Smart Contracts and Opportunities for Formal Methods

Andrew Miller[1(✉)], Zhicheng Cai[2], and Somesh Jha[2]

[1] University of Illinois at Urbana-Champaign, Champaign, USA
soc1024@illinois.edu
[2] University of Wisconsin Madison, Madison, USA

Abstract. Smart contracts are programs that run atop of a blockchain infrastructure. They have emerged as an important new programming model in cryptocurrencies like Ethereum, where they regulate flow of money and other digital assets according to user-defined rules. However, the most popular smart contract languages favor expressiveness rather than safety, and bugs in smart contracts have already lead to significant financial losses from accidents. Smart contracts are also appealing targets for hackers since they can be monetized. For these reasons, smart contracts are an appealing opportunity for systematic auditing and validation, and formal methods in particular. In this paper, we survey the existing smart-contract ecosystem and the existing tools for analyzing smart contracts. We then pose research challenges for formal-methods and program analysis applied to smart contracts.

1 Introduction

Smart contracts are programs that run atop of a financial infrastructure, and command the flow of money according to user-defined rules. Today, smart contracts have already been brought to reality on top of publicly deployed cryptocurrencies, most notably Ethereum, which is currently the #2 cryptocurrency, second to Bitcoin [63], and already hosts tens of millions of smart contract programs deployed by users.

On Ethereum today, there have been auction mechanisms to raise capital investment, totalling $1B in the month of 2017 alone. Ethereum contracts have also been used to implement decentralized order books and public auctions. A smart-contract based token exchange, IDEX, is the most widely active used smart contract today, processing $7.5M USD of exchange volume each day[1].

Smart contracts are appealing for many reasons, and seem to show great potential. They essentially provide users with "programmable money" that can be used to automatically enforce agreements between potentially distrusting parties. They can operate on data provided by authenticated sources (such as stock prices, account balances, press releases, etc.). They may even be used to implement decentralized, virtual corporations defined only by the smart contracts

[1] https://cryptocoincharts.info/markets/show/etherdelta.

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11247, pp. 280–299, 2018.
https://doi.org/10.1007/978-3-030-03427-6_22

programatically governing their behavior. There has been significant demand from within the financial industry, including investments in blockchain technology projects [43]. Though integration with smart contracts with existing financial infrastructure may take years [58].

In cryptocurrencies like Ethereum, smart contracts give end users the full power and expressivity of a Turing-complete language. With such great power can come equally devastating bugs with direct financial consequences. Furthermore, since smart contracts are tied directly to anonymous payment instruments, they are an attractive target for hackers. Recent high-profile disasters involving the TheDAO [59] and the Parity Wallet [36] have highlighted these risks. Attackers have exploited programming bugs to steal approximately $60M USD.

Smart contracts seem to be a compelling motivation for systematic approaches to system validation, and to formal methods and program analysis in particular. Writing a correct smart contract is no easier than writing bug-free code in any other programming language. In fact, smart contract bugs are often harder to fix. For one reason, most blockchain systems are designed for immutability, meaning they do not provide any built-in means to change smart contract code once it is running. It is perhaps no surprise that the recent disasters have led to public interest from the cryptocurrency community in improve verification tools.

In this paper, we provide a background on smart contracts, and in particular the experience over the past few years as Ethereum has brought smart contracts to a wider audience. We argue that not only do smart contracts provide an impetus to improve tooling around formal methods, they also highlight new areas and opportunities for fundamental research in formal methods.

2 Background

Blockchains and Cryptocurrencies. In a nutshell, blockchains are distributed ledgers maintaining a globally consistent log of user-submitted transactions. Blockchains come in many forms, permissioned and permissionless. These are often implemented as open peer-to-peer networks, based on proof-of-work mining. Starting with Bitcoin [63], public blockchains are often used to create a virtual currency. The main idea of a virtual currency is that user accounts are associated with public keys. Where users transfer currency between one another using public key digital signatures.

Smart Contracts in Ethereum. Besides just storing account values, many blockchains, most notably Ethereum, also feature a full-fledged "smart contract" programming languages. In Ethereum, contracts are implemented as a new type of account: ordinary user accounts are associated with public keys, while contract accounts are associated with a fragment of executable code. Users can create a new contract account by publishing a special transaction containing the bytecode for the new contract along with an initial endowment of Ether.

Just like user accounts, smart contract accounts can store and wield a balance of Ether currency. Unlike user accounts, whoever owns the private key determines

how the money is spent, the Ether belonging to a smart contract account can only be spent by executing the instructions of the smart contract code. Hence smart contracts can be thought of as programmable money.

The Ethereum blockchain currently stores more than one million contracts. Developers write in a high level language, the most popular of which is Solidity. As a programming model, Solidity smart contracts mostly resemble object-oriented programming. Contracts are defined as a class, including methods and member variables. Users can create an instance object of the class through a contract creation transaction. Once created, contracts are assigned a unique identifier, called its address, which is a 32-byte string such as `0x06012c8cf97BEaD5deAe237070F9587f8E7A266d`. Roughly, the address is a hash of the contract's code, and the state of the blockchain prior to its creation. An example of a smart contract written in Solidity is shown in Fig. 1.

3 Smart Contract Disasters in Ethereum

Most Ethereum contracts are used for some financial purpose, such as collecting investment funding [29]. Perhaps their most notable use is for Initial Coin Offerings (ICOs), which have been a successful mechanism for generating investment revenue (more than $1B USD invested in 2017), though these have also drawn the attention of regulators since many have been fraudulent.

