Short Paper: Formal Verification of Smart Contracts

Karthikeyan Bhargavan[2] Antoine Delignat-Lavaud[1] Cédric Fournet[1]
Anitha Gollamudi[3] Georges Gonthier[1] Nadim Kobeissi[2] Aseem Rastogi[1]
Thomas Sibut-Pinote[2] Nikhil Swamy[1] Santiago Zanella-Béguelin[1]

[1]Microsoft Research [2]Inria [3]Harvard University

{antdl,fournet,gonthier,aseemr,nswamy,santiago}@microsoft.com
{karthikeyan.bhargavan,nadim.kobeissi,thomas.sibut-pinote}@inria.fr agollamudi@g.harvard.edu

Abstract

Ethereum is a cryptocurrency framework that uses blockchain technology to provide an open distributed computing platform, called the Ethereum Virtual Machine (EVM). EVM programs are written in bytecode which operates on a simple stack machine. Programmers do not usually write EVM code; instead, they can program in a JavaScript-like language called Solidity that compiles to bytecode. Since the main application of EVM programs is as *smart contracts* that manage and transfer digital assets, security is of paramount importance. However, writing trustworthy smart contracts can be extremely difficult due to the intricate semantics of EVM and its openness: both programs and pseudonymous users can call into the public methods of other programs. This problem is best illustrated by the recent attack on TheDAO contract, which allowed roughly $50M USD worth of Ether to be transferred into the control of an attacker. Recovering the funds required a hard fork of the blockchain, contrary to the *code is law* premise of the system. In this paper, we outline a framework to analyze and verify both the runtime safety and the functional correctness of Solidity contracts in F*, a functional programming language aimed at program verification.

Categories and Subject Descriptors F.3 [*F.3.1 Specifying and Verifying and Reasoning about Programs*]

Keywords Ethereum, Solidity, EVM, smart contracts

1. Introduction

The blockchain technology, pioneered by Bitcoin [7] provides a globally-consistent append-only ledger that does not rely on a central trusted authority. In Bitcoin, this ledger records transactions of a virtual currency, which is created by a process called mining. In the *proof-of-work* mining scheme, each node of the network can earn the right to append the next block of transactions to the ledger by finding a formatted value (which includes all transactions to appear in the block) whose SHA256 digest is below some difficulty threshold. The system is designed to ensure that blocks are mined at a constant rate: when too many blocks are submitted too quickly, the difficulty increases, thus raising the computational cost of mining.

Ethereum is similarly built on a blockchain based on proof-of-work; however, its ledger is considerably more expressive than that of Bitcoin's: it stores Turing-complete programs in the form of Ethereum Virtual Machine (EVM) bytecode, while transactions are construed as function calls and can carry additional data in the form of arguments. Furthermore, contracts may also use non-volatile storage and log events, both of which are recorded in the ledger.

The initiator of a transaction pays a fee for its execution measured in units of *gas*. The miner who manages to append a block including the transaction gets to claim the fee converted to Ether at a specified gas price. Some operations are more expensive than others: for instance, writing to storage and initiating a transaction is four orders of magnitude more expensive than an arithmetic operation on stack values. Therefore, Ethereum can be thought of as a distributed computing platform where anyone can run code by paying for the associated gas charges.

The integrity of the system relies on the honesty of a majority of miners: a miner may try to cheat by not running the program, or running it incorrectly, but honest miners will reject the block and fork the chain. Since the longest chain is the one that is considered valid, miners are incentivized not to cheat and to verify that others do as well, since their block reward may be lost unless malicious miners can supply the majority of new blocks to the network.

While Ethereum's adoption has led to smart contracts managing millions of dollars in currency, the security of these contracts has become highly sensitive. For instance, a variant of a well-documented reentrancy attack was recently exploited in TheDAO [2], a contract that implements a decentralized autonomous venture capital fund, leading to the theft of more than $50M worth of Ether, and raising the question of whether similar bugs could be found by static analysis [6].

