac{\hat{L}(x) = \hat{v}}{\mathbf{CpLoc}\,\langle x\rangle \vdash \left\langle \hat{L}, \hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle} \qquad \text{(StoreLoc)}\ \frac{}{\mathbf{StLoc}\,\langle x\rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}\left[x \mapsto \hat{v}\right], \hat{S}\right\rangle}$$

$$\text{(WriteRef)}\ \frac{}{\mathbf{WriteRef} \vdash \left\langle \hat{L}, \hat{v_1}::\hat{v_2}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{S}\right\rangle} \qquad \text{(ReadRef)}\ \frac{}{\mathbf{ReadRef} \vdash \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \textsc{Ok}::\hat{S}\right\rangle} \qquad \text{(Pop)}\ \frac{}{\mathbf{Pop} \vdash \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{S}\right\rangle}$$

$$\text{(LoadConst)}\ \frac{}{\mathbf{LoadConst}\,\langle v\rangle \vdash \left\langle \hat{L}, \hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \mathsf{NonRef}::\hat{S}\right\rangle} \qquad \text{(Op)}\ \frac{}{\mathbf{Op} \vdash \left\langle \hat{L}, \hat{v_1}::\hat{v_2}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \mathsf{NonRef}::\hat{S}\right\rangle}$$

$$\text{(MoveTo)}\ \frac{}{\mathbf{MoveTo}\,\langle s\rangle \vdash \left\langle \hat{L}, \hat{v_1}::\hat{v_2}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{S}\right\rangle} \qquad \text{(MoveFrom)}\ \frac{}{\mathbf{MoveFrom}\,\langle s\rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \mathsf{NonRef}::\hat{S}\right\rangle}$$

$$\text{(Pack)}\ \frac{}{\mathbf{Pack}\,\langle s\rangle \vdash \left\langle \hat{L}, \hat{v_1}::\cdots::\hat{v_n}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \mathsf{NonRef}::\hat{S}\right\rangle} \qquad \text{(Unpack)}\ \frac{}{\mathbf{Unpack}\,\langle s\rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{v_1}::\cdots::\hat{v_n}::\hat{S}\right\rangle}$$

$$\text{(Branch)}\ \frac{}{\mathbf{Branch}\,\langle \mathrm{pc}\rangle \vdash \left\langle \hat{L}, \hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{S}\right\rangle} \qquad \text{(BranchCond)}\ \frac{}{\mathbf{BranchCond}\,\langle \mathrm{pc}\rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{S}\right\rangle}$$

$$\text{(Module-instruction list)}\ \frac{\Omega, P, P(\mathrm{pc}) \vdash \left\langle \hat{L}, \hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L''}, \hat{S''}\right\rangle \qquad \Omega, P \vdash \left\langle \mathrm{pc}+1, \hat{L''}, \hat{S''}\right\rangle \twoheadrightsquigarrow \left\langle \mathrm{pc}', \hat{L'}, \hat{S'}\right\rangle}{\Omega, P \vdash \left\langle \mathrm{pc}, \hat{L}, \hat{S}\right\rangle \twoheadrightsquigarrow \left\langle \mathrm{pc}', \hat{L'}, \hat{S'}\right\rangle}$$

$$\text{(Module-top)}\ \frac{\forall P \in \Omega'.\Omega', P \vdash \left\langle 0, \mathtt{absty}\,(\Omega(P).1), \hat{[]}\right\rangle \twoheadrightsquigarrow \left\langle \|\Omega(P).code\|, \hat{L'}, \mathtt{absty}\,(\Omega(P).2)\right\rangle}{\Xi(\Omega')}$$

We indicate the size of the list of instructions in a procedure declaration as $\|\cdot\|$. Recall that in a procedure declaration, the first type is the input type and the second one is the return type.

The analysis in the branch case goes through the whole code, essentially first passing over the else branch, and then over the then branch. Since here the control flow does not do a branch, our analysis' linear pass checks both branches.

This auxiliary functions returns the abstract values for all possible types.

$$\begin{aligned} \texttt{absty}(z) &= \mathsf{NonRef} && z \in \mathit{GroundType} \vee z \in \mathit{RecordType} \vee z \in \mathit{StorableType} \\ \texttt{absty}(a) &= \mathsf{OkRef} && a \in \mathit{ReferenceType} \end{aligned}$$

Below is the concretisation function that takes an abstract state $\langle \hat{S}, \hat{L} \rangle$ and returns a set of possible states that realise that state. This, in turn, relies on a concretisation function for abstract values $\hat{v}$ to a set of possible values. Intuitively, NonRef concretizes to a non-reference value, OkRef concretizes to a reference value that does not contain a field offset defined in the current module, and InternalRef concretizes to a reference value with one or more offsets defined in the current module.

$$\begin{aligned} \gamma\left(\langle \hat{S}, \hat{L} \rangle, \Omega\right) &= \left\{ \langle C, M, G, S \rangle \;\middle|\; \begin{array}{l} C = \langle P, \mathrm{pc}, \gamma\left(\hat{L}, M, G, \Omega\right) \rangle :: C' \\ \text{and } S = \gamma\left(\hat{S}, M, G, \Omega\right) \end{array} \right\} \\ \gamma\left(\hat{[]}, M, G, \Omega\right) &= [] \\ \gamma\left(\hat{v} :: \hat{S}, M, G, \Omega\right) &= \gamma(\hat{v}, M, G, \Omega) :: \gamma\left(\hat{S}, M, G, \Omega\right) \\ \gamma\left(x \mapsto \hat{v} :: \hat{L}, M, G, \Omega\right) &= x \mapsto \gamma(\hat{v}, M, G, \Omega) :: \gamma\left(\hat{L}, M, G, \Omega\right) \\ \gamma(\mathsf{NonRef}, M, G, \Omega) &= \{ v \mid v \in \mathit{StorableValue} \} \\ \gamma(\mathsf{OkRef}, M, G, \Omega) &= \left\{ \ell \;\middle|\; \begin{array}{l} \ell \in \mathit{Reference} \text{ and } \ell \in \texttt{dom}(M) \text{ and} \\ M(\ell) = a. \forall \rho \in \Omega.\mathit{types}. \langle a, \rho \rangle \notin G \end{array} \right\} \\ \gamma(\mathsf{InternalRef}, M, G, \Omega) &= \left\{ \ell \;\middle|\; \begin{array}{l} \ell \in \mathit{Reference} \text{ and } \ell \in \texttt{dom}(M) \text{ and} \\ M(\ell) = a. \exists \rho \in \Omega.\mathit{types}. \langle a, \rho \rangle \in G \end{array} \right\} \end{aligned}$$