ICOs so far have typically made use of a "token" contract, which has emerged as a standard convention. The simple Solidity program in Fig. 1 captures the basic functionality. Tokens have a finite supply, but can be owned by a user, and can be transferred to another user at the owner's discretion. Many ICOs build additional smart contract functionality in addition to the token interface, such as an auction mechanism or a crowd-voting mechanism. Implementation flaws of such smart contracts have already caused several significant disasters in practice. We now tell the stories behind a few of them, and later discuss how they motivated new research questions for formal methods.

3.1 The DAO

The DAO was originally developed as a fundraising platform by a company called slock.it. The idea behind slock.it is the vision of "smart property" as defined by Nick Szabo in his influential 1997 essay on smart contracts [74]. The initial product was a "smart lock," a physical lock that could be applied to bicycles or rental apartments. The lock could be remotely operated by a nearby base station, which also connected to the internet and the Ethereum peer-to-peer network. The opening of the lock could be triggered by a message sent to an Ethereum smart contract. The price for renting a particular bicycle could thus be set by dynamic market.

As ambitious engineers with prior Ethereum experience, slock.it also set out to solve the meta-problem of fundraising. Rather than seeking traditional venture capital funding, and rather than using an existing centralized crowdfunding

```
contract Token {
    mapping (address => uint) balance;

    function transfer(address to,
                      uint      amount) {
        require (balance[msg.sender] >= amount);
        balance[msg.sender] -= amount;
        balance[to] += amount;
    }
    ...
}
```

Fig. 1. Solidity smart contract example.—This excerpt is from an ERC20-compliant "token" contract, which defines a virtual currency that can be transferred between users and traded on exchanges like Etherdelta.

platform like Kickstarter, slock.it developed a multi-purpose Ethereum-based crowdfunding platform, called the Decentralized Anonymous Organization (or The DAO).

The phrase Decentralized Autonomous Corporation (DAC), was first coined by Larimer [51] and expounded on by Buterin [33] as a central motivation for building a flexible programming language on top of Ethereum.

> "Think of a crypto-currency as shares in a Decentralized Autonomous Corporation (DAC) where the source code defines the bylaws" – Daniel Larimer

Token holders would purchase DAO tokens by investing Ether. Token holders would be able to vote on the activities funded by the DAO. Would-be entrepreneurs would submit funding proposals for consideration by The DAO, who would then vote on whether or not to fund the proposal. If accepted, the entrepreneur would pay profits to the DAO, which would be disbursed back to the token holders in proportion to their investment.

The source code for the DAO defined a fairly complex deliberation and decision making structure. For example, in order to mitigate potential hostile takeovers by an investor, the DAO provided a way for a dissenting token holder to exit, or "split" from the DAO, withdrawing their remaining share of the assets. All of this is to say that the DAO's design was ambitious, and experts anticipated that it would have failed for subtle game-theoretic reasons [53]. Instead, the experiment was cut short by a more mundane flaw. The technical cause is interesting, and illustrates some of the challenges in designing smart contracts.

The DAO's Flaw: Re-entrancy Hazards. The technical flaw behind the DAO's failure is essentially due to a unintuitive behavior of method invocation involving untrusted code. The events surrounding the DAO flaw and its exploit are explained in detail by a blogpost by Phil Daian [35]. We illustrate the idea here with a simple example in Fig. 2. When the `ReentrantToken.withdraw` method

is invoked, it uses `msg.sender.call` to invoke the fallback `function()` method of the caller, transferring the requested Ether. If the caller is `AttackContract`, then this recursively invokes `withdraw` again, repeating until gas runs out or the call stack limit 1024 is reached. The Ether is transferred with each call, but the `balances` field is only updated *after* the recursive call completes, leading to multiple withdrawals.

As it turns out, the attacker was only able to withdraw a portion of the funds from the contract before a team of developers with Ethereum Foundation raced the attacker to withdraw the rest and return them to the original owners [41]. Furthermore, by a stroke of luck (one that defies explanation, involving a subtle design issue with the "split" functionality mentioned above [35]), the attacker's withdrawn funds entered a month-long purgatory, which enabled the Ethereum community to develop a "hardfork" remedy that reverted the theft at last minute [75]. What rules of engagement drive interventions in smart contracts?

```
contract ReentrantToken {
  mapping (address => int64) balances;
  ...
  function withdraw(uint amount) {
    if (balances[msg.sender] >= amount) {
      // The following line transfers control to untrusted code
      msg.sender.call.value(amount)();
      balances[msg.sender] -= amount;
    }
  }
}

contract AttackContract {
  ReentrantToken token = 0x{victimsaddress};

  function startAttack() {
    token.withdraw(100);
  }

  function() payable {
    // Continue the attack, triggering a
    // recursive call
    A.withdraw(100);
  }
}
```

Fig. 2. A toy example of a vulnerable reentrant smart contract (similar to the DAO).

3.2 Parity Wallet Failures

The Parity wallet is an Ethereum smart contract that is provided along with the Parity node, the second most popular Ethereum node software. Although the Parity software supports ordinary user accounts, it also gives the user the option to create a "wallet" account, which creates an instance of the Ethereum smart contract for the benefit of customizability and extra features:

> "The most common use-case are multi-signature wallets, that allow for transaction logging, withdrawal limits, and rule-sets for signatures required."

In Ethereum, each transaction must pay a transaction fee that depends on the amount of resources consumed, including each byte of data, and each opcode executed. Creating a contract means paying for each byte of bytecode. To reduce the costs of creating instances of the same transaction, the Parity wallet makes use of a form of smart contract inheritance.