In this paper, we outline a framework to analyze and formally verify Ethereum smart contracts using F* [9], a functional programming language aimed at program verification. Such contracts are generally written in Solidity [3],

a JavaScript-like language, and compiled down to bytecode for the EVM. We consider the Solidity compiler as untrusted and develop a language-based approach for verifying smart contracts. Namely, we present two tools based on F*:

Solidity* a tool to translate Solidity program to shallow-embedded F* programs (Section 2).

EVM* a decompiler for EVM bytecode that produces equivalent shallow-embedded F* programs that operate on a simpler machine without stack (Section 3).

These tools enable three different forms of verification:

1. Given a Solidity program, we can use Solidity* to translate it to F* and verify at the source level functional correctness specifications such as contract invariants, as well as safety with respect to runtime errors.
2. Given an EVM bytecode, we can use EVM* to decompile it and analyze low-level properties, such as bounds on the amount of gas consumed by calls.
3. Given a Solidity program and allegedly functionally equivalent EVM bytecode, we can verify their equivalence by translating each into F*. Thus, we can check the correctness of the output of the Solidity compiler on a case-by-case basis using relational reasoning [1].

1.1 Architecture of the Framework

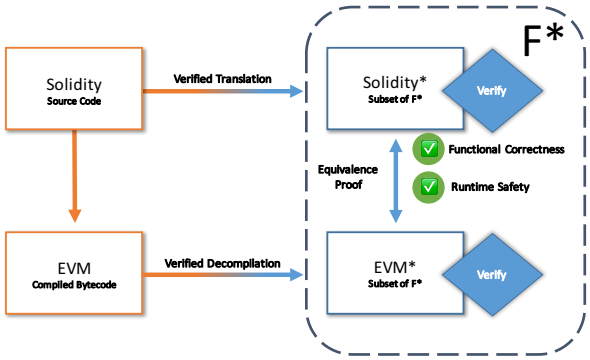

Figure 1. Overview of the architecture of our framework

Our smart contract verification framework is a two-pronged approach (Figure 1) based on F*. F* comes with a type system that includes dependent types and monadic effects, which we apply to generate automated queries to statically verify properties on EVM bytecode and Solidity sources.

While it is clearly favorable to obtain both the Solidity source code and EVM bytecode of a target smart contract, we design our architecture with the assumption that the verifier may only have the bytecode. At the moment of this writing, only 396 out of 112,802 contracts have their source code available on `http://etherscan.io`. Therefore we provide separate tools for decompiling EVM bytecode (EVM*), and analyzing Solidity source code (Solidity*).

⟨*solidity*⟩ ::= (⟨*contract*⟩)*

⟨*contract*⟩ ::= '`contract` ' @identifier '`{`' (⟨*st*⟩)*'`}`'

⟨*st*⟩ ::= ⟨*typedef*⟩ | ⟨*statedef*⟩ | ⟨*method*⟩

⟨*typedef*⟩ ::= '`struct` ' @identifier ' `{`' (⟨*type*⟩ @identifier '`;`')* '`}`'

⟨*type*⟩ ::= '`uint`' | '`address`' | '`bool`'
| '`mapping (`' ⟨*type*⟩ '`=>`' ⟨*type*⟩ '`)`'
| @identifier

⟨*statedef*⟩ ::= ⟨*type*⟩ @identifier

⟨*method*⟩ ::= '`function`' (@identifier)?'`()`' (⟨*qualifier*⟩)* '`{`'
('`var`' (@identifier ('`=`' ⟨*expression*⟩)? '`,`')+)?
(⟨*statement*⟩ '`;`')* '`}`'

⟨*qualifier*⟩ ::= '`private`' | '`public`' | '`internal`'
| '`returns (`' ⟨*type*⟩ (@identifier)? '`)`'

⟨*statement*⟩ ::= ε
| ⟨*type*⟩ @identifier ('`=`' ⟨*expression*⟩)? (*decl*)
| '`if(`' ⟨*expression*⟩ '`)`' ⟨*statement*⟩
('`else`' ⟨*statement*⟩)?
| '`{`' (⟨*statement*⟩ '`;`')* '`}`'
| '`return`' (⟨*expression*⟩)?
| '`throw`'
| ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*literal*⟩
| ⟨*lhs_expression*⟩ '`(`' (⟨*expression*⟩ '`,`')* '`)`'
| ⟨*expression*⟩ ⟨*binop*⟩ ⟨*expression*⟩
| ⟨*unop*⟩ ⟨*expression*⟩
| ⟨*lhs_expression*⟩ '`=`' ⟨*expression*⟩
| ⟨*lhs_expression*⟩

