Formal Verification of Smart Contracts: Short Paper

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al.

Formal Verification of Smart Contracts

Short Paper

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

[1]Inria
{karthikeyan.bhargavan,nadim.kobeissi,natalia.kulatova,thomas.sibut-pinote}@inria.fr

[2]Microsoft Research
{antdl,fournet,gonthier,aseemr,nswamy,santiago}@microsoft.com

[3]Harvard University
agollamudi@g.harvard.edu

ABSTRACT

Ethereum is a framework for cryptocurrencies which uses blockchain technology to provide an open global computing platform, called the Ethereum Virtual Machine (EVM). EVM executes bytecode on a simple stack machine. Programmers do not usually write EVM code; instead, they can program in a JavaScript-like language, called Solidity, that compiles to bytecode. Since the main purpose of EVM is to execute *smart contracts* that manage and transfer digital assets (called *Ether*), security is of paramount importance. However, writing secure smart contracts can be extremely difficult: due to the openness of Ethereum, both programs and pseudonymous users can call into the public methods of other programs, leading to potentially dangerous compositions of trusted and untrusted code. This risk was recently illustrated by an attack on TheDAO contract that exploited subtle details of the EVM semantics to transfer roughly $50M worth of Ether into the control of an attacker. In this paper, we outline a framework to analyze and verify both the runtime safety and the functional correctness of Ethereum contracts by translation to F*, a functional programming language aimed at program verification.

1. INTRODUCTION

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

Ethereum similarly founds a virtual currency, called the Ether, on a blockchain based on proof-of-work. Ethereum's ledger is significantly more general than Bitcoin's: it stores Turing-complete programs, in the form of Ethereum Virtual Machine (EVM) bytecode, and it enables transactions as function calls into that code, with additional data in the form of arguments. Programmable contracts may also access non-volatile storage and log events, both recorded in the ledger. Figure 1 shows a high-level view of the workflow in the Ethereum network.

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

The integrity of the system relies on the honesty of a majority of miners: a miner may try to cheat by not running the program, or running it incorrectly, but honest miners will reject the block and fork the chain. Since the longest chain is the one considered valid, (notably for paying fees and block rewards) each miner is incentivized not to cheat, and to check each other's work. Still, integrity only ensures that contract code is executed correctly, as specified by the EVM semantics, not that the contract behaves as intended by its programmer, which is the focus in this paper.

Ethereum currently runs smart contracts that manage millions of dollars, making their security highly sensitive. For instance, a variant of a well-documented reentrancy attack was recently exploited in TheDAO [2], a contract that implements a decentralized autonomous venture capital fund, leading to the theft of more than $50M worth of Ether. Recovering the funds required a hard fork of the blockchain, contrary to the *code is law* premise of the system. Such attacks raise the question of whether similar bugs could be prevented by static analysis [6], before uploading contracts to Ethereum.

PLAS'16, October 24 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2993600.2993611

