The Move Borrow Checker – Review

Formal Verification, EPFL

Guillem Bartrina I Moreno
guillem.bartrinaimoreno@epfl.ch

Franco Sainas
franco.sainas@epfl.ch

Marcin Wojnarowski
marcin.wojnarowski@epfl.ch

January 5, 2024

1 Introduction

A smart contract is a self-executing contract where the terms of the agreement are directly written into code. It runs on a blockchain, automating and enforcing the terms of the contract without the need for intermediaries. The immutability of these contracts, once deployed, and the substantial real-world value of the digital assets they manage underscore the critical importance of security measures [4]. In adversarial settings where malicious entities seek to exploit vulnerabilities, robust security measures are paramount to prevent unauthorized access, manipulation, or theft of these valuable assets[1]. Given the necessity for all network nodes to reach a consensus on execution outcomes, determinism also becomes a foundational attribute.

The Move programming language [2] is designed specifically for smart contracts. Unlike Solidity, the prevailing language for smart contracts, Move adopts a combination of move and reference semantics. In alignment with the Rust programming language, Move utilizes ownership and borrowing as a key mechanisms to effectively address its security challenges.

Our focus centers on the Move Borrow Checker[3], a tool that guarantees three pivotal memory properties in valid Move programs: the elimination of dangling references, referential transparency for immutable references, and the prevention of memory leaks. This review delves into the efficacy of the Move Borrow Checker in upholding these critical memory-related assurances.

1.1 Memory Challenges

The paper presents a few examples illustrating potential memory safety issues that may arise in Move in the absence of adequate checks by the borrow checker. We highlight the most noteworthy instances in this report.

A dangling reference is reference in a program that points to memory that has been deallocated or is no longer valid. The function defined here accepts a parameter `c` of type `Coin` and assigns a reference to the field `f` of `c` to a variable `r`. Subsequently, the coin `c` is moved to another variable, leading to a change in its memory location. Consequently, the memory referenced by `r` is rendered invalid.

```
fun dangle_after_move(c: Coin) {
    let r = &c.f; // the field f of c is borrowed by r
    let x = move c; // †
    let y = *r; // read from dangling ref
}
```

Here † indicates where the bug is introduced, causing the code to fail validation by the borrow checker.

The paper presents instances of dangling references across procedures as well. These two functions illustrate the potential to write Move code that returns unsafe references to local variables, which subsequently become dangling after the procedure terminates.

```
fun ret_local(): &u64 { let x = 7; &x } †
fun ret_param(x: u64): &u64 { &x } †
```

Such bugs have the potential to render a smart contract unusable. If not identified prior to deployment, they may result in the contract entering a sink state, effectively trapping funds within it permanently.

2 The Move language

In the Move language variables can either be owners of some value or be borrowing a value owned by a different variable. At any point in time there exists at most a single owner of a value. This separation of owned and borrowed values is visible in the type system: owned values are annotated with the type `T` while borrowed values with `&T` and `&mut T` for immutable and mutable references respectively. The state of borrows is tracked to eliminate previously introduced memory issues. Ownership can be transferred with the `move` keyword, but only under the condition that the value has no outstanding borrows, otherwise these references would be invalidated. At any point in the program all references have to point to a value that has an owner. And finally when a value is mutably borrowed, the value can be accessed only through that reference. Together these simple rules prevent problems with dangling references.

Since Move supports composite types, tracking ownership has to be more fine grained to allow borrows of fields. To do that Move maintains a notion of paths which is a sequence of field selections of some type. `v[p]` denotes the field selections of a value `v` by a path `p`. Values in Move are tree-shaped, so a specific field is represented by a unique path. This allows Move to identify borrows to the same value by simply comparing the paths of a borrow.

