Formal Verification of Ethereum Smart Contracts

Mario M. Alvarez
University of California, San Diego; ConsenSys
mmalvare@eng.ucsd.edu, mario.alvarez@consensys.net

Abstract
Blockchain technology has the potential to fundamentally transform how humans collaborate. It enables mutually untrusting parties to come to consensus on a view of reality, from ledgers of transactions to the state of an ecosystem of software applications (commonly referred to as Smart Contracts.) However, users of these applications need to trust that they are implemented correctly: Smart Contracts cannot be changed once deployed, and can process digital assets worth a substantial amount of real money, meaning their failure can lead to significant financial losses. This paper discusses recent and ongoing work in applying formal verification technologies to Blockchain Smart Contracts, in order to help improve the reliability of these crucial pieces of decentralized infrastructure.

1. Introduction: Blockchain
By creating an unambiguous, transparent ordering of events, Blockchain technology enables mutually untrusting parties to come to consensus on a view of reality, from ledgers of transactions to the state of an ecosystem of software applications (commonly referred to as Smart Contracts) [3]. The term Smart Contracts is somewhat deceptive, since not all Smart Contracts implement behavior that we would associate with contracts in the legal sense of the term. While I will continue to use the term Smart Contracts, they are better thought of as Objects in an object-oriented system, able to respond to messages embedded in Ethereum transactions by changing their internal state or sending messages to other Smart Contracts (objects).

This class of systems rose to prominence along with a relatively new type of consensus algorithm, known as Nakamoto Consensus after its pseudonymous inventor(s), the creator of the Bitcoin system [13]. Nakamoto consensus algorithms enable untrusting parties to collaborate in establishing an ordering of discrete events (blocks) by tying production of new blocks to proof that scarce resources have been consumed in the production of the block. While blockchains based on more traditional consensus protocols are possible, the inherent leaderlessness and robustness of Nakamoto Consensus makes it particularly attractive for implementing these types of protocols.

This technology has wide-ranging applications. Perhaps the most famous is the original application, Bitcoin [13], which enables transactions between parties that do not trust each other: transactions are combined into blocks, which are validated using a proof-of-work scheme (described below). Systems such as Ethereum [3] provide a generalization of Bitcoin-like systems. In addition to allowing users to transact between each other, they also enable the encoding of transaction logic in a Turing-complete virtual-machine language, the Ethereum Virtual Machine (EVM) [20]. This enables a range of novel decentralized applications (dApps), ranging from funding vehicles (tokens/token sales) to "breedable" digital collectibles (e.g. CryptoKitties) to new kinds of decentralized registries [8] (discussed later in this paper). These transaction-logic programs are Smart Contracts.

1.1 Mining
To support their integrity, most Blockchains rely on a community of miners who are subject to both standard cryptographic assumptions (e.g. not being able to forge hashes) as well as economic incentives (the study of such incentive systems is an emerging field known as cryptoeconomics). Proof-of-work is one technique for establishing consensus among miners (hence, consensus on the state of the blockchain) that has been widely deployed and whose characteristics are well-understood. In a proof-of-work scheme [13], miners have consensus on a current block and compete to see who can produce the next block. In order to produce a block, miners compete to find a packet of data whose cryptographic hash is less than a certain value. Miners are allowed to include user transactions in the blocks they mine (enabling them to be compensated by the transaction fees other users pay to the network), as well as a nonce, a free-form piece of data whose value is chosen in order to make the resulting hash of the entire block less than the threshold.

Assuming the cryptographic hashes involved are secure, in order to find a new block with a sufficiently low hash, miners must repeatedly try hashing block contents with different nonces - essentially, brute-force. Miners who find blocks are incentivized to share them, because if they do so other miners will recognize the block they share as valid, ensuring that miner receives the fees to which they are entitled (depending on the system, miners may also be awarded a grant of new currency when they mine a block). The miner network has a policy that ensures convergence on a single chain - in Bitcoin, this is the policy that all mining should be done on the longest known chain with the most accumulated cryptographic "work" [13]. In this way, miners seeking to maximize their own profit will come to consensus on which chain is the longest, as it will be the chain on which they will want to mine.

There are some caveats here - for instance, if a sufficient number of miners collude (this can certainly be done with 51% of total mining power, described in the Bitcoin Whitepaper [13], although some lower thresholds have been proposed since), the integrity guarantees of the system may be compromised. This is not a purely academic concern, either: in Bitcoin, certain mining pools of collaborating Bitcoin miners engaged in profit-sharing among themselves have at times approached 50% of the network's total hash power [19]. It is believed that rational miners would never participate in a 51% attack, since doing so would degrade the value of the currency that the miner is attempting to mine. However, actors seeking to destabilize and destroy a cryptocurrency network might do so (e.g. nation states trying to undermine Bitcoin).

