Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE

Scott Wesley[1], Maria Christakis[2], Jorge A. Navas[3], Richard Trefler[1], Valentin Wüstholz[4], and Arie Gurfinkel[1](✉)

[1] University of Waterloo, Waterloo, Canada
ar627383@dal.ca, arie.gurfinkel@uwaterloo.ca
[2] MPI-SWS, Kaiserslautern and Saarbrücken, Germany
[3] SRI International, Menlo Park, USA
[4] ConsenSys, Kaiserslautern, Germany

Abstract. Solidity *smart contract* allow developers to formalize financial agreements between users. Due to their monetary nature, smart contracts have been the target of many high-profile attacks. Brute-force verification of smart contracts that maintain data for up to 2^{160} users is intractable. In this paper, we present SmartACE, an automated framework for smart contract verification. To ameliorate the state explosion induced by large numbers of users, SmartACE implements *local bundle abstractions* that reduce verification from arbitrarily many users to a few *representative* users. To uncover deep bugs spanning multiple transactions, SmartACE employs a variety of techniques such as model checking, fuzzing, and symbolic execution. To illustrate the effectiveness of SmartACE, we verify several contracts from the popular OpenZeppelin library: an access-control policy and an escrow service. For each contract, we provide specifications in the Scribble language and apply fault injection to validate each specification. We report on our experience integrating Scribble with SmartACE, and describe the performance of SmartACE on each specification.

1 Introduction

Smart contracts are a trustless mechanism to enforce financial agreements between many users [46]. The Ethereum blockchain [52] is a popular platform for smart contract development, with most smart contracts written in Solidity. Due to their monetary nature, smart contracts have been the target of many high-profile attacks [14]. Formal verification is a promising technique to ensure the correctness of deployed contracts. However, Solidity smart contracts can

This work was supported, in part, by Individual Discovery Grants from the Natural Sciences and Engineering Research Council of Canada, and a Ripple Fellowship. Jorge A. Navas was supported by NSF grant 1816936.

© Springer Nature Switzerland AG 2022

B. Finkbeiner and T. Wies (Eds.): VMCAI 2022, LNCS 13182, pp. 425–449, 2022.
https://doi.org/10.1007/978-3-030-94583-1_21

```
 1 contract Auction {
 2   mapping(address => uint) bids;
 3   address manager;
 4   uint leadingBid;
 5   bool stopped;
 6   uint _sum;
 7
 8   modifier canParticipate() {
 9     require(msg.sender != manager);
10     require(!stopped);
11     _;
12   }
13   constructor(address _m) public { manager = _m; }
14   function () external payable { bid(); }
15   function bid() public payable canParticipate() {
16     require(msg.value > leadingBid);
17     _sum = _sum + msg.value - bids[msg.sender];
18     bids[msg.sender] = msg.value;
19     leadingBid = msg.value;
20   }
21   function withdraw() public canParticipate() {
22     require(bids[msg.sender] != leadingBid);
23     _sum = _sum + 0 - bids[msg.sender];
24     bids[msg.sender] = 0;
25   }
26   function stop() public {
27     require(msg.sender == manager);
28     stopped = true;
29   }
30 }
31 contract Mgr {
32   Auction auction;
33
34   constructor() public {
35     auction = new Auction(address(this));
36   }
37   function stop() public { auction.stop(); }
38 }
39
40 contract TimedMgr is Mgr {
41   event Stopped(address _by, uint _block);
42   uint start;
43   uint dur;
44
45   constructor(uint _d) public {
46     start = block.number;
47     dur = _d;
48   }
49   function stop() public {
50     require(start + dur < block.number);
51     emit Stopped(msg.sender, block.number);
52     super.stop();
53   }
54   function check() public returns (bool, uint) {
55     if (start + dur < block.number) {
56       return (false, block.number - dur - start);
57     }
58     else { return (true, 0); }
59   }
60 }
```

Fig. 1. A smart contract that implements a simple auction.

address and maintain data for up to 2^{160} users. Analyzing smart contracts with this many users is intractable in general, and calls for specialized techniques [51].

In this paper, we present SmartACE, an automated framework for smart contract verification. SmartACE takes as input a smart contract annotated with assertions, then checks that all assertions hold. This is in contrast to tools that check general patterns on unannotated smart contracts, such as absence of integer overflows (e.g., [47]), or access control policies (e.g., [10]). To ameliorate the state explosion induced by large numbers of users, SmartACE implements *local bundle abstractions* [51] to reduce verification from arbitrarily many users to a few *representative* users. SmartACE targets deep violations, that require multiple transactions to observe, using a variety of techniques such as model checking, fuzzing, and symbolic execution. To avoid reinventing the wheel, SmartACE models each contract in LLVM-IR [33] to integrate off-the-shelf analyzers such as SeaHorn [21], libFuzzer [34], and Klee [11].

As an example of the local bundle abstraction, consider `Auction` in Fig. 1. In `Auction`, each user starts with a bid of zero. Users alternate, and submit increasingly larger bids, until a designated manager stops the auction. While the auction is not stopped, a non-leading user may withdraw their bid. To ensure that the auction is fair, a manager is not allowed to place their own bid. Furthermore, the role of the manager is never assigned to the *zero-account* (i.e., the null user at address 0). It follows that `Auction` satisfies property **A0**: "*All bids are less than or equal to the recorded leading bid.*"

In general, `Auction` can interact with up to 2^{160} users. However, each transaction of `Auction` interacts with at most the zero-account, the auction itself, the manager, and an arbitrary sender. Furthermore, all arbitrary senders are interchangeable with respect to **A0**. For example, if there are exactly three active bids $\{2, 4, 8\}$ then **A0** can be verified without knowing which user placed which

```
1  // Initialize blockchain state.
2  block.number = *;
3  Auction _a = new Auction(address(2));
4  _a.address = address(1);
5  // Transaction loop.
6  while (true) {
7    // Apply interference.
8    _a.bids[address(3)] = *;
9    require(_a.bids[address(3)] <= _a.leadingBid);
10   // Update blockchain state.
11   block.number += *;
12   // Generate transaction.
13   uint method = *; msg.sender = *; msg.value = *;
14   require(msg.sender > address(1));
15   require(msg.sender < address(4));
16   require(value == 0 || method == 0);
17   // Execute transaction.
18   if (method == 0) _a.bid();
19   if (method == 1) _a.withdraw();
20   if (method == 2) _a.stop();
21 }
```

Fig. 2. A simplified test harness for `Auction` of Fig. 1

bid. This is because the leading bid is always 8, and each bid is at most 8. Due to these symmetries between senders, it is sufficient to verify `Auction` relative to a representative user from each class (i.e., the zero account, the auction itself, the manager, and an arbitrary sender), rather than all 2^{160} users. The key idea is that each representative user corresponds to one or *many* concrete users [40].