2.1 Global memory

Global storage, a distinctive characteristic of blockchain technology, is shared among all smart contracts and persists across all executions. Access to it is provided through association of an account addresses and a type with record values: (Addr × Type) → Record. Accessing global memory is designed in a way to fit into the existing reference semantics. Interacting with the global memory consists of three operations:

- `move_to<T>(a: address, value: T)` – sets a value at a given address
- `move_from<T>(a: address): T` – removes and gives ownership of the value under some address
- `borrow_global<T>(a: address): &mut T` – borrows mutably the value under some address

To fit these operations into the existing reference semantics it is enough to see borrows as coming from `T` instead of an owner. One issue is that global accesses are invisible across procedure boundaries. Move strives to infer all borrow relations of a procedure strictly from its signature to keep reasoning local. Therefore a special syntax has to be introduced to denote a procedure potentially taking ownership of a global resource. This is shown below with the `acquires` clause.

```
fun remove_t(a: address): T acquires T {
    return move_from<T>(a);
}
```

Move code is organized in modules. This allows to create logically safe abstractions. In practice this is enforced by limiting access to global memory indexed by some type T to the module that defined this type. Additionally the creation and destruction of record values can only happen within the module that defines the type of that value. An interesting consequence of that is if a module were to define a record `Coin` which represents a financial asset that module would be guaranteed that if it gives someone the ownership of a `Coin` instance through an inter-module boundary procedure call, that coin could not be destroyed (disappearance of funds) nor duplicated (duplication of funds) outside of the defining module.

2.2 Memory Model

Move compiler produces bytecode for a stack-based VM. All formalization is built on top of the bytecode representation. In result Move encompasses three distinct types of storage:

- Call Stack – This segment holds procedure frames containing a fixed set of local variables. These variables remain uninitialized at the beginning of the procedure. Initialized procedures, however, can store both values and references.

- Operand Stack – Primarily utilized for local computations and the exchange of arguments/return values among procedures, the operand stack is shared across procedures. At the initiation of program execution, the call stack holds one frame, and the operand stack is free of elements. If the program concludes successfully, the initial and final states of the operand stack coincide.

- Global Storage

The exact operational semantics of the language are skipped in this review. However a crucial piece to formalization is the notion of a concrete state.

Definition

A *concrete state* s represents the state of the program at a specific point. It is denoted by a tuple $\langle P, S, M \rangle$ where: P is the call stack, a list of frames. Each frame is a triple (ρ, ℓ, L) which represents a procedure ρ, the program counter ℓ, and a store of local variables L that maps variables to locations or references. S is the operand stack holding a list of values and references. Finally, M is the memory which maps locations to values.

3 Move Borrow Checker

The Borrow Checker is a key piece of the Move Bytecode Verifier, which is executed on every piece of code that is to be executed on the *blockchain.* Any program that passes the check is guaranteed to enjoy the three desirable properties we have stated: absence of dangling references, referential transparency and absence of memory leaks.

This check builds upon specific local annotations at each program point of each procedure. One can use these annotations to derive a (conservatively approximate) abstract state from a concrete one. Then, one may formulate the memory safety invariants as a predicate over a concrete state and abstract state. Finally, combining the operational semantics and the propagation of local annotations, one can show that any reachable concrete state in the program is connected to its corresponding abstract state by that invariant, effectively proving that the entire program enjoys all three properties.

Definitions

An *abstract location* represents either the contents of a local variable in some stack frame or the contents of the operand stack at some position. It can be looked up in a concrete state to return the corresponding location, reference or value.

A *borrow graph* is a directed graph whose nodes represent abstract locations and arcs, which are labelled with a path, represent borrowing relations. A borrow arc $x \xrightarrow{p} y$ indicates that $x[p]$ *is borrowed by* y. We say that $x \xrightarrow{p} y$ is *subsumed* by $x \xrightarrow{q} y$ if q is a prefix of p. Let G and H be borrow graphs, we define $G \sqsubseteq H$ iff every arc in G is subsumed by some arc in H. Semantically, this means that G imposes every restriction on concrete execution that H does, so every state that satisfies G also satisfies H.

3.1 Local annotations

The local annotations required by the analysis at every program point are in fact local abstract states, which capture local typing and borrowing information relative to that program point.

A *local abstract state* is a tuple $\langle \hat{L}, \hat{S}, B \rangle$ where: $\hat{L}$ maps local variables to their types; $\hat{S}$ contains the types of the *visible* elements in the operand stack; and B is the borrow graph of all abstract locations in $\hat{L}$ and $\hat{S}$. These abstract locations are defined as if the call and operand stacks were empty (indices start at 0), they can be viewed as offsets. A local abstract state is *well-formed* if $\hat{L}$ includes all input abstract locations, B is acyclic and no borrow arc in B is incident to an input abstract location.