⟨*lhs_expression*⟩ ::=
| @identifier
| ⟨*lhs_expression*⟩ '`[`' ⟨*lhs_expression*⟩'`]`'
| ⟨*lhs_expression*⟩ '`.`' @identifier

⟨*literal*⟩ ::= ⟨*function*⟩
| '`{`' (@identifier '`:`' ⟨*expression*⟩ '`,`')* '`}`'
| '`[`' (⟨*expression*⟩ '`,`')* '`]`'
| @number | @address | @boolean

⟨*binop*⟩ ::= '`+`' | '`-`' | '`*`' | '`/`' | '`%`'
| '`&&`' | '`||`' | '`==`' | '`!=`' | '`>`' | '`<`' | '`>=`' | '`<=`'

⟨*unop*⟩ ::= '`+`' | '`-`' | '`!`'

Figure 2. Syntax of the translated Solidity subset

2. Translating Solidity to F*

In the spirit of previous work on type-based analysis of JavaScript programs [8], we advocate an approach where the programmer can verify high-level goals of a contract using F*. In this section, we present a tool to translate Solidity to F*, and a simple automated analysis of extracted F* contracts.

Solidity programs consist of a number of contract declarations. Once compiled to EVM, contracts are installed using a special kind of account-creating transaction, which allocates an address to the contract. Unlike Bitcoin, where an

address is the hash of the public key of an account, Ethereum addresses can refer indistinguishably to a contract or a user public key. Similarly, there is no distinction between transactions and method calls: when sending Ether to a contract, it will implicitly call the fallback function (the unnamed method of the Solidity contract). In fact, compiled contracts in the blockchain consist of a single entry point that decides depending on the incoming transaction which method code to invoke. The methods of a Solidity contract have access to ambient global variables that contain information about the contract (such as the balance in `this.balance`), the transaction used to invoke the contract's method (such as the source address in `msg.sender` and the amount of ether sent in `msg.value`), or the block in which the invocation transaction is mined (such as the miner's timestamp in `block.timestamp`).

In this exploratory work, we consider a restricted subset of Solidity, shown in Figure 2. Notably, the fragment we consider does not include loops. The three main types of declarations within a contract are type declarations, property declarations and methods. Type declarations consist of C-like structs and enums, and mappings (associative arrays implemented as hash tables). Although properties and methods are reminiscent of object oriented programming, it is somewhat a confusing analogy: contracts are "instantiated" by the account creating transaction; this will allocate the properties of the contract in the global storage and call the constructor (the method with the same name as the contract). Despite the C++/Java-like access modifiers, all properties of a contract are stored in the Ethereum ledger, and as such, the internal state of all contracts is completely public. Methods are compiled in EVM into a single function that runs when a transaction is sent to the contract's address. This transaction handler matches the requested method signature with the list of non-internal methods, and calls the relevant one. If no match is found, a fallback handler is called instead (in Solidity, this is the unnamed method).

2.1 Translation to F*

We perform a shallow translation of Solidity to F* as follows:

1. contracts are translated to F* modules;
2. type declarations are translated to type declarations: enums become sums of nullary data constructors, structs become records, and mappings become F* maps;
3. all contract properties are packaged together within a `state` record, where each property is a reference;
4. each method gets translated to a function, no defunctionalization is required since Solidity is first-order only;
5. we rewrite `if` statements that have a continuation depending on whether one branch ends in `return` or `throw` (moving the continuation in the other branch) or not (we then duplicate the continuation in each branch).
6. to translate assignments, we keep an environment of local, state, and ambient global variable names: local variable declarations and assignments are translated to `let` bindings; globals are replaced with library calls; state properties are replaced with `update` on the `state` type;
7. built-in method calls (e.g.`address.send()`) are replaced by library calls.

