Finding Unchecked Low-Level Calls with Zero False Positives and Negatives in Ethereum Smart Contracts

Puneet Gill, Indrani Ray, Alireza Lotfi Takami, and Mahesh Tripunitara(✉)

Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada
{p24gill,iray,alotfitakami,tripunit}@uwaterloo.ca

Abstract. Smart contracts are a relatively new class of computer programs that are intended to run on a blockchain. Checking a smart contract for security vulnerabilities is recognized as an important problem in both research and practice. Motivated by recent empirical work that suggests that existing tools suffer high numbers of false positives and negatives for vulnerabilities that belong to commonly-occurring classes, we ask: for even one such class of vulnerabilities, can there exist a tool that is highly effective? We answer in the affirmative by construction: for the class of unchecked low-level calls, checking for which is undecidable in general and **PSPACE**-complete under finitizing assumptions we adopt, we propose an approach for Ethereum smart contracts that comprises a reduction to model-checking, encoding the property in Linear Temporal Logic (LTL) and use of an off-the-shelf model checker. We find that across almost 200 smart contracts drawn from curated and "wild" datasets from the publicly available benchmark that underlies the prior empirical work that points out that existing tools suffer high numbers of false positives and negatives, our approach is highly effective in that we see zero false positives and negatives.

Keywords: Smart contract · Security vulnerability · Unchecked low-level calls · Model checking · Linear Temporal Logic (LTL)

1 Introduction

A smart contract is a computer program written with an intent of running it on a blockchain [15]. A blockchain, as a substrate of running a smart contract, offers certain guarantees when events occur in the running of that smart contract; for example, when the smart contract transmits digital currency to another smart contract and the latter accepts it, both the sender and the receiver can be assured that these events have indeed occurred, and that this fact is recorded immutably.

We focus on smart contracts written for the Ethereum blockchain [11]. Ethereum provides a so-called Ethereum Virtual Machine (EVM) within which a smart contract runs. The language an EVM understands and executes is called

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
G.-V. Jourdan et al. (Eds.): FPS 2022, LNCS 13877, pp. 305–321, 2023.
https://doi.org/10.1007/978-3-031-30122-3_19

EVM code [33]. EVM code is a somewhat low-level language, with instructions similar to those we find in an assembly language. From the standpoint of expressive power, EVM has been called quasi-Turing-complete; the rationale from Wood [33] is: "the *quasi* qualification comes from the fact that the computation is intrinsically bounded through a parameter, *gas*, which limits the total amount of computation done". (In Sect. 4, we address the computational complexity of verification problems of interest to us more clearly). Typically, a programmer writes a smart contract in a higher-level language such as Solidity [10], which is then compiled to EVM code for deployment in Ethereum.

```
pragma solidity ^0.4.18;
contract Lotto {
    bool public payedOut = false;
    address public winner;
    uint public winAmount;

    function sendToWinner() public {
        require(!payedOut);
        winner.send(winAmount);
        payedOut = true;
    }
    function withdrawLeftOver() public {
        require(payedOut);
        msg.sender.send(this.balance);
    }
}
```

```
...63ffffffff1680631846f51a146100725780...

                    ⋮

[48] PUSH4 0xffffffff
[53] AND
[54] DUP1
[55] PUSH4 0x1846f51a
[60] EQ
[61] PUSH2 0x0072
[64] JUMPI
[65] DUP1

                    ⋮
```

Fig. 1. The Solidity code for `Lotto.sol` from the curated dataset of Durieux et al. [8] is to the left. To the right is a portion of its complied EVM code as bytecode to the top, and the bytecode written as human-recognizable instructions and their arguments below it. A number in square brackets, e.g., "`[48]`", is the byte number within the bytecode, starting at 0, of the particular instruction + arguments.

Figure 1 shows a smart contract written in Solidity and a portion of its encoding as EVM code. As the Solidity code suggests, there is the notion of a contract, within which one can specify functions. The `address` data type identifies a publicly visible contract that resides on the Ethereum blockchain. A function in a contract may be invoked from another contract, sometimes with arguments. A function may invoke another contract, for example, via a `send()` call as shown in the example code. EVM code, as we show to the right of the figure, is a string of bytes. Each byte is either an instruction, e.g., `PUSH4`, or an argument to an instruction, e.g., `0x0072`.

Our Work. Verification as to whether a smart contract contains security vulnerabilities is a well-recognized problem in both research and practice. There are a number of software tools that address this; we present some of the most closely related to our work in Sect. 2. Our work is motivated by a recent empirical assessment of several such tools [8], which points out that they suffer high numbers of false positives and false negatives when they check for vulnerabilities from common occurring classes [23] in real-world smart contracts. (A false positive is

a tool reporting a vulnerability where there is none. A false negative is a vulnerability that goes undetected by the tool). We think that the underlying reason is that existing tools "try to do everything, none of them well." Said differently, the poor performance of existing tools leads us to ask: if we were to focus on even one class of vulnerabilities only, can there exist a tool that for real-world smart contracts, is highly effective? Lest one mistakenly think that checking for only one class is somehow "easy", as we discuss in Sect. 3, even a basic verification problem, such as whether a contract is guaranteed to halt when run on some input, is intractable for Ethereum smart contracts.

