SOLC-VERIFY: A Modular Verifier for Solidity Smart Contracts

Ákos Hajdu[1(✉)] and Dejan Jovanović[2]

[1] Budapest University of Technology and Economics, Budapest, Hungary
hajdua@mit.bme.hu
[2] SRI International, New York City, USA
dejan.jovanovic@sri.com

Abstract. We present SOLC-VERIFY, a source-level verification tool for Ethereum smart contracts. SOLC-VERIFY takes smart contracts written in Solidity and discharges verification conditions using modular program analysis and SMT solvers. Built on top of the Solidity compiler, SOLC-VERIFY reasons at the level of the contract source code, as opposed to the more common approaches that operate at the level of Ethereum bytecode. This enables SOLC-VERIFY to effectively reason about high-level contract properties while modeling low-level language semantics precisely. The properties, such as contract invariants, loop invariants, and function pre- and post-conditions, can be provided as annotations in the code by the developer. This enables automated, yet user-friendly formal verification for smart contracts. We demonstrate SOLC-VERIFY by examining real-world examples where our tool can effectively find bugs and prove correctness of non-trivial properties with minimal user effort.

1 Introduction

Distributed blockchain-based applications are gaining traction as a secure and trustless alternative to more centralized solutions that require trusted intermediaries such as banks. The focus of early blockchain implementations, such as Bitcoin [34], was to provide the infrastructure for one particular application: digital money (cryptocurrency). Public blockchains allow arbitrary parties to transact with each other in a secure and trustless manner, with no central authority. In this setting a blockchain is a distributed ledger of transactions, where nodes in a peer-to-peer network are processing and validating transactions to maintain integrity. The next step in the evolution of blockchains was to extend the blockchain to a setting where the digital money can also be programmable. This is achieved by generalizing the ledger to allow deployment of programs (termed smart contracts [39]) that operate over ledger data. Blockchains with support for smart contracts provide a general distributed computing platform and allow

Á. Hajdu—The author was also affiliated with SRI International as an intern during this project.

S. Chakraborty and J. A. Navas (Eds.): VSTTE 2019, LNCS 12031, pp. 161–179, 2020.
https://doi.org/10.1007/978-3-030-41600-3_11

a set of mutually distrusting parties to execute and enforce their contractual terms (expressed as code) automatically. At the moment, the most popular such platform is the Ethereum blockchain [3,41].

However, smart contracts are often prone to errors with potentially devastating financial effects (see, e.g., [4] for a survey). The infamous DAO bug [16] is an illustrative example of the difficulties involved in deploying a smart contract. The DAO was a relatively small contract (2KLOC of Solidity code) that was heavily scrutinized by the wider Ethereum community before deployment. Nevertheless, an attacker managed to exploit a subtle reentrancy bug to steal $60M worth of cryptocurrency. Examples such as the DAO highlight the mission-critical nature of smart contracts. Although the code of the contract is usually small by the standards of modern software, if the contract attracts a large amount of investment, the code carries a significant amount of value per line of code. Moreover, since the contract code is stored on the blockchain, once deployed, the code is immutable and making upgrades or bug-fixes is impossible without complex solutions that involve a central authority. There has been a great interest in applying formal methods to verify smart contracts [4,23,32]. While there are ongoing projects based on identifying specific vulnerability patterns [8,13,20,22,29,35,40], theorem provers [24,25,37], finite automata [1,30] or SMT [2,26,27], they all have limitations in terms of scalability, precision, expressiveness and ease of use.

In this paper we present SOLC-VERIFY, a tool for formal verification of Ethereum smart contracts that integrates seamlessly with developer tools. SOLC-VERIFY follows the modular software verification approach (e.g., VCC [11], HAVOC [10], and ESC/Java [21]), in the context of Solidity. Given a Solidity contract, annotated with specifications, SOLC-VERIFY translates the contract into the Boogie intermediate verification language [15,28], and discharges verification conditions by SMT solvers [7]. Developers can define the expected behavior of their contracts using annotations within the contract code, including assertions, contract and loop invariants, and function pre- and post-conditions. Verification of smart contracts brings domain-specific challenges. To start with, the semantics of Solidity include Ethereum-specific constructs such as the blockchain state, transactions, and data-types not common in general programming languages. As an example, Ethereum smart contracts generally operate on 256-bit integers, making precise reasoning about low-level properties, such as the absence of overflows, infeasible with standard SMT techniques. Furthermore, some common high-level properties of smart contracts, such as "the sum of user balances is always equal to the total supply", cannot be expressed in first-order logic or in Solidity, and therefore need domain-specific treatment. SOLC-VERIFY addresses these issues through an SMT-friendly encoding of Solidity into Boogie that is expressive enough to capture the properties of interest, and takes advantage of recent advances in SMT solving to enable effective reasoning. We describe SOLC-VERIFY through examples and demonstrate how SOLC-VERIFY can both find non-trivial bugs in real-world examples and prove correctness after the bugs have been fixed (e.g., the BEC token [36] hack). As far as we know, SOLC-VERIFY is the first tool that allows specification and modular verification of Solidity smart contracts that is practical and automatic. solc-verify is implemented as an add-on to the open-source Solidity compiler and is available on GitHub.[1]

2 Background

Ethereum. Ethereum [3,41] is a generic blockchain-based distributed computing platform. The Ethereum ledger is a storage layer for a database of *accounts* and data associated with those accounts, where each account is identified by its *address*. Ethereum contracts are usually written in a high-level programming language, most notably Solidity [38], and then compiled into the bytecode of the Ethereum Virtual Machine (EVM). A compiled contract is deployed to the blockchain using a special transaction that carries the contract code and sets up the initial state with the constructor. At that point the deployed contract is issued an address and stored on the ledger. From then on, the contract is publicly accessible and its code cannot be modified. A user (or another contract) can interact with a contract through its public API by calling public functions. This can be done by issuing a *transaction* with the contract's address as the recipient. The transaction contains the function to be called along with the arguments, and an execution fee called *gas*. Optionally, some value of Ether (the native currency of Ethereum) can also be transferred with transactions. The Ethereum network then executes the transaction by running the contract code in the context of the contract instance. During their execution, each instruction costs some predefined amount of gas. If the contract overspends its gas limit, or there is a runtime error (e.g., an exception is thrown, or an assertion is triggered), the entire transaction is aborted and has no effect on the ledger (apart from charging the sender for the used gas).