C.1.1 Auxiliaries for the Encapsulator

State pc This auxiliary function returns the current pc:

$$\sigma.\mathrm{pc} = n \text{ if } \sigma = \langle \langle P, n, L \rangle :: C, M, G, S \rangle$$

Entry-Exit Reductions We say that a reduction is *entry*/*exit* if it starts from the entry point of a function until the *next* exit point (Rule Entry/Exit). We identify entry points (Rule Entry) to be:

- the first instruction;
- any instruction after a call;

while exit points (Rule Exit) are:

- call instructions;
- the last instruction.

Recall that we indicate the size of a stack (e.g., the call stack C) with $\|C\|$.

$$\frac{(\sigma.\mathrm{pc} = 0) \vee (\Omega(\sigma.\mathrm{pc} - 1) = \mathbf{Call}\,\langle p \rangle)}{\Omega, P \vdash \sigma : \mathit{entry}}\ \text{(Entry)} \qquad \frac{(\sigma.\mathrm{pc} = \|\Omega(P).\mathit{code}\|) \vee (\Omega(\sigma.\mathrm{pc}) = \mathbf{Call}\,\langle p \rangle) \vee (\Omega(\sigma.\mathrm{pc}) = \mathbf{Ret})}{\Omega, P \vdash \sigma : \mathit{exit}}\ \text{(Exit)}$$

$$\frac{\forall \sigma''.\, \Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{[]} \sigma'' \Longrightarrow \sigma' \quad \|\sigma''.C\| = \|\sigma.C\| = \|\sigma'.C\| \quad \Omega, P \vdash \sigma : \mathit{entry} \quad \Omega, P \vdash \sigma' : \mathit{exit}}{\vdash \Omega' \triangleright \Omega, P \vdash \sigma \Longrightarrow \sigma' : \mathit{entry}/\mathit{exit}}\ \text{(Entry/Exit)}$$

$$\frac{\Omega, P \vdash \sigma : \mathit{exit} \quad \Omega, P \vdash \sigma' : \mathit{entry}}{\vdash \Omega' \triangleright \Omega, P \vdash \sigma \xrightarrow{[]} \sigma' : \mathit{exit}/\mathit{entry}}\ \text{(Exit/Entry)}$$

Definition 5 (Same-Domain Reductions can be Split into Entry/Exit Reductions)**.**

$$
\begin{aligned}
&\text{if } \Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{\alpha!} \sigma' \\
&\text{then } \exists n.\, \Omega' \triangleright \Omega, P \vdash \sigma'_0 \xRightarrow{[]} \sigma_1 \xrightarrow{[]} \sigma'_1 \xRightarrow{[]} \cdots \xrightarrow{[]} \sigma'_n \xRightarrow{[]} \sigma_{n+1} \xrightarrow{\alpha!} \sigma' \\
&\text{and } \sigma = \sigma'_0 \\
&\text{and } \forall i \in 0..n/2. \vdash \Omega' \triangleright \Omega, P \vdash \sigma'_{2i} \xRightarrow{[]} \sigma_{2i+1} : \mathit{entry/exit} \\
&\text{and } \vdash \Omega' \triangleright \Omega, P \vdash \sigma_{2i+1} \xrightarrow{[]} \sigma''_{2i+1} : \mathit{exit/entry}
\end{aligned}
$$

Entry-Exit Encapsulation Given that a code environment is encapsulated, we know that all its functions are also encapsulated from their entry to their exit points.

As for semantic reductions, we define what it means for encapsulator reductions to be $\mathit{entry/exit}$. This holds if the reduction starts from a state that concretizes to an entry point of a function and ends in a state that concretizes into the *next* exit point (Rule Entry/Exit Encapsulator).

(Entry/Exit Encapsulator)

$$
\frac{\begin{array}{c}\forall \left\langle \mathrm{pc}'', \hat{L}'', \hat{S}'' \right\rangle . \Omega, P \vdash \left\langle \mathrm{pc}, \hat{L}, \hat{S} \right\rangle \rightsquigarrow \left\langle \mathrm{pc}'', \hat{L}'', \hat{S}'' \right\rangle \rightsquigarrow \left\langle \mathrm{pc}', \hat{L}', \hat{S}' \right\rangle \\ \not\vdash \Omega, P \vdash \left\langle \mathrm{pc}'', \hat{L}'', \hat{S}'' \right\rangle : \mathit{exit} \\ \Omega, P \vdash \left\langle \mathrm{pc}, \hat{L}, \hat{S} \right\rangle : \mathit{entry} \qquad \Omega, P \vdash \left\langle \mathrm{pc}', \hat{L}', \hat{S}' \right\rangle : \mathit{exit}\end{array}}{\vdash \Omega, P \vdash \left\langle \mathrm{pc}, \hat{L}, \hat{S} \right\rangle \rightsquigarrow \left\langle \mathrm{pc}', \hat{L}', \hat{S}' \right\rangle : \mathit{entry/exit}}
$$