2. Blockchain and Software Defects
However, Blockchain technology is not a magical bullet. It cannot remove the need for trust, it only moves it. Instead of trusting each
other or mutually trusting a third party, participants in a blockchain consensus system instead need to trust the implementation of that system itself. In Bitcoin, this trust is limited to the implementation of the miners that make up the Bitcoin network, as well as trusting that a sufficient number of miners (51% or more) are not colluding to undermine the integrity of the network.

In a system that enables Smart Contracts, such as Ethereum, we also need to trust the implementation of the smart contracts that power the services that we want to use. Assuming Ethereum itself is functioning correctly, we can guarantee that the Smart Contracts in the Ethereum system will be executed correctly according to the semantics of the code that implements them. Ethereum Smart Contracts have a clear semantics in the sense that they are bytecode programs that run on a small, well-defined virtual machine (the Ethereum Virtual Machine, or EVM) [20].

2.1 Bugs in Ethereum Smart Contracts Can Be Costly
Like any programs written in a Turing-complete language, programs in EVM can have subtle and often surprising bugs. One famous example was the hacking of the TheDAO [5], a system of smart contracts intended to support a “Decentralized Autonomous Organization”: a sort of leaderless investment fund in which users could become members by contributing valuable digital assets in the form of Ether, Ethereum’s native currency, in exchange for the ability to vote on proposals to determine how TheDAO would invest those assets. It was enormously popular, attaining $150M worth of Ether assets, equal to about 15% of all outstanding Ether at the time. However, one of the core contracts implementing the system had a reentrancy bug, allowing an anonymous attacker to drain all the Ether stored in the DAO into a contract under their own control, ultimately forcing a “hard-fork” of Ethereum, in which the community agreed to break with the rules of the protocol to restore DAO funds to their original owners.

Another famous Ethereum smart contract bug with serious financial consequences was the failure of the Parity Wallet Library smart contract [4]. The Parity Wallet provided a framework for safely storing Ether and other valuables via a “multi-signature” scheme, wherein the digital signatures of multiple parties with joint custody over funds contained in the wallet needed to be confirmed before funds could be released for use. Anyone who wished to could deploy their own copy of the Parity wallet; however, in order to save on deployment costs, all these contracts relied on a separate smart contract - the “Parity Library” - to implement core functionality. As it turned out, the Library contract was written in such a way that it was deployed without an “owner”, an Ethereum account authorized to perform privileged operations on that contract, including deleting it. Because of the way the Parity Library was written and configured, this meant that any Ethereum account could take ownership of the Library smart contract. Once that account had ownership privileges, it was then able to delete the Library contract, rendering all deployed Parity wallets that depended on the library useless, and trapping within them any valuables they had stored. Eventually, this vulnerability was discovered and exploited by a user, leading to about $300M worth of Ether stored in those wallets to be rendered permanently inaccessible.

These two high-profile cases - along with many other, smaller ones - demonstrate the potential for bugs in EVM programs to have financial ramifications. As can be seen in these two cases, there is a tension between the need for truly trustworthy smart contracts to be un-patchable once released (since otherwise whoever has the ability to update them could compromise them) and the severity of consequences that an error in such a contract can impose on that contract’s users.

2.2 Current Mitigations
Currently, Ethereum smart contract authors attempt to defend against them by hiring external auditors, from outfits such as Zeppelin or ConsenSys Diligence. The foundation of these audits is a manual code review. While this is an important practice, it is not guaranteed to find all the bugs present in a smart contract system, and even a minor flaw that goes unnoticed during an audit can be fatal both for the system in question and for the reputation of the auditors (since the auditors’ names are usually publicly attached to the systems they audit.) Thus, auditors have come to rely increasingly on automated, formal-methods-based tools of the sort described below in order to gain a higher level of assurance about the correctness of the code they audit and to complement the strengths of traditional code review.

3. Applying Formal Verification to Blockchain
As can be seen from the above, in order for smart contracts to live up to their potential as “decentralized neutral third parties”, which can be reasonably trusted by anyone in the ecosystem who wishes to use them, a higher level of certification may be needed. Experience shows that, in mission-critical contexts where the cost of a software bug could include large financial losses, or the loss of human life, formal verification may be the only way to gain a level of assurance high enough that the mission-critical technology can be responsibly deployed.