Solidity. Figure 1 shows a Solidity contract `SimpleBank` that illustrates some of the common features that Ethereum contracts use in practice. A contract can have *state variables*, which define the persistent data that the contract will store on the ledger. The state of `SimpleBank` consists of a single variable `balances`, which is a *mapping* from addresses to 256-bit integers. Further Solidity types include *value types*, such as Booleans, signed and unsigned integers (of various bit-lengths), addresses, fixed-size arrays, enums, and *reference types*, to be used with arbitrary-size arrays and structures. Once deployed, an instance of `SimpleBank` will be assigned its address and since no constructor is provided, its data will be initialized to default values (in this case an empty mapping).

Contracts define *functions* that can act on their state. Functions can receive data as arguments, perform computation, manipulate the state variables and interact with other accounts. In addition to declared parameters, functions also receive a `msg` structure that contains the details of the transaction. Our example contract defines two public functions `deposit` and `withdraw`. The `deposit` function is marked as `public` and `payable`, meaning that it can be called by anyone and is allowed to receive Ether as part of the call. This function reads

[1] https://github.com/SRI-CSL/solidity.

```
1  /** @notice invariant sum(balances) == this.balance */
2  contract SimpleBank {
3    mapping(address => uint256) balances;
4
5    function deposit() payable public {
6      balances[msg.sender] += msg.value;
7    }
8
9    function withdraw(uint256 amount) public {
10     require(balances[msg.sender] > amount);
11     if (!msg.sender.call.value(amount)("")) {
12       revert();
13     }
14     balances[msg.sender] -= amount;
15   }
16 }
```

Fig. 1. An example Solidity smart contract implementing a simple bank. Users can deposit and withdraw Ether with the corresponding functions, and the contract keeps track of user balances. The top level annotation states that the contract will ensure that the sum of individual balances is equal to the total balance in the bank.

```
/** @notice invariant x == y */
contract C {
  int x;
  int y;

  /** @notice precondition x == y
      @notice postcondition x == (y + n) */
  function add_to_x(int n) internal {
    x = x + n;
    require(x >= y); // Catch overflow
  }

  function add(int n) public {
    require(n >= 0);
    add_to_x(n);
    /** @notice invariant y <= x */
    while (y < x) {
      y = y + 1;
    }
  }
}
```

Fig. 2. An example Solidity smart contract illustrating the annotation features of SOLC-VERIFY, including contract-level invariants, pre- and post-conditions and loop invariants.

the amount of Ether received from `msg.value` and adds it to the balance of the caller, whose address is available in `msg.sender`. The `withdraw` function allows users to withdraw a part of their bank balance. The function first checks that the sender's balance in the bank is sufficient using a `require` statement. If the condition of `require` fails, the transaction is reverted with no effect. Otherwise the function sends the required amount of Ether funds by using a `call` on the caller address with no arguments (denoted by the empty string). The amount to be transferred is set with the `value` function. The recipient of the `call` can be another contract that can perform arbitrary actions on its own (within the gas limits) and can also fail (indicating it in the return value). If `call` fails, the whole transaction is reverted with an explicit `revert`, otherwise the balance of the caller is deducted in the mapping as well.

`SimpleBank` contains a classic reentrancy vulnerability that can be exploited to steal funds from the bank. As the control is transferred to the caller in line 11, before their balance is deducted in line 14, they are free to make another call to `withdraw` to perform a double (or multiple) spend. Although this flaw seems basic, it is the issue that lead to the loss of funds in the DAO hack [16].

3 Overview and Features

SOLC-VERIFY is implemented as an extension to the Solidity compiler. It takes a set of Solidity contracts including specification annotations and discharges verification conditions using the Boogie verifier and SMT solvers. An overview of the architecture is shown in Fig. 3.

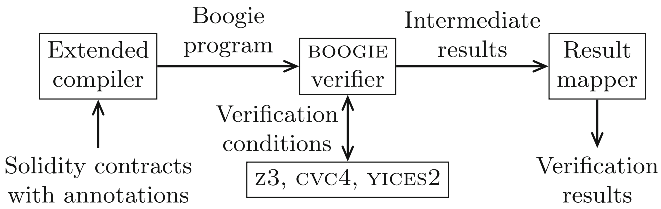

Fig. 3. Overview of the SOLC-VERIFY modules. The extended compiler creates a Boogie program from the Solidity contract, which is checked by the BOOGIE verifier using SMT solvers. Finally, results are mapped back and presented at the Solidity code level.

Specification. Solidity provides only a few error handling constructs (e.g., `assert`, `require`) for the programmer to specify expected behavior. Therefore, SOLC-VERIFY supports in-code *annotations* to specify contract properties, as illustrated in Fig. 2. Annotations are side-effect free Solidity expressions, which can reference any variable in the scope of the annotated element. *Contract-level invariants* (line 1) must hold before and after the execution of every public function and after the contract constructor. Non-public functions are inlined to a depth of one by default, but can also be specified with *pre- and postconditions* (lines 6–7). Moreover, *loop invariants* (line 16) can be attached to loops. As an extension, we also provide a special *sum* function over collections (arrays and mappings) in the specification language, as seen for example for `SimpleBank` in Fig. 1. The sum function is modeled internally by associating a ghost variable to the collection tracked by the sum: each collection update also updates the ghost variable. This encoding is a sufficient abstraction for our needs.