If a representative's class contains a single concrete user, then there is no difference between the concrete user and the representative user. For example, the zero-account and the auction each correspond to single concrete users. Similarly, the manager refers to a single concrete user, so long as the `manager` variable does not change. Therefore, the addresses of these users, and in turn, their bids, are known with absolute certainty. On the other hand, there are many arbitrary senders. Since `Auction` only compares addresses by equality, the exact address of the representative sender is unimportant. What matters is that the representative sender does not share an address with the zero-account, the auction, nor the manager. However, this means that at the start of each transaction the location of the representative sender is not absolute, and, therefore, the sender has a range of possible bids. To account for this, we introduce a predicate, called an *interference invariant*, to summarize the bid of each sender. An example interference invariant for `Auction` is **A0** itself.

Given an interference invariant, **A0** can be verified by SeaHorn. To do this, the concrete users in `Auction` must be abstracted by representative users. The abstract system (see Fig. 2), known as a *local bundle abstraction*, assigns the zero-account to address 0, the auction to address 1, the manager to address 2, the representative sender to address 3, and then executes an unbounded sequence of transactions (all feasible sequences are included). Before each transaction, the sender's bid is set to a nondeterministic value that satisfies its interference invariant. If the abstract system and **A0** are provided to SeaHorn, then SeaHorn verifies that all states reachable in the abstract system satisfy **A0**. It then follows from the symmetries between senders that **A0** holds for any number of users.

Prior work has demonstrated SmartACE to be competitive with state-of-the-art smart contract verifiers [51]. This paper illustrates the effectiveness of SmartACE by verifying several contracts from the popular OpenZeppelin library. For each contract, we provide specifications in the Scribble language. We report on our experience integrating Scribble with SmartACE, and describe the performance of SmartACE on each specification. As opposed to

other case studies (e.g., [2,10,16,17,23–26,30,32,35,37,45,47–49]), we do not apply SmartACE to contracts scraped from the blockchain. As outlined by the methodology of [16], such studies are not appropriate for tools that require annotated contracts. Furthermore, it is shown in [23] that most contracts on the blockchain are unannotated, and those with annotations are often incorrect. For these reasons, we restrict our case studies to manually annotated contracts.

This paper makes the following contributions: (1) the design and implementation of an efficient Solidity smart contract verifier SmartACE, that is available at https://github.com/contract-ace/smartace; (2) a methodology for automatic verification of deep properties of smart contracts, including aggregate properties involving sum and maximum; and (3) a case-study in verification of two OpenZeppelin contracts, and an open-bid auction contract, that are available at https://github.com/contract-ace/verify-openzeppelin.

The rest of this paper is structured as follows. Section 2 presents the high-level architecture of SmartACE. Section 3 describes the conversion from a smart contract to an abstract model. Section 4 describes challenges and benefits in integrating SmartACE with off-the-shelf analyzers. Section 5 reports on a case study that uses SmartACE and Scribble to verify several OpenZeppelin contracts. The performance of SmartACE, and the challenges of integrating with Scribble, are both discussed.

2 Architecture and Design Principles of SmartACE

SmartACE is a smart contract analysis framework guided by communication patterns. As opposed to other tools, SmartACE performs all analysis against a local bundle abstraction for a provided smart contract. The abstraction is obtained through source-to-source translation from Solidity to a *harness* modelled in LLVM-IR. The design of SmartACE is guided by four principles.

1. **Reusability:** The framework should support state-of-the-art and off-the-shelf analyzers to minimize the risk of incorrect analysis results.
2. **Reciprocity:** The framework should produce intermediate artifacts that can be used as benchmarks for off-the-shelf analyzers.
3. **Extensibility:** The framework should extend to new analyzers without modifying existing features.
4. **Testability:** The intermediate artifacts produced by the framework should be executable, to support both validation and interpretation of results.

These principles are achieved through the architecture in Fig. 3. SmartACE takes as input a smart contract with Scribble annotations (e.g., contract invariants and function postconditions), and optionally an *interference invariant*. Scribble processes the annotated smart contract and produces a smart contract with assertions. The smart contract with assertions and the interference invariant are then passed to a source-to-source translator, to obtain a model of the smart contract and its environment in LLVM-IR (see Sect. 3). This model is called a *harness*. Harnesses use an interface called libVerify to integrate with

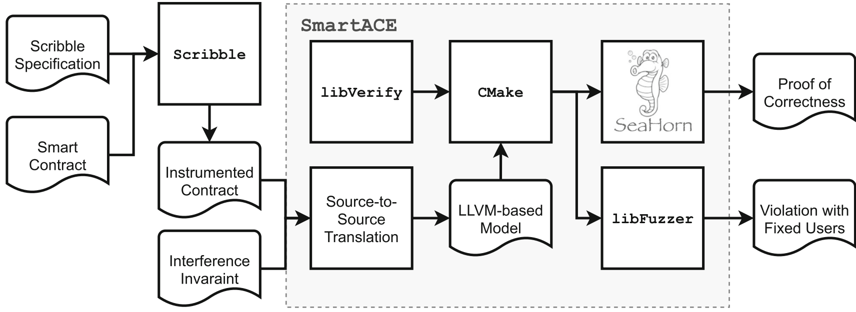

Fig. 3. The architecture of SMARTACE for integration with SEAHORN for model checking and LIBFUZZER for greybox fuzzing.

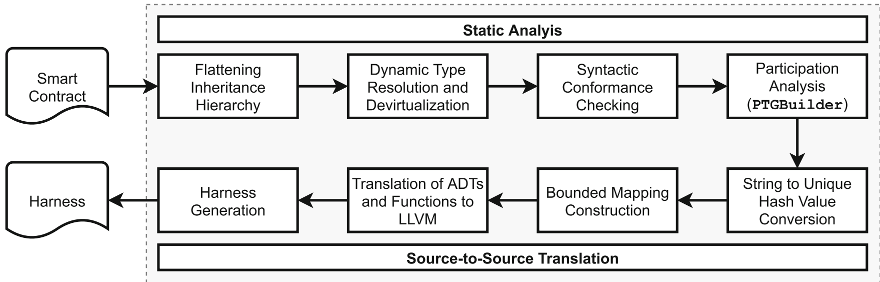

Fig. 4. The analysis and transformations performed by SMARTACE.

arbitrary analyzers, and are therefore analyzer-agnostic (see Sect. 4). When an analyzer is chosen, CMAKE is used to automatically compile the harness, the analyzer, and its dependencies, into an executable program. Analysis results for the program are returned by SMARTACE.

