Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL

Sidney Amani
Data61 (CSIRO)
NSW, Australia
Sidney.Amani@data61.csiro.au

Myriam Bégel*
ENS Paris-Saclay, Université Paris-Saclay
France
Myriam.Begel@ens-paris-saclay.fr

Maksym Bortin
Data61 (CSIRO)
NSW, Australia
Maksym.Bortin@data61.csiro.au

Mark Staples
Data61 (CSIRO) & School of CSE, UNSW
NSW, Australia
Mark.Staples@data61.csiro.au

Abstract

Blockchain technology has increasing attention in research and across many industries. The Ethereum blockchain offers *smart contracts*, which are small programs defined, executed, and recorded as transactions in the blockchain transaction history. These smart contracts run on the Ethereum Virtual Machine (EVM) and can be used to encode agreements, transfer assets, and enforce integrity conditions in relationships between parties. Smart contracts can carry financial value, and are increasingly used for safety-, security-, or mission-critical purposes. Errors in smart contracts have led and will lead to loss or harm. Formal verification can provide the highest level of confidence about the correct behaviour of smart contracts. In this paper we extend an existing EVM formalisation [8] in Isabelle/HOL by a sound program logic at the level of bytecode. We structure bytecode sequences into blocks of straight-line code and create a program logic to reason about these. This abstraction is a step towards control of the cost and complexity of formal verification of EVM smart contracts.

Keywords formal verification, blockchain, smart contracts, Ethereum, Isabelle/HOL

1 Introduction

Blockchain technology emerged to support financial transactions in the Bitcoin system, but has become increasingly important in many industries, with potential use in legal, medical, and supply chain industries. The Ethereum blockchain provides a general-purpose computational mechanism called *smart contracts*. These are programs that run on the Ethereum Virtual Machine (EVM) [19]. They and their effects are recorded in the blockchain history. They can be used to encode or execute agreements between parties. For example, a party invoking a smart contract could cause digital currency to be transferred to another party, or could record a state change which makes the other party eligible to invoke other transactions. Smart contracts provide new ways to implement multi-party relationships within blockchain-based applications. In addition to the direct financial value in these applications, blockchains are increasingly being used for safety-critical applications such as in pharmaceutical supply chains, or for mission-critical application such as in electrical power grids. For such reasons, it is highly desirable to know that smart contract implementations do not violate critical requirements, and formal modelling and verification can be applied in this context. To address these questions we seek:

(i) a trustworthy logical framework capable of expressing complex safety and security requirements;
(ii) a valid formal model of the EVM within the framework;
(iii) a sound program logic defined within the framework and able to reason about properties of smart contracts.

In our setting, we use the logical framework Isabelle/HOL, and an existing EVM formal model [8]. Thus, our remaining goal is a sound program logic. In this paper we propose such a logic for EVM bytecode. We target unstructured bytecode rather than a high-level programming language for the following reasons. First, our approach is independent of any high-level language (e.g. Solidity [5]) compiler, making our work more general and significantly less reliant on the correctness of higher-level tools. Second, bytecode is the actual programming language of Ethereum as all smart contracts appear only in this form on the blockchain. Altogether, this gives us the motivation to focus on reasoning about EVM bytecode, despite the absence of convenient programming constructs like conditionals, which we can take for granted in structured languages.

The main contributions of the paper are:

(i) an extension to the EVM formalisation [8] in the Isabelle/HOL theorem prover, covering smart contract correctness properties, and which gives a separate universal treatment of termination based on Ethereum's concept of execution 'gas';

*work done while at Data61 (CSIRO)

PL'17, January 01–03, 2017, New York, NY, USA
2017.

(ii) a sound program logic to verify smart contracts at the bytecode level; and
(iii) Isabelle tactics to support automated generation of verification conditions using the rules of the logic.

Our development is entirely formalised in Isabelle/HOL and has been accepted in the official EVM formalisation repository[1] maintained by the Ethereum foundation.

The paper is structured as follows. Section 2 describes the background for the presented work. Section 3 describes how we can capture correctness properties in a pre/postcondition style for EVM bytecode programs. Section 4 is devoted to our program logic, and Section 5 shows the soundness of the logic w.r.t. the correctness property. Section 6 presents a case study, which outlines the specification and verification of properties of bytecode generated by the Solidity compiler from a high-level smart contract, as well as how we automate generation of verification conditions using Isabelle tactics. Finally, Section 7 outlines some related work and Section 8 summarises the results and gives an outlook.

2 Background

The EVM is described in the Ethereum 'Yellow Paper' [19], which provides a foundation not only for its implementation, but also for formalisations in logic. One such formalisation has been done [8] using the 'meta-tool' *Lem* [12], which supports a variety of theorem provers including Isabelle/HOL. *Isabelle* [14] is a logical framework in form of a generic interactive theorem prover, whereas Isabelle/HOL encodes higher-order logic and is the most important and most developed part of the framework. Based on a small (meta)-logical inference kernel, Isabelle's LCF-style architecture ensures very high confidence about its soundness as a theorem prover.

However, the EVM model [8] needs to be validated to provide confidence that it meets the specification [19]. To this end, a validation test suite accompanies the model in Lem. Using this, the actual EVM, regarded as the reference implementation of [19], and the OCaml code generated by Lem are both applied to a large collection of contracts, cross-checking their outputs.

As we entirely focus on the Lem output in Isabelle, we wanted to have an additional validation of this particular EVM formalisation. To this end, we also invoked Isabelle's code generator and ran the test suite on the OCaml code generated by Isabelle.

Our 'double-validation' process is outlined in Figure 1. The use of the test suite from Isabelle has required some effort, mainly because of different representations of machine words: OCaml code from Lem uses efficient native modules, whereas the Isabelle side invokes a formally verified theory of machine words. Because of this, our suite needs much more time to pass the tests. Nonetheless, we gain a complementary indication that all three models of EVM follow the specification and behave equally.

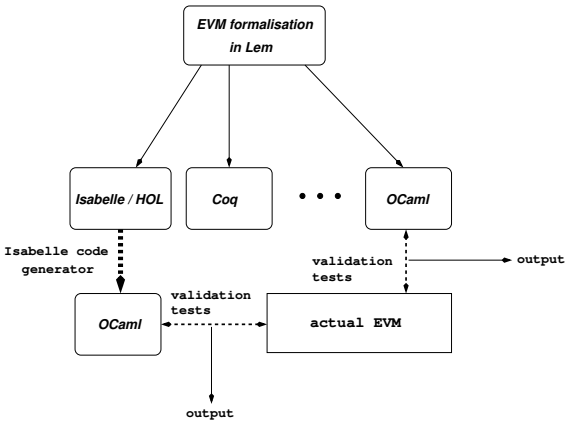

Figure 1. Validation of EVM models.