Formal Verification is a relatively broad term, and encompasses a wealth of different techniques. Many of these techniques are already being applied to enhance the trustworthiness of smart contracts. Perhaps the most commonly applied to smart contracts are automated static analyses, which aim to provide a “push-button” approach to getting strong guarantees about the correctness of smart contracts. Because of fundamental limitations on fully-automated analyses of Turing-complete programs, such tools will necessarily always be unsound - that is, able to prove statements which are not true - or incomplete - that is, unable to prove statements which are true. In practice, unsoundness translates into false-negatives: an unsound tool can tell us our program is correct, when in fact it still has a bug. Incompleteness, on the other hand, manifests itself as false-positives: an incomplete tool might be unable to prove the absence of a bug, and therefore report a possible bug, despite the fact that there is no bug in practice.

3.1 Running Examples: Smart Contract Vulnerabilities
3.1.1 “Graffiti”: Unintended Ownership Semantics
One representative example of a bug involves a contract inadvertently exposing the possibility of an unintended change in its state, or of pieces of the Smart Contract’s state having unexpected semantics (e.g. because of missing checks). One crucial piece of state in a smart contract is the “owner” variable that is part of many contracts, which stores the identifier of the account able to perform administrative actions such as deleting the smart contract.

This class of vulnerabilities is broadly similar to the Parity Wallet bug discussed in [4]. First, here is a small, vulnerable smart contract exposing a single 256-bit integer data cell that any user of Ethereum can update as they wish (we call this smart contract “graffiti”). It also contains a destructor method that can be used to remove the smart contract from the Ethereum blockchain. While the intent of this contract is that only the owner (the account that created it) should be able to destroy the contract, a crucial check is missing in the destructor that enables anyone to destroy it, not dissimilar to the Parity exploit (in the case of Parity, there were some additional complications, but this is the essential principle.)
contract Graffiti {
  address public owner;
  uint256 public data;
  constructor () public {
    owner = msg.sender;
  }
  function update(uint256 payload) public {
    data = payload;
  }
  function kill() public {
    selfdestruct(msg.sender);
  }
}

contract Graffiti {  
  address public owner;  
  uint256 public data;  
  constructor () public {  
    owner = msg.sender;  
  }  
  function update(uint256 payload) public {  
    data = payload;  
  }  
  function kill() public {  
    selfdestruct(msg.sender);  
  } 
}

Here is another smart contract which, when deployed by a malicious user, can exploit unanticipated reentrancy in the "pay" function to drain all users' funds from "MiniDAO":

contract MiniDaoExploit {
  address public dao;
  uint256 balance;
  constructor (address daoAddr) public {
    dao = daoAddr;  // pass in the deployed MiniDAO contract
    balance = 0;
  }
  function pay(uint256 profit) public {
    balance += profit;
    dao.withdrawAll(address(this));
  }
}

Finally, here is a corrected version of "MiniDAO". By making sensitive state changes before calling the (potentially reentrant) "transfer" function, this vulnerability is avoided.

contract MiniDao {
  mapping (address => uint256) balances;
  uint256 totalBalance;
  function withdrawAll(address payable a) public {
    if (totalBalance > 0 && balances[a] > 0) {
      totalBalance -= balances[a];
      balances[a] = 0;
      a.transfer(balances[a]);
    }
  }
}

In the following sections, we will discuss different classes of tools for formal verification of Ethereum smart contracts, and their application to cases like these.

3.2 Automated Static Analyses
3.2.1 Mythril

One automated verification tool that has been put into practice within the Ethereum community is Mythril, a tool developed by ConsenSys. Mythril is routinely used by ConsenSys Diligence, one of the premier Ethereum smart contract auditing firms, as part of its auditing process. Additionally, Mythril aims to show that these kinds of tools can give rise to novel business models: the Mythril team is currently working on building more advanced versions of Mythril and offering them as a service, where users pay for access using Ethereum-based tokens as part of a wider token economy supporting Mythril.[17]. This is referred to as the Mythril Platform, and the current version of Mythril described above is referred to as Mythril Classic. While the Platform is still under development, it should be able to take advantage of the wealth of smart contracts submitted to it in order to refine its analyses and discover new classes of important bugs.

Mythril is implemented on top of LASER [12], an intermediate layer for translating Ethereum programs into SMT queries, which are then dispatched by the Z3 SMT solver [7]. Mythril supports...
a wide variety of analyses of Ethereum programs. These include classic general-purpose analyses such as detecting integer over- and underflows, and detecting whether exception states are reachable. Mythril also incorporates analyses that are more specific to Ethereum, such as detecting the sort of re-entrancy attacks that brought down TheDAO and “suicidal” contracts that can be killed by anyone, like the Parity Wallet Library. [12] It also has analyses suitable to particular applications: for instance, smart contracts that implement forms of gambling can use Mythril’s analysis to detect whether the “random” outcomes of a contract execution can actually depend on values which are predictable to the user - a common problem in a Blockchain context, since EVM smart contracts are unable to hide state from their clients. Mythril can also produce diagrams to aid in auditing and understanding of smart-contract code, such as control-flow graphs of the smart contracts being analyzed. [12]

While I would consider Mythril to be an instance of formal verification for Ethereum smart contracts - particularly given its usage of Z3 to validate queries about EVM programs - Mythril is careful to self-destruct itself as “not [being] equivalent to formal verification”. They are understating their case here, but what they mean is that Mythril is necessarily incomplete: in its current form, it is a collection of analyses that each aim to capture a class of bugs, but it cannot be used to completely prove the correctness of a smart contract in the same way as the foundational techniques discussed below.