The SMARTACE architecture achieves its guiding principles as follows. To ensure *reusability*, SMARTACE uses state-of-the-art tools for contract instrumentation (SCRIBBLE), build automation (CMAKE), and program analysis (e.g., SEAHORN and LIBFUZZER). The source-to-source translation is based on the SOLIDITY compiler to utilize existing source-code analysis (e.g., AST construction, type resolution). To ensure *reciprocity*, the SMARTACE architecture integrates third-party tools entirely through intermediate artifacts. In our experience, these artifacts have provided useful feedback for SEAHORN development. To ensure *extensibility*, the LIBVERIFY interface is used together with CMAKE build scripts to orchestrate smart contract analysis. A new analyzer can be added to SMARTACE by first creating a new implementation of LIBVERIFY, and then adding a build target to the CMAKE build scripts. Finally, *testability* is achieved by ensuring all harnesses are executable. As shown in Sect. 4, executable harnesses provide many benefits, such as validating counterexamples from model checkers, and manually inspecting harness behaviour.

3 Contract Modelling

This section describes the translation from a smart contract with annotations, to a harness in LLVM-IR. A high-level overview is provided by Fig. 4. First, static analysis is applied to a smart contract, such as resolving inheritance and over-approximating user participation (see Sect. 3.1). Next, the analysis results are used to convert each `contract` to LLVM structures and functions (see Sect. 3.2). Finally, these functions are combined into a harness that schedules an unbounded sequence of smart contract transactions (see Sect. 3.3).

3.1 Static Analysis

The static analysis in SmartACE is illustrated by the top row of Fig. 4. At a high-level, static analysis ensures that a bundle conforms to the restrictions of [51], and extracts facts about the bundle required during the source-to-source translation. Bundle facts include a flat inheritance hierarchy [5], the dynamic type of each contract-typed variable, the devirtualization of each call (e.g., [4]), and the representative users (*participants*) of the bundle. Key design considerations in the analysis follow.

Reducing Code Surface. SmartACE over-approximates conformance checks through syntactic rules. Therefore, it is possible for SmartACE to reject valid smart contracts due to inaccuracies. For this reason, SmartACE uses incremental passes to restrict the code surface that reaches the conformance checker. The first pass flattens the inheritance hierarchy by duplicating member variables and specializing methods. The second pass resolves the dynamic type of each contract-typed variable, by identifying its allocation sites. For example, the dynamic type for state variable `auction` in `TimedMgr` of Fig. 1 is `Auction` due to the allocation on line 35. The third pass uses the dynamic type of each contract-typed variable, to resolve all virtual calls in the smart contract. For example, `super.stop` at line 52 devirtualizes to method `stop` of contract `Mgr`. The fourth pass constructs a call graph for the `public` and `external` methods of each smart contract. Only methods in the call graph are subject to the conformance checker.

Conformance Checking. The syntactic conformance check follows from [51] and places the following restrictions: (1) There is no inline assembly; (2) Mapping indices are addresses; (3) Mapping values are numeric; (4) Address comparisons must be (dis)equality; (5) Addresses never appear in arithmetic operations; (6) Each contract-typed variable corresponds to a single call to `new`.

Participation Analysis. A key step in local analysis is to identify a set of representative users. A representative user corresponds to one or arbitrarily many concrete users. In the case of one concrete user, the corresponding address is either *static* or *dynamic* (changes between transactions). Classifying representative users according to this criterion is critical for local analysis. A write of v to abstract location l is said to be *strong* if v replaces the value at l, and *weak* if v

is added to a set of values at locations referenced by l. It follows that a write to many concrete users is weak, whereas a write to a single concrete user is strong. Furthermore, if the address of the single concrete user is dynamic, then aliasing between representative users can occur. A representative user with weak updates is an *explicit participant*, as weak updates result from passing arbitrary users as inputs to transactions (e.g., `msg.sender`). A representative user with strong updates and a dynamic address is a *transient participant*, as dynamic addresses are maintained via roles, and may change throughout execution (e.g., `manager`). A representative user with strong updates and a static address is an *implicit participant*, as static addresses are determined by the source text of a contract, independent of transaction inputs and roles (e.g., the zero account). SmartACE implements the `PTGBuilder` algorithm from [51] that uses an intraprocedural taint analysis to over-approximate the maximum number of *explicit*, *transient*, and *implicit* participants. Recall that taint analysis [28] determines whether certain variables, called *tainted sources*, influence certain expressions, called *sinks*. In `PTGBuilder`, tainted sources are (a) input address variables, (b) state address variables, and (c) literal addresses, while sinks are (a) memory writes, (b) comparisons, and (c) mapping accesses. An input address variable, `v`, that taints at least one sink is an *explicit* participant[1]. Similarly, state address variables and literal addresses that taint sinks represent *transient* and *implicit* participants, respectively. For example, `PTGBuilder` on Fig. 1, computes 2 explicit participants due to `msg.sender` and `_m` in the constructor of `Auction`, 1 transient participant due to `manager` in `Auction`, and 3 implicit participants due to the addresses of the zero-account, `Auction`, and `TimedMgr`. This over-approximates true participation in several ways. For example, the constructor of `Auction` is never influenced by the equality of `msg.sender` and `_m`, and `TimedMgr` is always the manager of `Auction`.

3.2 Source-to-Source Translation

Source-to-source translation relies on the call graph and participants obtained through static analysis. The translation is illustrated by the bottom row of Fig. 4. A translation for Fig. 1 is given in Fig. 5. Note that the C language is used in Fig. 5, rather than LLVM-IR, as C is more human-readable.

Abstract Data Types (ADTs). An ADT is either a **`struct`** or a **`contract`**. Each **`struct`** is translated directly to an LLVM structure. The name of the structure is prefixed by the name of its containing **`contract`** to avoid name collisions. Each **`contract`** is translated to an LLVM structure of the same name, with a field for its address (`model_address`), a field for its balance (`model_balance`), and a field for each user-defined member variable. An example is given for `Auction` at line 3.