Definition 6 (Whole-Program Encapsulation Entails Entry/Exit Encapsulation)**.**

$$
\begin{aligned}
&\text{if } \Omega, P \vdash \left\langle \mathrm{pc}, \hat{L}, \hat{S} \right\rangle \rightsquigarrow \left\langle \mathrm{pc}', \hat{L}', \hat{S}' \right\rangle \\
&\text{then } \exists m. \Omega, P \vdash \left\langle \mathrm{pc}_0, \hat{L_0}, \hat{S_0} \right\rangle \rightsquigarrow \left\langle \mathrm{pc}_1, \hat{L_1}, \hat{S_1} \right\rangle \rightsquigarrow \cdots \rightsquigarrow \left\langle \mathrm{pc}_m, \hat{L_m}, \hat{S_m} \right\rangle \rightsquigarrow \left\langle \mathrm{pc}', \hat{L}', \hat{S}' \right\rangle \\
&\text{and } \left\langle \mathrm{pc}, \hat{L}, \hat{S} \right\rangle = \left\langle \mathrm{pc}_0, \hat{L_0}, \hat{S_0} \right\rangle \\
&\text{and } \forall j \in 0..m/2. \vdash \Omega, P \vdash \left\langle \mathrm{pc}_j, \hat{L_j}, \hat{S_j} \right\rangle \rightsquigarrow \left\langle \mathrm{pc}_{2j+1}, \hat{L_{2j+1}}, \hat{S_{2j+1}} \right\rangle : \mathit{entry/exit}
\end{aligned}
$$

C.1.2 Properties of Ξ_{ea}

Theorem 4 (The Escape Analysis is a Sound Encapsulator ($\iota \vdash \Omega' : \Xi_{ea}$))**.**

$$
\begin{aligned}
&\text{if } \Omega' \vDash \sigma \propto \iota : \mathit{strong} \\
&\text{and } \Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{\alpha!} \sigma' \\
&\text{and } \Xi_{ea}(\Omega'|_\iota) \\
&\text{then } \Omega' \vDash \sigma' \propto \iota : \mathit{unreachable}
\end{aligned}
$$

Proof. By Rule Strong Property with HP0 we have HPw $\Omega' \vDash \sigma \propto \iota : \mathit{unreachable}$.

By the semantics rules with HP2 we have:

- HPRF $\Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{[]} \sigma''$
- HPRA $\Omega' \triangleright \Omega, P \vdash \sigma'' \xrightarrow{\alpha!} \sigma'$

Additionally, by Rule Entry and Rule Exit we have:

- HPEN $\Omega, P \vdash \sigma : \mathit{entry}$
- HPEX $\Omega, P \vdash \sigma'' : \mathit{exit}$

By HP3 with Definition 6 (Whole-Program Encapsulation Entails Entry/Exit Encapsulation) we can split the encapsulation of the whole code into entry/exit encapsulations, so we have:

- $\vdash \Omega, P \vdash \langle \sigma.\mathrm{p}c, \hat{L}, \hat{S} \rangle \twoheadrightarrow \langle \sigma_1.\mathrm{p}c, \hat{L_1}, \hat{S_1} \rangle : \mathit{entry/exit}$

  This gives us HPS $\Omega, P \vdash \langle \sigma.\mathrm{p}c, \hat{L}, \hat{S} \rangle \twoheadrightarrow \langle \sigma_1.\mathrm{p}c, \hat{L_1}, \hat{S_1} \rangle$

- $\vdash \Omega, P \vdash \langle \sigma_2.\mathrm{p}c, \hat{L_2}, \hat{S_2} \rangle \twoheadrightarrow \langle \sigma''.\mathrm{p}c, \hat{L''}, \hat{S''} \rangle : \mathit{entry/exit}$

  This gives us HPS2 $\Omega, P \vdash \langle \sigma_2.\mathrm{p}c, \hat{L_2}, \hat{S_2} \rangle \twoheadrightarrow \langle \sigma''.\mathrm{p}c, \hat{L''}, \hat{S''} \rangle$

By well-typedness of code and Rule Module-top we get HPC: $\sigma \in \gamma(\langle \hat{S}, \hat{L} \rangle)$.
By Lemma 13 (Generalised Encapsulation) with HPw, HPRF, HPEN, HPEX, HPS, HPS2, HPC, HP3 we have

- HPP $\Omega' \vDash \sigma'' \propto \iota : \mathit{unreachable}$.
- HPC $\sigma'' \in \gamma(\langle \hat{S''}, \hat{L''} \rangle)$

By Lemma 14 (!Actions from Concretized States Preserve the Weak Invariant) with HPP, HPRA, HPS2, HPC we have what is needed.

□

♠

Lemma 13 (Generalised Encapsulation)**.**

$$
\begin{aligned}
&\text{if } \Omega' \vDash \sigma \propto \iota : \mathit{unreachable} \\
&\text{and } \Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{[]} \sigma' \\
&\text{and } \Omega, P \nvDash \sigma : \mathit{entry} \\
&\text{and } \Omega, P \nvDash \sigma' : \mathit{exit} \\
&\text{and } \sigma \in \gamma(\langle \hat{S}, \hat{L} \rangle) \\
&\text{and } \Omega, P \vdash \langle \sigma.\mathrm{p}c, \hat{L}, \hat{S} \rangle \twoheadrightarrow \langle \sigma''.\mathrm{p}c, \hat{L''}, \hat{S''} \rangle \\
&\text{and } \Omega, P \vdash \langle \sigma'''.\mathrm{p}c, \hat{L'''}, \hat{S'''} \rangle \twoheadrightarrow \langle \sigma'.\mathrm{p}c, \hat{L'}, \hat{S'} \rangle \\
&\text{and } \Xi_{ea}(\Omega'|_\iota) \\
&\text{then } \Omega' \vDash \sigma' \propto \iota : \mathit{unreachable} \\
&\text{and } \sigma' \in \gamma(\langle \hat{S'}, \hat{L'} \rangle)
\end{aligned}
$$

