Robust Safety for Move

Marco Patrignani
University of Trento
marco.patrignani@unitn.it

Sam Blackshear
Mysten Labs
sam@mystenlabs.com

***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 analysed 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 [16, 47]), 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 analyse a large representative benchmark set of real-world Move programs. This toolchain certifies >99% of the Move modules we analyse, validating that automatic enforcement of strong security properties like robust safety is practical for Move. Additionally, our results tell that security-centric language design can be effective in attaining strong security properties such as robust safety.

I. 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* [6, 22, 43], 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 [40], process isolation, programming patterns such as object capabilities [35], and specialized hardware [45]. Alternatively, one can enforce robust safety at the language level [1, 4, 6, 19, 22, 30, 39, 43]. The vision of this approach is that first, the programmer writes code and specifies key safety invariants (or, assertions). 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.

Unfortunately, no real-world programming language attains robust safety using this this approach, and we ascribe this to 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 [11]). The language is co-developed with the Move Prover [16, 47], a verification tool that checks whether Move code complies with invariants written in Move's integrated specification language *locally*.

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 large 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

2023 IEEE 36th Computer Security Foundations Symposium (CSF) | 979-8-3503-2192-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/CSF57540.2023.00045

DOI 10.1109/CSF57540.2023.00045

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 and flags references that may leak to untrusted code.
3) We implement the escape analysis and evaluate both its efficiency and precision on a large 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 that: 1) automatically enforced robust safety is a practically achievable goal for Move programmers and 2) security-centric language design (such as Move's) can be effective in attaining strong security properties such as robust safety.

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

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 companion techical report [38]. Our implementation is open source as part of the Move Prover tool (see Section IV).

II. 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 II-A, 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 II-B). 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 II-C).

A. *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 II-B.

```
module 0x1::NextCoin {
  use 0x1::Signer;

  struct Coin has key { value: u64 }
  struct Info has key { total_supply: u64 }

  const ADMIN: address = 0xB055;

  // below is the definition of an invariant
  spec { invariant: forall c: Coin, global<Info>(ADMIN).total_supply =
        sum(c.value) }

  public fun initialize(account: &signer) {
    assert(Signer::address_of(account) == ADMIN, 0);
    move_to<Info>(account, Info { total_supply: 0 })
  }

  // the next function temporarily violates
  // and then restores the invariant
  public fun mint(account: &signer, value: u64): Coin {
    let addr = Signer::address_of(account);
    assert(addr == ADMIN, 0);
    let info = borrow_global_mut<Info>(addr);
    info.total_supply = info.total_supply + value;
    // invariant temporarily violated
    Coin { value } // invariant restored
  }

  // this function violates the invariant
  public fun value_mut(coin: &mut Coin): &mut u64 {
    &mut coin.value // not safe!
  }
}
```

Listing 1. Implementation of a coin asset in Move.

a) 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.

b) Structs: The module defines two data structures `Coin` and `Info`. A `Coin` represents the currency allocated to users of the module while `Info` records 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.

c) 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).

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.

d) 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.

e) 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 II-B.

f) 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.

[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.

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 possesses, or copying/destroying existing Coins? Fortunately, Move has a bytecode verifier (a type system enforced at the bytecode level, as in the JVM [29] and CLR [32]) 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 [31] mutable or immutable borrow (e.g., &mut coin.value at Line 30).

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, 20, 44]. 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, 10], it is not powerful enough to enforce robust safety. We now explain why by describing Move code invariants.

B. 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. As such, the invariant talks about not just on a single object, but on a collection of them (i.e., all the Coins). 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).

a) 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 [13]); 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.

[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.

b) 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.

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.

C. 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 III.

We detect leaks of structs involved in programmer-specified safety invariants with an intraprocedural escape analysis. When the analysis begins analysing a procedure, it binds all mutable reference parameters to the abstract value OkRef. Borrowing an invariant-relevant field from OkRef produces the abstract value InvRef, indicating a pointer into module-internal state. The analysis flags a leak if an InvRef 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 InvRef. 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.

a) Is Robust Safety so Simple?: Our running example may leave the reader with the impression that it is trivial to enforce robust safety: just avoid leaking internal state! We emphasize that this principle is also sufficient to ensure robust safety 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 IV. Thus, our work validates the fact that Move's careful design enables efficient, precise verification of robust safety. We recap the benefits of our approach in more detail in Section III-D2, after providing more details.