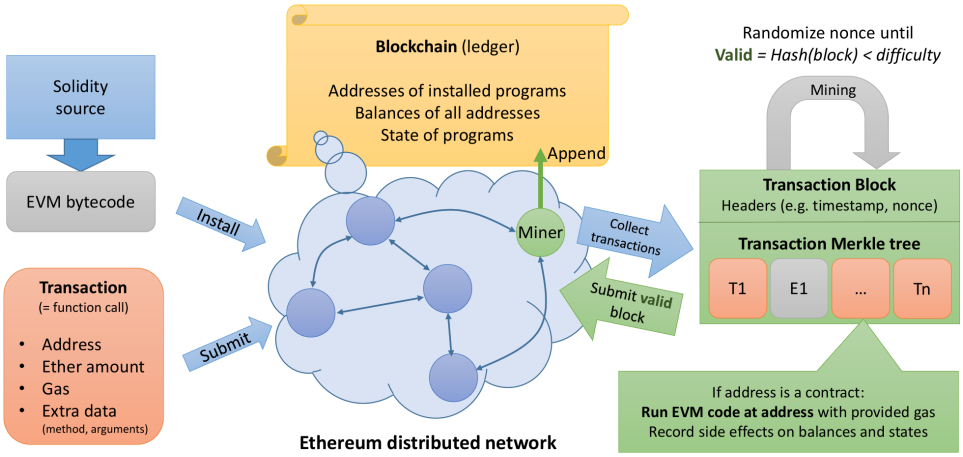

Figure 1: Overview of workflow in the Ethereum network

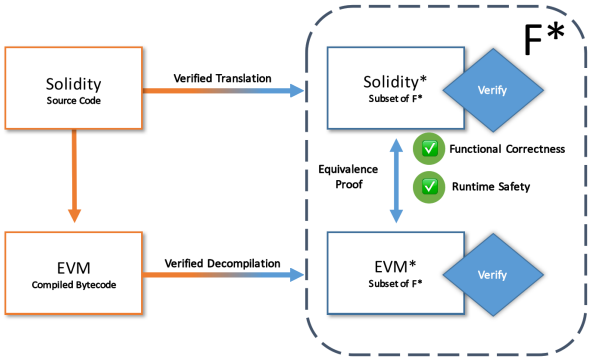

Figure 2: Outline of our verification architecture

In this paper, we outline a framework to analyze and formally verify Ethereum smart contracts using F* [9], a functional programming language aimed at program verification. Such contracts are generally written in Solidity [3], a JavaScript-like language, and compiled down to EVM bytecode. However, analyzing lower-level EVM code is also useful, inasmuch as the Solidity compiler may not be fully trusted, and malicious code may be crafted in bytecode instead of Solidity. While it is clearly favorable to obtain both Solidity source code and EVM bytecode for a given smart contract, we design our architecture with the assumption that the verifier may only have the bytecode. At the moment of this writing, only 396 out of 112,802 contracts have their source code available on http://etherscan.io.

Our smart contract verification architecture uses a two-pronged approach, summarized in Figure 2. We develop a language-based approach for verifying smart contracts, and present two prototype tools based on shallow embeddings in F* (that is, contracts are translated to F* programs that call an F* runtime library for all Ethereum operations):

Solidity* compiles Solidity contracts to F* (Section 2). It allows us to verify, at the source level, functional correctness specifications (such as contract invariants) and safety with respect to runtime errors.

EVM* decompiles EVM bytecode into more succinct F* code, hiding the details of the stack machine (Section 3). It allows us to analyze low-level properties, such as bounds on the amount of gas required to complete a call or a transaction.

The F* language comes with a rich type system that includes dependent types and monadic effects, which we apply to generate automated queries for an SMT solver that can then statically verify such properties of contracts.

This paper presents preliminary results for these two forms of verification, but in the future, we plan to use our tools to also test the correctness of the Solidity compiler on a case-by-case basis. Given a Solidity program and allegedly functionally equivalent EVM bytecode, we will translate both into F* and verify their equivalence using relational reasoning [1].

2. FROM SOLIDITY TO F*

In the spirit of previous work on type-based analysis of JavaScript programs [8], we advocate an approach where the programmer can verify high-level goals of a contract by embedding it in F*. In this section, we present both a tool to translate Solidity to F* and a sample automated analysis of embedded F* contracts based on F* typechecking.

Solidity programs consist of a number of contract declarations. Once compiled to EVM, contracts are installed using a special kind of account-creating transaction, which allocates an address to the contract. Unlike Bitcoin, where an address is the hash of the public key of an account, Ethereum addresses can refer indistinguishably to a contract or a user public key. Similarly, there is no distinction between transactions and method calls: sending Ether to a contract will implicitly call the fallback function (the unnamed method of the Solidity contract).

In fact, compiled contracts in the blockchain consist of a single entry point that decides depending on the incoming transaction (treated as a received message `msg`) which method code to invoke. The methods of a Solidity contract have access to ambient global variables that contain information about the contract (such as the current balance of the contract in `this.balance`), the transaction used to invoke the contract's method (such as the source address in `msg.sender` and the amount of Ether received in `msg.value`), or the block in which the invoking transaction is mined (such as the miner's timestamp in `block.timestamp`).