Proof. Take the reductions in HP2: they are all for functions in Ω. Also, they can be split into sequences of reductions from an entry point to the next exit point for each function. (intuitively, from the start of the reduction, code can call other functions in Ω until the attacker code is called or returned to).
By Definition 5 (Same-Domain Reductions can be Split into Entry/Exit Reductions) with HP2 we have HPRN:

$$\Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{[]} \sigma_1 \xrightarrow{[]} \sigma_1' \xRightarrow{[]} \cdots \xrightarrow{[]} \sigma_n' \xRightarrow{[]} \sigma'$$

and $\forall i \in 0..n/2.$, HPRI:

$$\vdash \Omega' \triangleright \Omega, P \vdash \sigma_{2i} \xRightarrow{[]} \sigma_{2i+1} : \mathit{entry/exit}$$

and HPRE:

$$\vdash \Omega' \triangleright \Omega, P \vdash \sigma_{2i+1} \xrightarrow{[]} \sigma''_{2i+1} : \mathit{exit/entry}$$

We proceed by induction over the amount of entry-to-entry reductions within the Ω' execution:

Base, $i = 0$**:** We have:

- $\Omega' \triangleright \Omega, P, i \vdash \sigma \overset{[]}{\Longrightarrow} \sigma'$
- HPR $\vdash \Omega' \triangleright \Omega, P \vdash \sigma \overset{[]}{\Longrightarrow} \sigma' : entry/exit$ (by definition)
- HPE $\Omega, P \vdash \langle \sigma.\mathrm{pc}, \hat{L}, \hat{S} \rangle \rightsquigarrow \langle \sigma'.\mathrm{pc}, \hat{L}', \hat{S}' \rangle$
  Since the reduction is entry/exit, σ'=σ'' and σ'''=σ

We need to prove:

- $\Omega' \vDash \sigma' \propto \iota : unreachable$.
- $\sigma' \in \gamma\left(\langle \hat{S}', \hat{L}' \rangle\right)$

By Lemma 16 (Entry/Exit Reductions Preserve Encapsulation and Weak Invariant) with HP1, HPR, HPE, HP5 we have what is needed.

Inductive, $i = i + 1$**:** In this case we have i steps:

$$\Omega' \triangleright \Omega, P \vdash \sigma \overset{[]}{\Longrightarrow} \sigma_1 \overset{[]}{\longrightarrow} \sigma'_1 \overset{[]}{\Longrightarrow} \cdots \overset{[]}{\Longrightarrow} \sigma_i \overset{[]}{\longrightarrow} \sigma'_i$$

which proceed as HPRD

$$\Omega' \triangleright \Omega, P \vdash \sigma'_i \overset{[]}{\Longrightarrow} \sigma_{i+1} \overset{[]}{\longrightarrow} \sigma'_{i+1} \overset{[]}{\Longrightarrow} \sigma'$$

By IH we have

- IHSi $\Omega' \vDash \sigma'_i \propto \iota : unreachable$
- IHCi $\sigma'_i \in \gamma\left(\langle \hat{S}_i, \hat{L}_i \rangle\right)$

and we need to prove:

- $\Omega' \vDash \sigma' \propto \iota : unreachable$
- $\sigma' \in \gamma\left(\langle \hat{S}', \hat{L}' \rangle\right)$

We also have the related encapsulation from HP8, since HP8 implies that all the code from entry to exit points are encapsulated:

- HPEI $\Omega, P \vdash \left\langle \sigma'_i.\mathrm{pc}, \hat{L}'_i, \hat{S}'_i \right\rangle \rightsquigarrow \langle \sigma_{i_1}.\mathrm{pc}, \hat{L_{i+1}}, \hat{S_{i+1}} \rangle$
- HPE+1 $\Omega, P \vdash \left\langle \sigma'_{i+1}.\mathrm{pc}, \hat{L'_{i+1}}, \hat{S'_{i+1}} \right\rangle \rightsquigarrow \langle \sigma'.\mathrm{pc}, \hat{L}', \hat{S}' \rangle$

By Lemma 15 (Entry to Enty Reductions Preserve Encapsulation and Weak Invariant) with IHSi, HPRD (for the next 3 hps), HPEI, HPE+1, HPCi, HP8, we have:

- HPS+1 $\Omega' \vDash \sigma'_{i+1} \propto \iota : unreachable$
- HPC+1 $\sigma'_{i+1} \in \gamma\left(\left\langle \hat{S'_{i+1}}, \hat{L'_{i+1}} \right\rangle\right)$

By Lemma 16 (Entry/Exit Reductions Preserve Encapsulation and Weak Invariant) with HPS+1, HPRD (last fat red), HPE+1, HPC+1 we have what is needed.

□

♠

Lemma 14 (!Actions from Concretized States Preserve the Weak Invariant)**.**

$$\begin{aligned}
&\text{if } \Omega \vDash \sigma \propto \iota : \mathit{unreachable} \\
&\text{and } \Omega' \triangleright \Omega, P \vdash \sigma \xrightarrow{\alpha!} \sigma' \\
&\text{and } \Omega, P \vdash \langle \sigma.\mathrm{pc}, \hat{L}, \hat{S} \rangle \rightsquigarrow \langle \sigma''.\mathrm{pc}, \hat{L''}, \hat{S''} \rangle \\
&\text{and } \sigma \in \gamma(\langle \hat{S}, \hat{L} \rangle) \\
&\text{then } \Omega \vDash \sigma \propto \iota : \mathit{unreachable}
\end{aligned}$$

Proof. **Rule Action-Call:** where the function being called is not defined in the trusted codeΩ'

This is a contradiction with Property 2.