Correctness. SOLC-VERIFY targets functional correctness of contracts with respect to completed[2] transactions and different types of failures. An *expected failure* is a failure due to an exception deliberately thrown to guard from the user (e.g., `require`, `revert`). An *unexpected failure* is any other failure (e.g., `assert`, overflow). We say that a contract is *correct* if all transactions (public function calls) that do not fail due to an expected failure also do not fail due to an unexpected failure and satisfy their specification.

Translation to Boogie. SOLC-VERIFY relies on the Solidity compiler that parses the contracts and builds an abstract syntax tree (AST) where names, references and types are resolved. SOLC-VERIFY then traverses the internal AST and produces a Boogie [15,28] representation of the program. We discuss the details and properties of the translation in more detail in Sect. 4.

Boogie and SMT. Boogie transforms the program into verification conditions (VCs) and discharges them using SMT solvers. By default, Boogie can use Z3 [33]

[2] Due to the usage of gas, total and partial correctness are equivalent. Furthermore, currently we do not model gas: running out of gas does not affect correctness as the transaction is reverted. However, we might model it in the future in order to verify liveness properties or to be able to specify an upper bound.

and CVC4 [6] but we also extended it to support YICES2 [18]. A notable feature of our encoding is that it allows quantifier-free VC generation, permitting to use SMT solvers that do not support quantifiers (e.g., YICES2). Boogie reports violated annotations and failing assertions in the Boogie program and SOLC-VERIFY maps these errors back to the Solidity code using traceability information. The final output of SOLC-VERIFY is a list of errors corresponding to the original contracts (e.g., line numbers, function names).

4 Translation Details and Properties

The core of SOLC-VERIFY is a translation from Solidity contracts to the Boogie IVL, supporting a majority of the Solidity language.[3]

Contracts. The input of the translation is a collection of contracts to be verified and the output is a single Boogie program with all contracts. SOLC-VERIFY can reason about single and multiple contracts as well. If the code of all contracts is available, SOLC-VERIFY can take all available annotations into account when reasoning. However, this can be unsafe as EVM addresses are not typed (any address can be cast to a contract type) and is to be used with care. SOLC-VERIFY also supports inheritance by relying on the compiler to perform flattening and virtual-call disambiguation.

Types. SOLC-VERIFY supports basic Solidity types such as Booleans, integers and addresses. Several modes are provided for modeling arithmetic operations that can be selected by the user. In the simplest mode, integers are unbounded *mathematical integers*. This mode does not capture the exact semantics of the operations (e.g., overflows) but is scalable and well supported by SMT solvers. Precise arithmetic can be provided by relying on the SMT theory of *bitvectors*. SOLC-VERIFY supports this mode but can suffer from scalability issues due to the 256-bit default integer size of Solidity. In order to provide both precision and scalability, SOLC-VERIFY provides a *modular arithmetic* mode that encodes arithmetic operations using mathematical integers with range assertions and precise wraparound semantics of all operations. Addresses are modeled with uninterpreted symbols as they can only be queried for equivalence. SOLC-VERIFY also supports mappings and arrays using SMT arrays [14,31]. Structures, enumerations and tuples are currently not supported but there are no technical difficulties in supporting them and they are planned in the future. Events (a logging mechanism) are ignored as they are not relevant for functional correctness.[4]

State Variables. State variables are mapped to global variables in Boogie. However, multiple instances of a contract can be deployed to the blockchain at

[3] The paper and the experiments are based on compiler version v0.4.25, but we keep SOLC-VERIFY up to date with the latest development branch.

[4] We might model events in the future to be able to specify that an event is expected to be triggered.

different addresses. Since aliasing of contract storage is not possible, SOLC-VERIFY models each state variable as a one-dimensional global mapping from contract addresses to their respective type (in essence treating the blockchain as a heap in a Burstall-Bornat model [9]). For example, the state variable `x` with type `int` at line 2 of Fig. 4 (left) is transformed to the global variable `x` with mapping type `[address]int` at line 1 of Fig. 4 (right).

```
1  contract A {
2    int public x;
3    function set(int _x) public { x = _x; }
4  }
5  contract B {
6    A a;
7    function setXofA(uint x) public { a.set(x); }
8    function getXofA() public returns (uint) {
9      return a.x();
10   }
11 }
```

```
1  var x: [address]int;
2  procedure set(_this: address, _x: int) {
3    x := x[_this :=_x];
4  }
5  var a: [address]address;
6  procedure setXofA(_this: address, x: int) {
7    call set(a[_this], x);
8  }
9  procedure getXofA(_this: address) returns (r: int) {
10   r := x[a[_this]];
11 }
```

Fig. 4. Solidity contract (left) and its Boogie translation (right), illustrating the representation of the blockchain data as a heap and the receiver parameter of functions.

Functions. Each function in Solidity is translated to a procedure in Boogie with an additional implicit receiver parameter [5] called `_this`, which identifies the address of the contract instance. As an example, consider the `set` function of the Solidity contract `A` in Fig. 4. Updating `x` in the Boogie program becomes an update of the map `x` using the receiver parameter `_this`. Consider also the call `a.set(x)` in the Solidity function `setXofA`. The Boogie program first gets the address of the `A` instance corresponding to the current `B` instance using `a[_this]`. Then it passes this address to the receiver parameter of the function `set`.