Mythril is capable of detecting both the vulnerabilities described in the examples above. In the case of the “Graffiti” contract that any user can cause to self-destruct, Mythril has a self-destruct analysis that will detect the possibility of execution paths leading to unchecked self-destruct instructions and report them. Likewise, Mythril also has as built-in reentrancy analysis that can track whether state changes happen after potentially reentrant external calls, enabling it to detect the error in “MiniDAO”.

Examples of how Mythril can be extended with new analyses using Python can be found in [12].

3.2.2 Oyente

Oyente is another tool for automated analysis of Ethereum smart contracts. Like Mythril, it allows for translation of EVM smart contracts into control-flow graphs (CFG); properties of these CFGs to be verified are then translated into Z3 queries. [11] While Oyente is perhaps less widely deployed than Mythril, it has gained recognition for its technical merits: it was accepted at ACM CCS’16, and won the Kaspersky Cybersecurity Research Competition in 2017. Additionally, the accompanying paper presents a detailed formal operational semantics of the Ethereum EVM, and even provides some recommendations for how the design of the EVM could be improved. [11]

The system as described in [11] has a reentrancy analysis that would be able to detect the “MiniDAO” vulnerability; however, it does not have an analysis that checks explicitly for unguarded executions of selfdestruct, meaning it would not be able to detect the bug in “Graffiti”. However, because its semantics includes the ability to generate CFGs as well as a definition of the selfdestruct instruction (known as suicide at the time of the paper’s publication), it would be relatively trivial to extend Oyente to incorporate such an analysis.

3.2.3 Securify

Another automated verification tool that has both an academic pedigree and industrial application is the tool SECURIFY, built by researchers at ETH Zurich [18]. Like Mythril and Oyente, Securify checks for certain patterns that correlate with insecure smart contracts, such as unconstrained re-entrancy (which has the potential to lead to bugs such as experienced by TheDAO, described above). Securify also exposes a publicly-accessible API enabling community members to submit smart contracts to their system for evaluation, enabling Securify to collect a large dataset on which they can evaluate their approach. In their paper published at CCS2018 [18], the Securify authors evaluate their system against Mythril and Oyente, and show that for the set of smart contracts they have collected, the Securify tool outperforms these competitors in terms of finding a greater number of bugs with fewer false positives.

Securify also provides a domain-specific language (DSL) for expressing new semantic patterns to search for in attempting to identify security vulnerabilities in smart contracts [18], enabling it to be extended to new analyses more easily than Mythril and Oyente. This may enable Securify to more easily evolve as new classes of vulnerabilities affecting Ethereum smart contracts are discovered.

Like Mythril and Oyente, Securify works by “decompiling” raw EVM bytecode into an intermediate representation that can be more easily analyzed. In addition to extracting control-flow from the bytecode into a CFG representation, Securify also converts the stack-based EVM code into a Static Single Assignment form, [18] which helps to remove some of the complexity involved in reasoning about stack-based computation.

In the cases of the examples presented above, Securify’s built-in analyses should enable it to detect the bug present in “MiniDAO”, as these kinds of reentrancy errors correspond to a built-in analysis (which is also part of the system’s acceptance-test suite - see DAOTest.java in Securify’s Github repo[1]).

Likewise, Securify also contains built-in analyses for detecting contracts in which funds can become locked due to selfdestruct instructions, enabling the analyses to also treat cases similar to “Graffiti”. (For tests covering these cases, see LockedEtherTest.java [1].)

3.3 Abstractions and Languages for Verification of EVM Smart Contracts

3.3.1 Effective Callback Freedom

In addition to the above-mentioned automated frameworks, some interesting new languages and abstractions have been proposed to help facilitate the creation of secure smart contracts. One of these is an abstraction called Effective Callback-Freedom (ECF), described by Grossman et al [9]. The aim of this abstraction is to describe how reentrancy in objects’ message handlers (e.g. Ethereum smart contracts’ API functions) has the potential to lead to unexpected state changes of the kind that brought down TheDAO. A dynamically ECF invocation of a smart contract is one for which there exists an execution with the same starting and ending states that does not use reentrant callbacks. In such cases, the reentrancy displayed by the execution is not “essential” to the behavior witnessed by the execution, since the same result could have been obtained without any reentrancy at all. This is why these executions are referred to as effectively callback-free.