3 Total Correctness of EVM Bytecode Programs

In his PhD thesis [13], Myreen introduced a general method of formal verification of machine code with a particular application to ARM. In this section we show how this general method can be adapted to EVM, with additional consideration given to EVM specific properties rooted in gas consumption.

In the general model, a state carries all the information needed to execute a program, including instructions (with respective reference numbers) constituting the program itself, a program counter that refers to the current instruction, a stack and so on. All these elements are treated uniformly as sets of so-called *state elements* and separated in a state using separation logic conjunctions $\wedge^*$ (denoted by $*$ in [17]). A single machine step is captured by a function *next* that takes the current instruction via program counter from the state and transforms the state in accordance with the instruction's specified behaviour. Of course, *next* might not always be able to pick an instruction as an execution can have terminated properly or with an exception. This is indicated within a state by the *not-continuing* flag, which is just an abbreviation for the state element *ContinuingElm False*, as opposed to *continuing* flag abbreviating *ContinuingElm True*. If *not-continuing* is present in a state, *next* leaves the state unchanged.

A Hoare-style property of a program c is captured by a triple $\models \{P\}\ c\ \{Q\}$, where P and Q are separation logic predicates on the state. The triple is true iff for any state s and predicate F such that

$$(P \wedge^* code(c) \wedge^* F)\ s$$

[1] https://github.com/pirapira/eth-isabelle/

holds, there exists a natural number k such that

$$(Q \wedge^* code(c) \wedge^* F)\ next^k(s)$$

holds. The predicate F is usually called in this context a *frame*, and keeping it allows us to reason about parts of states locally. $code(c)$ is another element specifying that the program code is present in the state, and $next^k$ denotes k-times iteration of *next*. Such triples are highly generic and we cannot conclude much from many of these, except that under given preconditions the program c will pass a state satisfying Q. This changes immediately in cases where Q is of the form *not-continuing* $\wedge^* Q'$, now stating that c has reached a terminating state satisfying Q'. Hence, showing $\models \{P\}c\{\textit{not-continuing} \wedge^* Q'\}$ amounts to showing termination of the program c in a Q'-state.

This generic technique applies seamlessly to EVM programs as shown by Hirai [8]. However, we realised that showing termination of a contract individually is an unnecessary burden because Ethereum was designed in such a way that all smart contracts are guaranteed to terminate (either successfully or due to an 'out-of-gas' exception). More specifically, to ensure that miners get compensated for their costs incurred by operating the Ethereum blockchain, each EVM instruction has a gas fee. When invoking a contract, the initiator provides a gas budget proportional to how computationally expensive the execution is expected to be and every step of execution is deducted from the budget. If the gas consumption exceeds the budget, an 'out-of-gas' exception is raised, the miner keeps all of the gas and the state of the contract prior to the invocation is restored.

These blockchain specifics give us a termination order. Thus, we augment the EVM formalisation [8] by the function $next_\mu$ which consequently iterates *next* on each state as long as possible, i.e. the *continuing* flag is present. Hence, the essential property we show about $next_\mu$ is that for any state s there exists $k \geq 0$ such that

(i) $next_\mu(s) = next^k(s)$ holds;
(ii) for any l, such that $0 \leq l < k$, the state $next^l(s)$ contains *continuing*;
(iii) for any l, such that $l \geq k$, the state $next^l(s)$ contains *not-continuing*.

In other words, $next_\mu(s) = next^k(s)$ where k is the least number such that $next^k(s)$ reaches a state with the *not-continuing* element.

Now, regarding the contract correctness, we strengthen $\models \{P\}\ c\ \{Q\}$ to a total input/output property $\models [P]\ c\ [Q]$ which is true iff for any state s and frame F

$$(P \wedge^* code(c) \wedge^* F)\ s$$

implies

$$(Q \wedge^* code(c) \wedge^* F)\ next_\mu(s)$$

To sum up, what we achieved so far is to factor out the termination part which we have shown once and for all, thus removing this obligation from the verification process completely.

In the next section we will present our program logic processing EVM bytecode, which (based on soundness presented in Section 5) will give us a sound device to derive verification conditions for contract properties of the form $\models [P]\ c\ [Q]$. Although the program logic does not support general loops yet, it is worth noting that factoring out the termination part means that we would not need to provide any variants to establish loop properties since gas consumption is a variant for any EVM-loop. On the other hand, we might then need to augment our triples by an exception condition to handle cases when an 'out-of-gas' exception is raised during a loop iteration, thus not reaching the loop exit condition. Alternatively, we also might augment any such condition by a check of remaining gas amount, such that any loop would just exit if not enough gas is available to perform the next iteration. This approach would however require good approximations of gas consumed per iteration.

4 Program Logic

A Hoare-style program logic comprises a collection of rules that allow us to derive semantic properties of compound programs from properties of its parts. In the case of structured languages we usually have, for instance, a rule telling us that a program **if** C **then** p_1 **else** p_2 exhibits a certain input/output behaviour if p_1 and p_2 do so, however, with the additional precondition C available for p_1 and $\neg C$ for p_2. The situation is not that simple when we have to reason about EVM bytecode. At this level, a conditional compound construct appears merely as a jump instruction that transfers the flow of execution to another part of the program. In this sense, a program logic that treats the entire bytecode program simply as a list of instructions would be in principle feasible, but intricate. To this end more sophisticated techniques are available, such as decompilation via extraction of *Control Flow Graphs* (CFG), in particular applied to the Java Virtual Machine (JVM) code [20]. The aim of CFG extraction is to split a program into *basic blocks*, i.e. sequences of instructions without jumps, and connect them using edges corresponding to jumps. The essential property of basic blocks is that they comprise straight-line code, i.e. the control flow always enters it at its first instruction and leaves only after the last one has been executed. Thus, reasoning about bytecode at the CFG-level resembles reasoning about structured programs in many aspects. In particular, a CFG provides enough structure to add annotations, such as loop invariants, enabling complete automation of the verification condition generation process.

However, full CFG extraction poses more advanced challenges in EVM context than for JVM, especially from the formal modelling perspective. This is because JVM jump instructions take their target address as an immediate value

argument which can be determined statically, whereas in EVM jump destinations must be obtained from the stack, i.e. dynamically. For that reason, our bytecode preprocessing currently addresses basic block extraction only, presented in the next section. Then, Sections 4.2, 4.3 and 4.4 present how our logic handles programs, blocks and instructions, respectively.

4.1 Extraction of Basic Blocks

We divide EVM instructions into three groups:

(i) JUMPDEST indicates a jump destination and hence beginning of a basic block;
(ii) JUMP, JUMPI, UNKNOWN and all of *Misc*-instructions[2] indicate the end of a basic block (UNKNOWN and Misc-instructions interrupt program execution);
(iii) all remaining instructions.