Functions can be declared view (cannot write state) or pure (cannot read or write state), but these restrictions are checked by the compiler. Additional user-defined function modifiers are a language feature of Solidity to alter or extend the behavior of functions. In practice, modifiers are commonly used to weave in extra checks and instructions to functions. For example, the `pay` function in Fig. 5 (left) includes the modifier `onlyOwner` (defined in line 4), which performs an extra check before calling the actual function (denoted by the placeholder `_`). SOLC-VERIFY simply inlines statements of all modifiers of a function to obtain a single Boogie procedure (e.g., `pay` procedure in Fig. 5 right).

Statements and Expressions. Most of the Solidity statements and expressions can be directly mapped to a corresponding statement or expression in Boogie with the same semantics, including variable declarations, conditionals, `while` loops, calls, returns, indexing, unary/binary operations and literals. There are also some statements and expressions that require a simple transformation, such as mapping `for` loops to `while` loops or extracting nested calls and assignments within expressions to separate statements using fresh temporary variables. SOLC-VERIFY currently does not support inline assembly and creating new contracts from within another contract (`new` expressions). Furthermore, the availability of some arithmetic operations depends on the expressiveness of the underlying domain (e.g., bitwise operations).

```
1  contract Wallet {
2    address owner;
3
4    modifier onlyOwner() {
5      require(msg.sender == owner);
6      _;
7    }
8    function receive() payable public {
9      // Actions could be performed here
10   }
11   function pay(address to, uint amount)
           public onlyOwner {
12     to.transfer(amount);
13   }
14 }
```

```
1  var _balance: [address]int;
2
3  var owner: [address]address;
4
5  procedure receive(_this: address, _msg_sender: address,
         _msg_value: int) {
6    _balance := _balance[_this := _balance[_this] + _msg_value];
7    // Actions could be performed here
8  }
9  procedure pay(_this: address, _msg_sender: address, _msg_value:
         int, to: address, amount: int) {
10   assume(_msg_sender == owner[_this]);
11   assume( _balance[_this] >= amount);
12   _balance := _balance[_this := _balance[_this] - amount];
13   _balance := _balance[to := _balance[to] + amount];
14 }
```

Fig. 5. A simple wallet, which can receive Ether from anyone but only the owner can make transfers. This example illustrates various Ethereum and blockchain features in Solidity (left) along with their representation in Boogie (right).

Transactions. Solidity includes Ethereum-specific functions and variables to query and manipulate balances and transactions. Some examples can be seen in Fig. 5 (left) with the corresponding translation in Fig. 5 (right). Each address is associated with its balance, which can be queried using the `balance` member of the address. Correspondingly, SOLC-VERIFY keeps track of the balances in a global mapping from addresses to integers (line 1 of Fig. 5 right).

Solidity offers the `msg.sender` field within functions (line 5 of Fig. 5 left) to access the caller address. SOLC-VERIFY maps this to Boogie by adding an extra parameter `_msg_sender` of type `address` to each procedure. When a procedure calls another, the current receiver address (`_this`) is passed in as the sender.

Solidity functions marked with the `payable` keyword (line 8 of Fig. 5 left) are capable of receiving Ether when called. The amount of Ether received can be queried from the `msg.value` field. SOLC-VERIFY models this in Boogie by including an extra parameter `_msg_value` and updating the global balances map at the beginning of the corresponding Boogie procedure (line 6 of Fig. 5 right). When calling a payable function in Solidity, the amount of Ether to be transferred can be set with the special `value` function (e.g., line 11 of Fig. 1). SOLC-VERIFY translates this to Boogie by reducing the balance of the caller before making the call and passing the value as the `_msg_value` argument.

The functions `send` and `transfer` are dedicated functions to transfer Ether between addresses. SOLC-VERIFY inlines these functions by manipulating the global mapping of balances directly. If the recipient is a contract, a special fallback function is executed, but the gas passed is limited to raising events, which is irrelevant for functional correctness.[5] For example, the transfer in line 12 of Fig. 5 (left) is mapped to lines 11–13 on the right. The sender not having enough funds is an expected transaction failure, which is modeled with an assumption.

The function `call` can call a function by its name on any address and can also pass arbitrary data. Since there can be an unknown code behind the called

[5] Gas costs of certain write operations were about to change with Constantinople, allowing a reentrancy attack, but it was reverted with the St. Petersburg upgrade [19].

address, SOLC-VERIFY treats such cases as an external call that can perform arbitrary computation.[6] SOLC-VERIFY does not support low-level function calls such as `callcode` and `delegatecall` as it is considered dangerous and would require encoding of the EVM details (contract layout, EVM semantics).

Error Handling. Solidity exceptions will undo all changes made to the global state by the current call. Deliberately thrown exceptions (`require`, `revert`, `throw`) are therefore mapped to assumptions in Boogie, which stop the verifier without reporting an error. Assertions are mapped to Boogie assertions, causing a reported error when their condition evaluates to false.

Detection of Overflows. Neither the EVM nor Solidity performs any checking of the results of arithmetic operations by default. Due to the wraparound semantics of integers, this allows unexpected overflows and underflows to occur undetected (e.g., the infamous BEC token [36]).

In general, overflows can be detected by checking the result of every operation after it has been computed. However, reporting every such overflow would result in an overwhelming number of false alarms. For example, it is common practice for Solidity developers to perform arithmetic operations first, and then check for overflows manually after the fact (see, e.g., line 10 of Fig. 2). This practice of overflow detection is an integral part of the SafeMath library [17] that is used in almost all deployed contracts on the Ethereum blockchain and is part of Solidity best practices [12].