The idea is that the main portion of the Parity wallet code is uploaded to a single instance, at address `0xbec591de75b8699a3ba52f073428822d0bfc0d7e`, which can be linked to by the individual per-user instances of the wallet. The wallet library defines most of the methods relevant to the wallet, such as "withdraw", while the per-user wallets dispatch to the code contained in the library. This kind of inheritance is achieved in Ethereum through the use of the `delegatecall` opcode, which was added fairly recently to the Ethereum Virtual Machine (EVM). An example illustrating inheritance can be found in Fig. 3.

The `delegatecall` takes in another contract's address as a parameter. The semantics of this opcode instruction runs the code from the target contract, in the context of the calling contract. This essentially achieves the prototype inheritance pattern; *the library is not a superclass, but rather an actual object instance.* As an object instance, methods can be invoked directly on the contract. This fact was overlooked, leading to a disaster totaling in tens of millions of dollars. In particular, while the subclass wallets featured an access control policy whereby only the contract creator can command the contract, the library object itself was uninitialized and had an open access policy. As a result, a random user, who later claimed to be a newcomer to Ethereum, was able to claim ownership of the library and destroy it. At the current time, all instances of this version of the Parity wallet, numbering at least 150 and controlling around $150M USD, are inoperable.

Other Common Bugs in Ethereum Smart Contracts. While the re-entrancy and `delegatecall` bugs are well known and quite severe, there are other classes of bugs that are also relevant to smart contracts (e.g. reliance on poor quality sources of randomness). Atzei et al. [28] provide a taxonomy of common classes of bugs in Ethereum smart contracts.

```
contract Wallet {
  address _walletLibrary = ${hardcoded address};
  ...
  function withdraw(uint amount) {
    _walletLibrary.delegatecall(''withdraw(uint)'', amount);
  }
  ...
}
```

Fig. 3. Prototype inheritance as found in the Parity wallet

4 Research Trends

4.1 Safer Smart Contract Languages

A wide variety of approaches to language design have now been proposed and in some cases tried in practice, as surveyed by Seijas et al. [69]. In Table 1 we provide a summary of such proposals. The simplest path to a better scripting language, taken by Vyper, is to modify the existing Solidity language through syntactic restriction. That is, Vyper is a safer subset of Solidity. Many other smart contract languages use a different programming model, such as functional programming languages, formal logics and automata.

Typed functional programming languages are promising for smart contracts because they are known to be amenable to formal analysis. For example, the Tezos alternative to EVM is called Michelson, and is designed as a typed abstract machine for (mostly) pure functional programs [12], while Liquidity is an Ocaml-inspired functional alternative to Solidity for high level contract programming. At the opposing end of the restrictive-expressive spectrum, the Bitcoin developer community has preferred smart contracts compatible with the existing UTXO model underlying Bitcoin Script [1], and that guarantee a property "reorg safety". Simplicity is a typed functional language for this regime. Phil Wadler has written a comparison of both Michelson and Simplicity, ultimately arguing for Plutus, another alternative typed functional language [76].

Other various programming models have also been proposed, which can be alternatives to EVM. Rholang is build on a core calculus called ρ-calculus (inspired by π-calculus) which provides asynchronous message-passing. Similarly, Scilla is based on communicating automata, while FSolidM is a formally finite-state machine based model for Ethereum. Owlchain, combines timed-automata-language (TAL) and web ontology language (OWL). The formal definitions underlying these models are also expected to simplify formal analysis, though the benefits of these have yet to be seen.

Many blockchain or cryptocurrency projects often differentiate themselves in their smart contract programming language, however several other factors seem to determine how their project evolves. They may differ also by their underlying consensus algorithms, or by the target applications that guide their choices of engineering tradeoffs.

Several proposals have been made for "sharded" blockchains, which achieve better scalability but pose an additional challenges for smart contracts. Instead of a single linearized chain replicated by every node in the blockchain network, the ledger is instead logically divided into separate namespaces, each of which is replicated by only a portion of the nodes. This model underlies Omniledger [48], RScoin [37], Aspen [38], and Scilla [71].

4.2 Program Analysis

As far as we know, there are 11 tools or frameworks attempting to detect various types of vulnerabilities, or give an assistance for programming. The table below (see Table 2) shows their detailed capabilities respectively. Here, we give a summary for them.

From the scale of checking abilities, SmartCheck provides the most types of checking and recommendations, and more serves as a dynamic suggestion-generated system for Solidity source code. Currently, most of its Solidity-related checking is already provided by Solidity IDE [18].

From the view of vulnerabilities, these tools almost cover all possible vulnerabilities mentioned in [4,24,28], including safer programming design pattern suggestion. The reentry vulnerability became the most popular one to tackle, and the reason is obvious because this vulnerability resulted in the infamous DAO attack [35].

However, apart from traditional bugs like integer overflow or usage of uninitialized variables, those vulnerabilities shown in contract programming do not have unified definition respectively, or their detection results highly depend on tools' own implementations.

From the view of programming analysis techniques, most of the tools choose static analysis and a majority of them support EVM bytecodes analysis. Securify and Mythril declare supporting on-chain contracts analysis. Porosity does some reverse engineering and provides a prototype for decompilation. Solgraph and Mythril can generate control flow graphs, and Manticore and Maian will generate transactions with inputs for later validation on each vulnerable path. Oyente also has a validation process after symbolic execution analysis. Oyente, Maian and ZEUS provide false-positive analysis, and all of them use manually-tagged data sets, selected from contracts with *verified*[2] Solidity source code.