We show a minimalistic Solidity contract and its F* translation in Figure 3. The only type annotation added by the translation is a custom `Eth` effect on the contract's methods, which we describe in Section 2.2. The `Solidity` library defines the `mapping` type (a reference to a map) and the associated functions `update_map` and `lookup`. Furthermore, it defines the numeric types used in Solidity, which are unsigned 256-bit by default.

2.2 An effect for detecting vulnerable patterns

The example in Figure 3 captures two major pitfalls of Solidity programming. First, many contracts fail to realize that `send` and its variants are not guaranteed to succeed (`send` returns a `bool`). This is highly surprising for Solidity programmers because all other runtime errors (such as running out of gas or call stack overflows) trigger an exception. Such exceptions (including the ones triggered by `throw`) revert all transactions and all changes to the contract's properties. This is *not* the case of `send`: the programmer needs to undo side effects manually when it returns `false`, e.g. `if(!addr.send(x)) throw`.

The other problem illustrated in `MyBank` is reentrancy. Since transactions are also method calls, calling `send` is a transfer of program control. Consider the following malicious contract:

```
contract Malicious {
  uint balance;
  MyBank bank = MyBank(0xdeadbeef8badf00d...);

  function Malicious(){
      balance = msg.value;
      bank.Deposit.value(balance)();
      bank.Withdraw.value(0)(balance); // forwarding gas
  }

  function (){ // fallback function
    bank.Withdraw.value(0)(balance);
  }
}
```

It attacks the `Withdraw` method of `MyBank` by calling recursively into it at the point where it does its `send`. The `if` condition in the second `Withdraw` call is still satisfied (because the balances are updated after `send`, and there is no check that it was successful). Even though the `send` in the second call to `Withdraw` is guaranteed to fail (because unlike method calls, `send` allocates only 2300 gas for the call), it still corrupts the balance by decreasing twice, causing an unsigned integer underflow. After corrupting the balance,

```
contract MyBank {
  mapping (address ⇒ uint) balances;

  function Deposit() {
    balances[msg.sender] += msg.value;
  }

  function Withdraw(uint amount) {
    if(balances[msg.sender] ≥ amount) {
      msg.sender.send(amount);
      balances[msg.sender] −= amount;
    }
  }

  function Balance() constant returns(uint) {
    return balances[msg.sender];
  }
}
```

```
module MyBank
open Solidity

type state = { balances: mapping address uint; }
val store : state = {balances = ref empty_map}

let deposit () : Eth unit =
  update_map store.balances msg.sender
      (add (lookup store.balances msg.sender) msg.value)

let withdraw (amount:uint) : Eth unit =
  if (ge (lookup store.balances msg.sender) amount) then
    send msg.sender amount;
    update_map store.balances msg.sender
        (sub (lookup store.balances msg.sender) amount)

let balance () : Eth uint =
  lookup store.balances msg.sender
```

Figure 3. A simple bank contract in Solidity translated to F⋆

the malicious contract can freely withdraw any remaining funds in the bank.

Using the effect system of F⋆, we now show how to detect some vulnerable patterns such as unchecked `send` results in translated contracts. The base construction is a combined exception and state monad (see [9] for details) with the following signature:

```
EST (a:Type) = h0:heap // input heap
  → send_failed:bool // send failure flag
  → Tot (option (a * heap) // result and new heap, or exception
          * bool) // new failure flag

return (a:Type) (x:a) : EST a =
  fun h0 b0 → Some (x, h0), b0

bind (a:Type) (b:Type) (f:EST a) (g:a → EST b) : EST b =
  fun h0 b0 →
    match f h0 b0 with
    | None, b1 → None, b1 // exception in f: no output heap
    | Some (x, h1), b1 → g x h1 b1 // run g, carry failure flag
```

The monad carries a `send_failure` flag to record whether or not a `send()` or external call may have failed so far. It is possible to enforce several different styles based on this monad; for instance, one may want to enforce that a contract always throws when a send fails. As an example, we defined the following effect based on EST:

```
effect Eth (a:Type) = EST a
  (fun _ b0 → not b0) // Start in non-failsure state
  (fun h0 b0 r b1 →
     // What to do when a send failed
     b1 ⟹ (match r with | None → True // exception
             | Some (_, h1) → no_mods h0 h1)) // no writes
```

The standard library then defines the post-condition of `throw` to `fun h0 b0 r b1 → b0=b1 ∧ is_None r` and the post-condition of `send` to `fun h0 b0 r b1 → r == Some (b1, h0)`.