In this work, we answer the above question in the affirmative. Our proof is by construction—we discuss our reduction to model-checking, the use of Linear Temporal Logic (LTL) to meaningfully encode the property to be checked for, and the use of the off-the-shelf model checker nuXmv [3]. Our choice of the class of vulnerabilities is unchecked low-level calls. An unchecked low-level call is an invocation to another contract whose return value is not checked. As we establish in Sect. 3, this problem is undecidable in general, and **PSPACE**-complete under the finitizing assumptions we adopt. Our empirical results on almost 200 contracts from the benchmark of Durieux et al. [8] suggest that our approach is highly effective—we see zero false positives and negatives.

The remainder of this paper is organized as follows. In the next section, we discuss related work. In Sect. 3 we address the computational complexity of the problem. In Sect. 4, we present our reduction to model-checking and encoding in SMV. In Sect. 5, we discuss our empirical validation. We conclude with Sect. 6 in which we discuss also future work.

2 Related Work

Our work addresses checking a smart contract for security vulnerabilities that belong to commonly occurring classes of such vulnerabilities. As such, our work is related to work that proposes classifications of security vulnerabilities in smart contracts, and tools and techniques for detecting vulnerabilities. From the standpoint of tools, we focus on one class of vulnerabilities only, but are highly effective for that class. That is the key distinction between our work and all prior work on tools. We leverage a particular classification [23], but otherwise make no contributions to that aspect.

From the standpoint of classifications, several exist—Rameder [27] and Tolmach et al. [31] provide comprehensive surveys. We adopt the classification and taxonomy of DASP [23], which in turn is adopted by the empirical work of Durieux et al. [8]. The work of Durieux et al. [8] proposes two datasets of smart contracts which they have made available publicly [9], and using which they have assessed several existing tools and demonstrated that those tools suffer from too many false negatives. One of these is a curated set, with several smart contracts in each of the categories of DASP. Their other dataset is a "wild" dataset of several thousand smart contracts. We rely on their datasets for our empirical assessment. There have been other datasets that have been proposed, such as that of Ghaleb and Pattabiraman [13].

As the surveys of Rameder [27] and Tolmach et al. [31] express, there are several tools which analyze smart contracts for security vulnerabilities; a comprehensive discussion is beyond the scope of this work. In the remainder of this section, we survey several of the prior tools. We emphasize that none, to our knowledge, claims to suffer zero false positives and negatives for even one class of security vulnerabilities. That is the main difference between our work and prior approaches.

A piece of work that is closest to ours is that of Nehai et al. [24]. That work proposes applying model checking, and specifically the use of nuSmv, to "capture the behaviour of the Ethereum blockchain, the smart contracts themselves and the execution framework". Many of the other tools are based on verification techniques that check for security assurance and correctness of smart contracts. For example, Bhargavan et al. [2] create a framework that compiles contracts written in Solidity to F* and checks for, e.g., safety with respect to runtime errors; and decompiles EVM bytecode into F* code to check for low-level properties such as bounds on the amount of gas required for a transaction. Other tools, such as the one created by Amani et al. [1] extend formalisations of the Ethereum Virtual Machine and use theorem provers such as Isabelle or Coq to check for reachability properties in smart contracts, such as termination. There are symbolic execution frameworks, such as Manticore [21], which implement state exploration, but do not target smart contract vulnerabilities that can be exploited by attacks.

Oyente [19], however, does so. This symbolic execution tool uses the Z3 solver to determine the feasibility of execution traces of smart contracts (input as bytecode) to detect given vulnerabilities such as transaction-order dependence. Kalra et al. [16] argue that Oyente is neither sound nor complete and propose a symbolic model checker, Zeus, for Solidity-based smart contracts which categorizes them into incorrect and unfair groups, and further detects vulnerabilities such as re-entrancy in the former and 'incorrect logic' in the latter. Zeus inserts assertions based on vulnerabilities into smart contract code and, after translation to LLVM bitcode, uses the SeaHorn verifier to determine assertion violations. Zeus' soundness claims are refuted by the creators of eThor [30]: a sound and static analyzer for bytecode. This tool creates semantic abstractions based off of the blockchain environment, gas modelling, the memory model, and the callstack and scans for reachability properties to search for re-entrancy bugs.

Several other symbolic execution tools exist. MadMax [14] uses another tool, Vandal [4] to translate EVM bytecode to an intermediate representation and then detects out-of-gas exceptions. Maian [25] (based on Oyente) detects three types of vulnerable contracts: suicidal (can be killed by anyone), prodigal (can send Ether to anyone), and greedy (cannot have Ether extracted from). Mythril [22] uses a symbolic interpreter for EVM bytecode known as LASER to find abstract program states and reason about their reachability (using a Z3 solver) given certain conditions to determine vulnerabilities (such as integer overflow); then uses concolic testing to deign whether the vulnerabilities are exploitable. Securify [32] is a security analyzer for smart contracts that takes as input the

bytecode of a contract and checks for compliance and violation security patterns such as locked Ether.