⟨*solidity*⟩ ::= (⟨*contract*⟩)*

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

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

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

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

⟨*statedef*⟩ ::= ⟨*type*⟩ @identifier

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

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

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

⟨*unop*⟩ ::= '`+`' | '`-`' | '`!`'

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

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

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

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

Figure 3: Syntax of the Solidity subset supported by Solidity*

In this exploratory work, we consider a restricted subset of Solidity, specified in Figure 3. Notably, the fragment we consider does not include loops, but does support recursion. The three main types of declarations within a contract are type declarations, property declarations and methods. Type declarations consist of C-like structs and enums, and mappings (associative arrays implemented as hash tables). Although properties and methods are reminiscent of object oriented programming, it is somewhat a confusing analogy: contracts are "instantiated" by the account creating transaction; this will allocate the properties of the contract in the global storage and call the constructor (the method with the same name as the contract). Despite the C++/Java-like access modifiers, all properties of a contract are stored in the Ethereum ledger, and as such, the internal state of all contracts is completely public. Methods are compiled in EVM into a single function that runs when a transaction is sent to the contract's address. This transaction handler matches the requested method signature with the list of non-internal methods, and calls the relevant one. When no match is found, a fallback handler is called.

2.1 Translation to F*

Our translation of Solidity to F* proceeds as follows:

1. contracts are translated to F* modules;
2. type declarations are translated to type declarations: enums become sums of nullary data constructors, structs become records, and mappings become F* maps;
3. all contract properties are packaged together within a `state` record, where each property is an F* reference;
4. contract methods are translated to F* functions;
5. `if` statements that have a continuation are rewritten, depending on whether one branch ends in `return` or `throw` (moving the continuation in the other branch) or not (we then duplicate the continuation in each branch).
6. assignments are translated as follows: we keep an environment of local, state, and ambient global variable names: local variable declarations and assignments are translated to `let` bindings; globals are replaced with library calls; state properties are replaced with `update` on the `state` type;
7. built-in method calls (e.g. `address.send()`) are replaced by library calls.

Figure 4 shows an example Solidity contract and its F* translation. The only type annotation added by the translation is a `Eth` effect on the contract's methods, explained in Section 2.2. The translated contract uses the `Solidity` library, which defines the `mapping` type (a reference to a map) and the associated functions `update_map` and `lookup`. This library also defines the base numeric type `uint`, for unsigned 256-bit integers, and arithmetic operators.

2.2 An effect for detecting vulnerable patterns

The example in Figure 4 illustrates two major pitfalls of Solidity programming. First, many contract writers fail to realize that `send` and its variants are not guaranteed to succeed (`send` returns a `bool`). This language feature is surprising for Solidity programmers because all other runtime errors (such as running out of gas or call stack overflows) trigger an exception. Such exceptions, including those triggered by `throw`, safely revert all transactions and all changes to the contract's properties. This is *not* the case of `send`: the programmer needs to undo any side effects manually when it returns `false`, e.g. by writing **if**(!addr.send(x)) **throw**.

The other pitfall illustrated in `MyBank` is reentrancy. Since transactions are also method calls, transferring funds to an-

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

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

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

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

```
module MyBank
open Solidity

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

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

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

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

Figure 4: A simple bank contract in Solidity translated to F⋆

other contract (either using `send`, or calling directly the fallback method) is a transfer of program control. In fact, `MyBank` suffers from a behavior similar to the one that allowed to drain TheDAO. Consider the malicious contract below:

```
contract Malicious {
  address owner;
  uint amount;
  MyBank bank;

  function Malicious(MyBank _bank) {
      bank = _bank;
      owner = msg.sender;
  }

  function Drain() {
     amount = msg.value;
     bank.Deposit.value(amount)(); // forwards all gas
     bank.Withdraw.value(0)(amount);
  }