To reduce the number of false overflow reports, SOLC-VERIFY uses the following approach. Whenever an arithmetic computation is performed, it computes the *overflow condition* that captures whether the overflow has occurred (i.e., if the result of the computation in modular arithmetic is different from the result over unbounded integers). However, instead of immediately checking this condition, it is accumulated in a dedicated Boolean overflow-detection variable. SOLC-VERIFY then checks for overflow at the end of every basic block with an assertion. This "delayed checking" gives space to developer to perform manual checking for the overflow (in which case the assertion will not trigger) and will avoid the false alarms. For example, the potential overflow in line 9 of Fig. 2 is not reported because in the very next line the programmer guards the overflow and reverts the transaction.

5 Examples and Experiments on Real World Contracts

To demonstrate SOLC-VERIFY we first discuss the coverage of currently-supported language features and scalability by examining the (unannotated) contracts currently deployed on the Ethereum blockchain. We also pick a subset of the unannotated contracts and manually check what SOLC-VERIFY can report on them. Finally, we examine two contracts that had been exploited in

[6] Contract invariants are also checked before external calls as they can perform a callback to the contract.

Table 1. Etherscan results with different solvers and arithmetic encodings. Each cell represents the number of successfully processed contracts (of 7836 total) and the average execution time per contract.

Encoding	`int`	`bv`	`mod`	`mod-overflow`
Translated	4096	3919	3926	3926
CVC4	4090 (0.71 s)	3837 (0.99 s)	3921 (0.72 s)	3911 (0.79 s)
YICES2	3892 (1.15 s)	3854 (0.86 s)	3903 (0.75 s)	3859 (0.87 s)
Z3	3897 (1.24 s)	3831 (1.10 s)	3892 (0.87 s)	3894 (0.88 s)

the past, and show how SOLC-VERIFY could have found the issues, with minimal annotation burden, and prove that the fixed versions of the contracts are correct.

5.1 Language Coverage

To analyze the coverage of currently-supported language features and the scalability of SOLC-VERIFY, we collected 37531 contracts available on Etherscan.[7] These contracts were compiled with various versions of the Solidity compiler and not all of them are supported by version 0.4.25 that SOLC-VERIFY used at the time of writing the paper. We therefore selected 7836 contracts that do compile. The results of running SOLC-VERIFY on the selected contracts is shown in Table 1. Columns correspond to different arithmetic modes, with the last column representing modular arithmetic with overflow checking enabled. The first row shows that roughly 50% of the contracts can be successfully translated to Boogie in each mode. Contracts that cannot be translated contain constructs not yet handled by SOLC-VERIFY, such as structures, enumerations or special transaction and blockchain members. Some features (e.g., exponentiation) also depend on the arithmetic mode, resulting in slight differences in feature coverage. The remaining three rows show the number of contracts for which SOLC-VERIFY terminates within 10 s with a given SMT solver as a backend. Note, that the effectiveness of the different SMT solvers on this set of contracts should be taken with a grain of salt. For example, the bitvector encoding seems to be nearly as efficient as modular arithmetic. However, this is because the assertions in these contracts do not depend on complex (e.g., nonlinear) arithmetic. With more complex invariants, the bitvector encoding becomes infeasible for reasoning, as we demonstrate it with the BEC token example later in this section. The takeaway of these results is that the average execution time per contract is around a second, meaning that SOLC-VERIFY is applicable and effective for a significant amount of real-world contracts, but scalability might depend on the complexity of the properties.

5.2 Unannotated Contracts

The contracts available at Etherscan are *not annotated* and SOLC-VERIFY can only consider `assert` and `require` statements, and overflows as implicit

[7] http://csl.sri.com/users/dejan/contracts.tar.gz.

specification. Furthermore, the ground truth about the contracts (whether they are correct or not) is unknown. Nevertheless, we systematically selected a subset of the contracts and manually checked the results given by solc-verify.

We took all 3897 contracts that solc-verify could translate and process with z3 in integer mode. At the first glance we discovered that a majority of the contracts (2754) use the popular SafeMath library [17], which has just recently adopted the proper usage of `assert` and `require`.[8] We updated these contracts to properly guard against user input with `require` (instead of `assert`). Afterwards, we checked for *assertion failures* and *overflows* using solc-verify.

Assertion Checking. Surprisingly, only 88 contracts (out of the 3897) contain assertions. solc-verify reported an error for 80 contracts, which we all checked manually. Out of those errors, 78 are clearly false alarms caused by a bad specification – the developer wrote `assert` where `require` should have been used – and fit into one of the following categories:

- Enforcing input validity with assertion (e.g., input arrays are of equal size).
- Enforcing time locks with an assertion (e.g., `now > 100`).
- Enforcing success of functions calls with an assertion (e.g. `addr.call()`).
- Enforcing permissions with an assertion (e.g., checking `msg.sender`).
- Enforcing correct result of arithmetic operations with an assertion.

As described in the Solidity documentation [38] `assert` should only be used to check for internal errors and invariants, and all cases highlighted above should use `require` instead. After replacing the spurious assertions with `require`, solc-verify reports no false alarms.

The 2 reported errors worth discussing in more detail are illustrated in Fig. 6. The example on the left is a pre-sale contract that accepts Ether until a sale cap is reached. The invariant of the contract, i.e. that `(raised <= max)` is enforced with a (stronger) assertion at the beginning of function. It could be argued that this fits within the mentioned prescribed usage for the `assert` construct. However, as solc-verify performs modular analysis, and nothing is assumed about the state before a function call, it will report such an assertion as a potential error. To fix this, the invariant `(raised <= max)` should be specified as a contract invariant, and `require` should be used to check the stronger precondition at function entry (followed by an `assert` at the end of the function).