Furthermore, we classify basic blocks with the following four types:

(i) *Terminal* — if the last instruction of the block interrupts execution;
(ii) *Jump* — if the last instruction is JUMP;
(iii) *Jumpi* — if the last instruction is JUMPI;
(iv) *Next* — otherwise, i.e. when control passes from the last instruction of the block to the instruction with the successor address.

block index	address	instruction	block type
0	0	OR	*Next*
	1	ADD	
	2	SWAP1	
3	3	JUMPDEST	*Jump*
	4	MLOAD	
	5	POP	
	6	JUMP	
7	7	DUP3	*Jumpi*
	8	PUSH1 0	
	10	ISZERO	
	11	JUMPI	
12	12	POP	*Terminal*
	13	RETURN	

Figure 2. A program split into four basic blocks, where grey instructions appear in the original code but are removed from the list of instructions of their block.

Figure 2 illustrates how we split EVM bytecode into basic blocks of different types, thereby indexing the blocks with addresses of their first instruction and removing all the jumps from the block contents. The entire extraction process is captured in the Isabelle development by means of the function *build-blocks* which maps a list of instructions to a list of tuples (n, xs, t), where n is the block index, xs is the list of instructions of the block, and t is the type of the block.

By splitting bytecode into basic blocks no information gets lost since we can connect the produced blocks in the right order and insert jumps back in accordance with block types. More precisely, we also have the function *connect-blocks* such that for any bytecode program c the identity

$$\textit{connect-blocks}(\textit{build-blocks}\, c) = c$$

holds.

Based on these preparations, the following predicates for reasoning at different levels will be defined inductively in the following three sections:

$\textit{blocks} \vdash_{\texttt{prog}} [P]\ (n, xs, t)\ [Q]$ – at the program level

$\vdash_{\texttt{block}} [P]\ xs\ [Q]$ – at the block level

$\vdash_{\texttt{instr}} [P]\ x\ [Q]$ – at the instruction level

where P, Q are state predicates, x is an instruction, xs is a list of instructions, *blocks* is a list of basic blocks, and (n, xs, t) — a basic block.

4.2 Program Rules

We start at the program level, where we have the following rules for each block type.

$$\text{(i)}\quad \frac{\vdash_{\texttt{block}} [P]\ xs\ [Q]}{\textit{blocks} \vdash_{\texttt{prog}} [P]\ (n, xs, \textit{Terminal})\ [Q]}$$

That is, a *Terminal*-block is simply passed to the level of blocks as we do not need to look at any other block after this has been processed.

The rule for a *Next*-block is different in this respect:

$$\text{(ii)}\quad \frac{\begin{array}{l}\vdash_{\texttt{block}} [P]\ xs\ [pc\ m \wedge^* R] \\ (m, ys, t) \in \textit{blocks} \\ \textit{blocks} \vdash_{\texttt{prog}} [pc\ m \wedge^* R]\ (m, ys, t)\ [Q]\end{array}}{\textit{blocks} \vdash_{\texttt{prog}} [P]\ (n, xs, \textit{Next})\ [Q]}$$

The state element $pc\ m$ determines the program counter after xs has been processed. Then we need to retrieve the next block associated to the index m from the structure *blocks*, which is expressed by $(m, ys, t) \in \textit{blocks}$, and proceed with (m, ys, t).

Further, in case of a *Jump*-block we additionally need to retrieve the address of the jump destination from the stack after the block has been processed. This yields the following, slightly more involved, rule:

$$\text{(iii)}\quad \frac{\begin{array}{l}\vdash_{\texttt{block}} [P]\ xs\ [R_1] \\ (j, ys, t) \in \textit{blocks} \\ \textit{head}\ ys = (j, \texttt{JUMPDEST}) \\ \textit{blocks} \vdash_{\texttt{prog}} [R_2]\ (j, ys, t)\ [Q]\end{array}}{\textit{blocks} \vdash_{\texttt{prog}} [P]\ (n, xs, \textit{Jump})\ [Q]}$$

where R_1 and R_2 abbreviate the conditions

[2] RETURN, STOP, SUICIDE, CREATE, CALL, CALLCODE, DELEGATECALL

$\langle h \leq 1023 \wedge g \geq 8 \rangle \wedge^* \mathit{continuing} \wedge^* \mathit{gas\text{-}pred}\ g \wedge^*$
$\mathit{pc}\ i\ \wedge^* \mathit{stack\text{-}height}\ (h+1) \wedge^* \mathit{stack}\ h\ j \wedge^* R$

and

$\mathit{continuing} \wedge^* \mathit{gas\text{-}pred}\ (g-8) \wedge^*$
$\mathit{pc}\ j\ \wedge^* \mathit{stack\text{-}height}\ h \wedge^* R$

respectively. Regarding the stack, *stack-height* $(h+1)$ and *stack* $h\ j$ specify a state where index h refers to the top of the stack containing j — the jump destination we are looking for. Regarding gas, *gas-pred* g binds the available amount of gas to g, whereas the part $\langle h \leq 1023 \wedge g \geq 8 \rangle$ is a *pure* condition, i.e. not dependent on state, and sets an upper bound for the height of the stack and a lower bound for the amount of gas, namely 8 units: as much as EVM requires to perform a jump, which is deducted by *gas-pred* $(g-8)$. Note that here and in the rules below we need to carry the *continuing* state element because the EVM model [8] imposes having either *continuing* or *not-continuing* in a state to process instructions.

The case of a conditional jump. i.e. a *Jumpi*-block, is similar, except that we also need to retrieve from the stack the value c to be compared to 0 and jump only if $c \neq 0$:

$$\text{(iv)}\quad \frac{\begin{array}{ll} \vdash_{\mathtt{block}}\ [P]\ \mathit{xs}\ [R_1] & \\ (j, \mathit{ys}, t) \in \mathit{blocks} & \\ (i, \mathit{zs}, t') \in \mathit{blocks} & \\ \mathit{head}\ \mathit{ys} = (j, \mathtt{JUMPDEST}) & \\ \mathit{blocks} \vdash_{\mathtt{prog}}\ [R_2]\ (j, \mathit{ys}, t)\ [Q] & \bullet \text{ if } c \neq 0 \\ \mathit{blocks} \vdash_{\mathtt{prog}}\ [R_3]\ (i, \mathit{zs}, t')\ [Q] & \bullet \text{ if } c = 0 \end{array}}{\mathit{blocks} \vdash_{\mathtt{prog}}\ [P]\ (n, \mathit{xs}, \mathit{Jumpi})\ [Q]}$$