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.

III. 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 III-A), as taken from the work of Blackshear et al. [9]. Then, it describes the threat model we consider (Section III-B): 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 fulfil (Section III-C). Finally, this section proves that any Move module certified by tools that satisfy these properties is robustly safe (Section III-D).

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 companion technical report [38].

A. 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 II-A 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 II-A). 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 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 storable 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.

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.

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

$$
\begin{aligned}
\text{instrs.}\quad & \mathbf{Call}\,\langle P\rangle \mid \mathbf{Ret} \mid \mathbf{BranchCond}\,\langle \mathrm{pc}\rangle \\
& \mid \mathbf{Branch}\,\langle \mathrm{pc}\rangle \\
\text{global instrs.}\quad & \mathbf{MoveTo}\,\langle s\rangle \mid \mathbf{MoveFrom}\,\langle s\rangle \mid \mathbf{Exists}\,\langle s\rangle \\
& \mid \mathbf{BorrowGlobal}\,\langle s\rangle \mid \mathbf{Pack}\,\langle s\rangle \mid \mathbf{Unpack}\,\langle s\rangle \\
\text{local instrs.}\quad & \mathbf{MvLoc}\,\langle x\rangle \mid \mathbf{StLoc}\,\langle x\rangle \mid \mathbf{CpLoc}\,\langle \ell\rangle \\
& \mid \mathbf{BorrowLoc}\,\langle x\rangle \mid \mathbf{Pop} \mid \mathbf{LoadConst}\,\langle a\rangle \mid \mathbf{Op} \\
& \mid \mathbf{ReadRef} \mid \mathbf{WriteRef} \mid \mathbf{BorrowFld}\,\langle f\rangle
\end{aligned}
$$

Figure 1. Instructions of the Move language.

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.

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

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

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

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

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

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

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 the value of its content in memory ($M(L(x))$) 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 II-A), 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.

2) Static Semantics: As we mentioned in Section II 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.

B. 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 III-B1) and invariants, i.e., the specific formulation of security properties that must hold on trusted code (Section III-B2). 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 III-B3).

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.

Currently, Move programs are smart contracts deployed on blockchains, and as such we focus on a blockchain-based attacker; this affects how code interacts and what security we can enforce on data, as we now discuss. We identify two main classes of attackers based on whether the dependencies of trusted code with attacker code is immutable or not and call them the *immutable* attacker and the *mutable* one.

When the trusted code is deployed with an immutable attacker, it knows that any existing code cannot change, so the attacker is any code that gets deployed *temporally after* the trusted code. This attacker can call the trusted code and the trusted code can return to it, but any code that the trusted code calls is not attacker. In fact, the publisher of the trusted code can verify both the trusted code and any of its dependency before publishing. On the other hand, when trusted code is deployed with a mutable attacker, verifying existing code is not helpful, since it can change in the future. In this case, the attacker can both call and return to the trusted code. The mutable attacker exists also beyond blockchain settings, it is the typical attacker considered in robust safety works, since typically one does not know what code the trusted code will link against, only their signatures [40, 43]. In this paper we focus primarily on immutable attackers, though we demonstrate how to attain robust safety for immutable ones too in Section III-C2b.

All blockchain data being public suggests 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).[3]

To clearly capture the power of attackers (A), we formalise them as pairs consisting of a code environment (Ω) and a main function (P_{main}). We impose minimal constraints 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 $\Omega^\dagger$. Any attacker function can call trusted code, then immutable attacker functions cannot be called by trusted code, while mutable attacker functions can be. 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 $\Omega_0\left(\Omega^\dagger + A\right)$.

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, so

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

they contain all of the objects with an invariant on (in the sense of Section II-B). For each memory location, invariants define a logical condition that must hold for the content of that location (as in Listing 1). Invariants can describe relationships between structs or global storage locations in distinct modules as long as the modules have a dependency relationship. Some invariants describe a dynamic footprint (e.g., all values of type `Coin`, the storage location of type `Bank` under every possible address) that the verification process must approximate statically [16].