The authors demonstrate that if smart contracts can be proven to obey the ECF condition, modularity is enhanced: one can reason about their behavior in isolation, without having to worry about how the behavior of other smart contracts in the system can affect their internal state in unexpected ways by means of reentrancy. [9]

In addition to defining this abstraction, Grossman’s team also describes and implements and algorithm for efficiently checking real-world smart contract traces for effective callback-freedom. The algorithm abstracts smart-contract traces into a graph whose nodes are invocations of smart contracts, and then attempts to topologically sort this graph in order to check for cycles. The presence of a cycle in the graph indicates a reentrant invocation of a smart contract that violates ECF. Because of the efficiency
of topological sorting, this method can be done on-line; that is, immediately following the execution of any Ethereum transaction (leading to a trace of potentially nested smart contract calls), the algorithm can be run to check for suspect reentrancy. [9]

The authors propose that if an ECF checker had been built into the Ethereum virtual machine at the time of the DAO hack, the loss of funds could have been prevented: their checker would have detected the suspect reentrancy immediately and rejected the transactions containing them. [9] Additionally, the authors run their algorithm on the entire history of Ethereum transactions, observing that “only 10 executions out of 100 million” represented non-malicious execution traces that violated ECF, adding strength to their claim that ECF is a reasonable constraint on smart contract behavior that does not rule out very many legitimate executions.

In the case of the examples described above, the ECF detection system described in [9] would be able to detect malicious executions of the vulnerable “MiniDAO” contract dynamically, at runtime, and enable them to be aborted (assuming the Ethereum EVM specification were updated to include such behavior).

3.3.2 Scilla

Scilla, a project by Sergey et al [16], is a new language for writing EVM programs that aims to enhance security by restricting the flexibility of contracts that can be written. Rather than allowing programs with arbitrary Turing-complete executions, Scilla smart contracts are isomorphic to communicating automata, transition between a fixed set of states by means of transition functions which are guaranteed to terminate.

Scilla enforces that places where reentrancy occurs in smart contracts must be tail calls after which no further state manipulation or computation can occur in the smart contract that makes them. The authors observe that this condition is sufficient to prevent the sort of exploit that affected TheDAO. Scilla supplies a formal semantics for its programs (specified in terms of the automatatheoretic formalism described above) and defined in Coq for easy integration into formal interactive proof developments (for more on such developments, see the section on foundational proofs, below). In particular, more advanced properties of Scilla programs, such as temporal properties over a series of transactions of the contracts (e.g. “the balance may increase or decrease over time, but it never drops below 1 ETH”). [16]

Scilla would help to guarantee by construction that the “MiniDAO” bug cannot happen, by ensuring that all calls to external contract code are tail-calls (as is the case in the corrected version of “MiniDAO”), but not the vulnerable version). Additionally, it would help prevent the more complex bug that affected the Parity wallet by making state changes (such as reassigning the owner of the smart contract) more fully explicit, though not necessarily the bug affecting “Graffiti”, which involves a missing check rather than a change of ownership.

3.4 K Semantic Framework

The K Semantic Framework [15] provides another approach to verifying Ethereum smart contracts that falls somewhere in between fully-automated and interactive verification. The K framework provides a system for specifying the semantics of languages, and this system is connected to a formalism based on parallel graph-rewriting. This allows K to provide a wealth of tools to the user at low cost. For instance, given a programming language semantics specified in K’s language, the K Framework can generate an interpreter for that language without any additional user effort. The interpreters created in this way can leverage the high-performance graph-rewriting engine on which K is built to achieve reasonable performance in practice.

3.4.1 KEVM

The K framework has been used to implement a semantics for the Ethereum Virtual Machine [20], a project called KEVM. By using K’s graph-rewriting system, KEVM is able to verify properties about EVM programs. KEVM also derives an interpreter for the EVM that demonstrates acceptable performance: the authors of KEVM noted that they were able to take their automatically-generated interpreter and use it to process transactions “in the wild” on the real Ethereum blockchain. The performance of this VM is such that it could be used as the basis for an Ethereum “full node”, able to mine and validate blocks (which entails processing executions of EVM programs), with the advantage that its behavior is guaranteed to match the semantics of EVM as specified in K. Because the K style of specification more closely matches the operational-semantics-like approach used to specify EVM in [20], the official reference manual of EVM behavior, it can be particularly useful as a reference implementation of the EVM against which other (more performant) implementations can be validated.