The example on the right is a token transfer function. The function checks whether the sender has enough balance, and then it transfers the tokens to the recipient. Finally, the assertion checks that no overflow has occurred using an `assert` statement on the result of the addition. As is, solc-verify reports an error because increasing the balance of the recipient might overflow. As argued above, if the purpose of the assertion is to guard against overflows `require` should be used instead. On the other hand, one could argue that for fixed-cap tokens such an overflow should never occur since no address can hold enough

[8] For discussion, see https://github.com/OpenZeppelin/openzeppelin-solidity/issues/1120.

```
uint max    = 1000 ether;
uint raised = 0;

function() payable {
  assert(raised < max);
  require(msg.value != 0);
  require(raised + msg.value <= max);
  raised += msg.value;
}
```

```
mapping (address => uint) balances;

function transfer(address to, uint val) {
  require(balances[msg.sender] >= val);
  require(msg.sender != to);
  balances[msg.sender] = balances[msg.sender] - val;
  balances[to] = balances[to] + val;
  assert(balances[to] >= val);
}
```

Fig. 6. Examples of failing assertions reported by SOLC-VERIFY.

tokens to trigger the overflow. This assumption can be explicitly specified, i.e., by stating a contract invariant `sum(balances) <= cap`. With this invariant, SOLC-VERIFY avoids the false alarm by inferring that overflow is no longer possible.

Overflow Checking. We also checked for overflows and manually verified the results for the 68 contracts (out of 3897) that have at least 100 transactions. SOLC-VERIFY reports 33 alarms of which 29 are false and 4 can be considered as real. All false alarms are due to implicit assumptions on the magnitude of used numbers. There are 20 false alarms due to missing range assumptions for array lengths causing false overflow alarms for loop counters. For example, in a loop `for (uint i = 0; i < array.length; i++) {}` SOLC-VERIFY reports that `i++` might overflow. It is reasonable to assume that array lengths remain small due to the gas costs associated with growing an array. Other false alarms are caused by implicit assumptions on Ether balances or time. For example, it is assumed that a counter for the total amount of Ether received by a contract, or multiplying `msg.value` by 20000 cannot overflow because the amount of Ether is limited. Similarly, adding days or even weeks to the current timestamp will not overflow any time soon. We plan to include such implicit assumptions to a limited extent but, in general, it is best if the developer explicitly specifies them. The four issues found that could be considered real are the following:

– A pre-sale contract sets the `hardCap` in its constructor based on a `cap` provided as argument with `hardCap = cap*(10**18)`. Although the constructor is only called once by the deployer, providing a large cap can result in an unintentional overflow.
– A crowd-sale contract sets the unit cost based on the argument `perEther` by calculating `unitCost = 1 ether / (perEther*10**8)`. The problematic function is guarded so that it can only be called by the contract owner. Nevertheless, overflow can happen and can lead to an inconsistent unit price.
– A utility contract for mass distribution of tokens has a function to transfer an array of values to an array of recipients as a batch. The total amount transferred is kept accumulated in a contract counter and can overflow. However, as the counter is not used otherwise, the overflow might be benign.
– A food store contract first calculates the cost based on the bundles ordered, by computing `cost = bundles * price`, where `bundles` is provided by the caller. The function then checks if `msg.value >= cost` holds, but this check can be bypassed with the overflow, opening the door for a potential exploit.

```
1  library SafeMath {
2    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
3      uint256 c = a * b;
4      require(a == 0 || c / a == b);
5      return c;
6    }
7    // Similar for add, sub, div
8  }
9
10 /** @notice invariant totalSupply == sum(balances) */
11 contract BecToken {
12   using SafeMath for uint256;
13
14   uint256 public totalSupply;
15   mapping(address => uint256) balances;
16
17   function batchTransfer(address[] _receivers, uint256 _value) public returns (bool) {
18     uint cnt = _receivers.length;
19     uint256 amount = uint256(cnt) * _value; // Overflow
20     // uint256 amount = uint256(cnt).mul(_value); // Correct version
21     require(cnt > 0 && cnt <= 20);
22     require(_value > 0 && balances[msg.sender] >= amount);
23     balances[msg.sender] = balances[msg.sender].sub(amount);
24     /** @notice invariant totalSupply == sum(balances) + (cnt - i) * _value
25         @notice invariant (i <= cnt) */
26     for (uint i = 0; i < cnt; i++) {
27       balances[_receivers[i]] = balances[_receivers[i]].add(_value);
28     }
29     return true;
30   }
31 }
```

Fig. 7. Annotated part of the BECToken contract relevant for the "batchOverflow" bug [36]. While the contract uses the `SafeMath` library for most of its operations, there is a multiplication in line 19 that can overflow.

5.3 Annotated Contracts

While SOLC-VERIFY can find violations to implicit specifications in unannotated contracts, its main target is to allow developers to check custom, high-level properties by the means of annotations. We demonstrate this by annotating two contracts, finding bugs and proving the correctness of the fixed versions.

Reentrancy Detection (DAO). Reentrancy is a common source of vulnerabilities and the cause of the infamous DAO bug [16]. As explained in Sect. 2, the `SimpleBank` contract presented in Fig. 1 suffers from the same reentrancy bug. Using SOLC-VERIFY, the developer can specify the consistency of the bank contract state through a contract-level invariant, and SOLC-VERIFY can detect the bug. For example, we can annotate the contract with a property `sum(balances) == this.balance`. As the balance of the contract is deducted before the external call, the contract invariant is violated and SOLC-VERIFY reports a (real) error. However, if the user fixes the issue by first reducing the balance of the recipient in the mapping and then transferring the amount, the invariant will hold before making the external call and SOLC-VERIFY proves the specification successfully. For both the buggy and correct versions of the contract, the verification with SOLC-VERIFY is instant.