Simply by typechecking extracted methods in the `Eth` effect, we can detect dangerous patterns such as the send() followed by an unconditional write to the `balances` table in `MyBank`. Note that the safety condition imposed by `Eth` is not sufficient to prevent reentrency attacks, as there is no guarantee that the state modifictions before and after send preserve the functional invariant of the contract. Therefore, this analysis is useful for detecting dangerous patterns and enforcing a failure handling style, but it doesn't replace a manual F⋆ proof that the contract is correct.

Evaluation Despite the limitations of our tool (in particular, it doesn't support many syntactic features of Solidity), we are able to translate and typecheck 46 out of the 396 contracts we collected on `https://etherscan.io`. Out of these, only a handful are valid in the `Eth` effect. This is a clear sign that a large scale analysis of published contract is likely to uncover widespread vulnerabilities; we leave such analysis to future work.

3. Decompiling EVM Bytecode to F⋆

In this section we present EVM⋆, a decompiler for EVM bytecode that we use to analyze contracts for which the Solidity source is unavailable (as is the case for the majority of live contracts in the Ethereum blockchain), as well as low-level properties of contracts. A third use case of the decompiler that we do not further explore in this paper is to use EVM⋆ together with Solidity⋆ to check the equivalence between a Solidity program and the bytecode output by the Solidity compiler, thus ensuring not only that the compiler did not introduce bugs, but also that any properties verified at the source level are preserved. This equivalence proof could be done, for instance, using rF⋆ [1] a version of F⋆ with relational refinement types.

EVM* takes as input the bytecode of a contract as stored in the blockchain and translates it into a representation in F*. The decompiler performs a stack analysis to identify jump destinations in the program and detect stack under- and overflows. The result is an equivalent F* program that, morally, operates on a machine with infinite single-assignment registers which we translate as let bindings.

The EVM is a stack-based machine with a word size of 256 bits [10]. Bytecode programs have access to a word-addressed non-volatile storage modeled as a word array, a word-addressed volatile memory modeled as an array of bytes, and an append-only non-readable event log. The instruction set includes the usual arithmetic and logic operations (e.g. ADD, XOR), stack and memory operations (e.g. PUSH, POP, MSTORE, MLOAD, SSTORE, SLOAD), control flow operations (e.g. JUMP, CALL, RETURN), instructions to inspect the environment and blockchain (e.g. BALANCE, TIMESTAMP), as well as specialized instructions unique to EVM (e.g. SHA3, CREATE, SUICIDE). As a peculiarity, the instruction JUMPDEST is used to mark valid jump destinations in the code section of a contract, but behaves as a NOP at runtime. This is convenient for identifying potential jump destinations during decompilation, as jumping to an invalid address halts execution.

The static analysis done by EVM* marks stack cells as either of 3 types: 1. Void for initialized cells, 2. Local for results of operations, and 3. Constant for immediate arguments of PUSH operations The analysis identifies jumpable addresses and blocks, contiguous sections of code starting at a jumpable address and ending in a halting or control flow instruction (we treat branches of conditionals as independent blocks). A block summary consists of the address of its entry point, its final instruction, and a representation of the initial and final stacks summarizing the block effects on the stack. An entry point may be either the 0 address, an address marked with JUMPDEST, an immediate argument of a PUSH used in a jump, or a fall-through address of a conditional.

As a result of the static analysis, EVM* emits F* code, using variables bound in let bindings instead of stack cells. Many instructions can be eliminated in this way; the analysis keeps an accurate account of the offsets of instructions in the remaining code. Because the instructions eliminated may incur gas charges, we keep track of the fuel consumption by instrumenting the code with calls to burn, a library function whose sole effect is to accumulate gas charges. Figure 4 shows the F* code decompiled from the `Balance` method of the `MyBank` contract in Fig. 3.