In order to better describe contracts logic model and specifications, CertiK has its own verification labeling languages, and ZEUS uses an intermediate-level abstract language. Oyente develops their own EVM semantics, EtherLite, and Maian modifies it in their implementation.

Most of those tools are developed in Python, or other languages like Java (SmartCheck), OCaml (Dr.Y), JavaScript (Solgraph) and C++ (porosity). Manticore provides a Python API for analysis of EVM bytecode.

Besides, there are some other efforts in language (Sect. 4.2) or semantics design, which could help with formal verification of smart contracts (see Table 3).

[2] Here "verified" means the Solidity source code corresponds to the EVM bytecodes.

Table 1. Languages

Languages	Descriptions (motivation, expressivity, type system, analysis-friendly features, etc.)	Reference
Scilla (intermediate-level) (Zilliqa)	- motivated by achieving expressivity and tractability - based on communicating automata [50] - provide limited translation from higher-level languages (i.e., Solidity) - provide translation into Coq for verification, along with contract protocols, semantics, safety/liveness properties and proof machinery	Paper [71], code [70]
FSolidM (Ethereum, framework, higher-level)	- aims to develop more secure smart contracts - a formal, finite-state machine based model - provide several plugins (i.e., design patterns) to enhance security and functionality, targeting at vulnerabilities as reentry bugs and transaction ordering, or design patterns as time constraint and authorization - primarily for Ethereum, but it may applied on other platforms - provide translation into Solidity	Paper [55], code [54]
Rholang (higher-level, RChain)	- primitively for RChain, but could be used in other settings - focus on message-passing and formally modeled by the ρ-calculus, a **r**eflective, **h**igher-**o**rder extension of the π-calculus, which is good for concurrent settings [57]	Code [19]
Vyper (Ethereum, higher-level)	- mainly target at security and auditability - provide the following features: bounds and overflow checking, support for signed integers and decimal fixed point numbers, decidability, strong typing, small and understandable compiler code, and limited support for pure functions - does not support the following features: modifiers, class inheritance, inline assembly, operator overloading, recursive calling, infinite-length loops and binary fixed point - statically typed language	Code [5], doc [27]
Type-coin (Bitcoin)	- a logical commitment mechanism - the logic is linear and not rich to handle complex situations	Paper [34]
Simplicity (Bitcoin)	- type-safety, no unbounded loops, no named variables - no function types and thus no higher-order functions	Paper [66], blog [31]
Michelson (Tezos)(lower-level)(functional)	- a strongly-typed, stack-based language - It doesn't include many features like polymorphism, closures, or named functions - more as a way to implement pieces of business logic than as a generic "world computer" - Programs written in Michelson can be reasonably analyzed by SMT solvers and formalized in Coq without the need for more complicated techniques like separation logic - To provide a straightforward platform for business logic, to provide a readable bytecode, and to be introspectable	Paper [12], web [26]

(*continued*)

Table 1. (*continued*)

Languages	Descriptions (motivation, expressivity, type system, analysis-friendly features, etc.)	Reference
	- Entirely original implementation in OCaml - Isolated economical rules, self-amendable via voting - purely PoS - Blockchain state in a git-like persistent store - Highly functional, defensive coding style for the critical parts - designed with formal certification in mind	
Liquidity (Tezos) (higher-level)(functional)	- It uses the syntax of OCaml, and strictly complies to Michelson security restrictions	code [7], web [65]
Plutus (higher-level) and Plutus Core (lower-level) (IOHK)	- compiled to Plutus Core (lower level), Lisp-like syntax - a pure functional strictly typed programming language, with user-defined data types and polymorphism - several issues: unbounded integers supporting, non-supporting abstract data types and data constructors	Code [17], paper [56]
Owlchain (BOSCoin)	- a decidable programming framework, which consists of the Web Ontology Language and the Timed Automata Language. - OWL is defined as W3C standard, a declarative language that provides decidability - separate declaration from processing - TAL, Timed Automata Language, is a new language that is used to create operators. It is a finite state programming environment with two constraints: time limit and pure functions. Timed automata modeling can detect undefined areas (reachability problem) in the code that developers missed. Pure function can eliminate side effects that can occur during development	Article [32]

Most of proposals relate to functional languages, perhaps due to the advantages to perform static analysis.

For semantics, usually a tool has its own semantics (like Oyente has EtherLite). A representative work is [39], because they provide the first complete small-step semantics of EVM bytecodes and formalize it in F*. Also, this paper points out that, though smart contracts are written in a Turing complete language, their computations are bounded by *gasLimit*, thus it becoming a "quasi" Turing-complete language.

4.3 Off-chain Protocols and Cryptography

Off-chain payment channels have emerged as an important topic in smart contracts, in both industry and academia. Once a payment channel is established between two parties, they can send rapid micropayments to each other without