teEther [18] is an automated framework which scans bytecode for instructions (such as `CALL`) that can be used to extract Ether, and then creates exploits. Another tool (which focuses on instructions related to inter-contract reasoning, memory, and hash functions) is the bounded model checker EthBMC [12]. The creators of the tool test it against others upon toy smart contract examples before conducting a large-scale analysis and comparing their results with those received from their experiments using teEther and Maian. That work has since been followed-up by those of Perez et al. [26] Zhang et al. [35], Rodler et al. [28], Chinen et al. [7], Cecchetti et al. [5] and Chen et al. [6]. All these pieces of work except those of Rodler et al. [28] and Cecchetti et al. [5] address detection of exploits and vulnerabilities; those two pieces of work propose defenses, the former via patching and the latter via a new type system.

Finally, specifically as it pertains to unchecked low-level calls, there is work on simply replacing such calls entirely in the source-code [34]. However, that work has its own set of limitations, such as the inability to identify all low-level calls, and whether the replacement indeed results in a contract that is equivalent to the original.

3 Computational Complexity

Our approach, which we discuss in the next section, is based on model-checking. In this section, we articulate the underlying foundations in computational complexity. We have two sets of results: that in general, checking for unchecked low-level calls in Ethereum smart contracts is undecidable, and that under finitizing assumptions we make, it is **PSPACE**-complete. To establish these results, we appeal to the halting problem for smart contracts. We omit full proofs on account of lack of space, and provide sketches only.

As we mention in Sect. 1, EVM has been called quasi-Turing-complete, with the quasi- being attributed to the fact that a sequence of computations is bounded by a parameter called gas, and every computation consumes some gas. Therefore, assuming that the value for gas is bounded by a constant, EVM is not Turing-complete. However, in our observation, gas is not the only limiting factor in this regard. The only storage offered in EVM are: (i) a *program counter*, which is 256 bits, (ii) a *stack*, which has a maximum size of 1024, each entry 256 bits, (iii) *memory*, addressed by 256 bits, each entry 8 bits, and, (iv) *storage*, addressed by 256 bits, each entry 256 bits. As all of these are bounded by constants, we do not have the unboundedness required to encode an arbitrary Turing machine. However, as the number 2^{256} is large, certainly from the standpoint of verification in practice, we may assume that any value whose upper-bound is 2^{256} is unbounded.

Theorem 1. *If we assume that any value bounded by 2^{256} only is unbounded, and gas is unbounded, then whether a run of a contract encoded in EVM halts on some input is undecidable.*

Our proof is by construction—we have an encoding of a Turing machine in Solidity, which then compiles to EVM. Our Solidity code uses the `uint` data type which is 256 bits for anything that may be unbounded, for example, the number of states, the size of the alphabet and the tape.

Corollary 1. *Detecting unchecked low-level calls in EVM is undecidable.*

We can reduce the halting problem that underlies the proof for Theorem 1 to the complement of the problem of finding unchecked low-level calls, and then appeal to the fact that a problem is undecidable if and only if its complement is undecidable. We wrap every instruction that causes a smart contract in EVM to stop running with a `CALL` instruction and a check of its return value. Thus, the contract halts if and only if the return value from the `CALL` is checked. There are only a few instructions in EVM that cause a contract to stop running, e.g., `STOP` and `REVERT`.

Theorem 2. *Suppose the number of memory and storage locations allocated and accessed by a smart contract C is at worst polynomial in the size of C and in $\log g$ where g is the value of the gas parameter. Then the problem of determining whether C terminates on some input is **PSPACE**-complete.*

The proof for **PSPACE**-hardness follows from an encoding of a Linear-Bounded Automaton (LBA) using our construction for a Turing machine to which we refer in the Proof for Theorem 1, and the fact that the halting problem for LBAs is **PSPACE**-hard. To prove that the problem is in **PSPACE**, we can construct a non-deterministic algorithm, denote it $\mathcal{A}$, to decide the problem such that $\mathcal{A}$ allocates at worst polynomial space only. We then appeal to Savitch's theorem that **NPSPACE** = **PSPACE** [29]. $\mathcal{A}$ simply allocates the maximum space that may be needed and runs the smart contract on the input. For any uninitialized or unknown value, e.g., the return from a `CALL` to an external contract, $\mathcal{A}$ chooses a return value non-deterministically. We know, from our assumptions, that the space $\mathcal{A}$ must allocate is at worst polynomial in the size of the input.

We need the "log" qualification for the input gas g for the reason that g may be encoded, for example, in binary, in which case, under the assumption that each instruction consumes constant gas, the contract C may run, on some input, for time $\Theta(g)$, and therefore may allocate space as much as $\Theta(g)$, where $\Theta(\cdot)$ represents a tight asymptotic bound.

Corollary 2. *Determining whether an EVM contract has an unchecked low-level call under the finitizing assumptions of Theorem 2 is **PSPACE**-complete.*

Similar to the proof for Corollary 1, we appeal to the fact that a problem is **PSPACE**-complete if and only if its complement is **PSPACE**-complete.