We write $\langle \hat{L}, \hat{S}, B \rangle \sqsubseteq \langle \hat{L}', \hat{S}', B' \rangle$ iff $\hat{L} = \hat{L}'$, $\hat{S} = \hat{S}'$ and $B \sqsubseteq B'$. Semantically, every concrete state represented by $\langle \hat{L}, \hat{S}, B \rangle$ can also be conservatively represented by $\langle \hat{L}', \hat{S}', B' \rangle$.

Propagation of local abstract state

The abstract execution of a given bytecode instruction is a form of type & borrow propagation. Therefore, by defining appropriate propagation rules for every bytecode instruction, one can establish the basis for calculating these local annotations at every program point. These propagation rules are of the form $\rho, \mathrm{op} \vdash \langle \hat{L}, \hat{S}, B \rangle \rightarrow \langle \hat{L}', \hat{S}', B' \rangle$. They preserve 'well-formedness' by construction.

Well-typed programs

The notion of a well-typed program is defined in terms of these local annotations. A program $\mathcal{P}$ is *well-typed* if, for all its procedures ρ:

- Initially, the only local variables are the arguments and the operand stack and borrow graph are empty.
- The 'restrictions' imposed after executing an instruction continue to every possible next instruction. In other words, let $\langle \hat{L}, \hat{S}, B \rangle$ be the local abstract state obtained by applying the propagation rule of the current instruction, we have that $\langle \hat{L}, \hat{S}, B \rangle \sqsubseteq \langle \hat{L}', \hat{S}', B' \rangle$ for every local abstract state $\langle \hat{L}', \hat{S}', B' \rangle$ in the possible next program points.

Computing local annotations

One can define the following well-behaved join operation between two borrow graphs G and H: take the union of arcs in G and H and then drop every borrow arc in the result that is subsumed by another.

The computation of the local annotations at each program point is performed **locally for each procedure** based on abstract interpretation. Starting from certain initial local abstract states, the corresponding type & borrow propagation rules are applied and their result is joint in an iterative fashion until a fixpoint is reached.

It may happen that the borrow graph becomes cyclic after some join operation, in such case the procedure is aborted and a type error is reported. Otherwise, if it reaches fixpoint, we are guaranteed that the computed local annotations define a well-typed program.

3.2 Abstract state

An *abstract state* is a tuple $\langle \hat{P}, \hat{S}, B \rangle$ where: abstract call stack $\hat{P}$ is a list of frames, each frame is a list of triples $(\rho, l, \hat{L})$ comprising a procedure ρ, a program counter l and a map from variables to types $\hat{L}$. It defines a set of abstract locations of local variables in stack frames. Abstract operand stack $\hat{S}$ is a list of types. It defines a set of abstract locations in operand stack. And borrow graph B includes all abstract locations defined by $\hat{P}$ and $\hat{S}$. An abstract state is similar to a concrete state but it carries typing information instead of locations, values or references and it includes a borrow graph instead of the memory.

Abstraction function

One can define the abstraction function *Abs* on well-typed programs, which maps concrete states $\langle P, S, M \rangle$ to abstract states $\langle \hat{P}, \hat{S}, B \rangle$. It constructs the abstract state only by looking at the concrete state and all the local annotations at every program point. The function itself is fairly straightforward, just involves a bit of playing around with abstract location types, indexes and offsets, and some renaming and joining operations on borrow graphs. By the properties of a well-typed program, we are guaranteed that B is acyclic.

3.3 Invariants

One wishes to prove memory safety invariants about the execution of any well-typed program. These invariant can be stated as a predicate $Inv(s, \hat{s})$ over a concrete state s and an abstract state $\hat{s}$.

Inv is defined the conjunction of $InvA$, $InvB$, $InvC$, $InvD$.

Type agreement

s and $\hat{s}$ are said to be *shape-matching* is they have the same call stack height, the same operand stack height and there is agreement between the corresponding procedure names, program counters and the set of local variables in each stack frame. Moreover, they are said to be *type-matching* if the values of s represented by the abstract locations of $\hat{s}$ conform to the types. $InvA$ captures that s and $\hat{s}$ are shape and type -matching.

No memory leaks

$InvB$ captures that there are no memory leaks, that is: every local variable on the call stack contains a different location and that the memory does not contain any location not present in a local variable.