Table 2. Tools and frameworks for analyzing smart contracts

Tool/Framework	Capabilities	Reference
CertiK (Demo)	- target at fully trustworthy blockchain ecosystems in the future - specifications for each function can be expressed using CertiK labels, indicating pre-condition, post-condition and invariants respectively, as comments in Solidity programs	White paper [2]
Dr.Y's Ethereum Contract Analyzer	- a symbolic execution tool, reflecting contract behavior to some point	Code [44]
Maian	- check locked money - detect unchecked suicide or Ether sending - generate inputs to validate through private blockchain	Paper [64], code [8]
Manticore	- detect potential overflow and underflow conditions on "ADD", "MUL" and "SUB" instructions - detect potential uses of uninitialized memory or storage - calculate code coverage - generate inputs which could trigger unique code paths (Solidity source code needed) - Other: offer a Python API for analysis of EVM bytecodes	Article [11], code [9], doc [10]
Mythril	- detect reentry bugs and external calls to untrusted contracts - detect unchecked suicide or Ether sending - check mishandled exceptions (i.e., detect unchecked CALL return value) - check integer underflows - detect usage of "tx.origin" [21] - check dependence on predictable variables (e.g., coinbase, gaslimit, timestamp, number, etc.) - Other: generate control flow graph, blockchain exploration and some utilities - support on-chain contracts analysis	Article [60,61], doc [13], code [62],
Oyente	- detect reentry bugs - check mishandled exceptions (i.e., detect unchecked CALL return value) - check transaction-order-dependency (a.k.a. money concurrency, or front running) - check timestamp dependency - check possible assertion failure (Solidity source code required) - calculate code coverage	Paper [52], web access [16], code [15]
Porosity	- find potential reentrancy vulnerability - support decompilation and disassembly	Code [3], white paper [73], article [72]
SmartCheck (target at Solidity)	- detect reentry bugs - check locked money - detect possibly infinite or impractical loops - detect unchecked low-level call	Code [23], web access [22]

(*continued*)

Table 2. (*continued*)

Tool/Framework	Capabilities	Reference
	- check integer overflow and underflow, and recommend to use the SafeMath library [14] - check timestamp dependence - Other: more better programming design pattern recommendation - Other: recommendations for standard ERC-20 function usages, and check style guide violation - Other: some checking for recommended Solidity programming style	
Securify	- check reentry bugs - check mishandled exception - check transaction-order-dependency - check insecure coding patterns, e.g., unchecked transaction data length, use of ORIGIN instruction and missing input validation - check unexpected Ether flows, such as locked Ether [68] - check use of untrusted inputs in security operations, i.e., checking whether the inputs to the SHA3 depend on block information (timestamp, number, coinbase)	Web access [20]
Solgraph	- highlight potential unchecked money receiver - generate function control flow of a Solidity contract	Code [67]
ZEUS	- support self-defined policy verification, e.g., reentry bugs, unchecked "send", possibly vulnerable failed "send", integer overflow, transaction state dependency (i.e., usage of "tx.origin"), block state dependency (including all "block" parameters) and transaction order dependency - specification limited to quantifier-free logic with integer linear arithmetic	Paper [47]

any transaction fees. The idea is that the parties send messages to each other in the typical case, off-chain, and only use the smart contract to close. Payment channels are also the building blocks for payment channel networks, which are a highly anticipated scalability proposal for cryptocurrencies.

Payment channels and state channels are multi-faceted protocols, relying not just on the smart contract, but also on a cryptographic scheme involving digital signatures and hash functions, as well as the reconciliation of state stored at different parties. Reasoning about these applications relies on more than just analyzing the smart contract directly.

Off-chain Payment Channels. A smart contract payment channel protocol should provide the following (informal) properties:

Table 3. Language design and model translation

Languages or semantics	Descriptions (motivation, expressivity, type system, analysis-friendly features, etc.)	Reference
SMAC (modular reasoning)	- introduce ECF (Effectively callback free) property for modular object-level analysis - develop online detection algorithm which can apply to Ethereum full node, and monitor non-ECF executions, including the infamous DAO bug	Paper [40]
eth-isabelle (semantics)	- define the complete instruction set of EVM in Lem, a language that can be compiled into Coq, Isabelle/HOL and HOL4 - can prove invariants and safety properties	Paper [46], code [45]
TU Wien F* (2018) (Ethereum)	- present the complete small-step semantics of EVM bytecode in the F* proof assistant - define a number of central security properties, such as call integrity, atomicity, and independence from miner controlled parameters	Paper [39], code [25]
F* (2016) (Ethereum)	- motivated by formal verification - partial semantics for converting Solidity to F*, EVM to F* - show the correspondence between Solidity and EVM to some point	Paper [30]
KEVM (semantics, high-level, Ethereum)	- a complete K Semantics of the Ethereum Virtual Machine (EVM)	Code [6], paper [42]

– (Timing properties.) Payments are processed very quickly (no blockchain transactions), and closure is guaranteed within a predictable time (small number of blockchain transactions).
– (Integrity properties.) If Bob thinks he has received $X, then he is guaranteed to get at least $X when the channel closes. And Alice should get back everything except what she has paid.

A payment channel protocol is given in Algorithm 1, comprising a local program for the sender (Alice), a local program for the recipient, and a smart contract program. Alice initially deposits $\$X$ by making an on-chain transaction, into a smart contract running the given pseudocode. Alice can then make numerous micropayments to Bob, by sending signed messages that indicate Bob's latest credit. Each payment can be very fast and efficient, since it requires only point-to-point interactions between Alice and Bob; it does not require any on-chain transaction. At any time, either party can request to "close" the channel, in which case Alice submits her most recent signed message. The smart contract is only activated when the channel closes.

The generalization of a payment channel, a "state channel", allows two or more parties to maintain an off-chain replicated state machine that can be synchronized on demand (or in case of a dispute) with the blockchain.