where R_1, R_2, R_3 abbreviate the conditions

$\langle h \leq 1022 \wedge g \geq 10 \rangle \wedge^* \mathit{continuing} \wedge^* \mathit{gas\text{-}pred}\ g \wedge^*$
$\mathit{pc}\ (i-1)\ \wedge^* \mathit{stack\text{-}height}\ (h+2)\ \wedge^* \mathit{stack}\ (h+1)\ j \wedge^*$
$\mathit{stack}\ h\ c\ \wedge^* R$

and

$\mathit{continuing} \wedge^* \mathit{gas\text{-}pred}\ (g-10) \wedge^*$
$\mathit{pc}\ j\ \wedge^* \mathit{stack\text{-}height}\ h \wedge^* R$

and

$\mathit{continuing} \wedge^* \mathit{gas\text{-}pred}\ (g-10) \wedge^*$
$\mathit{pc}\ i\ \wedge^* \mathit{stack\text{-}height}\ h \wedge^* R$

respectively.

4.3 Block Rules

At the level of basic blocks we need only two simple rules that handle the cases of non-empty and empty lists of instructions to be processed:

$$\text{(i)}\quad \frac{\vdash_{\mathtt{instr}}\ [P]\ x\ [R] \qquad \vdash_{\mathtt{block}}\ [R]\ \mathit{xs}\ [Q]}{\vdash_{\mathtt{block}}\ [P]\ x{::}\mathit{xs}\ [Q]}$$

and

$$\text{(ii)}\quad \frac{P \Rightarrow Q}{\vdash_{\mathtt{block}}\ [P]\ \mathit{Nil}\ [Q]}$$

where $P \Rightarrow Q$ means $P\ s$ implies $Q\ s$ for any state s.

4.4 Instruction Rules

To be able to verify any possible EVM bytecode program we need to provide a rule for each of 70 EVM instructions. Presently we provide rules for 36 commonly used instructions, and extend this set gradually 'on-demand'. However, the behaviour of instructions referring to the Ethereum global state, such as `CALL` for inter-contract calls, cannot be fully captured by the EVM state alone. As a result, in the EVM semantics (and accordingly in our logic) these instructions lead to an environment action that must be modelled separately. Such modelling is however possible and we discuss it further in Section 8.

The following rule for `PUSH1`, the operation pushing one byte on the stack, is quite representative as it shows how we specify the necessary conditions for the instruction to be performed by the EVM in the precondition, as well as the effect of the operation on state elements in the postcondition:

$$\vdash_{\mathtt{instr}}\ [P]\ (n, \mathtt{PUSH1}\ x)\ [Q]$$

where P stands for

$\langle h \leq 1023 \wedge g \geq 3 \rangle \wedge^* \mathit{continuing} \wedge^* \mathit{gas\text{-}pred}\ g \wedge^*$
$\mathit{pc}\ n\ \wedge^* \mathit{stack\text{-}height}\ h\ \wedge^* F$

and Q for

$\mathit{continuing} \wedge^* \mathit{gas\text{-}pred}\ (g-3) \wedge^*$
$\mathit{pc}\ (n+2)\ \wedge^* \mathit{stack\text{-}height}\ (h+1) \wedge^* \mathit{stack}\ h\ x \wedge^* F$

In particular, we have stated that the height of the stack increases by 1 (*stack-height* $(h+1)$) and that the top index h of the stack points to the value x (*stack* $h\ x$) among the effects of `PUSH1`. As the size of the instruction including the byte to push is 2 bytes, the program counter increases by 2. It is also worth noting that we incorporate frames into such instruction-specific rules by carrying a variable F in pre- and postconditions, as shown above. This is in contrast to the more common way, which is introducing a generic *frame* rule (cf. [17]) of the form

$$\frac{\vdash_{\mathtt{instr}}\ [P]\ i\ [Q]}{\vdash_{\mathtt{instr}}\ [P \wedge^* F]\ i\ [Q \wedge^* F]}$$

and removing F from all instruction-specific rules. Although the rule is sound, this treatment leads to a considerable overhead in the verification process, since we would need to apply the frame rule each time an instruction-specific rule is applied.

Apart from the frame rule we still have two generic rules at the instruction level:

$$\text{(i)}\quad \frac{\vdash_{\texttt{instr}} [P']\ i\ [Q'] \qquad P \Rightarrow P' \qquad Q' \Rightarrow Q}{\vdash_{\texttt{instr}} [P]\ i\ [Q]}$$

$$\text{(ii)}\quad \vdash_{\texttt{instr}} [\langle \mathit{False} \rangle]\ i\ [Q]$$

The rule (i) is the usual 'consequence' rule, allowing us to adjust pre- and postconditions, whereas (ii) is needed to discharge trivial proof obligations having unsatisfiable preconditions. Such obligations arise frequently from conditional jumps (rule (iv), Section 4.2) where the condition is fully evaluated prior to the actual jump, such that we need to follow only one of the emerging branches.

The following section puts the program logic and the results of Section 3 together by means of a soundness property and outlines its proof.

5 Soundness

As our program logic is separated in three layers, we establish its soundness in three steps.

At the level of instructions, soundness basically amounts to the property

$$\frac{\vdash_{\text{instr}} [P]\ x\ [Q] \qquad (P \wedge^* \mathit{code}([x]) \wedge^* F)\ s}{(Q \wedge^* \mathit{code}([x]) \wedge^* F)\ \mathit{next}(s)} \tag{1}$$

which we prove by structural induction on $\vdash_{\text{instr}} [P]\ x\ [Q]$. By this, we need to show that the pre- and postconditions, as specified in each rule for individual instructions, are indeed covered by the behaviour of the respective instruction. Note that $\mathit{code}([x])$ ensures that $x = (\mathit{addr}, \mathit{instr})$ is present in the code-element of s, such that *next* executes precisely the instruction *instr*, if *pc addr* is present in s as well. This, in turn, is obtained from the preconditions of instruction rules, such as the PUSH1-rule from the previous section.