Rule Action-Returnback: to a function defined outside Ω'

By Rule Return we know that none of the parameters are InternalRef.

From the definition of concretization for OkRefand NonRef, we derive that there are no globals associated with the returned values.

□

♠

Lemma 15 (Entry to Enty Reductions Preserve Encapsulation and Weak Invariant)**.**

$$\begin{aligned}
&\text{if } \Omega' \vDash \sigma \propto \iota : \mathit{unreachable} \\
&\text{and } \vdash \Omega' \triangleright \Omega, P \vdash \sigma \xRightarrow{[]} \sigma' : \mathit{entry/exit} \\
&\text{and } \vdash \Omega' \triangleright \Omega, P \vdash \sigma'' \xRightarrow{[]} \sigma''' : \mathit{entry/exit} \\
&\text{and } \vdash \Omega' \triangleright \Omega, P \vdash \sigma' \xrightarrow{[]} \sigma'' : \mathit{exit/entry} \\
&\text{and } \vdash \Omega, P \vdash \langle \sigma.\mathrm{pc}, \hat{L}, \hat{S} \rangle \rightsquigarrow \langle \sigma'.\mathrm{pc}, \hat{L'}, \hat{S'} \rangle : \mathit{entry/exit} \\
&\text{and } \vdash \Omega, P \vdash \langle \sigma''.\mathrm{pc}, \hat{L''}, \hat{S''} \rangle \rightsquigarrow \langle \sigma'''.\mathrm{pc}, \hat{L'''}, \hat{S'''} \rangle : \mathit{entry/exit} \\
&\text{and } \sigma \in \gamma(\langle \hat{S}, \hat{L} \rangle) \\
&\text{and } \Xi_{ea}(\Omega'|_\iota) \\
&\text{then } \Omega' \vDash \sigma'' \propto \iota : \mathit{unreachable} \\
&\text{and } \sigma'' \in \gamma(\langle \hat{S''}, \hat{L''} \rangle)
\end{aligned}$$

Proof. By Lemma 16 (Entry/Exit Reductions Preserve Encapsulation and Weak Invariant) with HP1, HP2, HP5, HP7 we get

- HPS $\sigma' \in \gamma(\langle \hat{S'}, \hat{L'} \rangle)$
- HPW $\Omega' \vDash \sigma' \propto \iota : \mathit{unreachable}$

By Rule Entry and Rule Exit there are only these combinations here:

1. σ' performs a **Call** $\langle p \rangle$ to σ'', which is the starting address of a function.

   From Rule Module-top we know that σ'' is encapsulated with abstract types that match its signature ($\texttt{absty}(\cdot)$).

   The well-typedness of the code ensures that the parameters being passed match the signature.

   The definition of $\texttt{absty}(\cdot)$ and of $\gamma(\cdot)$ ensure TH2.

   Since $\texttt{absty}(\cdot)$ never returns an InternalRef, from the definition of concretization for OkRefand NonRef, we derive that there are no globals associated with the returned values.

2. σ' performs a **Ret** to σ'', which is the address after a call done in the past (this also covers the case where σ' is the last instruction of a function, which is a return)

   By HP8 we know that σ'' is the state after a call, so we know there exists $\sigma?$ which is the state where the call is done such that

   $$\Omega, P, \mathbf{Call}\,\langle P_0\rangle \vdash \left\langle \sigma?.\mathrm{pc}, \hat{L}?, \hat{S}?\right\rangle \rightsquigarrow \left\langle \sigma''.\mathrm{pc}, \hat{L}'', \hat{S}''\right\rangle$$

   We now case-analyse the returned values in σ'' to see if their abstract value matches their concretisation. As before, we rely on the well-typedness to know that the returned values match their signature.

   As before, the definition of $\texttt{absty}(\cdot)$ and of $\gamma(\cdot)$ ensure TH2.

   For TH1 we have 2 cases:

   (a) If the returned value is not an InternalRef, so from the definition of concretization for OkRefand NonRef, we derive that there are no globals associated with the returned values.

   (b) If the returned value is InternalRef, notice that the called function P_0 is defined in Ω, so nothing is passed to the attacker and TH1 holds.

□

♠

Lemma 16 (Entry/Exit Reductions Preserve Encapsulation and Weak Invariant)**.**

$$\begin{aligned}
&\text{if } \Omega' \vDash \sigma \propto \iota : \mathit{unreachable}\\
&\text{and } \vdash \Omega' \triangleright \Omega, P \vdash \sigma \overset{[]}{\Longrightarrow} \sigma' : \mathit{entry/exit}\\
&\text{and } \vdash \Omega, P \vdash \left\langle \sigma.\mathrm{pc}, \hat{L}, \hat{S}\right\rangle \twoheadrightarrow \left\langle \sigma'.\mathrm{pc}, \hat{L}', \hat{S}'\right\rangle : \mathit{entry/exit}\\
&\text{and } \sigma \in \gamma\left(\left\langle \hat{S}, \hat{L}\right\rangle\right)\\
&\text{then } \Omega' \vDash \sigma' \propto \iota : \mathit{unreachable}\\
&\text{and } \sigma' \in \gamma\left(\left\langle \hat{S}', \hat{L}'\right\rangle\right)
\end{aligned}$$

Proof. This proof proceeds by induction over $\Rightarrow$ with Lemma 17 (Single Reductions Preserve Encapsulation and Weak Invariant). □