Algorithm 1. A Smart Contract protocol for Off-chain Payments

Alice and Bob are represented by hardcoded public keys

Local code for Alice (the sender):

[Initially]:
    credit := $\$X_0$ // initial deposit
[on input ("pay", $\$X$)]:
    assert $\$X \leq$ credit
    credit := credit $- \$X$
    $\sigma \leftarrow$ Sign($\$X_0 -$ credit) as Alice
    send $(\sigma, \$X_0 -$ credit) to Bob
[on input ("close")]: send ("close") to the Contract

Local code for Bob (the recipient):

[Initially]:
    credit := 0
[on receiving (σ, credit$'$) from Alice)]:
    assert σ is a valid signature on credit$'$ from Alice
    assert credit$' \leq \$X_0$
    if credit$'$ > credit
        credit := credit$'$
        $\sigma := \sigma'$
[on input ("close")]: send "close" to the Contract
[on contract event ("close"):
    send ("evidence", σ, credit) to the Contract

Smart Contract Code:

[Initially]:
    lastKnownCredit := 0
[on contract input ("close") from Alice or Bob (only once)]:
    within delay $O(\Delta)$:
        send ($\$X_0 -$ lastKnownCredit) to Alice
        send (lastKnownCredit) to Bob
[on contract input ("evidence", σ, credit)] from Alice or Bob:
    assert σ is a valid signature on credit from Alice
    assert credit$' \leq \$X_0$
    if credit > lastKnownCredit
        lastKnownCredit := credit

Functionality Model for the Payment Channel. The payment channel protocol above was given an informal specification. To give a precise security definition, an appealing approach is to use the simulation based security framework used by cryptographers. The main idea behind the simulation-based security framework is that instead of expressing properties as indistinguishability games, we provide an explicit program, called an ideal functionality, that exhibits all the properties

Algorithm 2. An Ideal Functionality for Off-chain Payments

```
[Initially]
    Alice and Bob are represented by hardcoded public keys
    credit@A := $X_0 // initial deposit
    credit@B := $0
[on input ("pay", $X) from Alice]:
    assert $X ≤ credit@A
    credit@A := credit@A − $X
    within O(1) delay:
        credit@B := credit@B + $X
[on input ("close") from Alice or Bob]:
    within O(Δ) delay:
        send at least credit@A to Alice
        send at least credit@B to Bob and halt
```

at once. This has the advantage that all the salient security properties of a protocol can be defined in effectively one place.

An ideal functionality for the payment channel protocol is given in Algorithm 2. Note that the functionality is structurally simpler than the protocol (it executes in one location rather than three), and does not contain any cryptography.

It is also easy to see that the functionality exhibits the desired properties. The phrases "$O(1)$ delay" and "$O(\Delta)$ delay" denote the desired time bounds, which would be automatically inferred or written as annotations by the programmer. Here Δ refers to a worst-case bound on the time it takes to submit and confirm a blockchain transaction. Hence the fact that the "pay" command completes in $O(1)$ time reflects the fact that the protocol uses only off-chain messages. The credit@A and credit@B expressions denote the respective local views of Alice and Bob; the functionality explicitly sends a final payment to each consistent with their local views. Note that it is possible for a payment to interleave with channel closure; in this case, Bob may receive more than he expected.

Several Instances of Cryptographic Protocols Where the Smart Contract Acts as a Verifier. OpenVote uses Ethereum as the tallier for a cryptographic, sealed ballot election. Users submit encrypted ballots to the Ethereum blockchain to be tallied, along with a zero-knowledge proof (ZKP) that their vote is correctly formatted (i.e., contains an encryption of just one vote for just one candidate). The use of the Ethereum blockchain in place of an election authority avoids the need to trust any privileged party to carry out the election.

Ethereum has recently included support for the `ALT_BN128` elliptic curve, which is used in particular for a generic proof system called zkSNARKs. An example contract is provided where in order to claim a prize, a prover must demonstrate knowledge of a solution to a Sudoku puzzle, but without revealing the solution itself. Further cryptographic applications include privacy-preserving auctions and insurance contracts [49].

In general, it seems likely that many future applications will involve the use of increasingly sophisticated cryptographic primitives within smart contract programs.

5 Challenges and Opportunities for Formal Methods

Smart contracts present three non-traditional challenges to developers, making smart contracts more difficult to implement than code in other contexts.

1. **Composition with untrusted and adversarial code.** Smart contracts are implemented on a distributed system, and are freely accessible by the public. Composition within smart contracts tends to involve untrusted code. The DAO involved transferring control flow to an attacker's smart contract, which carried out the attack. For this reason we would likely want the capability to combine runtime certification with static analysis. We would use static analysis, but may need to write defensive code that provides runtime enforcement of guarantees against untrusted code. Compositional verification technologies are useful here, but the challenge is to write a model E of the environment for the smart contract. Note that E will have to account for other code fragments that the smart contract might interact with and also a model of the underlying infrastructure (e.g. blockchain) that the contract is executing on.
2. **Distributed and asynchronous setting.** Smart contracts are often just one component of a more complicated distributed protocol. Smart contracts often play the role of a "verifier" in a cryptographic protocols. The Ethereum platform enables application developers to make use of built-in primitives, such as hash functions, digital signatures, and now more recently, pairing-friendly elliptic curves, the ingredients for zkSNARK proofs.
   In general, a smart contract protocol may involve local code and custom cryptography, which are just as important to the correct functionality and design of the application of the smart contract itself. However, one challenge here is that the guarantees that the smart contract requires from the cryptographic function will heavily depend on the functionality of the smart contract. Future smart contract programming languages and analysis techniques will need to take this into account.
3. **Economic incentives.** Unlike in traditional software, in the smart-contract setting, many of the desired properties one wishes to establish are *economic*. For example, participants might want to verify that their expected payoff for participating in a contract is non-negative, since otherwise they have little reason to participate. Analyzing a smart contract often involves reasoning about game-theoretic properties like incentive compatibility. Effective tools may need to take this reasoning into account. Formalisms, such as mean-payoff games, might be useful in this context, but verifying properties of these expressive formalisms remains a challenge.