No dangling references

Given shape-matching s and $\hat{s}$, a borrow arc $m \xrightarrow{p} n$ is *realized* in the concrete state s if the path p leads from the content of m to the content of n. Concretely, this can happen if either m contains a location and n a reference or both contain references.

InvC captures the fact that there are no dangling references, that is, every reference is rooted in a memory location present in some local variable on the call stack: the borrow graph is acyclic, for all value-typed abstract locations in $\hat{s}$ there is no incident arc in the borrow graph, for all **reference**-typed abstract locations in $\hat{s}$ there is an incident arc in the borrow graph that is realized in the concrete state s.

Referential transparency

InvD captures the fact that if there is no borrow arcs out of an abstract location of type value or mutable reference, is is guaranteed that any mutation via that abstract location (either the stored value or the value pointed) does not invalidate any other reference. Concretely, for any distinct abstract locations m and n such that n is of type reference and that the concrete contents of m and n are, respectively (a) a location and a reference rooted on that location or (b) a reference and another reference to a prefix of the first one, we have one of the following: both m and n are of type immutable reference; m is not of type immutable reference and there is a path in the borrow graph from m to n comprising arcs realized in s; or the contents of m and n are the same and there is a path in the borrow graph from n to m comprising arcs realized in s.

3.4 Main theorem

Finally, one puts to use the abstraction function together with the operational semantics and the propagation of local annotations to prove the main theorem:

Theorem 1 *Let $\mathcal{P}$ be a well-typed program,*
If s is a concrete state with $Inv(s, Abs(s))$ and $\mathcal{P} \vdash s \rightarrow s'$, then $Inv(s', Abs(s'))$

Which informally states that in a well-typed program, if a concrete state is connected to its corresponding abstract state by the invariants, then any other concrete state reachable from it is also connected to its corresponding abstract state by the same invariants. Authors present the proof in supplementary materials. Since the proof is not the main contribution of this paper and is quite long and involved, we omit even sketching it in our review.

3.5 Putting all together

Given that the program is well-typed, one can easily show that for the initial state s_0, which only has one stack frame with the inputs to the transaction and has empty operand stack and memory, it holds that $Inv(s_0, Abs(s_0))$.

Hence, the whole analysis of the Borrow Checker reduces to compute the appropriate local annotations using the fixpoint algorithm mentioned in a section. In case it succeeds, we know that the resulting local annotations define a well-typed program, so by application of the previous theorem on the fact the invariants hold for the initial state s_0, we get that **the invariants will hold for the entire program execution**.

Remark: this analysis builds upon some assumptions on the correct structure of the program that must be checked prior to its execution. These are: ***Stack usage analysis****, which ensures that the shape-matching property holds;* ***Value type analysis****, which checks that values are used if they have not been moved and that instructions are applied to values of appropriate types; and* ***Acquires analysis****, which checks acquire annotations on procedures (related to global memory).*

4 Conclusions

The careful design of the language allows to prove memory safety properties on all programs that are considered well-typed. The language presented is stated by the authors to be only a subset of the Move language which encompasses its most important features. Basing verification on the bytecode rather than source code allows Move to work in adversarial environments with untrusted code. However, the language is noticeably restricted. To not introduce potential unsafety many conservative decisions have to be made. Authors draw comparison to Rust which validates reference by means of lifetimes by mentioning the benefits of the borrow graph over lifetimes. Yet the drawbacks are not mentioned: lifetimes encode a larger set of executions. To name a few restrictions from which Move suffers from while Rust does not: storing references in records, usage of closures, specifying exact reference relationships. A more restricted language results in awkward workarounds to satisfy the borrow checker and a frustrating programming experience.

References

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A Survey of Attacks on Ethereum Smart Contracts (SoK)". In: *The post*. 2017. URL: https://api.semanticscholar.org/CorpusID:15494854.

[2] Sam Blackshear et al. "Move: A Language With Programmable Resources". In: 2019. URL: https://api.semanticscholar.org/CorpusID:201681125.

[3] Sam Blackshear et al. *The Move Borrow Checker*. 2022. arXiv: 2205.05181 [cs.PL].

[4] Cheyenne DeVon. "Crypto investors lost nearly $4 billion to hackers in 2022". In: *CNBC* (2023). URL: https://www.cnbc.com/2023/02/04/crypto-investors-lost-nearly-4-billion-dollars-to-hackers-in-2022.html.