KEVM has also been used to perform verification of realistic Ethereum smart contracts. For instance, KEVM was used to specify and verify an implementation of the ERC-20 token protocol. ERC-20 compliant contracts form a critical part of the Ethereum ecosystem, because they are used to implement the digital “tokens” that are frequently used in Ethereum to raise funds and to influence user behavior using cryptoeconomic incentive systems.

For example, one exciting idea tokens make possible is that of a token-curated registry (TCR). In a TCR, users collaborate to curate a list - say, a list of trustworthy news sources, as in the case of Civil, a ConsensSys project. Users who wish to join the list need to acquire and stake special tokens in order to apply for membership in the registry. Once they have applied for entrance, token-holders take a weighted vote on the merits of the application and decide whether to grant membership. Having a token for the registry (separate from, say, Ether) creates a particular system of incentives: the higher-quality the list, the more applications it will see, driving up the price of its tokens. From the token-holder-voter’s point of view, these tokens act as a store of value, but their value is tied directly to the perceived quality of the list. Thus, if a registry like Civil lets in low-quality news sources, the token-holders find the value of their holdings diminished. Thus, they have an incentive to behave in good faith.

The authors of KEVM have also used the K framework to build a translation from the for the Viper smart contract language, a Python-like language that is gaining in popularity in the Ethereum ecosystem. [6] This provides an important tool for reasoning about the correctness of Viper programs by means of their translation to EVM, which is captured by an extension to the KEVM framework. In defining this semantics, they found several bugs with implications for smart contract security in the existing production Viper compiler.

While the K-framework can be used in an automated way to check assertions about EVM programs, it also provides a layer for interfacing with external tools. This can be useful, for example, if a proof requires manual inductive reasoning that cannot easily be expressed within the K framework. [15] However, this would mean that K is being used in an interactive mode, sharing many of the similarities and pitfalls to the interactive verification tools discussed below.

When combined with a Solidity compiler, KEVM can be used to prevent both the bugs described in the examples above (“Graffiti” and “MiniDAO”). However, because K is a semantics framework rather than a static analysis framework, the user would be expected to encode the desired smart contract properties (“Graffiti” has no unguarded selfdestruct instructions; “MiniDAO” does not update any state after external calls) in the K specification language. Once
4. Foundational Interactive Verification

By foundational verification, researchers generally mean systems for stating and proving theorems (proof assistants) in such a way that the correctness of a purported proof can be checked mechanically by a trusted algorithm, known as the kernel of the proof assistant. This kernel should adhere to the De Bruijn Criterion: the kernel of any foundational theorem-proving system should be as small as possible, in order to build trust in the system by minimizing its trusted codebase (TCB). In a foundational system, we only need to trust the correctness of the kernel and of the definitions users (and library developers) build on top of it.

This distinguishes foundational approaches from systems like Z3, in which the entirety of the Z3 codebase must be considered "trusted" in order to trust the proofs it generates. Z3 is a large project, and as such has experienced bugs that compromise its soundness. This is a particularly acute problem for Z3's theory solvers, subsystems that Z3 uses to answer queries over specialized domains such as strings, floating-point numbers, or bit-vectors. [7] Bugs in a theory solver can propagate throughout the rest of the system, invalidating proofs that do not appear to directly depend on these theories.

4.0.1 How Foundational Verification Works

Foundational systems for software verification - including the systems discussed in this section - tend to be of an interactive flavor. In this paper, I will use the two terms interchangeably. In an interactive theorem prover, a push-button solution is not provided. Instead, the user is able to define programs and data in a programming language, write specifications in a logical specification language, and then prove the correctness of programs with respect to their specifications by using a special scripting language called a tactic language. During a proof, the theorem prover presents the user with a view of the proof state, consisting of one or more goals. Tactic scripts manipulate this proof state to either solve the goals, removing them from the proof state, or manipulate the goals to produce a new proof state, which is (ideally) closer to a completed proof. Since the process of completing a proof in a system like this involves an iterative process of trying tactics and analyzing the proof state produced to determine which tactics to try next, these systems are said to be interactive rather than automated. However, some tactics may provide a significant degree of automation to aid in this interactive process. In the ISABELLE/HOL proof assistant, for instance, the AUTO tactic can solve many goals in one shot, provided they have the right structure. [14]

One significant advantage that interactive, foundational systems have over their automated counterparts is support for mathematical induction. Many theorems which are true do not hold inductively: that is to say, the induction principles involved (say, the induction principle for a data type or inductive relation that is at the center of the proof) may require the initial theorem to be in a particular form, or else the inductive hypotheses generated by the principle are not useful in proving the conclusion of the inductive case of the theorem. The typical approach in interactive systems is to strengthen the statement of the theorem into a new theorem that implies the old one but holds inductively (that is, generates acceptable inductive hypotheses in its inductive cases). Armed with a strengthened statement of the theorem, the base and inductive cases can then be proved interactively through a series of tactic applications.