6 Conclusion

This paper surveys the ecosystem of smart contracts, such as various platforms, high-profile bugs, and existing analysis tools. Of course, one can use existing analysis techniques and tools to analyze smart contracts, and one should do so. However, we believe that uniqueness of the smart contracts also brings some unique challenges for the formal-methods community, which will require new techniques and novel research ideas.

References

1. Bitcoin Script Wiki. https://en.bitcoin.it/wiki/Script. Accessed 21 Apr 2018
2. CertiK: building fully trustworthy smart contracts and blockchain ecosystems. https://certik.org/docs/white_paper.pdf. Accessed 28 Mar 2018
3. Decompiler and security analysis tool for blockchain-based ethereum smart-contracts. https://github.com/comaeio/porosity. Accessed 25 Mar 2018
4. Ethereum smart contract security best practices. https://consensys.github.io/smart-contract-best-practices/. Accessed 29 Mar 2018
5. ethereum/vyper - new experimental programming language. https://github.com/ethereum/vyper. Accessed 26 Mar 2018
6. K semantics of the ethereum virtual machine (EVM). https://github.com/kframework/evm-semantics. Accessed 26 Mar 2018
7. Liquidity: a smart contract language for Tezos. https://github.com/OCamlPro/liquidity. Accessed 19 Apr 2018
8. MAIAN: automatic tool for finding trace vulnerabilities in ethereum smart contracts. https://github.com/MAIAN-tool/MAIAN. Accessed 24 Mar 2018
9. Manticore 0.1.7 release - symbolic execution tool. https://github.com/trailofbits/manticore. Accessed 24 Mar 2018
10. Manticore documentation, release 0.1.0. https://media.readthedocs.org/pdf/manticore/latest/manticore.pdf. Accessed 25 Mar 2018
11. Manticore: Symbolic execution for humans. https://blog.trailofbits.com/2017/04/27/manticore-symbolic-execution-for-humans/. Accessed 24 Mar 2018
12. Michelson: the language of smart contracts in Tezos. https://www.tezos.com/static/papers/language.pdf. Accessed 19 Apr 2018
13. Mythril 0.14.9 - security analysis tool for ethereum smart contracts. https://pypi.python.org/pypi/mythril. Accessed 24 Mar 2018
14. Openzepplin/zeppelin-solidity/contracts/math/safemath.sol. https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol. Accessed 24 Mar 2018
15. Oyente - an analysis tool for smart contracts, version 0.2.7. https://github.com/melonproject/oyente. Accessed 9 Oct 2017
16. Oyente web access. https://oyente.melon.fund. Accessed 24 Mar 2018
17. Plutus language prototype. https://github.com/input-output-hk/plutus-prototype. Accessed 20 Apr 2018
18. Remix - solidity IDE, v0.4.21. http://remix.ethereum.org. Accessed 28 Mar 2018
19. Rholang. https://github.com/rchain/rchain/tree/master/rholang. Accessed 26 Mar 2018
20. Securify - formal verification of ethereum smart contracts. https://securify.ch/. Accessed 24 Mar 2018

21. Security considerations - pitfalls - tx.origin. https://solidity.readthedocs.io/en/develop/security-considerations.html#tx-origin. Accessed 24 Mar 2018
22. Smartcheck. https://tool.smartdec.net/. Accessed 24 Mar 2018
23. Smartcheck - a static analysis tool that detects vulnerabilities and bugs in solidity programs. https://github.com/smartdec/smartcheck. Accessed 24 Mar 2018
24. Solidity v0.4.21. https://solidity.readthedocs.io/en/v0.4.21/. Accessed 28 Mar 2018
25. The source code of ethereum virtual machine bytecode f* formalization. https://secpriv.tuwien.ac.at/tools/ethsemantics. Accessed 26 Mar 2018
26. Try Michelson. https://try-michelson.com/. Accessed 19 Apr 2018
27. Vyper. https://vyper.readthedocs.io/en/latest/index.html. Accessed 27 Mar 2018
28. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
29. Bartoletti, M., Pompianu, L.: An empirical analysis of smart contracts: platforms, applications, and design patterns. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 494–509. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_31
30. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS 2016, pp. 91–96. ACM, New York (2016). https://doi.org/10.1145/2993600.2993611
31. Blockstream: Simplicity itself for blockchains. https://blockstream.com/2017/10/30/simplicity.html. Accessed 20 Apr 2018
32. BOScoin: Smart contracts and trust contracts: part 3. https://medium.com/@boscoin/smart-contracts-trust-contracts-part-3-6cf76bf5882e. Accessed 21 Apr 2018
33. Buterin, V.: Bootstrapping a decentralized autonomous corporation: part I. Bitcoin Mag. (2013)
34. Crary, K., Sullivan, M.J.: Peer-to-peer affine commitment using bitcoin. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 479–488. ACM, New York (2015). https://doi.org/10.1145/2737924.2737997
35. Daian, P.: Analysis of the DAO exploit (2016). http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/
36. Daian, P.: An in-depth look at the parity multisig bug, July 2017. http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/
37. Danezis, G., Meiklejohn, S.: Centrally banked cryptocurrencies. In: NDSS (2016)
38. Gencer, A.E., van Renesse, R., Sirer, E.G.: Short paper: service-oriented sharding for blockchains. In: Kiayias, A. (ed.) FC 2017. LNCS, vol. 10322, pp. 393–401. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70972-7_22
39. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts. arXiv preprint arXiv:1802.08660 (2018)
40. Grossman, S.: Online detection of effectively callback free objects with applications to smart contracts. Proc. ACM Program. Lang. 2(POPL), 48 (2017)
41. Higgins, S.: Ethereum developers launch white hat counter-attack on the DAO, June 2016. https://www.coindesk.com/ethereum-developers-draining-dao/
42. Hildenbrandt, E., et al.: KEVM: a complete semantics of the ethereum virtual machine. http://hdl.handle.net/2142/97207. Accessed 27 Mar 2018