4 Reduction

From Corollary 2 in the previous section, we have a sufficient condition for our problem to be **PSPACE**-complete, and therefore, reduction to model checking that is complete for **PSPACE** is an appropriate approach. This is exactly what we do. Our implementation encodes the model checking problem in SMV [3], but more abstractly, the output of our reduction is a Kripke structure [17]. To us, a state is uniquely determined by the value of several variables that we associate with a state. We then define state-transitions between those states, some of which are non-deterministic. The variables are for the basic components of the state of an EVM: (1) The stack, memory and storage of the EVM, (2) the current stack-head, (3) the next instruction to be executed, and, (4) the gas remaining.

Program Counter. To keep track of the instructions, we have a variable which we call `operationName`, which is an enumeration of every instruction that appears in the EVM code that is input, annotated with its byte location. For example, suppose we have the EVM code `6005600401`. This corresponds to:

```
[0] PUSH1 0x05
[2] PUSH1 0x04
[4] ADD
```

Then, our reduction would introduce:

```
operationName : {begin, PUSH1_0, PUSH1_2, ADD_4,  end};
```

The `begin` and `end` are keywords we introduce to delimit the sequence of instructions. The state-transitions of the instructions would be the following (in SMV syntax, the value of the variable in the next state is shown after the colon, ":"):

```
next(operationName) :=  case
    operationName = begin : PUSH1_0;
    operationName = PUSH1_0 : PUSH1_2;
    operationName = PUSH1_2 : ADD_4;
    operationName = ADD_4 : end;
    TRUE : operationName;
esac;
```

Unless we have a `JUMP` or `JUMPI` instruction, our program counter increments sequentially, i.e., by the number of bytes the previous instruction + operands consume. We maintain also an `operationArray[ ]`, which identifies an operand, if any, in the EVM code for any operation. In the above example of two pushes and then an add, the argument to each of the two pushes is in the EVM code, and the argument to the add is not. Consequently, we would initialize and never change our `operationArray[ ]` as follows (`0ud`x`...` is the SMV syntax for an unsigned x-bit value):

```
operationArray[0] := 0ud8_5;
operationArray[2] := 0ud8_4;
```

Stack, Memory and Storage. For each of the stack, memory and storage, we maintain arrays. For the stack we maintain also a `stack_head`. We also exercise a design choice on the size of each entry in the stack. While we can certainly set those to 256-bit, for most smart contracts we observe in the benchmarks, this is too large. We can estimate the size we need via a simple static analysis for the `PUSH` instructions, and other instructions such as `ADD` and `MUL` that increase the size of a stack-entry. In EVM code, the push instructions are `PUSH1`,...,`PUSH32`, where `PUSH`x means x bytes are pushed. For memory and storage, the situation is different. There are two memory-store instructions, `MSTORE`, which stores 256 bits, and `MSTORE8`, which stores 1 byte—we find only the former in the datasets we have adopted [9]. There is only one storage-store instruction, `SSTORE`, which stores 256 bits. However, the value stored in memory and storage comes from the stack. Therefore, we again rely on our estimate of the maximum size stored on the stack for the sizes to be stored in memory and storage. Also, with the stack, in EVM, the maximum number of entries is 1024. However, this is again an upper-bound that is often loose. A count of the number of `PUSH` instructions gives us a tighter upper-bound on the number of possible entries in the stack. We may need to account for the possibility that the same `PUSH` instruction may be executed more than once. This depends on the specific property we are checking for. For unchecked low-level calls, we know that we need to account for at most a constant number of executions of a `PUSH` instruction.

```
stack : array 0 .. 9 of unsigned word[16];
stack_head : 0 .. 9;
memory : array 0 .. 3 of unsigned word[16];
memory_offsets : array 0 .. 3 of unsigned word[16];
storage : array 0 .. 4 of unsigned word[16];
storage_keys : array 0 .. 4 of unsigned word[16];
```

The `memory_offsets` and `storage_keys` are needed to identify to exactly which memory and storage locations a corresponding instruction refers. Consider, as an example, the following EVM code.

```
[0] PUSH1 0x80
[2] PUSH1 0x40
[4] MSTORE
```

The `MSTORE` instruction takes two arguments, both off the stack: an offset or location within memory, and the value to be stored. In the above example, the offset is `0x40` and the value to be stored is `0x80`. We would store the offset in some index, call it i, of `memory_offsets[ ]`, and ensure that the value in `memory[i]` is the value stored at that offset. We discuss below under "Instructions that need more than one transition" as to the manner in which we handle the `MSTORE`, `SSTORE` and their corresponding load instructions via state-transitions in our model.