Primitive Types. Primitive types include all integer types, along with **`bool`**, **`address`**, and **`enum`** (unbounded arrays are not yet supported in SmartACE). Integer types are mapped to singleton structures, according to their signedness

[1] One exception is `msg.sender` which is always an explicit participant.

```
1 struct Map_1 { sol_uint256_t data_0; /* ... */ sol_uint256_t data_4; };
2 struct Auction {
3   sol_address_t model_address;  sol_uint256_t model_balance;
4   struct Map_1 user_bids;
5   sol_address_t user_manager;   sol_uint256_t user_leadingBid;
6   sol_bool_t user_stopped;      sol_uint256_t user___sum;
7 };
8 void TimedMgr_Method_stop(
9   struct TimedMgr *self, sol_address_t sndr, sol_uint256_t value,
10  sol_uint256_t bnum, sol_uint256_t time, sol_bool_t paid, sol_address_t orig) {
11  sol_require(self->user_start.v + self->user_dur.v < bnum.v, 0);
12  sol_emit("Stopped(msg.sender, block.number)");
13  Mgr_Method_For_TimedMgr_stop(self, /*...*/ time, Init_sol_bool_t(0), orig);
14 }
15 sol_bool_t TimedMgr_Method_check(
16  struct TimedMgr *self, /*...globals...*/, sol_uint256_t *out_1) {
17  (*out_1) = Init_sol_uint256_t(0);
18  if (self->user_start.v + self->user_dur.v < bnum.v) {
19    out_1->v = bnum.v - self->user_dur.v - self->user_start.v;
20    return Init_sol_bool_t(0);
21  }
22  return Init_sol_bool_t(1);
23 }
24 void Auction_Method_1_bid(struct Auction *self, /*...globals...*/) {
25  sol_require(value.v > self->user_leadingBid.v, 0);
26  Write_Map_1(&self->user_bids, sndr, value);
27  self->user_leadingBid = value;
28 }
29 void Auction_Method_bid(struct Auction *self, /*...globals...*/) {
30  if (paid.v == 1) self->model_balance.v += value.v;
31  sol_require(sndr.v != self->user_manager.v, 0);
32  sol_require(!self->user_stopped.v, 0);
33  Auction_Method_1_bid(self, /*...globals...*/);
34 }
```

Fig. 5. Partial modelling of the types and methods in Fig. 1 as C code (LLVM).

and bit-width. For example, the type of `leadingBid` is mapped to `sol_uint256_t` (see line 5). Each **`bool`** type is mapped to the singleton structure `sol_bool_t`, which contains the same underlying type as **`uint8`** (see line 6). Each **`address`** type is mapped to the singleton structure `sol_address_t`, which contains the same underlying type as **`uint160`** (see line 5). Each **`enum`** is treated as an unsigned integer of the nearest containing bit-width. Benefits of singleton structures, and their underlying types, are discussed in Sect. 4.

Functions. Methods and modifiers are translated to LLVM functions. Methods are specialized according to the flattened inheritance hierarchy, and modifiers are specialized to each method. To avoid name collisions, each function is renamed according to the contract that defines it, the contract that is calling it, and its position in the chain of modifiers. For example, the specialization of method `Mgr.stop` for `TimedMgr` is `Mgr_Method_For_TimedMgr_stop`. Likewise, the specializations of method `Auction.bid` and its modifier `canParticipate` are `Auction_Method_1_bid` and `Auction_Method_bid`, respectively. Extra arguments are added to each method to represent the current call state (see `self` through to `orig` on line 9). Specifically, `self` is **`this`**, `sndr` is **`msg.sender`**, `value` is **`msg.value`**, `bnum` is **`block.number`**, `time` is **`block.timestamp`**, and `orig` is **`msg.origin`**. A special argument, `paid`, indicates if **`msg.value`** has been added to a contract's balance (see line 13,

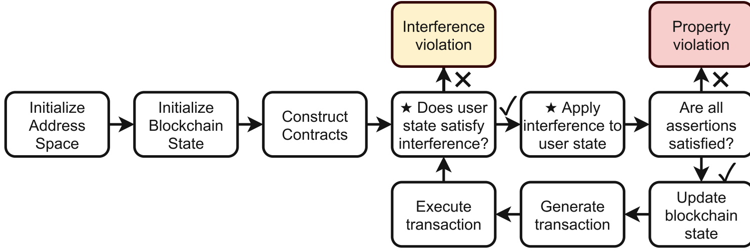

Fig. 6. The control-flow of a test harness. Each ⋆ denotes an optional step.

where `paid` is set to false). If `paid` is true, then the balance is updated before executing the body of the method (see line 30). Multiple return values are handled through the standard practice of output variables. For example, the argument `out_1` in `TimedMgr_Method_check` represents the second return value of `check`.

Statements and Expressions. Most expressions map directly from Solidity to LLVM (as both are typed imperative languages). Special cases are outlined. Each **`assert`** maps to `sol_assert` from libVerify, which causes a program failure given argument **`false`**. Each **`require`** maps to `sol_require` from libVerify, which reverts a transaction given argument **`false`** (see line 31). For each **`emit`** statement, the arguments of the event are expanded out, and then a call is made to `sol_emit` (see line 12). For each method call, the devirtualized call is obtained from the call graph, and the call state is propagated (see line 13 for the devirtualized called to **`super`**`.stop`). For external method calls, `paid` and **`msg.sender`** are reset.

Mappings. Each **`mapping`** is translated to an LLVM structure. This structure represents a bounded mapping with an entry for each participant of the contract. For example, if a contract has N participants, then a one-dimensional **`mapping`** will have N entries, and a two-dimensional **`mapping`** will have N^2 entries. Since **`mapping`** types are unnamed, the name of each LLVM structure is generated according to declaration order. For example, `bids` of `Auction` is the first mapping in Fig. 1, and translates to `Map_1` accordingly (see line 1). Accesses to `Map_1` are encapsulated by `Read_Map_1` and `Write_Map_1` (see line 26).

Strings. Each string literal is translated to a unique integer value. This model supports string equality, but disallows string manipulation. Note that string manipulation is hardly ever used in smart contracts due to high gas costs.