Next, at the level of blocks we show

$$\frac{\vdash_{\text{block}} [P]\ \mathit{xs}\ [Q] \qquad (P \wedge^* \mathit{code}(\mathit{xs}) \wedge^* F)\ s}{(Q \wedge^* \mathit{code}(\mathit{xs}) \wedge^* F)\ \mathit{next}^{|\mathit{xs}|}(s)} \tag{2}$$

where the usage of $\mathit{next}^{|\mathit{xs}|}$ is justified, since we consider a basic block *xs* which requires precisely $|\mathit{xs}|$ steps to be processed completely. By induction on $\vdash_{\text{block}} [P]\ \mathit{xs}\ [Q]$ we need to consider the cases when *xs* is non-empty or empty. In case $\mathit{xs} = x :: \mathit{zs}$ we can assume $\vdash_{\text{instr}} [P]\ x\ [R]$ and $\vdash_{\text{block}} [R]\ \mathit{zs}\ [Q]$ such that

$$\frac{(R \wedge^* \mathit{code}(\mathit{zs}) \wedge^* F)\ s}{(Q \wedge^* \mathit{code}(\mathit{zs}) \wedge^* F)\ \mathit{next}^{|\mathit{zs}|}(s)}$$

holds by the induction hypothesis. Furthermore, from (1) and $\vdash_{\text{instr}} [P]\ x\ [R]$ we can further conclude

$$\frac{(P \wedge^* \mathit{code}([x]) \wedge^* F)\ s}{(R \wedge^* \mathit{code}([x]) \wedge^* F)\ \mathit{next}(s)}$$

which combined establish (2). Thereby we also make use of the equality

$$\mathit{code}(x :: \mathit{zs}) = \mathit{code}([x]) \wedge^* \mathit{code}(\mathit{zs})$$

which holds since $x = (\mathit{addr}, \mathit{instr})$ does not occur in *zs* due to the unique *addr*.

Next, in case *xs* is empty, we can assume $P \Rightarrow Q$ which immediately gives us (2).

The ultimate soundness statement is at the program level:

$$\frac{\mathit{build\text{-}blocks}\ c \vdash_{\text{prog}} [P]\ \mathit{first\text{-}block}\ [Q] \qquad 0 < |c| < 2^{256}}{\models [P]\ c\ [Q]} \tag{3}$$

where *first-block* is a shorthand for the block with the smallest index in *build-blocks c*, and the assumption $0 < |c| < 2^{256}$ is necessary to avoid dealing with empty programs as well as programs with more than 2^{256} instructions (imposed by the design of EVM). In other words, in order to establish an input/output property specified by $\models [P]\ c\ [Q]$, we can transform *c* into its basic blocks *bs*, pick the first block *b* from *bs*, and apply the rules of our program logic to derive

$$\mathit{bs} \vdash_{\text{prog}} [P]\ b\ [Q]$$

However, in order to show (3) we need some preparations to be able to apply structural induction on the program logic rules. To this end we deploy our function *connect-blocks* and state the proposition

$$\frac{\mathit{bs} \vdash_{\text{prog}} [P]\ b\ [Q] \qquad b \in \mathit{bs} \qquad \mathit{wf\text{-}blocks}\ \mathit{bs}}{\models [P]\ \mathit{connect\text{-}blocks}\ \mathit{bs}\ [Q]} \tag{4}$$

where *wf-blocks* is our well-formedness predicate capturing all necessary technical details about block structure, essentially retaining the property

$$\frac{0 < |c| < 2^{256}}{\mathit{wf\text{-}blocks}(\mathit{build\text{-}blocks}\ c)}$$

for any program *c*.

Thus, the proposition (4) is a generalisation of (3), since for any *c* we can instantiate *bs* by *build-blocks c* and *b* by *first-block* in (4), and use the identity

$$\mathit{connect\text{-}blocks}(\mathit{build\text{-}blocks}\ c) = c$$

from Section 4.1. to obtain (3).

Unfolding the definition of $\models [P]$ *connect-blocks bs* $[Q]$ in (4) we further obtain

$$\frac{\begin{array}{l} bs \vdash_{\text{prog}} [P]\ b\ [Q] \\ b \in bs \\ \textit{wf-blocks}\ bs \\ (P \wedge^* \textit{code}(\textit{connect-blocks}\ bs) \wedge^* F)\ s \end{array}}{(Q \wedge^* \textit{code}(\textit{connect-blocks}\ bs) \wedge^* F)\ \textit{next}_\mu(s)} \quad (5)$$

and can proceed by induction on $bs \vdash_{\text{prog}} [P]b[Q]$. By this, we have to consider four cases: one for each type of the block b. So, for instance, if $b = (n, xs, \textit{Terminal})$, i.e. a terminal block, we have $\vdash_{\text{block}} [P]\ xs\ [Q]$. Since b is a part of the block list bs by assumption, we can separate some bs' such that

$$\begin{array}{l} (P \wedge^* \textit{code}(\textit{connect-blocks}\ bs) \wedge^* F)\ s \quad = \\ (P \wedge^* \textit{code}(xs) \wedge^* \textit{code}(\textit{connect-blocks}\ bs') \wedge^* F)\ s \end{array}$$

Hence, we can instantiate frame F in (2) by

$$\textit{code}(\textit{connect-blocks}\ bs') \wedge^* F$$

and consequently obtain

$$(Q \wedge^* \textit{code}(\textit{connect-blocks}\ bs) \wedge^* F)\ \textit{next}^{|xs|}(s)$$

As we consider a terminal block, the state $\textit{next}^{|xs|}(s)$ is the first one containing the *not-continuing* element, i.e.

$$\textit{next}^{|xs|}(s) = \textit{next}_\mu(s)$$

holds, concluding this case.

Although slightly more involved, the proof of the remaining three cases follows the same principles, making however additional use of the induction hypothesis.

6 Case study

Our development provides the ground work for full functional correctness of Ethereum smart contracts. These contracts are typically implemented in a high-level language called Solidity. In this case study, we demonstrate the practicality of our framework by formally specifying and verifying properties of the bytecode generated by Solidity compiler from an escrow agreement smart contract. Section 6.1 describes the design and implementation of the contract; Section 6.2, its specification; Section 6.3, its verification; and Section 6.4, the machinery we developed to maximize proof automation.

6.1 Design and Implementation

We designed an escrow agreement smart contract for the Ethereum blockchain. In our scenario, a *buyer* wants to purchase a good from a *seller*, but there is no particular reason to assume that they trust each other. To this end both parties rely on an escrow agent: an *arbiter* they both trust. In this agreement the arbiter creates the contract specifying the expected amount of Ether (the Ethereum crypto-currency), waits for the buyer to transfer the money to the contract and decides whether to pay the seller or refund the buyer. Apart from the arbiter's decision to pay or to refund, the purchase process is governed by the smart contract.

```
contract Escrow {                                                    1
    address buyer;                                                   2
    address seller;                                                  3
    address arbiter;                                                 4
    uint256 amount;                                                  5
                                                                     6
    function Escrow(address _buyer,                                  7
                    address _seller,                                 8
                    uint256 _amount) public {                        9
        require (amount > 0);                                        10
        buyer = _buyer;                                              11
        seller = _seller;                                            12
        arbiter = msg.sender;                                        13
        amount = _amount;                                            14
    }                                                                15
                                                                     16
    function addfund() payable public  {                             17
        require (amount > 0 &&                                       18
                 msg.value == amount &&                              19
                 msg.sender == buyer);                               20
        amount = 0;                                                  21
    }                                                                22
                                                                     23
    function refund() public {                                       24
        require (amount == 0 &&                                      25
                 msg.sender == arbiter);                             26
        selfdestruct(buyer);                                         27
    }                                                                28
                                                                     29
    function pay() public {                                          30
        require (amount == 0 &&                                      31
                 msg.sender == arbiter);                             32
        selfdestruct(seller);                                        33
    }                                                                34
}                                                                    35
```

Figure 3. Escrow smart contract allowing an arbiter to clear a transaction between two potentially distrusting parties.