Overflow Detection (BEC Token). We now consider the BEC token vulnerability [36] that has been exploited and resulted in significant financial losses. The relevant part of the contract is shown in Fig. 7. The contract is a typical token contract, tracking balances of users in terms of their BEC tokens and allowing transfers of tokens between users. The function `batchTransfer` shown in the figure is intended to be used for transferring some value of BEC tokens to a group of recipients in a batch. To do so, the contract multiplies the requested value with the number of recipients. Unfortunately, this multiplication can result in an overflow (line 19), causing the total transfer amount to be invalid (e.g., 0). This allows attackers to "print" large amounts of tokens and send them to other users, while keeping their own balance constant. Running SOLC-VERIFY with the modular encoding of arithmetic successfully detects the overflow issue of BEC token and does not report any other potential overflows. After fixing the contract (line 20), SOLC-VERIFY shows that no overflows are possible. We also annotated the BEC contract with a specification that the contract maintains the correct token balances throughout the operation. As before, we add the invariant `totalSupply == sum(balances)` to the contract, and adapt it to the loop invariant. The loop invariant introduces extra complexity as it involves nonlinear arithmetic and illustrates the need for precise reasoning at large bit-sizes. Running SOLC-VERIFY on the annotated contract in the bitvector mode does not terminate regardless of the SMT solver used.[9] On the other hand, using modular arithmetic with overflow detection SOLC-VERIFY discharges all VCs (with 256-bit integers) in seconds for both the buggy and correct version of the contract (with CVC4).

Other Tools. As far as we know, SOLC-VERIFY is the only available tool that can reason effectively and precisely about Solidity code with specifications. The Solidity compiler includes an experimental SMT checker [2], which is currently limited to basic require/assert and overflow checking. For the BEC token the latest version (v0.5.10) reports every arithmetic operation as a potential overflow, including all false alarms in the `SafeMath` library. It cannot detect the reentrancy issue in the SimpleBank example because external calls and the `revert` function is not supported. Furthermore, it incorrectly reports that the condition for revert is always true (possibly because `call` is skipped and the default return value is false). ZEUS [26] is not available publicly for comparison.[10] VERISOL [27] does not support libraries (like `SafeMath`) or the `call` function, which can cause reentrancy so we could not apply it to our examples.

Two notable static analysis tools are MYTHRIL [13] and SLITHER [20]. MYTHRIL (v0.20.0) correctly reports the overflow issue with the BEC token in 200s, but it also reports all spurious overflows. MYTHRIL detects the reentrancy issue with the bank contract, but it also reports the same issue with the corrected version of the contract. SLITHER (v0.5.2), on the other hand, has a dedicated DAO-like reentrancy issue check and correctly handles both the buggy and correct version of the bank contract. However, SLITHER doesn't support overflow checking and therefore doesn't detect the BEC token issue.

[9] With bit-size of 16 bits, z3 can discharge the VCs in 2295 s while other solvers do not terminate.

[10] We could only obtain a spreadsheet of results from the authors.

Our goal, as demonstrated by the annotated examples, is to provide a tool that allows developers to check their own high-level annotations and business logic properties. This makes SOLC-VERIFY a good complementary to other automated verification tools that mainly target well known vulnerability patterns.

6 Related Work

The popularity of blockchain technology and many high-profile attacks and vulnerabilities have put focus on the need for formal verification for smart contracts [4,23,32]. We mention prominent advances relying on vulnerability patterns, theorem provers, finite automata and SMT, and relate them to our work.

Vulnerability Pattern-Based Approaches. Bhargavan et al. [8] translate a fragment of Solidity and EVM to F* and use its type and effect system to check for vulnerable patterns and gas boundedness. Grishchenko et al. [22] extend this work on EVM by checking security properties such as call integrity, atomicity, and independence from miner controlled parameters. SECURIFY [40] decompiles EVM and infers data- and control-flow dependencies in Datalog to check for compliance and violation patterns. OYENTE [29] is a symbolic execution tool that can check various patterns, including transaction ordering dependency, timestamp dependency, mishandled exceptions and reentrancy. MAIAN [35] uses symbolic analysis with concrete validation over a sequence of invocations to detect fund locking, fund leaking and contracts that can be killed. MYTHRIL [13] uses symbolic analysis to detect a variety of security vulnerabilities. SLITHER [20] is a static analysis framework with dedicated vulnerability checkers. Approaches based on vulnerability patterns, as the ones mentioned above, can be effective at discharging specific properties, but are limited to built-in patterns (or a domain specific language [40]). Furthermore, as they are mainly EVM-based it makes reasoning about more general properties difficult. Our approach focuses on Solidity and allows high-level, user-defined properties to be checked effectively.

Theorem Prover-Based Approaches. KEVM [24] is an executable formal semantics of EVM based on the K framework including a deductive program verifier to check contracts against given specifications. Hirai [25] formalizes EVM in Lem, a language used by various theorem provers and proves properties using interactive theorem proving. Scilla [37] is an intermediate language between smart contracts and bytecode, using the Coq proof assistant for reasoning. Theorem prover-based approaches offer the ability to capture precise, formal semantics of the contracts but can be cumbersome as properties also need to be formalized in the language of the theorem prover. Moreover, user interaction and assistance is usually required impeding usability for contract developers.[11] In our approach the developer can specify the properties directly within the contract, as Solidity annotations and modular verification is fully automated. Although loop invariants might be required, complex loops are rare in contracts.

[11] For an example of the difficulties in manually analyzing even trivial issues, see https://runtimeverification.com/blog/erc-20-verification/.