Addresses. Implicit participation is induced by literal addresses. This means that the value of a literal address is unimportant, so long as it is unique and constant. For reasons outlined in Sect. 3.3, it is important to set the value of each literal address programmatically. Therefore, each literal address is translated to a unique global variable. For example, **`address`**`(0)` translates to `g_literal_address_0`.

```
 1 sol_bool_t paid; paid.v = 1;
 2 // Address space initialization.
 3 struct TimedMgr sc_1;
 4 struct Auction *sc_2;
 5 sc_2 = &sc_1.user_auction;
 6 g_literal_address_0 = 0;
 7 sc_1.model_address.v = 1;
 8 sc_2.model_address.v = 2;
 9 // Blockchain initialization.
10 sol_uint256_t bnum;
11 bnum.v = ND_UINT(1,256,"bnum");
12 sol_uint256_t time;
13 time.v = ND_UINT(2,256,"time");
14 // Contract construction.
15 sol_address_t sndr;
16 sndr.v = ND_RANGE(3,3,5,"sndr");
17 sol_uint256_t value; value.v = 0;
18 sol_uint256_t arg___d;
19 arg___d.v = ND_UINT(4,256,"_d");
20 Init_TimedMgr(
21   &sc_1, sndr, value, bnum, time,
22   paid, sndr, arg___d);
23 // Transaction Loop.
24 while (sol_continue()) {
25   sol_on_transaction();
26   // Interference.
27   if (sol_can_interfere()) {/*...*/}
28   // Update blockchain state.
29   if (ND_RANGE(5,0,2,"inc_time")) {
30     bnum.v =
31       ND_INCREASE(6,bnum.v,1,"bnum");
32     time.v =
33       ND_INCREASE(7,time.v,1,"time");
34   }
35   // Generate transaction.
36   switch (ND_RANGE(8,0,6,"call")) {
37   case 0: {
38     /*...generate arguments...*/
39     TimedMgr_Method_stop(/*...*/);
40     break;
41   } /*...other public methods...*/
42   }
43 }
```

Fig. 7. The harness for Fig. 1. Logging is omitted to simplify the presentation.

3.3 Harness Design

A harness provides an entry-point for LLVM analyzers. Currently, SmartACE implements a single harness that models a blockchain from an arbitrary state, and then schedules an unbounded sequence of transactions for contracts in a bundle. A high-level overview of this harness is given in Fig. 6. The harness for `Auction` in Fig. 1 is depicted in Fig. 7.

Modelling Nondeterminism. All nondeterministic choices are resolved by interfaces from libVerify. `ND_INT(id,bits,msg)` and `ND_UINT(id,bits,msg)` choose integers of a desired signedness and bit-width. `ND_RANGE(id,lo,hi,msg)` chooses values between `lo` (inclusively) and `hi` (exclusively). `ND_INCREASE(id,old,msg)` chooses values larger than `old`. In all cases, `id` is an identifier for the call site, and `msg` is used for logging purposes.

Address Space. An abstract address space restricts the number of addresses in a harness. It assigns abstract address values to each contract address and literal address symbol. Assume that there are N contracts, M literal addresses, and K non-implicit participants. The corresponding harness has abstract addresses 0 to $(N+M+K-1)$. Constraints are placed on address assignments to prevent impossible address spaces, such as two literal addresses sharing the same value, two contracts sharing the same value, or a contract having the same value as the zero-account. The number of constraints must be minimized, to simplify symbolic analysis. In SmartACE, the following partitioning is used. `Address(0)` is always mapped to abstract address 0 (see line 6). Abstract addresses 1 to N are assigned to contracts according to declaration order (see lines 7–8). Literal addresses are assigned arbitrary values from 1 to $(N+M)$. This allows contracts to have literal

addresses. Disequality constraints ensure each assignment is unique. Senders are then chosen from the range of non-contract addresses (see line 16).

Blockchain Model. SmartACE models `block.number`, `block.timestamp`, `msg.value`, `msg.sender`, and `msg.origin`. The block number and timestamp are maintained across transactions by `bnum` at line 10 and `time` at line 12. Before transaction generation, `bnum` and `time` may be incremented in lockstep (see lines 29–33). Whenever a method is called, `msg.sender` is chosen from the non-contract addresses (e.g., line 16). The value of `msg.sender` is also used for `msg.origin` (e.g., the second argument on line 22). If a method is `payable`, then `msg.value` is chosen by `ND_UINT`, else `msg.value` is set to 0 (e.g., line 17).

Transaction Loop. Transactions are scheduled by the loop on line 24. The loop terminates if `sol_continue` from libVerify returns `false` (this does not happen for most analyzers). Upon entry to the loop, `sol_on_transaction` from libVerify provides a hook for analyzer-specific bookkeeping. Interference is then checked and re-applied, provided that `sol_can_interfere` returns `true` at line 27. A transaction is picked on line 36 by assigning a consecutive number to each valid method, and then choosing a number from this range. Arguments for the method are chosen using `ND_INT` and `ND_UINT` for integer types, and `ND_RANGE` for bounded types such as `address`, `bool` and `enum` (see lines 15–19 for an example).

Interference. A harness may be instrumented with interference invariants to enable modular reasoning. Interference invariants summarize the data of all concrete users abstracted by a representative user, relative to the scalar variables in a smart contract (e.g., `leadingBid`, `stopped`, and `_sum` in Fig. 1). An interference invariant must be true of all data initially, and maintained across each transaction, regardless of whether the representative user has participated or not. As illustrated in Fig. 6, interference is checked and then re-applied before executing each transaction. Note that checking interference after a transaction would be insufficient, as this would fail to check the initial state of each user. To apply interference, a harness chooses a new value for each mapping entry, and then assumes that these new values satisfy their interference invariants. To check interference, a harness chooses an arbitrary entry from a mapping, and asserts that the entry satisfies its interference invariant. Note that asserting each entry explicitly would challenge symbolic analyzers. For example, a two-dimensional mapping with 16 participants would require 256 assertions.

Limitations. The harness has three key limitations. First, as gas is unlimited, the possible transactions are over-approximated. Second, there is no guarantee that time must increase (i.e., a fairness constraint), so time-dependent actions may be postponed indefinitely. Third, reentrancy is not modeled [20], though this is sufficient for *effectively callback free* contracts as defined in [42].

4 Integration with Analyzers

CMake and libVerify are used to integrate SmartACE with LLVM analyzers. Functions from libVerify, as described in Table 1, provide an interface between a harness and an analyzer (usage of each function is described in Sect. 3). Each implementation of libVerify configures how a certain analyzer should interact with a harness. Build details are resolved using CMake scripts. For example, CMake arguments are used to switch the implementation of primitive singleton structures between native C integers and Boost multiprecision integers. To promote extensibility, certain interfaces in libVerify are designed with many analyzers in mind. A key example is bounded nondeterminism.

In libVerify, the functions `ND_INT` and `ND_UINT` are used as sources of nondeterminism. For example, SeaHorn provides nondeterminism via symbolic values, whereas libFuzzer approximates nondeterminism through randomness. In principle, all choices could be implemented using these interfaces. However, certain operations, such as "*increase the current block number*," or "*choose an address between* 3 *and* 5," require specialized implementations, depending on the analyzer. For this reason, libVerify provides multiple interfaces for nondeterminism, such as `ND_INCREASE` and `ND_RANGE`. To illustrate this design choice, the implementations of `ND_RANGE` for SeaHorn and libFuzzer are discussed.

