Verification of Programs Written in Libra's Move Language

Master's Thesis Description

Constantin Müller

Supervisors: Vytautas Astrauskas, Marco Eilers, Federico Poli, Prof. Peter Müller

26 March 2020

1 Background

Libra [1] is a blockchain-based cryptocurrency initiated by Facebook that aims to underpin a new global financial infrastructure. The idea behind a blockchain is to keep a growing list of transactions that are linked cryptographically, making past transactions immutable and permanent. These transactions allow the transfer of resources, e.g., sending money from one person to another. Smart contracts are applications that perform transactions on the blockchain based on a set of rules that are encoded in a blockchain-specific programming language. These applications can, for example, provide banking and insurance services, allowing the blockchain to be the basis of an extensive financial infrastructure. Due to the proposed scale and monetary nature of the Libra project, security and reliability are of the utmost importance. In particular, the Libra developers envisage a system in which the behaviour of programs running on the blockchain is verifiable.

2 The Move Language

With these goals in mind, Libra developers are working on the *Move* [2] programming language. Move programs can either be *modules*, which are similar to smart contracts in other blockchain languages, or *transaction scripts*, which are arbitrary scripts for resource transfers. The Move language has not been finalised yet, however, there is a Move intermediate representation (IR) that can be directly translated to Move bytecode while being human-readable/writable. Move aims to be much safer than traditional blockchain programming languages,

such as Solidity, by providing *first-class resources*. These are resource types, inspired by linear logic [3], whose values cannot be copied or destroyed, they can only be moved. Resources are held by accounts and are part their state. Modules – unlike Ethereum smart contracts – cannot hold resources and have no global state. Listing 1 provides an example of how the use of first-class resources prevents a developer from ”losing track” of a resource, which is a common source of bugs in smart contracts. Here, the caller of `distribute` withdraws a certain amount of *LibraCoin*, a first-class resource, to distribute it between two payees. In the Move IR, every resource must be `move`d exactly once. If a variable's value is to be used more than once, one must `copy` the value, which is prohibited for first-class resources. This ensures that if, for example, line 26 is missing because the developer has forgotten to send coins to `payee2`, a bytecode verification error occurs.
First-class resources are defined as part of a module, where privileged procedures are defined that can create, modify and destroy them. So while transaction scripts and their specifications can make use of the properties of first-class resources, the procedures inside the module that defines a resource can break the invariants expected by their clients. Therefore, the properties that need to be verified for a module will differ from the ones for transaction scripts.

```
1  import 0x0.LibraAccount;
2  import 0x0.LibraCoin;
3
4  // Distributes a given ‘amount‘ between ‘payee1‘ and ‘payee2‘
5  // according to ‘weight1‘ and ‘weight2‘.
6  main(payee1: address, payee2: address, amount: u64,
7          weight1: u64, weight2: u64) {
8      // The amount to be sent to payee2
9      let amount2: u64;
10     // The coin resource to be transferred to payee1
11     let coin1: LibraCoin.T;
12     // The coin resource to be transferred to payee2
13     let coin2: LibraCoin.T;
14
15     // Calculate the amount of LibraCoin to be sent to payee2.
16     amount2 = copy(amount) * copy(weight2) /
17         (move(weight1) + move(weight2));
18     // coin1 now holds all the coins to be sent.
19     coin1 = LibraAccount.withdraw_from_sender(move(amount));
20     // Set coin2 to hold ‘amount2‘ coins
21     // while mutating coin1 to hold the rest.
22     coin2 = LibraCoin.withdraw(&mut coin1, move(amount2));
23
24     // Send the coin resources to the payees.
25     LibraAccount.deposit(move(payee1), move(coin1));
26     LibraAccount.deposit(move(payee2), move(coin2));
27     return;
28 }
```

Listing 1: A transaction script that withdraws a given amount of LibraCoin from the caller and distributes it between two recipients based on their weight.

While Move has other verification-friendly features, such as the requirement that the target of every function call can be statically determined, Move's language features do not ensure correctness as such, which is why the Libra developers are planning to develop a formal specification language for Move. Currently they are working on an experimental bytecode verifier that uses Boogie [4] as a backend. We believe that the Viper verification infrastructure [5] can be used to power such a specification language, similar to how it has been used for verifying Ethereum smart contracts written in Vyper[1] [6]. Listing 2 provides an example of a module definition for a referendum that is vulnerable to an attack. The idea behind this module is that there is a proposal stored at account address `prop_addr` and every account can either vote for or against it. These votes are stored in the `in_favor` and the `against` vectors. The developer has, however, forgotten to prevent people from voting multiple times, which would be prohibited by the invariant shown on lines 16-17. The simplest way to satisfy this constraint would be to uncomment line 36, which sends a `Vote` resource to any voter. Since any address can hold only one resource of any given kind, the second time the `vote` function is called, it would fail on line 55 and no additional values would be pushed into either of the vectors.

```
 1 module Referendum {
 2     import 0x0.Vector;
 3 
 4     resource Proposal {
 5         in_favor: vector<address>,
 6         against: vector<address>,
 7         accepted: bool
 8     }
 9 
10     resource Vote {
11         dummy: bool // every resource must contain a variable
12     }
13 
14     // Proposal creation etc. ...
15 
16     // INVARIANT: in_favor_ref.count({{sender}}) +
17     //            against_ref.count({{sender}}) <= 1
18     public vote(prop_addr: address, vote_for: bool)
19             acquires Proposal {
20         let sender: address;
21         let prop_ref: &mut Self.Proposal;
22         let in_favor_ref: &mut vector<address>;
23         let against_ref: &mut vector<address>;
24 
25         sender = get_txn_sender();
26         // This borrows the proposal from the account holding it
27         // allowing us to change its contents.
28         prop_ref = borrow_global_mut<Proposal>(copy(prop_addr));
29         // A mutable reference to the in_favor vector.
30         in_favor_ref = &mut copy(prop_ref).in_favor;
31         // A mutable reference to the against vector.
```