  function Claim() { owner.send(this.balance); }

  function () { // fallback method
    if(msg.gas > 50000)
      bank.Withdraw.value(0)(amount);
  }
}
```

This contract can drain the balance of a `MyBank` contract by calling recursively into it at a point where it transfers Ether. The attack is triggered by calling the `Drain` method, which deposits and immediately withdraws from `MyBank`. When the `Withdraw` method transfers the funds back, it also transfers control back to `Malicious` by calling into its fallback method, which in turn issues a reentrant call to `Withdraw`. The `if` condition in the second `Withdraw` call is still satisfied because the balance is updated only after the original call to `Malicious` yields control, and as a result the same amount of Ether can be transferred more than once. Further re-entrant calls are only limited by the gas supplied to the initial call to `Drain`. Observe also that returning from a subsequent call to the fallback method of `Malicious` would result in decrementing the entry in the `balances` mapping more than once, causing an integer underflow. A variant of the attack could corrupt the balance in this way before freely withdraw any remaining funds from the bank. Note that using `send(amount)` instead of `msg.sender.call.value(amount)` in `MyBank` would mitigate the attack, because unlike method calls, `send` allocates only 2,300 gas for the call.

Using the effect system of F⋆, we now show how to detect some vulnerable patterns in translated contracts, such as not checking the result of external calls. The base construction is a combined exception and state monad (see [9] for details) with the following signature:

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

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

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

This F⋆ code captures the type of F⋆ computations that return values of type `a`. The monad carries a `send_failed` flag to record whether or not a `send()` or external call may have failed so far. It is possible to enforce several different styles based on this monad; for instance, one may want to enforce that a contract always throws when a `send` fails, by essentially coding a small reference monitor into the type system. As an example, we defined the following effect based on EST:

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

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

By typechecking extracted methods in the `Eth` effect, we can prevent dangerous patterns, such as an external call followed by an unconditional state update. These patterns trigger an F⋆ typechecking error at compile time. Note that the

safety condition imposed by `Eth` is not sufficient to prevent reentrency attacks, as there is no guarantee that the state modifications before and after an external call preserve the functional invariant of the contract. Therefore, this analysis is useful for detecting dangerous patterns and enforcing a failure handling style, but it does not replace a manual F* specification and proof for the contract.

2.3 Evaluation

Despite the limitations of our tool (in particular, it does not support many syntactic features of Solidity), we were able to translate and typecheck 46 out of the 396 contracts we collected from https://etherscan.io. We could already find several that are invalid in the `Eth` effect. While these results are still preliminary, they indicate that a larger scale analysis of published contracts would have substantial value.

3. DECOMPILING BYTECODE TO F*

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

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

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

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

Based on the static analysis, EVM* emits F* code, using variables bound in let bindings instead of stack cells. Many instructions can be eliminated in this way, so the analysis keeps an accurate account of the offsets of instructions in the remaining code. Because the instructions eliminated may incur gas charges, we precisely keep track of their fuel consumption by instrumenting the code with calls to burn, a library function whose sole effect is to accumulate gas charges, equipped with an F* type that fully captures that effect.

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

Figure 5: Decompiled code of `Balance` method in the `MyBank` contract, instrumented with gas consumption.

Figure 5 shows the F* code decompiled from the `Balance` method of the `MyBank` contract in Fig. 4; the corresponding EVM bytecode is 396 bytes long.

We wrote a reference cost model for bytecode operations that can be used to prove bounds on the gas consumption of contract methods. As an example, Fig. 6 shows a type annotation for the entry point of the `MyBank` contract decompiled to F* that states that a method call to the `Balance` function will consume at most 390 units of gas. This annotation can then be automatically discharged by F* typechecking.

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

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

Figure 6: A proof of a bound on the gas consumed by a call to the `Balance` method of `MyBank`.

4. CONCLUSION

Our preliminary experiments using F* to verify smart contracts suggests that the type and effect system of F* is flexible enough to capture and prove properties of interest for contract programmers. Our approach, based on shallow embeddings and typechecking within an existing verification framework, is convenient for exploring the formal verification of contracts coded in Solidity and EVM bytecode. On the other hand, static tools dedicated to these languages may be easier to use. (For instance, we miss tools to explain F* typechecking errors in simple source-code terms.)

In parallel with this work, Luu et al. [6] use symbolic execution to detect flaws in EVM bytecode programs, and an experimental Why3 [5] formal verification backend is now available from the Solidity web IDE [4].

The examples we considered are simple enough that we did not have to write a full implementation of EVM in F*. We plan to complete a verified reference implementation, and use it to verify that the output of the Solidity compiler is functionally equivalent to source contracts. We implemented EVM* and Solidity* in OCaml. It would be interesting to implement and verify parts of these tools using F* instead. For instance, we may try to prove that the stack and control flow analysis done in EVM* is sound with respect to a stack machine semantics.

5. ACKNOWLEDGMENTS

We thank Nicola Atzei, Massimo Bartoletti and Tiziana Cimoli from Università di Cagliari for pointing out a flaw in the code of the attack in Section 2 in an early draft.

References

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

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

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

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

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

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

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

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

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

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