43. Hileman, G.: State of blockchain Q1 2016: blockchain funding overtakes bitcoin (2016). http://www.coindesk.com/state-of-blockchain-q1-2016/
44. Hirai, Y.: Dr. Y's ethereum contract analyzer. https://github.com/pirapira/dry-analyzer. Accessed 25 Mar 2018
45. Hirai, Y.: A lem formalization of EVM and some Isabelle/HOL proofs. https://github.com/pirapira/eth-isabelle. Accessed 25 Mar 2018
46. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
47. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: Analyzing safety of smart contracts. In: NDSS (2018)
48. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: OmniLedger: a secure, scale-out, decentralized ledger via sharding (2018)
49. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 839–858. IEEE (2016)
50. Kuske, D., Muscholl, A.: Communicating automata. http://eiche.theoinf.tu-ilmenau.de/kuske/Submitted/cfm-final.pdf. Accessed 27 Mar 2018
51. Larimer, D.: Overpaying for security: the hidden costs of bitcoin, September 2013. https://letstalkbitcoin.com/is-bitcoin-overpaying-for-false-security
52. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269. ACM (2016)
53. Mark, D., Zamfir, V., Sirer, E.G.: A call for a temporary moratorium on "the DAO", May 2016
54. Mavridou, A.: Smartcontracts - the FSolidM framework. https://github.com/anmavrid/smart-contracts. Accessed 26 Mar 2018
55. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: a finite state machine based approach. arXiv preprint arXiv:1711.09327 (2017)
56. McAdams, D.: Formal specification of the plutus (core) language. https://github.com/input-output-hk/plutus-prototype/tree/master/docs/spec. Accessed 20 Apr 2018
57. Meredith, L., Radestock, M.: A reflective higher-order calculus. Electron. Notes Theor. Comput. Sci. **141**(5), 49–67 (2005). https://doi.org/10.1016/j.entcs.2005.05.016. Proceedings of the Workshop on the Foundations of Interactive Computation (FInCo 2005). http://www.sciencedirect.com/science/article/pii/S1571066105051893
58. Morgan Stanley Research: Global insight: blockchain in banking: disruptive threat or tool? (2016)
59. Morris, D.Z.: Blockchain-based venture capital fund hacked for $60 million, June 2016. http://fortune.com/2016/06/18/blockchain-vc-fund-hacked/
60. Mueller, B.: Analyzing ethereum smart contracts for vulnerabilities. https://hackernoon.com/scanning-ethereum-smart-contracts-for-vulnerabilities-b5caefd995df. Accessed 24 Mar 2018
61. Mueller, B.: Introducing Mythril: a framework for bug hunting on the ethereum blockchain. https://hackernoon.com/introducing-mythril-a-framework-for-bug-hunting-on-the-ethereum-blockchain-9dc5588f82f6. Accessed 24 Mar 2018
62. Mueller, B.: Mythril. https://github.com/ConsenSys/mythril. Accessed 24 Mar 2018
63. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)

64. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the Greedy, Prodigal, and Suicidal Contracts at Scale. ArXiv e-prints, February 2018
65. OCamlPRO: Liquidity online. http://www.liquidity-lang.org/edit/. Accessed 19 Apr 2018
66. O'Connor, R.: Simplicity: a new language for blockchains. CoRR abs/1711.03028 (2017). http://arxiv.org/abs/1711.03028
67. Revere, R.: solgraph - visualize solidity control flow for smart contract security analysis. https://github.com/raineorshine/solgraph. Accessed 24 Mar 2018
68. Securify: Automatically detecting the bug that froze parity wallets. https://medium.com/@SecurifySwiss/automatically-detecting-the-bug-that-froze-parity-wallets-ad2bebebd3b0. Accessed 24 Mar 2018
69. Seijas, P.L., Thompson, S.J., McAdams, D.: Scripting smart contracts for distributed ledger technology. IACR Cryptology ePrint Archive 2016/1156 (2016). http://eprint.iacr.org/2016/1156
70. Sergey, I.: Scilla-Coq, state-transition systems for smart contracts. https://github.com/ilyasergey/scilla-coq. Accessed 25 Mar 2018
71. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language. arXiv preprint arXiv:1801.00687 (2018)
72. Suiche, M.: DEF CON 25: Porosity. https://blog.comae.io/porosity-18790ee42827. Accessed 24 Mar 2018
73. Suiche, M.: Porosity: a decompiler for blockchain-based smart contracts bytecode. https://www.comae.io/reports/dc25-msuiche-Porosity-Decompiling-Ethereum-Smart-Contracts-wp.pdf. Accessed 25 Mar 2018
74. Szabo, N.: The idea of smart contracts. Nick Szabos Papers and Concise Tutorials, vol. 6 (1997)
75. Vigna, P.: Cryptocurrency platform ethereum gets a controversial update. Wall Street J. (2016). http://www.wsj.com/articles/cryptocurrency-platform-ethereum-gets-a-controversial-update-1469055722
76. Wadler, P.: Simplicity and Michelson: a programming language that is too simple. https://iohk.io/blog/simplicity-and-michelson/. Accessed 21 Apr 2018