The interface `ND_RANGE(id,lo,hi,msg)` returns a value between `lo` (inclusively) and `hi` (exclusively). Efficient implementations are given for SeaHorn and libFuzzer in Fig. 8a and Fig. 8b, respectively. The SeaHorn implementation is correct, since failed assumptions in symbolic analysis simply restrict the domain of each symbolic variable. Intuitively, assumptions made in the future can influence choices made in the past. This design does not work for libFuzzer, as failed assumptions in libFuzzer simply halt execution. This is because all values in libFuzzer are concrete. Instead, a value is constructed between `lo` and `hi` through modular arithmetic. In contrast, many symbolic analyzers struggle with non-linear constraints such as modulo. Therefore, neither implementation is efficient for both model checking and fuzzing.

SmartACE has been instantiated for greybox fuzzing, bounded model checking (BMC), parameterized compositional model checking (PCMC), and symbolic execution. The current version of libVerify supports libFuzzer for fuzzing, SeaHorn for model checking, and Klee for symbolic execution. Other analyzers, such as AFL [54] and SMACK [13], can also be integrated by extending libVerify. Each implementation of libVerify offers unique analysis benefits.

Interactive Test Harness. A default implementation of libVerify provides an interactive test harness. Nondeterminism, and the return values for `sol_continue`, are resolved through standard input. Events such as `sol_emit` are printed to standard output. The `sol_on_transaction` hook is used to collect test metrics, such as the number of transactions. As mentioned in Sect. 2, providing an interactive harness improves the testability of SmartACE.

Table 1. Summary of the LIBVERIFY interface.

Interface	Description
`sol_continue()`	Returns `true` if the transaction execution loop should continue
`sol_can_interfere()`	Returns `true` if interference should be applied and validated
`sol_require(cond, msg)`	If `cond` is `false`, then `msg` is logged and the transaction aborts
`sol_assert(cond, msg)`	If `cond` is `false`, then `msg` is logged and the program fails
`sol_emit(expr)`	Performs analyzer-specific processing for a call to `emit expr`
`ND_INT(id, n, msg)`	Returns an `n`-bit signed integer
`ND_UINT(id, n, msg)`	Returns an `n`-bit unsigned integer
`ND_RANGE(id, lo, hi, msg)`	Returns an 8-bit unsigned integer between `lo` (incl.) and `hi` (excl.)
`ND_INCREASE(id, cur, strict, msg)`	Returns a 256-bit unsigned integer that is greater than or equal to `cur` If `strict` is true, then the integer is strictly larger than `cur`

```
1 int rv = *;
2 assume(lo <= rv && rv < hi);
3 return rv;
```

(a) An implementation for SEAHORN.

```
1 int rv = rand();
2 rv = lo + (rv % (hi - lo));
3 return rv;
```

(b) An implementation for LIBFUZZER.

Fig. 8. Possible implementations of `ND_RANGE(n,lo,hi,msg)`.

Greybox Fuzzing. Fuzzing is an automated testing technique that explores executions of a program via input generation [38]. In greybox fuzzing, coverage information is extracted from a program to generate a sequence of inputs that maximize test coverage [56]. The harness for greybox fuzzing is instantiated with N participants, and each participant has strong updates. In general, greybox fuzzing is a light-weight technique to test edge-cases in contracts. As opposed to other smart contract fuzzing techniques, SMARTACE performs all fuzzing against a local bundle abstraction. This ensures that all implicit participants are in the address space. To illustrate the benefit of local bundle abstractions in fuzzing, consider the property for Fig. 1: "*The user with address 100 never places a bid.*". Without a local bundle abstraction, a counterexample requires 101 users (`address(0)` to `address(100)`). With a local bundle abstraction, only 4 users are required (e.g., the zero-account, the two contracts, and `address(100)`).

Table 2. Analysis results for each case study. For bug finding, n is the number of users, FUZ is greybox fuzzing, and SYM is symbolic execution. BMC results marked by (†) were obtained using an additional bound of 5 transactions. Omitted results indicate that a system memory limit was exceeded.

Benchmark		Verification		Bug Finding ($n = 5$)			Bug Finding ($n = 500$)		
Contract	**Prop.**	**Manual (s)**	**Auto. (s)**	**BMC (s)**	**FUZ (s)**	**SYM (s)**	**BMC (s)**	**FUZ (s)**	**SYM (s)**
`Ownable`	**O1**	1	1	1	1	90	1	1	85
	O2	1	1	1	1	25	1	1	27
	O3	1	1	1	1	25	1	1	27
`RefundEscrow`	**R1**	2	2	2	1	454	140	22	—
	R2	2	3	2	2	5	277	32	3124
	R3	2	7	5	26	5	1800	74	—
	R4	12	17	3	6	90	1724	296	—
	R5	3	4	3	2	6	2010$^{(\dagger)}$	33	—
`Auction`	**A1**	9	59	2	4	39	564	21	123
	A2	69	246	4	3	533	4392	397	—

Symbolic Execution. Symbolic execution is a sophisticated technique that can be used to find bugs in programs. At a high-level, symbolic execution converts program paths into logical constraints, and then solves for inputs that violate program assertions [12]. Symbolic execution is very precise, but its performance is negatively impacted by the number of paths through a program, which is often unbounded. As in the case of greybox fuzzing, the symbolic execution harness is instantiated with N participants, each with strong updates. Symbolic execution targets deeper violations than greybox fuzzing, at the cost of analysis time.

BMC. Model checking is a technique that, with little human input, proves properties of a program [15,43]. In bounded model checking (BMC), properties are proven up to a bound on execution (e.g., on the number of loop iterations or users) [7]. The harness for BMC is instantiated with N participants, each with strong updates. BMC either proves a bundle is safe up to N users, or finds a counterexample using at most N users (e.g., see [29]). As the harness is executable, SmartACE is able to compile and execute counterexamples found by SeaHorn. With SeaHorn, integers can be bit-precise [31], or over-approximated by linear integer arithmetic [8]. The number of transactions can be bounded, or an inductive invariant can be discovered for the transaction loop.

PCMC. PCMC is a modular reasoning technique for network verification [40]. Given an interference invariant, PCMC either proves a bundle is safe for any number of users, or finds a counterexample to compositionality (i.e., the interference invariant is inadequate). The harness is instantiated with representative users, and at most the transient and implicit participants are concrete (this is configurable). Increasing the number of concrete participants refines the abstraction, but also increases the size of the state space. As with BMC, integers may be bit-precise or arithmetic, and all counterexamples are executable. If SeaHorn is used as a model checker, then interference invariants are inferred from their initial conditions (i.e., all mapping entries are zero), and their usage throughout the harness. This technique is called *predicate synthesis*.