We wrote a reference cost model for bytcode operations that can be used to prove bounds on the gas consumption of contract methods. As an example, Fig. 5 shows a type annotation for the entry point of the `MyBank` contract decompiled to F* that proves that a method call to the `Balance` function will consume at most 390 units of gas.

```
let x_29 = pow [0x02uy] [0xA0uy] in
let x_30 = sub x_29 [0x01uy] in
let x_31 = get_caller () in
let x_32 = land x_31 x_30 in
burn 17 (* opcodes: SUB, CALLER, AND, PUSH1 00, SWAP1, DUP2 *);
mstore [0x00uy] x_32;
burn 9 (* opcodes: PUSH1 20, DUP2, DUP2 *);
mstore [0x20uy] [0x00uy];
burn 9 (* opcodes: PUSH1 40, SWAP1, SWAP2 *);
let x_33 = sha3 [0x00uy] [0x40uy] in
let x_34 = sload x_33 in
burn 9 (* opcodes: PUSH1 60, SWAP1, DUP2 *);
mstore [0x60uy] x_34;
loadLocal [0x60uy] [0x20uy] (* returned value *)
```

Figure 4. Decompiled version of the `Balance` method of the `MyBank` contract, instrumented with gas consumption.

```
val myBank: unit → ST word
  (requires (fun h → sel h mem = 0 ∧ sel h gas = 0 ∧
    nonZero (eqw
      (div (get_calldataload [0x00uy]) (pow [0x02uy] [0xE0uy]))
      [0xF8uy; 0xF8uy; 0xA9uy; 0x12uy]))) // hash of Balance method
  (ensures (fun h0 _ h1 → sel h1 gas ≤ 390))

let myBank () =
  burn 6 (* opcodes: PUSH1 60, PUSH1 40 *);
  mstore [0x40uy] [0x60uy];
  ...
  let x_28 = eqw [0xF8uy; 0xF8uy; 0xA9uy; 0x12uy] x_3 in
  burn 10 (* opcode JUMPI *);
  if nonZero x_28 then
  begin (* offset: 165 *)
    // decompiled code of Balance method
  end
```

Figure 5. A proof of a bound on the gas consumed by a call to the `Balance` method of `MyBank`.

4. Conclusion

Our preliminary experiments in using F* to verify smart contracts show that the type and effect system of F* is flexible enough to express and prove non-trivial properties. In parallel, Luu et al. [6] used symbolic execution to detect flaws in EVM bytecode programs, and an experimental Why3 [5] formal verification backend is now available from the Solidity web IDE [4].

The examples we considered are simple enough that we did not have to write a full implementation of EVM bytecode. We plan to complete a verified reference implementation and use it to verify that the output of the Solidity compiler is functionally equivalent to the sources.

We implemented EVM* and Solidity* in OCaml. It would be interesting to implement and verify parts of these tools using F* instead. For instance, we could prove that the stack and control flow analysis done in EVM* is sound with respect to a stack machine semantics.

References

[1] G. Barthe, C. Fournet, B. Grégoire, P.-Y. Strub, N. Swamy, and S. Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, pages 193–205. ACM, 2014.

[2] V. Buterin. Critical update re: Dao vulnerability. `https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability`, 2016.

[3] Ethereum. Solidity documentation – Release 0.2.0. `http://solidity.readthedocs.io/`, 2016.

[4] Ethereum. Solidity-browser. `https://ethereum.github.io/browser-solidity`, 2016.

[5] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *22nd European Symposium on Programming, ESOP '13*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[6] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. Cryptology ePrint Archive, Report 2016/633, 2016. `http://eprint.iacr.org/2016/633`.

[7] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`.

[8] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P. Strub, and G. M. Bierman. Gradual typing embedded securely in javascript. In *POPL '14*, pages 425–438. ACM, 2014.

[9] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16*, pages 256–270. ACM, 2016.

[10] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. `http://gavwood.com/paper.pdf`.