Automata-Based Approaches. FSolidM [30] is a finite state machine-based designer for smart contracts that can generate Solidity code. Security features and design patterns (e.g., locking, access control) can be included in the state machine. Abdellatif and Brousmiche [1] model contracts and the blockchain manually in BIP and use statistical model checking to simulate uncertainties in the environment. Such model-based approaches are orthogonal to our approach, as we are working on the source code directly. This has the advantage that the developer does not need to learn a new (modeling) language and an extra step of transformation (from model to source) is eliminated.

SMT-Based Approaches. Zeus [26] translates Solidity to LLVM bitcode and employs existing verifiers such as SeaHorn and Smack. Besides certain vulnerability patterns, it claims to have support for user-defined properties to some extent. However, it is not publicly available for comparison. VeriSol [27] checks for conformance between workflow policies and smart contract implementations on the Azure blockchain. While the core of their method is a translation to Boogie (similar to ours), it targets a specific problem limited in scope and does not yet support features needed for typical smart contracts (see Sect. 5.3). The Solidity compiler itself also includes a built-in experimental SMT checker [2], which executes the body of each function symbolically and checks for implicit specifications, such as assertion failures, dead code and overflows. Their approach is however, limited, by false overflow alarms and missing features (e.g., `call`, `revert`). Furthermore, it has no support for developer-supplied specification beyond `require` and `assert` statements. Some of the challenges they mention in their future work are solved by our approach, including contract level invariants and the reduced number of false overflow alarms.

7 Conclusion

We presented solc-verify, a tool for automated verification of Solidity smart contracts based on modular program reasoning and SMT solvers. Working at the source level, solc-verify allows users to specify high-level properties such as contract invariants, loop invariants, pre- and post-conditions and assertions. solc-verify then discharges verification conditions with SMT solvers to verify contract properties in a modular and scalable way. The approach offers precise and scalable, yet automated and user-friendly formal verification for Solidity smart contracts. solc-verify can already be used on real-world contracts and can effectively find bugs and prove correctness of non-trivial properties with minimal user effort.

References

1. Abdellatif, T., Brousmiche, K.: Formal verification of smart contracts based on users and blockchain behaviors models. In: 9th IFIP International Conference on New Technologies, Mobility and Security, pp. 1–5. IEEE (2018)

2. Alt, L., Reitwiessner, C.: SMT-based verification of solidity smart contracts. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 376–388. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_28
3. Antonopoulos, A., Wood, G.: Mastering Ethereum: Building Smart Contracts and DApps. O'Reilly Media, Inc., Sebastopol (2018)
4. 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
5. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. J. Object Technol. **3**(6), 27–56 (2004)
6. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
7. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 305–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
8. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: ACM Workshop on Programming Languages and Analysis for Security, pp. 91–96. ACM (2016)
9. Bornat, R.: Proving pointer programs in hoare logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000). https://doi.org/10.1007/10722010_8
10. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamarić, Z.: A reachability predicate for analyzing low-level software. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 19–33. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_4
11. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2
12. ConsenSys: Ethereum smart contract security best practices (2018). https://consensys.github.io/smart-contract-best-practices/
13. ConsenSys: Mythril classic: security analysis tool for Ethereum smart contracts (2019). https://github.com/ConsenSys/mythril-classic
14. De Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: Formal Methods in Computer-Aided Design, pp. 45–52. IEEE (2009)
15. DeLine, R., Leino, K.R.M.: BoogiePL: a typed procedural language for checking object-oriented programs. Technical report MSR-TR-2005-70, Microsoft Research (2005)
16. Dhillon, V., Metcalf, D., Hooper, M.: The DAO hacked. In: Dhillon, V., Metcalf, D., Hooper, M. (eds.) Blockchain Enabled Applications, pp. 67–78. Apress, Berkeley (2017). https://doi.org/10.1007/978-1-4842-3081-7_6
17. Dourlens, J.: Safemath to protect from overflows (2017). https://ethereumdev.io/safemath-protect-overflows/
18. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
19. Ethereum Constantinople/St. Petersburg upgrade announcement (2019). https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement/

20. Feist, J., Greico, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, pp. 8–15. IEEE (2019)
21. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: ACM SIGPLAN 2002 conference on Programming Language Design and Implementation, pp. 234–245. ACM (2002)
22. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 243–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_10
23. Harz, D., Knottenbelt, W.: Towards safer smart contracts: a survey of languages and verification methods (2018). http://arxiv.org/abs/1809.09805
24. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Rosu, G.: KEVM: a complete semantics of the Ethereum virtual machine. Technical report, IDEALS (2017)
25. 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
26. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: Network and Distributed Systems Security Symposium (2018)
27. Lahiri, S.K., Chen, S., Wang, Y., Dillig, I.: Formal specification and verification of smart contracts for Azure blockchain (2018). http://arxiv.org/abs/1812.08829
28. Leino, K.R.M.: This is Boogie 2 (2008)
29. 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)
30. Mavridou, A., Laszka, A.: Tool demonstration: FSolidM for designing secure Ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 270–277. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_11
31. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress, pp. 21–28 (1962)
32. Miller, A., Cai, Z., Jha, S.: Smart contracts and opportunities for formal methods. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 280–299. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_22
33. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
34. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008). http://www.bitcoin.org/bitcoin.pdf
35. Nikolić, 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, pp. 653–663. ACM (2018)
36. NIST National Vulnerability Database: CVE-2018-10299: Beauty Ecosystem Coin (BEC) issue (2018). https://nvd.nist.gov/vuln/detail/CVE-2018-10299
37. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language (2018). http://arxiv.org/abs/1801.00687
38. Solidity documentation (2018). https://solidity.readthedocs.io/en/v0.4.25/
39. Szabo, N.: Smart contracts (1994)

40. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67–82. ACM (2018)
41. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2017). https://ethereum.github.io/yellowpaper/paper.pdf