5 Case Study: Verifying OpenZeppelin Contracts

We illustrate the effectiveness of SmartACE and Scribble by applying them to analyze the OpenZeppelin library[2]. OpenZeppelin is a widely used Solidity library (more than 12'000 stars on GitHub) that implements many Ethereum protocols. From this library, we identify and verify key properties for the `Ownable` and `RefundEscrow` contracts. Properties are specified in the Scribble specification language[3]. To validate our results, we use fault injection to show that both the harness and the property instrumentation behave as expected. Faults are detected using SeaHorn (bounded in the number of users), libFuzzer, and Klee. To highlight properties not reflected in prior smart contract research, we conclude by verifying two novel properties for `Auction` from Fig. 1. All evaluations were run on an Intel® Core i7® CPU @ 1.8 GHz 8-core machine with 16 GB of RAM running Ubuntu 20.04. Timing results are given in Table 2.

5.1 Verification of `Ownable`

A simplified implementation of `Ownable` is presented in Fig. 9. This contract provides a simple access-control mechanism, in which a single user, called the *owner*, has special privileges. Initially, the owner is the user who creates the contract. At any point during execution, an owner may transfer ownership to another user by calling `transferOwnership`. An owner may also renounce ownership by calling `renounceOwnership`. When ownership is renounced, the owner is permanently set to `address(0)` and all privileges are lost. These behaviours are captured informally by three properties:

O1. If `transferOwnership(u)` is called successfully, then the new owner is `u`.
O2. If ownership changes, then the sender is the previous owner.
O3. If ownership changes and `renounceOwnership` has been called at least once, then the new owner is `address(0)`.

O1 is a post-condition for `transferOwnership`. In Scribble, post-conditions are specified by function annotations. However, function annotations are checked upon function return, using the latest value of each local variable. This means that if `u` was changed during the execution of `transferOwnership`, then the annotation refers to the newest value of `u`. To overcome this, `old(u)` is used to refer to the original value of `u`. The Scribble annotation is added at line 17 of Fig. 9.

O2 is an assertion for each update to `_owner`. In Scribble, invariants can be placed on state variable updates using state variable annotations. State variable annotations are checked after each update, even if the update is made during setup in a constructor. However, **O2** refers to "*ownership changes*" which assumes implicitly that some user already owns the contract. Therefore, the invariant should only be checked after construction. This is achieved by adding

[2] https://github.com/OpenZeppelin/openzeppelin-contracts/
[3] https://docs.scribble.codes/

```
1  contract Ownable {
2    bool _ctor = false;
3    bool _called = false;
4
5    /// #if_updated _ctor ==> msg.sender == old(_owner);
6    /// #if_updated _called ==> _owner == address(0);
7    address private _owner;
8
9    constructor() public {
10     _owner = msg.sender;
11     _ctor = true;
12   }
13
14   modifier onlyOwner() {
15     require(_owner == msg.sender); _;
16   }
17   /// #if_succeeds old(u) == _owner;
18   function transferOwnership(address u) public
         onlyOwner {
19     require(u != address(0)); _owner = u;
20   }
21   function renounceOwnership() public onlyOwner {
22     _called = true;
23     _owner = address(0);
24   }
25 }
```

Fig. 9. A simplified implementation of `Ownable` from OpenZeppelin. All comments are Scribble annotations, and all highlighted lines are instrumentation used in annotations.

```
1  /// #invariant  _fn_1 ==> address(this).balance == 0;
2  /// #invariant !_fn_1 ==> address(this).balance ==
         unchecked_sum(_d);
3  contract RefundEscrow is Ownable {
4    bool _fn_1 = false; bool _fn_2 = false;
5    address _u;
6
7    enum State { Active, Refunding, Closed }
8    address payable private immutable _beneficiary;
9    mapping(address => uint256) private _d; // Deposits.
10   /// #if_updated !_fn_2 ==> old(_d[_u]) <= _d[_u];
11   State private _state = State.Active;
12
13   constructor(address payable b, address u) public {
14     require(b != address(0)); _beneficiary = b;
15     _u = u;
16   }
17   function deposit(address p) public payable
         onlyOwner {
18     require(_state == State.Active);
19     _d[p] += msg.value;
20   }
21   function withdraw(address payable p) public {
22     require(_state == State.Refunding);
23     uint256 payment = _d[p]; _d[p] = 0;
24     p.transfer(payment);
25   }
26   function close() public onlyOwner {
27     require(_state == State.Active);
28     _state = State.Closed;
29   }
30   function enableRefunds() public onlyOwner {
31     _fn_2 = true;
32     require(_state == State.Active);
33     _state = State.Refunding;
34   }
35   function beneficiaryWithdraw() public {
36     _fn_1 = true;
37     require(_state == State.Closed);
38     beneficiary().transfer(address(this).balance);
39   }
40 }
41
```

Fig. 10. A simplified implementation of `RefundEscrow` from OpenZeppelin. All comments are Scribble annotations, and all highlighted lines are instrumentation used in annotations. The field `_deposits` is renamed `_d`.

a flag variable `_ctor` at line 2 that is set to **`true`** after the constructor has terminated (see line 11). The Scribble annotation is added at line 5 of Fig. 9.

O3 is also an assertion for each update to `_owner`. However, the techniques used to formalize **O2** are not sufficient for **O3**, as **O3** also refers to functions called in the past. To determine if `renounceOwnership` has been called, a second flag variable `_called` is added at line 3 that is set to **`true`** upon entry to `renounceOwnership` at line 22. The Scribble annotation is added at line 6 of Fig. 9.

SmartACE verified each property within 1 s. Furthermore, as `Ownable` does not maintain user-data, verification did not require interference invariants. To validate these results, a fault was injected for each property. Bounded models were then generated using 5 and 500 users to analyze the impact of parameterization. All faults were detected using each of BMC, greybox fuzzing, and symbolic execution. Both BMC and greybox fuzzing were able to detect each fault within 1 s, whereas symbolic execution required up to 90 s per fault. In this case study, the number of users did not impact analysis time.