Figure 3 shows our implementation of the contract. In Solidity, the functionality of a smart contract is encapsulated into a *contract interface* which has a well-defined interface with public and private elements, similarly to a class in object-oriented programming (OOP). Creating a contract on the blockchain instantiates the contract interface. Just like class instantiations in OOP, contracts are stateful objects where *storage variables* are stored on the blockchain.

Every variable declared as a member of a contract interface is a storage variable, and hence persists across multiple invocations of the contract. Lines 2–5 in Figure 3 declare variables carrying blockchain addresses of the three parties involved in the agreement, as well as *amount* that serves two purposes. It not only stores the amount of Ether for the

purchase, but also enforces the order in which the contract operations must be invoked.

The constructor part (lines 10–14) is run by the Ethereum platform to initialise the contract when it gets deployed. The environment variable *msg* carries the information about the Ethereum transaction that led to a contract deployment, including the address of the account that sent the transaction (*msg.sender*). Solidity's built-in function *require* throws an exception if the specified condition evaluates to false. In this case, only the amount of gas consumed so far is retained; the rest of the transaction is rolled back.

We use *require* (line 10) to ensure that the arbiter creates an escrow agreement with *amount* greater than zero. This condition is critical because *addfund* (line 21) expects *amount* to be strictly positive, before setting it to zero. This way we guarantee that the buyer can only add fund to the contract once. Note that *msg.value* is an environment variable telling us how much Ether has been transferred as part of the transaction.

Further, *addfund* is annotated as *payable*, which indicates that this function is allowed to be invoked by an Ethereum transaction along with a non-zero amount of Ether. By default, all contract functions are not payable in order to prevent accidental loss of Ether.

According to the *require*-clause on line 20, only the buyer shall be able to successfully invoke *addfund*. To this end, the exact amount of Ether (as specified by the arbiter) must be transferred along with the invocation. Similarly, only the arbiter is entitled to successfully invoke the functions *refund* (lines 25–27) and *pay* (lines 31–33). Moreover, in both cases *amount* is required to be zero, i.e. the buyer must already have placed the funds in the contract. Both functions make use of the EVM *selfdestruct* mechanism, which transfers the totality of the funds held by the contract to the address passed as argument, and subsequently destroys the contract. Note that the amount transferred by *selfdestruct* corresponds to the balance of the contract stored in the Ethereum global state, which in our case is just the *amount* specified by the arbiter during contract construction.

The keyword *public* is used in the contract to indicate that the respective function must be exported in the contract interface, thus getting accessible to the users. For the compiler this means adding the function to the abstract binary interface (ABI) and generating *dispatch* code to jump to this function within bytecode using the hash value, obtained from its signature. Since dispatcher code is added by the compiler, it is only visible at the bytecode level. Hence, verifying contract's bytecode implicitly means verifying the contract's dispatcher as well. As shown in the following section, in order to specify the behaviour of our contract, we compare the input hash to *addfund_hash*, *refund_hash* and *pay_hash*, and, in particular, require a safe behaviour in cases when the input hash does not match any of these hash values.

6.2 Specification

```
definition
  spec_Escrow :: [address, address, address,
                  address, 256 word, 32 word,
                  256 word]
                 ⇒ contract_action
where
  spec_Escrow sender buyer seller arbiter
              amount hash value =
 (if hash = addfund_hash ∧ sender = buyer ∧
     value = amount ∧ amount > 0 then
    ContractReturn []
  else if hash = refund_hash ∧
          sender = arbiter ∧ value = 0 ∧
          amount = 0 then
    ContractSuicide buyer
  else if hash = pay_hash ∧ sender = arbiter ∧
          value = 0 ∧ amount = 0 then
    ContractSuicide seller
  else
    ContractFail [ShouldNotHappen])

definition
  spec_amount :: [address, address, address,
                  256 word, 32 word, 256 word]
                 ⇒ 256 word
  spec_amount sender buyer seller amount
              hash value =
  (if hash = addfund_hash ∧ sender = buyer ∧
      value = amount ∧ amount > 0 then
    0
   else
    amount)
```

Figure 4. Functional specification of the Escrow smart contract.

Figure 4 and Figure 5 show our handwritten specification of *Escrow* and the functional correctness theorem we proved in Isabelle/HOL. The *spec_Escrow* definition is used in the main theorem to specify the return values expected by the contract. The arguments have the same names as the storage variables or the environment variables they carry the value for, except for *hash* which corresponds to the first 32-bit word of the input data byte string passed as argument to the bytecode. Note, that the actual hash values, denoted by the constants *addfund_hash*, *pay_hash* and *refund_hash*, are obtained from the ABI information generated by the Solidity compiler.

The specification *spec_Escrow* requires that

(i) *addfund* is only successful if it is called by the buyer who must simultaneously transfer the amount of Ether specified by the arbiter;

(ii) *refund* triggers a transfer of the totality of the funds held by the contract to the buyer via the self-destruct

mechanism only if called by the arbiter and *amount* = 0 holds;
(iii) *pay* behaves in the same way, except sending the funds to the seller;
(iv) any other input leads to a failure of the contract, cancelling the transaction.

In addition to specifying the return action of the contract, we also want to ensure that *addfund* can only be called once. This is done with the *spec_amount* definition which is used in the main theorem to specify the value of the *amount* storage variable after an invocation of the contract. It states that when the precondition of *addfund* to return successfully is met, *amount* is set to zero, thus disabling that precondition. As a result, invocations of *pay* and *refund* become enabled and of *addfund* — disabled.

```
theorem verify_escrow:
 ∃r. ⊨ [pc 0 ∧* stack_height 0 ∧*
        sent_data (word_rsplit hash) ∧*
        sent_value v ∧* caller sender ∧*
        memory_usage 0 ∧* continuing ∧*
        gas_pred 40000 ∧*
        storage 0 (ucast buyer) ∧*
        storage 1 (ucast seller) ∧*
        storage 2 (ucast arbiter) ∧*
        storage 3 (ucast amount) ∧*
        ... ]
         (build_blocks bytecode_Escrow)
       [action (spec_Escrow sender buyer seller
                  arbiter amount hash v) ∧*
        storage 0 (ucast buyer) ∧*
        storage 1 (ucast seller) ∧*
        storage 2 (ucast arbiter) ∧*
        storage 3 (spec_amount sender buyer
                     seller amount hash v) ∧*
        r]
```

Figure 5. The correctness statement in our program logic.