♠

Lemma 17 (Single Reductions Preserve Encapsulation and Weak Invariant)**.**

$$\begin{aligned}
&\text{if } \Omega' \vDash \sigma \propto \iota : \mathit{unreachable}\\
&\text{and } \Omega' \triangleright \Omega, P, i \vdash \sigma \xrightarrow{[]} \sigma'\\
&\text{and } \Omega', P, i \vdash \left\langle \hat{L}, \hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}', \hat{S}'\right\rangle\\
&\text{and } \sigma \in \gamma\left(\left\langle \hat{S}, \hat{L}\right\rangle\right)\\
&\text{and } \Omega, P \nvDash \sigma : \mathit{exit}\\
&\text{and } \Omega, P \nvDash \sigma' : \mathit{entry}\\
&\text{then } \Omega' \vDash \sigma' \propto \iota : \mathit{unreachable}\\
&\text{and } \sigma' \in \gamma\left(\left\langle \hat{S}', \hat{L}'\right\rangle\right)
\end{aligned}$$

Proof. This proof proceeds by case analysis on $\xrightarrow{[]}$. For simplicity, we treat all cases together, though some would be the local or global sub-cases.

Rule [MoveLoc], $i = \textbf{MvLoc}\,\langle x\rangle$ Here the value is not a reference, so its abstract type is an NonRef.

The encapsulator reduction is Rule MoveLoc

Since this reduction does not change G nor M, TH1 follows from Lemma 18 (Globals and Memory Invariance Preserve Unreachability).

TH2 follows from the definition of $\gamma(\cdot)$.

Rule [MoveLocRef], $i = \textbf{MvLoc}\,\langle x\rangle$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [CopyLoc], $i = \textbf{CpLoc}\,\langle \ell\rangle$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [CopyLocRef], $i = \textbf{CpLoc}\,\langle \ell\rangle$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [StoreLoc], $i = \textbf{StLoc}\,\langle x\rangle$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [StoreLoc-Ref], $i = \textbf{StLoc}\,\langle x\rangle$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [BorrowLoc], $i = \textbf{BorrowLoc}\,\langle x\rangle$ We have:

$$\frac{L(x) = \ell}{\frac{\textbf{BorrowLoc}\,\langle x\rangle \vdash \langle M, L, S\rangle \rightarrow_{loc} \langle M, L, \langle \ell, []\rangle :: S\rangle}{\Omega \vdash \langle \langle P, \mathrm{pc}, L\rangle :: C, M, G, S\rangle \rightarrow \langle \langle P, \mathrm{pc}+1, L\rangle :: C, M, G, \langle \ell, []\rangle :: S\rangle}}$$

and the encapsulator reduction is Rule BorrowLoc

$$\frac{L(x) = \ell}{\Omega', P, i \vdash \langle \hat{L}, \hat{S}\rangle \rightsquigarrow \langle \hat{L}, \mathsf{OkRef} :: \hat{S}\rangle}$$

Since this reduction does not change G nor M, TH1 follows from Lemma 18 (Globals and Memory Invariance Preserve Unreachability).

TH2 is $\langle \langle P, \mathrm{pc}+1, L\rangle :: C, M, G, \langle \ell, []\rangle :: S\rangle \in \gamma\big(\langle \hat{L}, \mathsf{OkRef} :: \hat{S}\rangle\big)$.

The L is unchanged, so that part follows from HP4.

The S part is extended with a OkRef, so apart from HP4, we need to show that $L(x) \in \gamma(\mathsf{OkRef})$, which holds from the semantics assumption that $L(x) = \ell$ since ℓ is a valid location in M are required by $\gamma(\mathsf{OkRef})$.

Rule [Borrowfld], $i = \textbf{BorrowFld}\,\langle f\rangle$ We have:

$$\frac{v = \langle \ell, p\rangle \quad \ell \in \mathtt{dom}\,(M) \quad M(\ell)[p] = \langle \{(f, v_f), \cdots\}, t\rangle}{\frac{\textbf{BorrowFld}\,\langle f\rangle \vdash \langle M, L, v :: S\rangle \rightarrow_{loc} \langle M, L, \langle \ell, p :: f\rangle :: S\rangle}{\Omega \vdash \langle \langle P, \mathrm{pc}, L\rangle :: C, M, G, v :: S\rangle \rightarrow \langle \langle P, \mathrm{pc}+1, L\rangle :: C, M, G, \langle \ell, p :: f\rangle :: S\rangle}}$$

and we have two cases for the encapsulator reductions.

Since this reduction does not change G nor M, TH1 follows from Lemma 18 (Globals and Memory Invariance Preserve Unreachability), so we prove TH2 in both cases below.

1. Rule BorrowFld-InvRelevant

$$\frac{f \in \iota}{\iota, \textbf{BorrowFld}\,\langle f\rangle \vdash \langle \hat{L}, v :: \hat{S}\rangle \rightsquigarrow \langle \hat{L}, \mathsf{InternalRef} :: \hat{S}\rangle}$$

For TH2, L is unchanged and S is extended with $\langle \ell, p::f \rangle$, which we need to prove $\in \gamma(\mathsf{InternalRef})$.

The additional premise is that v, so $\langle c, p \rangle$, comes from a field with an invariant.

Since the new field may point to a value that is a global with an invariant on, the abstract value being InternalRefcovers this case too.

2. Rule BorrowFld-InvIrrelevant

$$\frac{f \notin \iota}{\iota, \mathbf{BorrowFld}\,\langle f \rangle \vdash \langle \hat{L}, \hat{v}::\hat{S} \rangle \rightsquigarrow \langle \hat{L}, \hat{v}::\hat{S} \rangle}$$

For TH2, L is unchanged and S is extended with $\langle \ell, p::f \rangle$, which we need to prove $\in \gamma(\hat{v})$.

We know that the abstract value of v is $\hat{v}$, and that the field being read is $\notin \iota$, so the read value is not a reference pointing to invariants.