[1]Not to be confused with Viper.

```
        against_ref = &mut copy(prop_ref).against;

        // The following line would ensure that
        // double voting is not permitted.
        // Self.send_vote_resource();

        if (move(vote_for)) {
            Vector.push_back<address>(
                move(in_favor_ref), move(sender));
        } else {
            Vector.push_back<address>(
                move(against_ref), move(sender));
        }

        return;
    }

    // PROPERTY: fails if sender holds Vote else send Vote
    send_vote_resource() {
        let vote: Self.Vote;
        vote = Vote {
            dummy: true
        };
        move_to_sender<Vote>(move(vote));
        return;
    }

    // ...
}
```

Listing 2: A referendum module with annotations that should ensure verification fails if voting twice is permitted.

In this way, we can use Move's type system to provide strong guarantees about a program's behaviour without the programmer having to invest a lot of time into thinking about program verification.

3 Core Goals

The goal of this project is to design a specification and verification technique for Move programs that exploits the type system guarantees of the Move language, and to implemented this in a prototype verifier. While the design of the Move language has not been fixed, the verifier will be built for the intermediate representation, in expectation that large parts will be directly usable in Move. The core subgoals are thus:

- Searching for and analysing common smart contract security issues and evaluating to what extent the Move language addresses them. We will consider security issues that either exist in or are addressed by other smart contract languages, such as Solidity [7], Vyper [8] and Flint [9].
- Finding interesting properties to prove about Move programs, using, in particular, the guarantees provided by Move's type system and its first-

class resources. Since module procedures are privileged with regards to their respective type and can create and destroy resources, they will need additional properties that ensure correctness. Together, these properties should be able to capture the essence of common smart contracts.

- Evaluating which of these properties can already be proved by the experimental bytecode-to-Boogie verifier and which advantages using Viper has.
- Designing a specification language for Move that is able to encode these properties. Since the goal is that verification of Move programs becomes ubiquitous, it is crucial that the specification language is simple and intuitive even for developers who are not experts in program verification. This should be done by maximising the information provided by Move's type system.
- Implementing a prototype static verifier for the Move IR with the specification language.
- Building examples of larger Move programs and verifying those. These should include parts of the Move standard library.

4 Extension Goals

As Libra and Move are changing rapidly, some extension goals may change during the course of this project. They may, however, include the following:

- Developing proposals for the Move language that simplify verification. These could be language constructs that are easy to reason about or restricting language features that would make reasoning more difficult.
- Providing counterexamples to failing Move programs, i.e., inputs on which the specification fails.
- Verifying larger parts of Move standard library.

References

[1] Z. Amsden et al. *The Libra Blockchain.*
`https://developers.libra.org/docs/assets/papers/the-libra-blockchain/2019-09-26.pdf`

[2] S. Blackshear et al. *Move: A Language With Programmable Resources.*
`https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources/2019-09-26.pdf`

[3] D. Walker. *Substructural Type Systems*
`https://mitpress-request.mit.edu/sites/default/files/titles/content/9780262162289_sch_0001.pdf`

[4] K. R. M. Leino. *This is Boogie 2.*
`https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml178.pdf`

[5] P. Müller, M. Schwerhoff and A. J. Summers. *Viper: A Verification Infrastructure for Permission-Based Reasoning.*
`http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=MuellerSchwerhoffSummers16.pdf`

[6] R. Sierra, M. Eilers and P. Müller. *Verification of Ethereum Smart Contracts Written in Vyper.*
`https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Robin_Sierra_MA_Report.pdf`

[7] Ethereum Foundation. *Solidity, the Contract-Oriented Programming Language.*
`https://solidity.readthedocs.io/en/v0.6.3/`

[8] Ethereum Foundation. *Vyper.*
`https://vyper.readthedocs.io/en/latest/index.html`

[9] F. Schrans, S. Eisenbach, S. Drossopoulou. *Writing Safe Smart Contracts in Flint.*
`https://dl.acm.org/doi/10.1145/3191697.3213790`