Now, the main theorem, shown in Figure 5, states the overall input/output property we proved. It has the form of Hoare triples introduced in Section 3. The precondition initialises the machine state, e.g. program counter is 0, stack is empty, etc. Importantly, *hash* is the 32-bit word contract argument used by the dispatcher code as well as the specification. We convert it into a string of bytes to form the input data of the contract using *word_rsplit*. Further, *gas-pred* sets a gas budget sufficient for invoking any of the functions, whereas *storage* binds storage entries to the free variables *buyer*, *seller*, *arbiter* and *amount*. Using *ucast* we convert 160-bit words, representing Ethereum addresses, into the storage unit, i.e. 256-bit word. The constant *bytecode_Escrow* denotes the actual bytecode of the contract, which we convert into a list of basic blocks with *build-blocks*.

Finally, in the postcondition we specify that

(i) the resulting action complies with *spec_Escrow*;
(ii) the storage variables *buyer*, *seller*, and *arbiter* contain the same values as in the input state;
(iii) the storage variable *amount* contains the value specified by *spec_amount*;

whereas the rest of the output state *r* remains unspecified.

6.3 Verification

Smart contract bytecode is divided into two sections: the pre-loader and runtime code. The pre-loader bootstraps the contract by deploying it on the Ethereum network and running its constructor. The runtime code only contains the core functionality of the contract that can be invoked by other blockchain agents. Our verification considers only the runtime code of *Escrow*, thus excluding its constructor.

We obtain the runtime code as a byte string of hexadecimal opcodes from the Solidity compiler and convert it into a list of EVM instructions via an Isabelle/HOL function. In this sense we work directly on the bytecode output of the Solidity compiler.

In this case, *bytecode_Escrow* comprises 191 EVM instructions that are split into 45 basic blocks, including 12 of type *Jumpi* (i.e. conditional jumps), 23 *Terminal*, 6 *Next* and 4 *Jump*. Once the automation support, described in the next section, reached maturity, proving correctness of the bytecode was rather a routine task.

Some creativity was required to handle all the cases where the contract ends with an exception. The existential variable in the postcondition of our correctness statement means that we have to provide a witness for the unspecified part of the state. The witness is automatically constructed by running our proof automation tactics, however, for this to work the existential variable must remain a unification variable until we reach a *Terminal* basic block. The *Terminal* block rule replaces the unification variable by a concrete value, and to ensure that each path gets a new unification variable, we manually distinguish each case in the proof.

A difficult proof arose when parsing the contract input data in order to extract the argument passed to the dispatcher. This involved dealing with operations on words of different size, which is notoriously hard. In particular, the instruction `CALLDATALOAD` reads a 256-bit machine word from the input data array. Since the hash value passed in the input data is only a 32-bit word, the dispatcher does the following word arithmetic to convert the value:

$$w32\text{-}hash = (w256\text{-}input\text{-}data \gg 224)\ \&\ \texttt{0xffffffff}$$

With the help of Isabelle's machine word library [2], we proved that when the hash is packed in the input data, the above bit-wise operations return the same hash value.

The total development of our framework is ≈6000 lines of Isabelle/HOL theories, excluding the existing formalisation of EVM model, etc. The size of the top-level specification of *Escrow* is ≈30 lines and the functional correctness proof of

the contract is ≈40 lines of proof specific to this example, which compares favourably with the ≈500 lines of reusable proof automation machinery we developed. We describe this automation next.

6.4 Automation

When reasoning about bytecode, even the verification of small contracts can involve long, tedious and repetitive proofs. Hence, our program logic was purposefully designed to be amenable to proof automation.

The inductively defined inference rules, presented in Section 4, are designed to be used in a syntax-driven verification condition generator (VCG). Shaping the rules in a way that the conclusion of at most one of them can match the subgoal at each point in the proof makes it easy to write a VCG that just tries applying all of the rules one after another. This tactic can be implemented with a few lines of Eisbach [11] — Isabelle/HOL's high-level tactic language. We developed a VCG for each level of our program logic. For instance, the program-level tactic looks as follows:

```
method prog_vcg =
 (prog_jumpi_vcg | prog_jump_vcg
 | prog_terminal_vcg | prog_next_vcg)+
```

where e.g., *prog_jumpi_vcg* is another tactic applying the *Jumpi*-rule (iv) in Section 4.2 and solving its resulting subgoals by invoking more specific tactics we designed. Altogether, *prog_vcg* tries each of the tactics separated by the | symbol, whereas the + sign at the end means that it will do that repetitively until none of the tactics is applicable.

A common source of friction we experienced during the early development phase of our framework was with proving goals of the form: $\forall s.\ R\ s \longrightarrow P\ s$ where P and R are separation logic predicates comprising the same separation conjunctions but in different orders. Such proof obligations arise when we weaken a precondition to be able to apply an $\vdash_{\texttt{instr}}$ rule. Since each $\vdash_{\texttt{instr}}$ rule expects a separation logic expression with conjunctions in a specific order, we routinely have to reorder these to match a given precondition.

To ease the pain, we reuse the separation logic algebra framework [9], which provides a set of generic Isabelle tactics to manipulate separation logic terms. We instantiated the algebra with the EVM machine state and created Isabelle tactics which reorder separation conjunctions such that, e.g. the first term in P matches the first one of R. Once the first elements match, we leverage the tactics of the separation algebra framework to remove the first element from both R and P.

An issue we encountered is that when R contains variables, we have to reorder terms in P in such way that the variables in R get instantiated in the correct order. When such problem occurs, we resort to manual reordering of terms.

7 Related Work

As a consequence of the repeated exploitation of security flaws in smart contracts, a notable amount of approaches and tools have already been proposed (e.g. [1, 3, 10]).

The major trend is to apply various kinds of static analysis not only at the level of structured contract languages (e.g. Solidity's Why3 backend [4, 16]) but also at the bytecode level [1, 10, 18]. The obvious advantage of static analysis approaches is that full automation can be achieved for contract properties that can be confirmed statically, e.g. certain orders of transactions [1]. The tools Oyente [10] and Porosity [18] decompile bytecode into a control flow graph and perform control flow analysis in order to detect common smart contract security defects such as re-entrancy bugs. These tools are of great value when added to the development process so they can unveil mistakes in early stages, but they do not prove functional correctness.