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 $\texttt{cond}\,(\iota, M)$ evaluates the condition for all locations in memory M 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.

$$\frac{G_i = G|_{\texttt{domG}(\iota)} \qquad M_i = M|_{G_i} \qquad \texttt{cond}\,(\iota, M_i) = true}{M, G \vdash \iota}\ \text{(Invariant Satisfaction)}$$

This abstract characterisation lets us model invariants such as the one on Line 10 in Listing 1. In fact the $M|_{G_i}$ returns all the memory locations that contain structs with an invariant on, i.e., the `Info` **`struct`** as well as all all minted `Coin` **`struct`**`s`. With $\texttt{cond}\,(\iota, M_i)$, we express in an abstract fashion the condition that the first field of the former (`Info.total_supply`) equals the `sum` of all the first fields of the latter ones (`Coin.value`).

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 an invariant. This is used to identify the sub-part of a code environment that needs to be encapsulated, as we discuss later in Section IV-B2.

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 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) 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 where 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 II-B. 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 III-D1.

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. 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 (!) [41]. 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 α, we indicate its M and G elements as $\texttt{mg}\,(\alpha)$. 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:

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

a) Trace Semantics: We now define a big-step trace semantics on top of the small-step operational semantics of Section III-A1 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 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 $\Omega^\dagger \triangleright \Omega, P \vdash \sigma \stackrel{\alpha}{\Longrightarrow} \sigma'$. 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 ($same$).

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

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

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

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

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

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

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.

Let us now explain which of these rules apply to external code calling the `mint` function of Listing 1. First, Rule Action-Call is triggered when calling `mint`, producing action `call` $mint\ M, G?$. Rule Action-No handles the body of `mint` until it returns, where Rule Action-Return produces action `ret` $M', G'!$. The globals G' now contain a new address pointing to a memory location in M' where the newly-minted coin (the one being allocated and returned on line Line 25) is stored. All these single actions are concatenated into a trace by Rule Trace-Both.

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:

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

C. 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 III-A2), a prover that checks whether invariants hold locally for trusted code (Section III-C1) and an encapsulator ensuring trusted code does not leak globals that have an invariant (Section III-C2). 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 - Locality).

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

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 ($\not\cap$) with those with an invariant, we call this the unreachability property (Rule Weak Property - Unreachability). 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 $\Omega^\dagger$.

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

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).

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

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.

1) Prover: The prover (Λ) is a tool that statically 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 that code and linking it against attacker code. We denote the prover running on the trusted code $\Omega^\dagger$ as: $\Lambda(\Omega^\dagger)$.

Ideally, the prover statically shows 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 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 $\Lambda \vdash_{loc} \Omega^\dagger : \iota$, as captured by Definition 1 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)**.**

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

What we mentioned this far is an abstract prover Λ. A concrete prover instance would be the Move Prover [16, 47], which 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 Move Prover translates both invariants and Move bytecode into Boogie [5], which uses Z3 [15] to prove that the invariants hold or find a counterexample. We believe the Move Prover fulfils Definition 1, but since it is not the focus of this work, we leave that result for future work.

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 2); 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 $\Xi \vdash_{enc} \Omega^\dagger : \iota$, as captured by Definition 2. 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)**.**

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

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 later in Section IV-B2.

We now describe a concrete encapsulator, denoted with Ξ_{imm}, that satisfies Definition 2 under the immutably attacker of Section III-B1 (Section III-C2a). This is fulfilled in practice, since currently, most Move programs are smart contracts deployed on blockchains. Afterwards, we describe a slightly-different encapsulator, denoted with Ξ_{mut}, that satisfies Definition 2 with respect to the mutable attacker of Section III-B1 (Section III-C2b).