The `stack[ ]` array changes based on (i) the particular instruction, and (ii) the current value of the `stack_head`. Given the quirks of the SMV syntax, we need to enumerate every possible next-state value of each entry of the stack. Thus, if our `stack[ ]` was specified to be of 10 entries, indexed `0...9` as we show above, then, we would have a `next(stack[i])` case statement for every `i` $= 0, \ldots, 9$. For example:

```
next(stack[6]) := case
    ...
    operationName = PUSH1_2 & stack_head = 6 : operationArray[2];
    operationName = DUP2_4 & stack_head = 6 : stack[4];
    ...
```

To explain the above, we first clarify that our stack grows downwards, i.e., as we push more items onto the stack, the value of `stack_head` increases. Also, the value of `stack_head` is the next entry in the stack; thus, for example, if `stack_head` is 6, then the top of at the stack is at 5, and we initialize `stack_head` to 0. Thus, in the above snippet, the only way the value in `stack[6]` can change in a state-transition is if the current value of `stack_head` is 6. If the instruction is `PUSH1`, then we get the value to be stored in `stack[6]` from our EVM code, which in turn is in our `operationArray[ ]` (see above for a discussion of this variable). If the instruction is `DUP2`, we need to make a copy of the value from one below the current head of the stack, i.e., `stack[4]`, and store that in `stack[6]`—`DUP`x pushes a copy of the value that is at depth x in the stack, where $x = 1$ refers to the top of the stack, $x = 2$ one below the top and so on.

The `stack_head` also changes based on the instruction and the current value of `stack_head`. As we say above, the stack grows downwards, i.e., we increment the `stack_head` as items are added to the stack and decrement it as items are popped. The only other detail is that in our implementation, our stack is circular, and every change to the `stack_head` is performed modulo its maximum size. This has enabled us to quickly experiment with small stack-sizes albeit while risking correctness, specifically, false negatives.

```
next(stack_head) := case
    operationName = CALL_0 : max(0, stack_head + 4) mod 10;
    operationName = SWAP4_1 : stack_head;
    operationName = POP_2 : max(0, stack_head + 9) mod 10;
    ...
```

In the above example, the maximum size of the stack has been set to 10, with the items indexed $0, \ldots, 9$. The `CALL` instruction pops 7 items off the stack and pushes 1, for a net of 6 items popped. Consequently, we update the stack head to `stack_head` $+ 4 \mod 10$. The `SWAP`x instruction swaps item at depth $x + 1$ on the stack with the item at the top; it does not change `stack_head`. The `POP` instruction decrements `stack_head` by 1.

Instructions that Need One Transition Only. We observe that for most instructions in EVM, we require one state-transition in our model only. The simplest

examples are those involving basic arithmetic, such as `ADD` and `MUL` which are supported directly in SMV. For example:

```
next(stack[2]) := case
    ...
    operationName = MUL_28 & stack_head = 4 : (stack[2] * stack[3]);
    ...
next(stack_head) := case
    ...
    operationName = MUL_28 : max(0, stack_head + 9) mod 10;
    ...
```

The above shows that if our instruction is `MUL`, then our stack decreases by a net of 1—`MUL` pops two items off the stack, multiplies them and pushes the result. Thus, if our current `stack_head` is 4, we multiply `stack_head[2]` and `stack_head[3]` and store the result in `stack[2]`.

There are also numerous other examples of instructions that require one state-transition only in our model. For example, the `CALL` instruction is crucial for us in our empirical validation on unchecked low-level calls (see Sect. 5). For our purposes, modeling `CALL` is somewhat surprisingly straightforward. The instruction causes another contract to be called with optional arguments. If we return from the other contract, we get a return value. From our standpoint, `CALL` causes a net decrement of 6 to the stack, and the return value that is stored on the top of stack is chosen non-deterministically, because we do not know what value will be returned. In the example above, we have shown the manner in which `stack_head` changes with a `CALL`. The following shows the change to an entry in the stack.

```
next(stack[1]) := case
    operationName = CALL_23 & stack_head = 8 : CALL_23_return_value;
    ...
```

The variable `CALL_23_return_value` is declared but never assigned a value; the model checker non-deterministically chooses a value for it.

Instructions that Need More than One Transition. For the other instructions, we require more than one transition in our model. Consequently, we need to be careful that any termination condition we specify to the model checker does not match an "intermediate" state because such as state would not exist in the EVM. Two examples of such instructions are `JUMP` (unconditional jump) and `JUMPI` (conditional jump). We adopt the mnemonic "`_DUMMY_`" to refer to the intermediate state. Below, we discuss `JUMP`; `JUMPI` is realized similarly. The only valid destination for a jump instruction is a byte that has the opcode `JUMPDEST`. Consequently, in our reduction, we only need to check to which `JUMPDEST` a particular `JUMP` or `JUMPI` seeks to jump in a particular instance. Suppose we have a `JUMP` instruction in the EVM code at byte #7, and a `JUMPDEST` at byte #12. Then, our `operationName` would be declared as:

```
operationName : {..., JUMP_7, JUMP_7_DUMMY, ..., JUMPDEST_12, ...};
```

For each JUMPDEST_x, we introduce a boolean variable jump_destination_is_x. Its value is initialized and changes as follows (as before, we assume that our stack has maximum size 10, indexed $0, \ldots, 9$):

```
init(jump_destination_is_12) := FALSE;
...
next(jump_destination_is_12) := case
  ...
  operationName = JUMP_7 & stack_head = 6 & stack[5] = 0ud16_12 : ↩
      TRUE;
  ...
```