Our approach is more general as we aim to specify and verify contract properties in pre/postcondition style where the conditions can comprise any higher-order logic formula describing an EVM state. For that reason, the degree of automation is limited in our case such that the user will need to interact with the proof system to discharge elaborated claims.

Bhargavan et al. [3] proposed a technique using an intermediate functional language called F^*, which is more amenable to verification. It provides not only translation of a subset of Solidity programs to F^*, but also decompilation of EVM bytecode to F^* as well. This use of decompilation makes the approach similar to ours, since our program logic, in fact, resembles decompilation. As explained in Section 4, we split bytecode program into blocks without jumps and determine the actual jump destinations dynamically 'on-the-fly', by applying logic rules. By contrast, Bhargavan et al. perform static stack analysis to this end. However, a more striking difference is that our approach is homogeneous since every step of our verification process is performed and justified within a single, trusted logical framework without any translations to or from other formalisms. Such translations must be either assumed to behave correctly in some sense or formally modelled and verified, whereas we aimed to avoid both of these options.

KEVM [7] is a formal semantics of the EVM written using the K-framework. Like the Lem semantics [8] we use, KEVM is executable and therefore can run the Ethereum foundation's validation test suite. Reasoning about KEVM programs involves specifying properties in Reachability Logic and verifying them with a separate analysis tool. K supports translation for analysis with tools of varying power, including symbolic execution, faster concrete execution, or Isabelle. As explained earlier, we preferred the option of working in a single trusted logical framework.

8 Conclusions and Future Work

In this paper we have presented our approach to the verification of Ethereum smart contracts at the level of EVM bytecode. Building strictly on the thoroughly validated formal EVM model [8] in Isabelle/HOL, we have augmented it using the fact that EVM gas consumption allows us to state properties of contracts in pre/postcondition style with all termination considerations discharged. Further, we have outlined how we split contracts into a structure of basic blocks as well as how a sound program logic proceeds from such blocks down to the level of instructions. The presented case study has demonstrated the applicability of our program logic to real bytecode, verifying an escrow agreement smart contract implemented in Solidity. Moreover, the case study outlined how we use Isabelle tactics to automate large parts of verification condition generation process.

To further foster formal verification of Ethereum smart contracts, we could restore more of Solidity's control structures as well as function calls. For instance, restoring loops would require using heuristics to detect them and the program logic would be proved sound only for the subset of EVM bytecode accepted by this heuristic. Similarly detecting function calls in EVM bytecode requires complex stack analysis because the EVM provides no support to call subroutines and for stack unwinding. Thus, heuristics must be used to distinguish call sites and stack unwinding from other stack-manipulating instructions. However, the proposal [6] is currently being discussed, and suggests adding static jumps as well as instructions to call and return from sub-routines (internal functions) to the EVM instruction set. If implemented, it would greatly facilitate decompilation and reasoning about EVM bytecode, as most of the aforementioned issues would go away.

As mentioned in Section 4.4, our framework currently does not support reasoning about inter-contract message calls. When a contract A makes a message call to contract B, the execution of A terminates in our formal model with a *ContractCall* action. Hence, within our program logic we can only prove properties about the state of A right before it calls B. To reason further about such interactions, the Ethereum global state must be modelled separately to capture the behaviour of B. For example, Hirai [8] has extended his framework, on which ours is based, to formally verify that a smart contract throws an exception if it is subject to a re-entrant invocation (B attempts to call A). As future work, we could imagine an EVM formalisation parameterised by a contract environment which directly invokes the target contract when a message call occurs.

Another promising avenue for research is certifying compilers. We believe that the Ethereum community would greatly benefit from an EVM backend for the CakeML [15] verified compiler. Since CakeML is a functional programming language, it would provide a different approach to specification and verification of high-level properties of smart contracts.

Acknowledgments

We would like to thank Yoichi Hirai, Peter Höfner, Corey Lewis, Guillaume Vizier and Paul Rimba for all comments and suggestions.

References

[1] 2017. Securify. http://securify.ch. (2017).

[2] Joel Beeren, Matthew Fernandez, Xin Gao, Gerwin Klein, Rafal Kolanski, Japheth Lim, Corey Lewis, Daniel Matichuk, and Thomas Sewell. 2016. Finite Machine Word Library. *Archive of Formal Proofs* (June 2016). http://isa-afp.org/entries/Word_Lib.html, Formal proof development.

[3] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. ACM, New York, NY, USA, 91–96.

[4] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 – where programs meet provers. In *European Symposium on Programming*. Springer, 125–128.

[5] Ethereum foundation. 2017. Solidity documentation. https://solidity.readthedocs.io/en/develop/. (2017).

[6] Ethereum foundation. 2017. Subroutines and Static Jumps for the EVM. https://github.com/ethereum/EIPs/issues/615. (2017).

[7] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. 2017. *KEVM: A Complete Semantics of the Ethereum Virtual Machine.* Technical Report.

[8] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *WTSC'17, 1st Workshop on Trusted Smart Contracts, International Conference on Financial Cryprography and Data Security*.

[9] Gerwin Klein, Rafal Kolanski, and Andrew Boyton. 2012. Mechanised Separation Algebra. In *International Conference on Interactive Theorem Proving*, Lennart Beringer and Amy Felty (Ed.). Springer, Princeton, USA, 332–337.

[10] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.

[11] Daniel Matichuk, Toby Murray, and Makarius Wenzel. 2016. Eisbach: A Proof Method Language for Isabelle. *Journal of Automated Reasoning* 56, 3 (March 2016), 261–282. https://doi.org/10.1007/s10817-015-9360-2

[12] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 175–188.

[13] Magnus Oskar Myreen. 2009. *Formal verification of machine-code programs.* Ph.D. Dissertation. University of Cambridge, UK.

[14] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* Lecture Notes in Computer Science, Vol. 2283. Springer.

[15] Scott Owens, Michael Norrish, Ramana Kumar, Magnus O. Myreen, and Yong Kiam Tan. 2017. Verifying efficient function calls in CakeML. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 18.

[16] Christian Reitwiessner. 2016. Formal Verification of Smart Contracts. https://chriseth.github.io/notes/talks/formal_ic3_bootcamp. (2016).
[17] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*. IEEE, 55–74.
[18] Matt Suiche. 2017. Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode. https://github.com/comaeio/porosity/blob/master/defcon2017/dc25-msuiche-Porosity-Decompiling-Ethereum-Smart-Contracts-wp.pdf. (2017).
[19] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014).
[20] Jianjun Zhao. 1999. Analyzing Control Flow in Java Bytecode. In *Proc. 16th Conference of Japan Society for Software Science and Technology*. 313–316.