a) A Concrete Encapsulator for Blockchain Move *Code:* Ξ_{imm} is a static intraprocedural escape analysis that formalises the intuition presented in Section II-C. The analysis abstracts the concrete values bound to local variables and stack locations using a lattice with three abstract values: NonRef, OkRef, InvRef. We indicate abstract values as $\hat{v}$ and abstract locals (resp. globals) as $\hat{L}$ (resp. $\hat{S}$). The lattice ordering is NonRef $\sqsubseteq$ InvRef and OkRef $\sqsubseteq$ InvRef. Intuitively, NonRef represents any non-reference value, OkRef represents a reference that does not point to resource defined in trusted code, and InvRef represents a reference that *may* point to a resource defined in trusted code. The goal of the analysis is to prevent an InvRef 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_{imm} to a module Ω (still denoted as $\Xi_{imm}(\Omega)$) makes Ξ_{imm} 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 $\Omega, P, \iota, i \vdash \left\langle \hat{L}, \hat{S} \right\rangle \rightsquigarrow \left\langle \hat{L}', \hat{S}' \right\rangle$, which reads *"under 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 $\hat{S}'$"*.

Rule Ξ_{imm}-BorrowFld-Relevant states that when borrowing a field that has an invariant on, it may point to a resource defined in trusted code and thus InvRef. Rule Ξ_{imm}-BorrowFld-Irrelevant propagates the abstract values when the field has no invariant on itself. Rule Ξ_{imm}-BorrowGlobal applies to globals, since one such reference should never be leaked, the borrowed global is InvRef. Rule Ξ_{imm}-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 InvRef (Rule Ξ_{imm}-Return). Finally, as mentioned, the analysis is intraprocedural, so we conservatively assume that each value returned by a call is the join of all function inputs of that call–i.e., if any function input is InvRef, we assume any function output is alos InvRef.

(Ξ_{imm}-BorrowFld-Relevant)
$$\frac{f \in \iota}{\Omega, P, \iota, \mathbf{BorrowFld}\,\langle f\rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \mathsf{InvRef}::\hat{S}\right\rangle}$$

(Ξ_{imm}-BorrowFld-Irrelevant)
$$\frac{f \notin \iota}{\Omega, P, \iota, \mathbf{BorrowFld}\,\langle f\rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle}$$

(Ξ_{imm}-BorrowGlobal)
$$\Omega, P, \iota, \mathbf{BorrowGlobal}\,\langle s\rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \mathsf{InvRef}::\hat{S}\right\rangle$$

(Ξ_{imm}-BorrowLoc)
$$\Omega, P, \iota, \mathbf{BorrowLoc}\,\langle x\rangle \vdash \left\langle \hat{L}, \hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \mathsf{OkRef}::\hat{S}\right\rangle$$

(Ξ_{imm}-Return)
$$\frac{\|\Omega(P).\texttt{rety}\| = n \qquad \forall i \in 1..n.\ \hat{v_i} \neq \mathsf{InvRef}}{\Omega, P, \iota, \mathbf{Ret} \vdash \left\langle \hat{L}, \hat{v_1} :: \hat{v_n}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{v_1} :: \hat{v_n}::\hat{S}\right\rangle}$$

Figure 4. Ξ_{imm} escape analysis (excerpts).

As mentioned, we have proven that Ξ_{imm} is a valid encapsulator, i.e., it satisfies Definition 2 (as captured by Theorem 1). We describe the implementation of Ξ_{imm} in Section IV.

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

$$\Xi_{imm} \vdash_{enc} \Omega^{\dagger} : \iota$$

Intuitively, this holds because code that is encapsulated with Ξ_{imm} 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 Ξ_{imm}-Return prevents that so long as the reference is InvRef. To ensure that any reference to a global with invariants that we load in the stack is InvRef, Ξ_{imm} ensures that any way to load those references tags them as InvRef. This is exactly what Rules Ξ_{imm}-BorrowFld-Relevant and Ξ_{imm}-BorrowGlobal do.

Technically, the statement of Theorem 1 contains concrete states, yet applying Ξ_{imm} (i.e., $\Xi_{imm}(\Omega^{\dagger}|_{\iota})$) 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.

b) A Concrete Encapsulator for Mutable Attackers: We now discuss how to devise an escape analysis that lets us attain robust safety in the case of a mutable attacker, as defined in Section III-B1. Recall that the mutable attacker consists of code that cannot be verified, and whose functions can be called by the trusted code. Thus, to define Ξ_{mut}, the only change we need to introduce is that calls cannot send InvRef on function calls to functions defined outside the trusted code (Rule Ξ_{mut}-Call). In the same rule, the expected returned values are obtained from calculating the abstract value from the concrete types returned by the function ($\Omega(P).\texttt{rety}$). This is done via function $\texttt{absty}\,(\cdot)$, which maps non-reference types to NonRef and reference types to OkRef. Since these values are returned by an attacker, they cannot be InvRef.