It should be noted that, by supplying inductive versions of the necessary theorems manually, users may be able to achieve these proofs in automated systems. Even with strengthenings of the necessary theorems so that they hold inductively, an automated theorem prover may not be able to prove the inductive cases from the hypotheses generated by the strengthened theorem, because of the size of the search space involved. Thus, when reasoning using induction, an interactive approach has many advantages.

4.0.2 Eth-Isabelle

Many projects in foundational verification for Ethereum are based on the ISABELLE/HOL proof assistant [14], an interactive theorem prover built on higher-order classical logic. The kernel of Isabelle - referred to as ISABELLE/PURE [14], is a procedure based on resolution - that is, using substitution and unification to solve goals with one or more premises and a single conclusion. All of the higher-level systems built on top of Isabelle make use of this trusted core, achieving a clean separation of concerns consistent with the foundational discipline. Isabelle's popularity in the Ethereum ecosystem stems from the success of the ETH-ISABELLE project [10], a formal specification of the semantics of the EVM built within Isabelle. By virtue of being built inside Isabelle, a system with native support for interactive inductive reasoning, ETH-ISABELLE provides benefits over KEVM, which depends on external tools for some of these more advanced styles of reasoning.

ETH-ISABELLE provides a formal semantics of the Ethereum Virtual Machine inside of the ISABELLE theorem prover. In particular, it provides a complete description of EVM states, as well as interpreters for EVM instructions and programs. ETH-ISABELLE also defines a Hoare-style relational semantics on top of this interpreter, which enables more compositional reasoning about EVM programs. Because the core of ETH-ISABELLE's semantics is an interpreter implemented in Isabelle's functional programming language, it can easily be extracted into a relatively efficient OCaml program. Thus, like KEVM, ETH-ISABELLE can be used as a reference implementation for the core of an Ethereum client. However, unlike KEVM, ETH-ISABELLE does not model the global state of the entire ecosystem of Ethereum smart contracts. While on the face of it this would seem like a large limitation, it can be overcome by using modular reasoning of the style discussed above in Grossman et al [9].

Because of the need for expert effort in interactive verification, ETH-ISABELLE has seen less adoption than the automated approaches discussed above. Nonetheless, it has been used to prove properties about some interesting smart contracts, including a multi-signature wallet and an Escrow smart contract that handles secure exchange of digital valuables.

In the case of the examples above, ETH-ISABELLE could be used to prove the absence of unchecked selfdestruct instructions in the corrected "Graffiti example", and could be used to show that all executions of the corrected "MiniDAO" example are indeed Effectively Callback Free, guaranteeing that the smart contract's internal state (its internal Ether balance) maintains the invariant that the sum of user balances adds up to the total balance of Ether.

4.1 My work: Elle

In the last 18 months of my work at ConsenSys, I have been developing ELLE, a foundational verified compiler from a structured language to EVM. [2] One of the great challenges of reasoning about EVM bytecode programs is the fact that EVM has no native notion of scope: all jumps are to absolute locations in the body of the smart contract. This means that information about the control structures present in the original source code is lost during the translation to bytecode. While this structure can be partially reconstructed - as is done in the automated systems discussed above - it would be better to have a convenient representation that captures these essential structures without obscuring other low-level
properties of the code. This is what ELLE aims to be: it provides a framework for composing EVM sub-programs with trivial control flow into larger, structured programs whose control-flow behavior is structured and understandable. In ELLE, all labels and jumps are scoped, and jumps are only allowed to targets that are in scope.

As a first application for the ELLE compiler backend, I have built a compiler that translates from the LLL language to EVM. LLL (short for “low-level language” or “lisp-like language”) is essentially a macro assembler for EVM with an S-expression-like syntax. However, one of the few abstractions LLL provides over raw EVM is structured control flow: LLL provides if, for, and other familiar primitives that make programs easier to write and reason about.

4.2 Elle’s Generality

In fact, ELLE is more than just a compiler from a structured language to EVM - it implements a discipline for achieving a “separation of concerns” between control flow and instructions, and thus can be seen as a discipline for building verified compiler backends for a variety of different platforms. To use ELLE with any machine language, all that is needed is to identify which instructions affect control flow and which do not. Instructions that change the flow of control (i.e., change the machine’s program counter other than incrementing it) are forbidden within ELLE programs, because otherwise arbitrary control flow would be possible and the modularity reasoning guarantees ELLE provides by virtue of its scoping system would be invalidated. Next, implementations for ELLE’s jump, jumpi (conditional jump), and label (jump target) instructions need to be provided. For all other operations, ELLE simply allows the insertion of machine-code instructions as ELLE statements, thus achieving a clean separation of concerns between control flow - which is handled by ELLE - and all other computation, which is simply delegated to the underlying machine.