That is, we set the value of jump_destination_is_12 to true if and only if the current operationName is a JUMP, and the value at the head of the stack is the same as the byte number of this JUMPDEST. The reason is that in EVM, JUMP's operand, which is the destination of the jump, is the value at the top of the stack. Note also that no two jump_destination_is_x variables can be simultaneously true. If the instruction is JUMPI instead, we would check also whether the value immediately below the top of stack, i.e., stack[4] in our above example, is > 0, because that is exactly where the condition for the JUMPI resides. We can then carry out the state-transition that effects the changes to operationName to the appropriate JUMPDEST. That is, we effect one state-transition to setup the correct destination for the jump (and check the condition, in the case of JUMPI), and a next state-transition to correctly update the operationName, i.e., our version of the program counter.

```
next(operationName) :=  case
  ...
  operationName = JUMP_7: JUMP_7_DUMMY;
  operationName = JUMP_7_DUMMY & jump_destination_is_12: JUMPDEST_12;
  operationName = JUMP_7_DUMMY & jump_destination_is_317: JUMPDEST_317;
  ...
```

For MSTORE and SSTORE, we use "_DUMMY_" variables for a similar reason that we need to setup the location at which we store. The only difference between MSTORE and SSTORE is that the former stores a value at at offset within memory while the latter's storage space is indexed by a key, often computed by exercising a cryptographic hash function, Keccak-256 or SHA-3. Corresponding to the hash function, EVM has an instruction, byte value 0x20, opcode KECCAK256 [33]. The computation of this hash value has itself been identified in some prior work as a source of difficulty in verifying smart contracts [12]—owever, we observe that while we could certainly implement either of those hash functions in SMV, we have simplified the work by realizing the Adler-32 checksum instead [20]. From our standpoint, there is no consequence except ease of implementation.

Gas. We capture gas in a variable, and use the various values on consumption of gas that are specified for EVM [33].

4.1 Unchecked Low-Level Calls

An unchecked low-level call is an invocation of a function of another contract from this contract. If and when that call eventually returns, a return value is pushed onto the stack of this contract. A contract is vulnerable if and only if there exists an instance of execution in which that return value is not checked. In Solidity, there are a number of ways to call another contract, e.g., `call()`, `send()` and `staticcall()`. In EVM code, there is one way only: the `CALL` instruction. Also, in Solidity, there are several different ways of checking the return value from such a call; e.g., using an `if` statement or by invoking the `require()` convenience function. In EVM code, the check is achieved using the `ISZERO` instruction. We adopt the algorithm in Fig. 2 to find unchecked low-level calls.

1 $S \leftarrow$ trim_contract()
2 **foreach** *contract snippet* $s \in S$ **do**
3 Statically check whether the initial `CALL` is followed by an `ISZERO`
4 If no, halt and report "is vulnerable"
5 If yes, reduce s to a model-checking instance
6 Add an LTLSPEC (see below)
7 Invoke the model checker; if it does not return a counterexample, report "is vulnerable"

Fig. 2. Our algorithm for finding unchecked low-level calls.

In `trim_contract()`, we first find each instance of the `CALL` instruction. We then scan forward in the EVM code from that instruction till we hit an instruction that we consider terminating from the standpoint of our trimming: `JUMP`, `JUMPI`, `STOP`, `RETURN`, `REVERT` or another `CALL`. We expect there to be at least one `ISZERO` before we hit one of these instructions; if not, we report that the contract is vulnerable in Line (4) of the above algorithm. If we proceed past Line (4) for a particular snippet s, we know that in s, the `CALL` is followed by at least one `ISZERO`. Corresponding to each `ISZERO` in a snippet s, we ask whether it checks the return value from the `CALL`. Assume that we have an `ISZERO` in byte # 12 and another in byte # 35 of the snippet. We first generate a random value, 59061 in the example below. We then ask whether there exists a reachable state in which the top of the stack contains that value when we hit any `ISZERO` instruction that follows the `CALL` in the snippet. In the example below, the maximum stack-size is 10, with the entries indexed $0, \ldots, 9$.

```
LTLSPEC G !(
(CALL_0_return_value = 0ud16_59061 & stack_head = 1 & stack[0] = ↩
    CALL_0_return_value & operationName = ISZERO_12) |
(CALL_0_return_value = 0ud16_59061 & stack_head = 1 & stack[0] = ↩
    CALL_0_return_value & operationName = ISZERO_35) |
(CALL_0_return_value = 0ud16_59061 & stack_head = 2 & stack[1] = ↩
    CALL_0_return_value & operationName = ISZERO_12) |
...
```

`CALL_`x`_return_value` is a variable we declare and allow the model checker to assign non-deterministically. If the model checker finds a counter example to the above LTLSPEC, then that is evidence that some instance of `ISZERO` that follows the `CALL` in the snippet s checks the return value from the `CALL`, and therefore no vulnerability exists. Otherwise, we know and report that a vulnerability exists. We recognize that there is a small probability of a false negative here because it is possible that the top of the stack takes the random value we generate, 59061 in the above example, not because it corresponds to the return value from the `CALL`, but on account of some other computations. We can simply repeat to exponentially decrease this probability.