(Ξ_{mut}-Call)
$$\frac{\|\Omega(P_0).type\| = n \qquad \forall i \in 1..n.\ \hat{v_i^a} \neq \mathsf{InvRef} \qquad \hat{v_1^r} \cdots \hat{v_j^r} = \texttt{absty}\,(\Omega(P).\texttt{rety}) \qquad P_0 \notin \Omega^{\dagger}}{\Omega, P, \mathbf{Call}\,\langle P_0\rangle \vdash \left\langle \hat{L}, \hat{v_1^a} \cdots \hat{v_n^a}::\hat{S}\right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{v_1^r} \cdots \hat{v_j^r}::\hat{S}\right\rangle}$$

Figure 5. Ξ_{mut} escape analysis (excerpts).

Similarly to Ξ_{imm}, we have proven that Ξ_{mut} is a valid encapsulator (as captured by Theorem 2).

Theorem 2 (Ξ_{mut} is a Valid Encapsulator)**.**

$$\Xi_{mut} \vdash_{enc} \Omega^{\dagger} : \iota$$

D. Robust Safety for Move

We now have all the technical setup to state (Definition 3) and prove robust safety for trusted code (Theorem 3). After presenting and discussing the definition, we analyse the robust aspect from a security perspective (Section III-D1) and compare ours to existing robust safety definitions and proofs (Section III-D2).

Any trusted code $\Omega^{\dagger}$ that is verified (in the sense of Section III-A2), proved (in the sense of Section III-C1) and encapsulated from Ξ (in the sense of Section III-C2), 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 $\triangleright_{RS}\ \Omega^{\dagger} : \iota, \Xi, \Lambda$.

Definition 3 (Robust Safety for Move)**.**

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

The first premise of the definition 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 by Λ and encapsulated by Ξ. Note that the result is general, no matter what prover and encapsulator are used, so long as those prover and encapsulators are valid in the sense of Definitions 1 and 2. The final premise introduces the trace yielded by the interaction of the trusted code with the attacker and the conclusion of the definition confirms that the trace does not violate the invariants.

Theorem 3 (Move Modules are Robustly-Safe)**.**

$$\triangleright_{RS}\ \Omega^{\dagger} : \iota, \Xi, \Lambda$$

1) 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.

2) Comparison with Existing Robust Safety Statements and Proofs: To the best of our knowledge, Move is the first large language with tools that provide robust safety, early work on robust safety [1, 4, 6, 19, 22, 30, 39] focussed on formal calculi without a real-world implementation. Moreover, existing work on robust safety ties the robust safety result to the verification of trusted code with a *specific* tool. In the more modern robust safety results [40, 43] (which apply to calculi with complex features not amenable to smart-contract programming), this tool comes in the form of a semantic type systems built on top of the Iris separation logic [27] – a non-trivial tool to understand and master.

In contrast, our definition of robust safety identifies the requirements of *multiple* tools, and singles out the *individual guarantees* of each one tool. By performing such separation of requirements, our work requires less complexity from each tool individually, which can build upon simple, well-established techniques (such as the escape analysis). Moreover, as tools (and languages) evolve, this separation makes it simpler to provide new tools for robust safety, maximising proof reuse. For example, if a new version of Move relies on a different logic of invariants, it is sufficient to change the Move Prover, and thus re-prove it fulfils Definition 2, without changing the escape analysis, nor its proofs. Finally, the proof reuse makes is easier to prove robust safety for different attacker models, as we do with the Ξ_{imm} and Ξ_{mut} escape analyses.

IV. Evaluation

In this section, we present an implementation of the escape analysis Ξ_{imm} described in Section III-C2 (Section IV-A). Then, we measure its performance on a large set of Move benchmarks (Section IV-B).