4.3 Elle’s Semantics

Elle’s semantics are defined as a fairly standard big-step operational semantics. Unlike some presentations of operational semantics, rather than explicitly referring to the semantics of sub-programs in inductive cases, Elle uses a “virtual program counter”, which describes the position in the syntax tree that is currently being evaluated. This enables Elle’s semantics to never lose track of essential context when evaluating sub-programs deeper in the syntax tree, which is important since sub-trees may contain JUMP statements that bring the flow of execution out of that sub-tree. A zipper-like data structure could also be used to achieve a similar effect, and might be more efficient computationally, but would also have the effect of complicating Elle’s proof of correctness (rather than passing around the full syntax tree and a location within that tree to execute, the semantics would pass around a zipper, and predicates describing “correct” zippers for Elle syntax would have to be defined). Refactoring Elle to use zippers is potential future work, but may represent a premature optimization.

Elle’s source language is more suitable as an intermediate representation for use by other compilers than for use directly by programmers. As an example of a realistic compiler for a more ergonomic language that leverages the Elle system, I have implemented FOURL, a reimplementation of the programming language LLL, a macro assembler targeting EVM that is prized for generating efficient bytecode.

The architecture of FOURL is as follows: FOURL programs are S-expressions, which are parsed into a simple S-expression abstract-syntax datatype. A second pass of the FOURL compiler then processes these S-expressions, expanding any macros present in the raw S-expressions. A third pass identifies any raw data to be embedded in the generated bytecode; this is used to store large constants that can be retrieved by using the CODECOPY EVM instruction. Finally, FOURL identifies the main body of the program (the code to be run when the contract is deployed; i.e., the constructor) and the code to be stored permanently on the blockchain when the contract is deployed (i.e., the contract itself). It then compiles both of these using the verified ELLE backend, and returns the result of combining together these two programs. (As with constants, the constructor code will use the CODECOPY instruction to place the contract’s bytecode in the correct location.)

4.4 Elle: Future Work

Moving forward, Elle can provide a foundation on which even higher-level languages can be built, taking advantage of Elle’s formal guarantees of correctness to eliminate sources of bugs in the later stage of the compiler pipeline.

Projects ranging from reimplementations of the Solidity compiler, to improving the implementation of the Vyper language, to supporting implementations of languages with novel abstractions such as SCILLA, could benefit from using the infrastructure provided by ELLE. Because, in order to work on Ethereum, all these languages need to ultimately compile down to EVM code, all share the same need for trustworthy EVM code generation, a need met by ELLE. A verified Solidity compiler built on ELLE could be used to conduct proofs about “Graffiti” and “MiniDAO” at the source-language level rather than needing to do all reasoning directly on the compiled bytecode.

Elle can also complement automated-reasoning approaches to EVM smart contracts. Certain verification tools specify a semantics of and do reasoning at the level of the source language, in order to take advantage of the abstractions present in the source code and avoid the need for decompilation. However, these tools implicitly assume that the source-language program is faithfully translated into EVM code, despite the fact that there typically does not exist a verified compiler from those source languages to EVM. By providing a building block to enable the construction of verified compilers for higher-level EVM-targeting languages, ELLE can play an important part in closing this gap in formality.

5. Conclusion

Because of the unique characteristics of Ethereum smart contracts - small programs with sometimes surprising semantics that cannot be patched once deployed and can be responsible for significant amounts of digital valuables - they present a compelling opportunity for deployment of formal verification technologies. While a great deal of work so far has gone into automated analyses for Ethereum programs at the bytecode level, fundamental theoretical results in computer science limit what such analyses can do fully automatically.

One way around these limitations is to use interactive proof systems, such as the platform provided by ETH-ISABELLE, to manually state and prove theorems about Ethereum programs, and have the proofs checked for correctness by a small, trusted kernel. ELLE builds on the virtual-machine semantics specified by ETH-ISABELLE to allow for sound compilation of structured programs to EVM. This enables analyses of these programs at a higher level without the need for decompilation.

Elle can form a useful part of the wider ecosystem of Ethereum verification technologies, facilitating the construction of correct compilers for higher-level languages for Ethereum of the sort proposed by the SCILLA project. It can also facilitate the verification of compilers for these languages, enabling them to prove themselves correct with respect to ELLE’s semantics.

The future for formal verification of programs that run on systems like Ethereum and other blockchain systems looks bright. Through their application, we can move toward a world in which
we can trust decentralized applications, enabling them to live up to their promise of being able to replace trusted third-parties in communications and transactions of all kinds. It seems clear that formal verification will have an important role to play in unlocking the transformative potential latent in Blockchain.

References