5 Empirical Validation

A model-checker that can take specifications in SMV as input is nuXmv [3], and that is indeed what we have used. We have employed it in bounded model-checking more, where we infer the bound from the size of each contract snippet we check. For our empirical validation, we use the curated and "wild" datasets of Durieux et al. [8]. Their curated dataset comprises smart contracts written in Solidity classified into the 10 categories of DASP [23]. One of these classes is unchecked low-level calls. Each contract in the curated set is labelled where the vulnerability exists. For example, the contract in Fig. 1 from Sect. 1 suffers from an unchecked low-level call in the line `winner.send(winAmount)`.

When one considers the smart contracts from the entire curated set, we know that at least one contract in every file within the `unchecked_low_level_calls` subset contains the vulnerability. But, there may also be contracts in the curated set outside of that subset that suffer from the unchecked low-level calls vulnerability. Consequently, we organize our discussions below as follows. We first focus on files in the `unchecked_low_level_calls` subset of the curated set. As every one of those files contains the vulnerability, we cannot report any false positives. The question is whether we report any false negatives. We then consider the remainder of the files from the entire curated set. We report the manner in which our approach performs on them; in this case, both false positives and negatives are possible. Finally, we discuss our assessment on the "wild" dataset.

	Curated, unchecked low-level calls subset	*Curated, all others*	*Wild*
# files	52	91	47,581
# files we assessed	52	91	100
# contracts in files we assessed	84	129	664
# instances of `CALL`	269	160	805
# we determine are vulnerable	87	23	5
# false positives	0	0	0
# false negatives	0	0	0

Fig. 3. Table with statistics and our empirical results.

The table in Fig. 3 reports statistics on the datasets and our results. The first row of the table reports the number of smart contract files in each of the three datasets. The second row reports the number of files we assessed empirically against our implementation: it was all the files from the curated set, and 100 files chosen randomly from the "wild" set. The third row reports the total number of contracts in the Solidity code in each dataset, and the fourth row reports the total number of `CALL` instructions in the EVM code across all the contracts in the files we assessed. Thus, as we say in the table in the fourth row, we assessed more than 1000 contract snippets in the algorithm we discuss above. The fifth row, "# we determine vulnerable", is the number of those instances of the `CALL` instruction that we deem to be vulnerable. The final two rows reports the number of false positives and negatives, which are both zero. Our unit of measurement for the number of false positives and negatives is at the granularity of a file.

Comparison to Other Tools. The work of Durieux et al. [8] allows us to compare our results with prior tools. Their work reports "Vulnerabilities identified per category by each tool," (Table 5 in that work) from which we get a lower-bound on the number of false negatives for those tools. We observe that for unchecked low-level calls, every tool they studied suffers from a high number of false negatives—the best performer, Mythril [22], was able to detect only 5 of the total 12 instances of vulnerabilities. Our work is on a later, larger version of the dataset used in Durieux et al. [8] and we achieve zero false positives and negatives on the larger dataset.

Other Vulnerabilities. While the above results suggest that for unchecked low-level calls, our approach is highly effective, a natural question is whether we can extend it to address other commonly occurring classes of vulnerabilities in smart contracts. We think that the answer to this question is 'yes'. For example, it is possible for us to adopt the reduction from Sect. 4 with a different algorithm and LTL property than the ones from Sect. 4.1 to address reentrancy [23]. We leave this for future work.

6 Conclusions and Future Work

We have taken a different mindset than existing work towards checking for security vulnerabilities in smart contracts. Rather than trying to check for several different kinds of such vulnerabilities and as a consequence, building a tool that suffers high numbers of false positive and negatives as has been observed for such tools, we validate the hypothesis that if we focus on a class of commonly occurring vulnerabilities only, we can build a tool that suffers zero false positives and negatives for real-world smart contracts. The class which is our focus is as computationally hard as any other class we may want to check for. Our empirical results on a publicly available benchmark are highly promising—our tool suffers zero false positives and negatives.

Some future work is to extend our reduction to be able to detect other classes of vulnerabilities that are of particular interest in the context of smart contracts, such as reentrancy and front-running [23]. There is also the question as to whether we can detect new classes of vulnerabilities that are not members of known, commonly occurring classes. More broadly, there is the question of identifying characteristics unique to smart contracts from the standpoint of security vulnerabilities in them.