We evaluate the Ξ_{imm} analysis according to two criteria:

Performance. We claim the escape analysis is fast: it adds negligible overhead over the companion verifier (Section IV-B1);

Precision. We claim the escape analysis is precise: it rarely flags Move code that is robustly safe w.r.t its safety invariants (Section IV-B2).

A. Implementation

We have implemented the escape analysis in approximately 300 lines of Rust code on top of the Move Prover analysis framework.[4] 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 the Move Prover specification language[5] as the invariant language ι. This specification language lets programmers write source code invariants similar to our example in Section II-B. The invariants are converted to SMT and checked by the Move Prover against the compiled Move bytecode.

Unlike our minimalistic formalism in Section III, 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 read. 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 auditing. 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.

B. Benchmarks

We ran our analysis on the benchmarks shown in Figure 6. 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. Although our benchmark set is small, it represents a substantial fraction of the publicly available Move code on GitHub. Move is a young language that it only beginning to gain transaction, so these benchmarks contain a representative sample of production Move code.

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 6. 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.

1) Evaluating Performance: The results in Figure 6 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

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

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

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 analysis, 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 analyses in the pipeline. Anecdotally, these steps take roughly half of the escape analysis running time.

2) Evaluating Precision: The results also support our claim that the analysis is precise. Only three functions (0.1% of the total analysed) in three distinct modules (0.9% of the total analysed) 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)≤ 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 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.

a) Discussion: These examples demonstrate an interesting and perhaps counter-intuitive 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* [26] 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 specification 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 [14] 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.

Finally, we note that a simpler analysis that flags any return of a (mutable) reference would be too restrictive to be practical. Returning references is common in real-world Move code because of the ubiquity of non-copyable types.

C. 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.

V. Related Work

a) 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 [18] source language and its executable Ethereum Virtual Machine (EVM) [46] 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 [11] 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 [12], dynamic analysis [23], 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 [42] 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. Finally, recent work [34] proposes a proof methodology and programming discipline for addressing the challenges of verifying class invariants in the presence of common smart contract features such as callbacks. The emphasis on avoiding "reference leaks" is directly related to the properties enforced by our escape analysis.

b) 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 [36] and capability machines like CHERI [45] are low-level approaches to isolating memory from untrusted code running in the same process. The Singularity OS project [28] used the type system of Sing# (a variant of C#) to enforce strong ownership of memory that crosses trust boundaries. The Joe-E [33] language defines a secure subset of Java to enable capability-based programming patterns.

WASM [25] is designed to isolate untrusted applications from the trusted host system. Recent work on WASM has studied a variety of mechanisms [17, 25] (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.

c) Robust Safety: The robust safety property originated in the context of modular model checking [24] and has then been widely applied to reason about security protocols that interact with adversaries [1, 4, 6, 19, 22, 30, 39]. 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. [43], 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 [35]. 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 [27]. 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. [40] 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 [21, 37]. 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.

VI. 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 encapsulator can augment existing tools like the Move prover to enable practical enforcement of robust safety for Move programmers.

The authors would like to thank John Mitchell for useful feedback and discussions. This work was partially supported by a gift from Novi; the Office of Naval Research for support through grant N00014-18-1-2620, Accountable Protocol Customization; the Italian Ministry of Education through funding for the Rita Levi Montalcini grant (call of 2019).

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] Sam Blackshear, John Mitchell, Todd Nowacki, and Shaz Qadeer. The move borrow checker, 2022.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] David Dill, Wolfgang Grieskamp, Junkil Park, Shaz

Qadeer, Meng Xu, and Emma Zhong. Fast and reliable formal verification of smart contracts with the move prover. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 183–200, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99524-9.

[17] 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.

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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

[33] 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.

[34] Bertrand Meyer, Alisa Arkadova, Alexander Kogtenkov, and Alexandr Naumchev. The concept of class invariant in object-oriented programming. *CoRR*, abs/2109.06557, 2021. URL https://arxiv.org/abs/2109.06557.

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

[36] 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.

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

[38] Marco Patrignani and Sam Blackshear. Robust safety for move. *CoRR*, abs/2110.05043, 2021. URL https://arxiv.org/abs/2110.05043.

[39] 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.

[40] 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.

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

[42] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Am-

rit 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.

[43] 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.

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

[45] 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.

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

[47] 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.