So the reference will be in $\gamma(\mathsf{OkRef})$, and we can conclude that the read value is in $\gamma(\hat{v})$ since $\mathsf{OkRef} \sqsubseteq \mathsf{InternalRef}$.

Rule [ReadRef], $i = \mathbf{ReadRef}$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [Writeref], $i = \mathbf{WriteRef}$ The encapsulator reduction is Rule WriteRef.

As above, the reduction does not touch G nor M (since the type restriction prevents Rule Attacker Part of State from changing) and the abstract type concretizes to the expected value.

Rule [Pop], $i = \mathbf{Pop}$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [LoadConst], $i = \mathbf{LoadConst}\,\langle v \rangle$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [Op], $i = \mathbf{Op}$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [Pack], $i = \mathbf{Pack}\,\langle s \rangle$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [Unpack], $i = \mathbf{Unpack}\,\langle s \rangle$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [MoveFrom], $i = \mathbf{MoveFrom}\,\langle s \rangle$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [MoveTo], $i = \mathbf{MoveTo}\,\langle s \rangle$ The encapsulator reduction is Rule MoveTo.

The semantics rule ensures we are creating a new global and a new memory location, which are not under attacker influence, so TH1 follows from Lemma 18 (Globals and Memory Invariance Preserve Unreachability).

TH2 is a trivial consequence since we only eliminate bindings.

Rule [BorrowGlobal], $i = \mathbf{BorrowGlobal}\,\langle s \rangle$ We have:

$$\frac{\frac{\rho = \langle P.mid, s \rangle \quad G(\langle a, \rho \rangle) = \ell}{P, \mathbf{BorrowGlobal}\,\langle s \rangle \vdash \langle M, G, a::S \rangle \rightarrow_{glob} \langle M, G, \langle \ell, [] \rangle::S \rangle}}{\Omega \vdash \langle \langle P, pc, L \rangle::C, M, G, a::S \rangle \rightarrow \langle \langle P, pc+1, L \rangle::C, M, G, \langle \ell, [] \rangle::S \rangle}$$

and the encapsulator reduction is Rule BorrowGlobal

$$\frac{}{\mathbf{BorrowGlobal}\,\langle s \rangle \vdash \langle \hat{L}, \hat{v}::\hat{S} \rangle \rightsquigarrow \langle \hat{L}, \mathsf{InternalRef}::\hat{S} \rangle}$$

Since this reduction does not change G nor M, TH1 follows from Lemma 18 (Globals and Memory Invariance Preserve Unreachability).

TH2 is $\langle \langle P, pc+1, L \rangle::C, M, G, \langle \ell, [] \rangle::S \rangle \in \gamma\left(\langle \hat{L}, \mathsf{InternalRef}::\hat{S} \rangle\right)$.

The L is unchanged, so that part follows from HP4.

The S part is extended with a InternalRef, so apart from HP4, we need to show that $\langle \ell, [] \rangle \in \gamma(\mathsf{InternalRef})$.

By definition of $\gamma(\mathsf{InternalRef})$ this is true if ℓ is a valid location in M that can point to a global, which holds.

Rule [Call], $i = \textbf{Call}\,\langle p \rangle$ This is not possible since σ is not an exit state.

Rule [Return], $i = \textbf{Ret}$ This is not possible since σ' is not an exit state.

Rule [Branch], $i = \textbf{Branch}\,\langle \mathrm{p}c \rangle$ As above, the reduction is analogous to the call case and it does not touch G nor M and the abstract type concretizes to the expected value.

Rule [BranchTrue], $i = \textbf{BranchCond}\,\langle \mathrm{p}c \rangle$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

Rule [BranchFalse], $i = \textbf{BranchCond}\,\langle \mathrm{p}c \rangle$ As above, the reduction does not touch G nor M and the abstract type concretizes to the expected value.

□

♠

Lemma 18 (Globals and Memory Invariance Preserve Unreachability)**.**

$$\begin{aligned} &\text{if } \Omega' \vDash \sigma \propto \iota : \mathit{unreachable} \\ &\text{and } \Omega, \sigma \vdash M_a, G_a : \mathit{attackerpart} \\ &\text{and } \Omega, \sigma' \vdash M_a, G_a : \mathit{attackerpart} \\ &\text{then } \Omega' \vDash \sigma' \propto \iota : \mathit{unreachable} \end{aligned}$$

Proof. Trivial unfolding of Rule Weak Property - Atk Changes. □

♠

References

[1] Martín Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, September 1999. ISSN 0004-5411. doi: 10.1145/324133.324266. URL https://doi.org/10.1145/324133.324266.

[2] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. Taming callbacks for smart contract modularity. *Proc. ACM Program. Lang.*, 4(OOPSLA):209:1–209:30, 2020. doi: 10.1145/3428277. URL https://doi.org/10.1145/3428277.

[3] Zachary Amsden, Ramnik Arora, Shehar Bano, Mathieu Baudet, Sam Blackshear, Abhay Bothra, George Cabrera, Christian Catalini, Konstantinos Chalkias, Evan Cheng, Avery Ching, Andrey Chursin, George Danezis, Gerardo Di Giacomo, David L. Dill, Hui Ding, Nick Doudchenko, Victor Gao, Zhenhuan Gao, François Garillot, Michael Gorven, Philip Hayes, J. Mark Hou, Yuxuan Hu, Kevin Hurley, Kevin Lewi, Chunqi Li, Zekun Li, Dahlia Malkhi, Sonia Margulis, Ben Maurer, Payman Mohassel, Ladi de Naurois, Valeria Nikolaenko, Todd Nowacki, Oleksandr Orlov, Dmitri Perelman, Alistair Pott, Brett Proctor, Shaz Qadeer, Rain, Dario Russi, Bryan Schwab, Stephane Sezer, Alberto Sonnino, Herman Venter, Lei Wei, Nils Wernerfelt, Brandon Williams, Qinfan Wu, Xifan Yan, Tim Zakian, and Runtian Zhou. The Libra Blockchain. https://developers.libra.org/docs/the-libra-blockchain-paper, 2019.