References

1. Amani, S., Begel, M., Bortin, M., Staples, M.: Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, New York, NY, USA, pp. 66–67 (2018). https://doi.org/10.1145/3167084
2. 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, New York, NY, USA, pp. 91–96 (2016). https://doi.org/10.1145/2993600.2993611
3. Bozzano, M., et al.: nuxmv 2.0.0 user manual (2019). https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf
4. Brent, L., et al.: Vandal: a scalable security analysis framework for smart contracts (2018). https://doi.org/10.48550/ARXIV.1809.03981
5. Cecchetti, E., Yao, S., Ni, H., Myers, A.C.: Compositional security for reentrant applications. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 1249–1267 (2021). https://doi.org/10.1109/SP40001.2021.00084
6. Chen, W., et al.: SADPonzi: detecting and characterizing Ponzi schemes in Ethereum smart contracts. Proc. ACM Meas. Anal. Comput. Syst. **5**(2), 1–30 (2021). https://doi.org/10.1145/3460093
7. Chinen, Y., Yanai, N., Cruz, J.P., Okamura, S.: RA: hunting for re-entrancy attacks in Ethereum smart contracts via static analysis. In: 2020 IEEE International Conference on Blockchain (Blockchain), pp. 327–336 (2020). https://doi.org/10.1109/Blockchain50366.2020.00048
8. Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE 2020, New York, NY, USA, pp. 530–541 (2020)
9. Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Smartbugs repository. https://github.com/smartbugs/smartbugs. Accessed July 2022
10. Ethereum: Solidity. https://docs.soliditylang.org/. Accessed July 2022
11. ethereum.org: Welcome to ethereum. https://ethereum.org/. Accessed July 2022
12. Frank, J., Aschermann, C., Holz, T.: ETHBMC: a bounded model checker for smart contracts. In: USENIX Security Symposium, USENIX 2020, pp. 2757–2774 (2020)
13. Ghaleb, A., Pattabiraman, K.: How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, pp. 415–427. Association for Computing Machinery, New York (2020)

14. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: MadMax: surviving out-of-gas conditions in ethereum smart contracts. In: Proceedings of the ACM on Programming Languages. OOPSLA, vol. 2, pp. 1–27. New York (2018)
15. IBM: What is blockchain technology? https://www.ibm.com/topics/what-is-blockchain. Accessed July 2022
16. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: Network and Distributed Systems Security (NDSS) Symposium, NDSS 2018, pp. 18–21 (2018). https://doi.org/10.14722/ndss.2018.23082, http://pages.cpsc.ucalgary.ca/~joel.reardon/blockchain/readings/ndss2018_09-1_Kalra_paper.pdf
17. Kripke, S.: Semantical considerations on modal logic. Acta Philos. Fennica 83–94 (1963)
18. Krupp, J., Rossow, C.: teEther: gnawing at Ethereum to automatically exploit smart contracts. In: USENIX Security Symposium, USENIX 2018, pp. 1317–1333 (2018)
19. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: CCS, CCS 2016, New York, NY, USA, pp. 254–269 (2016)
20. Maxino, T.: Revisiting Fletcher and Adler Checksums. DSN 2006 Student Forum (2006). https://doi.org/10.1184/R1/6625619.v1, https://kilthub.cmu.edu/articles/journal_contribution/Revisiting_Fletcher_and_Adler_Checksums/6625619
21. Mossberg, M., et al.: Manticore: a user-friendly symbolic execution framework for binaries and smart contracts. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), ASE 2019, pp. 1186–1189 (2019). https://doi.org/10.1109/ASE.2019.00133
22. Mueller, B.: Smashing Ethereum smart contracts for fun and real profit. In: 9th Annual HITB Security Conference (HITBSecConf) (2018)
23. NCC Group: Decentralized security project (2022). https://dasp.co
24. Nehai, Z., Piriou, P.Y., Daumas, F.: Model-checking of smart contracts. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pp. 980–987 (2018). https://doi.org/10.1109/Cybermatics_2018.2018.00185
25. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, New York, NY, USA, pp. 653–663 (D2018). https://doi.org/10.1145/3274694.3274743
26. Perez, D., Livshits, B.: Smart contract vulnerabilities: vulnerable does not imply exploited. In: USENIX Security Symposium, pp. 1325–1341 (2021)
27. Rameder, H.: Systematic review of Ethereum smart contract security vulnerabilities, analysis methods and tools. Diploma thesis (2021). https://doi.org/10.34726/hss.2021.86784
28. Rodler, M., Li, W., Karame, G.O., Davi, L.: EVMPatch: timely and automated patching of ethereum smart contracts. In: USENIX Security Symposium, pp. 1289–1306 (2021)
29. Savitch, W.J.: Relationships between nondeterministic and deterministic tape complexities. J. Comput. Syst. Sci. **4**(2), 177–192 (1970). https://doi.org/10.1016/S0022-0000(70)80006-X

30. Schneidewind, C., Grishchenko, I., Scherer, M., Maffe, M.: eThor: practical and provably sound static analysis of Ethereum smart contracts. In: CCS, CCS 2020, New York, NY, USA, pp. 621–640 (2020). https://doi.org/10.1145/3372297.3417250
31. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Comput. Surv. **54**(7), 1–38 (2021)
32. Tsankov, P., Dan, A., Cohen, D.D., Gervais, A., Buenzli, F., Vechev, M.: Securify: practical security analysis of smart contracts. In: CCS, CCS 2018, New York, NY, USA, pp. 67–82 (2018). https://doi.org/10.1145/3243734.3243780
33. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Technical report Version 3078285, Ethereum & Parity (2022). https://ethereum.github.io/yellowpaper/paper.pdf
34. Xi, R., Pattabiraman, K.: When they go low: Automated replacement of low-level functions in Ethereum smart contracts. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 995–1005 (2022)
35. Zhang, M., Zhang, X., Zhang, Y., Lin, Z.: TXSPECTOR: uncovering attacks in Ethereum from transactions. In: USENIX Security Symposium, pp. 2775–2792 (2020)