[4] Michael Backes, Catalin Hritcu, and Matteo Maffei. Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. *Journal of Computer Security*, 22(2):301–353, 2014. doi: 10.3233/JCS-130493. URL http://dx.doi.org/10.3233/JCS-130493.

[5] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005. doi: 10.1007/11804192_17. URL https://doi.org/10.1007/11804192_17.

[6] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8:1–8:45, February 2011. ISSN 0164-0925. doi: 10.1145/1890028.1890031. URL http://doi.acm.org/10.1145/1890028.1890031.

[7] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2 (POPL), December 2017. doi: 10.1145/3158093. URL https://doi.org/10.1145/3158093.

[8] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. Move: A language with programmable resources. https://developers.libra.org/docs/move-paper, 2019.

[9] 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, 2020.

[10] Vitalik Buterin. Critical update re DAO, 2016. URL https://ethereum.github.io/blog/2016/06/17/critical-update-re-dao-vulnerability.

[11] Consensys. Smart contract best practices, 2021. URL https://consensys.github.io/smart-contract-best-practices/known_attacks.

[12] Devin Coughlin and Bor-Yuh Evan Chang. Fissile type analysis: modular checking of almost everywhere invariants. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 73–86. ACM, 2014. doi: 10.1145/2535838.2535855. URL https://doi.org/10.1145/2535838.2535855.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450373500. doi: 10.1145/512950.512973. URL https://doi.org/10.1145/512950.512973.

[14] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24.

[15] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Progressive memory safety for webassembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2019, June 23, 2019*, pages 4:1–4:8. ACM, 2019. doi: 10.1145/3337167.3337171. URL https://doi.org/10.1145/3337167.3337171.

[16] Ethereum Foundation. Solidity documentation, 2018. URL http://solidity.readthedocs.io.

[17] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007. ISSN 0164-0925. doi: 10.1145/1275497.1275500. URL http://doi.acm.org/10.1145/1275497.1275500.

[18] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 1987.

[19] Google. Sandboxed api, 2019. https://github.com/google/sandboxed-api.

[20] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *J. Comput. Secur.*, 11(4): 451–519, July 2003. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=959088.959090.

[21] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.*, 2(POPL):48:1–48:28, 2018. doi: 10.1145/3158136. URL https://doi.org/10.1145/3158136.

[22] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, May 1994. ISSN 0164-0925. doi: 10.1145/177492.177725. URL https://doi.org/10.1145/177492.177725.

[23] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with webassembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200. ACM, 2017. doi: 10.1145/3062341.3062363. URL https://doi.org/10.1145/3062341.3062363.

[24] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of LISP-like structures. In *POPL*, 1979.

[25] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. doi: 10.1017/S0956796818000151.

[26] James R. Larus and Galen C. Hunt. The singularity system. *Commun. ACM*, 53(8):72–79, 2010. doi: 10.1145/1787234.1787253. URL https://doi.org/10.1145/1787234.1787253.

[27] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[28] Sergio Maffeis, Martín Abadi, Cédric Fournet, and Andrew D. Gordon. *Code-Carrying Authorization*, pages 563–579. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-88313-5. doi: 10.1007/978-3-540-88313-5_36. URL http://dx.doi.org/10.1007/978-3-540-88313-5_36.

[29] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. *Ada Lett.*, 34(3):103–104, October 2014. ISSN 1094-3641. doi: 10.1145/2692956.2663188. URL http://doi.acm.org/10.1145/2692956.2663188.

[30] Erik Meijer, Redmond Wa, and John Gough. Technical overview of the common language runtime, 2000.

[31] Adrian Mettler, David A. Wagner, and Tyler Close. Joe-e: A security-oriented subset of java. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010. URL https://www.ndss-symposium.org/ndss2010/joe-e-security-oriented-subset-java.

[32] Mark Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical report, 2003.

[33] J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *J. Funct. Program.*, 13(5):957–959, 2003. doi: 10.1017/S0956796802004446. URL https://doi.org/10.1017/S0956796802004446.

[34] Mozilla. Script security, 2019. Technical Report. https://developer.mozilla.org/en-US/docs/Mozilla/Gecko/Script_security.

[35] Marco Patrignani, Dave Clarke, and Davide Sangiorgi. Ownership Types for the Join Calculus. In *FMOODS/-FORTE 2011*, volume 6722 of *LNCS*, pages 289–303, 2011.

[36] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. The high-level benefits of low-level sandboxing. *PACMPL*, 4(POPL):32:1–32:32, 2020. doi: 10.1145/3371100. URL https://doi.org/10.1145/3371100.

[37] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, USA, 2001. ISBN 0521781779.

[38] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with scilla. *Proc. ACM Program. Lang.*, 3(OOPSLA):185:1–185:30, 2019. doi: 10.1145/3360611. URL https://doi.org/10.1145/3360611.

[39] David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017. October 22 - 27, 2017*, 2017.

[40] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*, 1990.

[41] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 20–37. IEEE Computer Society, 2015. doi: 10.1109/SP.2015.9. URL https://doi.org/10.1109/SP.2015.9.

[42] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014. URL https://ethereum.github.io/yellowpaper/paper.pdf.

[43] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark W. Barrett, and David L. Dill. The move prover. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 137–150. Springer, 2020. doi: 10.1007/978-3-030-53288-8_7. URL https://doi.org/10.1007/978-3-030-53288-8_7.