5.2 Verification of RefundEscrow

A simplified implementation for RefundEscrow is presented in Fig. 10. An escrow is used when a smart contract (the owner) must temporarily hold funds from its users. In the case of RefundEscrow, the owner deposits funds on behalf of its users. If some condition is reached (as determined by the owner), the escrow is closed and a beneficiary may withdraw all funds. Otherwise, the owner may enable refunds, and each user can withdraw their funds without the intervention of the owner. In this case study, we consider five properties of RefundEscrow:

R1 If the state changes, then ownership has not been renounced.
R2 If close has been called, then all deposits are immutable.
R3 If close has been called, then enableRefunds has not been called.
R4 If beneficiaryWithdraw has been called, then the balance of the refund escrow is 0, otherwise the balance is the sum of all deposits.
R5 If enableRefunds has not been called, then all deposits are increasing.

The first three properties are not parameterized and can be formalized using the same techniques as in the previous case study (Sect. 5.1). **R4** is formalized using the unchecked_sum operator and a contract invariant, as illustrated on lines 1–2 of Fig. 10. In Scribble, unchecked_sum is used to track the sum of all elements in a mapping, without checking for integer overflow. Note that a contract invariant was required, as **R4** must be checked each time RefundEscrow receives payment. **R5** is formalized using a new technique, as illustrated on line 10 of Fig. 10. The key observation is that **R5** is equivalent to, "*For every address* _u,*if* enableRefunds *has not been called, then* old(_d[_u]) *is less than or equal to* _d[_u]." Then, RefundEscrow does not satisfy **R5** if and only if there exists some witnessing address _u that violates the new formulation. Therefore, **R5** can be checked by non-deterministically selecting a witness, and then validating its deposits across each transactions. In Fig. 10, line 15 non-deterministically selects a witness via user input, and line 10 validates each deposit made on behalf of the witness. Therefore, the annotation on line 10 is equivalent to **R5**.

Since RefundEscrow maintains user-data, all verification required interference invariants (see Sect. 3.3). SmartACE verified each property within 17 s using predicate synthesis. For **R1** to **R4**, all users were abstract, whereas **R5** required concrete transient participants to reason exactly about _d[_u]. For comparison, SmartACE was then used to verify each property with user-provided interference invariants. It was found that a "*trivial*" interference invariant, that includes all deposits, was sufficient to verify each property within 12 s. As in the previous case study, faults were then injected, and detected using 5 and 500 users. With 5 users, BMC required up to 5 s, greybox fuzzing required up to 26 s, and symbolic execution required up to 454 s. However, with 500 users, BMC increased to 33 min, fuzzing increased to 5 min, and symbolic execution exceeded system resource limits for most properties. In this case study, reducing the number of users significantly reduced analysis time.

```
 1 /// #invariant _monotonic && _max == leadingBid;
 2 /// #invariant leadingBid <= unchecked_sum(bids);
 3 /// #invariant bids[_u] == 0 || bids[_u] != bids[_v];
 4 contract Auction {
 5   /* ... State Variables ... */
 6   address _u; address _v;
 7   uint _max = 0; bool _monotonic = true;
 8
 9   constructor(address _m, address u, address v) {
10     manager = _m;
11     _u = u; _v = v; require(_u != _v);
12   }
13   function bid() public payable canParticipate() {
14     uint _pre = bids[msg.sender];
15     require(msg.value > leadingBid);
16     bids[msg.sender] = msg.value;
17     leadingBid = msg.value;
18     uint _post = bids[msg.sender];
19     if (_max < _post) { _max = _post; }
20     if (_post < _pre) { _monotonic = false; }
21   }
22
23   /* ... Other Functions and Modifiers ... */
24 }
```

Fig. 11. An annotated version of `Auction` from Fig. 1. All comments are SCRIBBLE annotations, and all highlighted lines are instrumentation used in annotations.

5.3 Verification of `Auction`

Recall `Auction` from Fig. 1. In this case study the following two properties are formalized and verified:

A1. The maximum bid equals `leadingBid` and is at most the sum of all bids.
A2. Any pair of non-zero bids are unequal.

A1 involves the maximum element of `bids`, and is not addressed by existing smart contract analyzers. The challenge in verifying **A1** is that the exact value of `max(bids)` depends on all previous writes to `bids`. Specifically, each time the largest bid is overwritten by a smaller bid, the value of `max(bids)` must be set to the *next largest* bid. However, if the maximum bid is monotonically increasing, then `max(bids)` is equal to the largest value previously written into `bids`. This motivates a formalization that approximates `max(bids)`. In this formalization, two variables are added to `Auction`. The first variable tracks the largest value written to `bids` (see line 19 in Fig. 11). The second variable is **`true`** so long as `max(bids)` is monotonically increasing (see line 20 in Fig. 11). Together, these two variables help formalize **A1**, as illustrated by lines 1–2 in Fig. 11.

A2 compares two arbitrary elements in `bids`, and cannot be reduced to pre- and post-conditions. However, the technique used for **R5** in Sect. 5.2 generalizes directly to **A2** as shown on line 3 in Fig. 11. In the formalization, there are now two instantiated users: `u` and `v`. On line 11, an assertion is added to ensure that these users are unique (i.e., a “pair” of users).

SMARTACE verified each property within 246 s using predicate synthesis. For **A1**, all users were abstract, whereas **A2** required concrete transient participants to reason exactly about `bids[_u]` and `bids[_v]`. For comparison, SMARTACE was then used to verify each property with user-provided interference invariants. Unlike in the previous study (Sect. 5.2), a trivial interference invariant was insufficient to prove **A1**. However, the discovery of a non-trivial invariant was aided by counterexamples. Initially, the trivial invariant was used, and a counterexample was returned in which each user's initial bid was larger than 0. This suggested that each element of `bids` must be bounded above, which motivated a second invariant: `bids[i] <= leadingBid`. This new invariant was shown to be compositional, and adequate to prove **A1**. Using the provided interference invariants, each property was verified within 56 s.

As in the previous case studies, faults were then injected, and detected using 5 and 500 users. With 5 users, BMC required up to 4 s, greybox fuzzing required up to 4 s, and symbolic execution required up to 533 s. However, with 500 users, BMC increased to 73 min, fuzzing increased to 6 min, and symbolic execution exceeded system resource limits for **A1**. As in Sect. 5.2, reducing the number of users significantly reduced analysis time.

5.4 Discussion

Inter-transactional Analysis. SmartACE is an *inter-transactional* verification tool. That is, SmartACE verifies properties across unbounded sequences of transactions. In contrast, *intra-transactional* verification tools (e.g., [2,26]) verify pre- and post-conditions for single transactions. Inter-transactional verification is a more challenging problem, as it requires an invariant for contract state between transactions. In our study, inter-transactional verification was required to support properties involving calls made in the past (e.g., **O3** and **R3**), and to eliminate unreachable contract states (e.g., the interference invariant used to prove **A1**). While there are many techniques for inter-transactional verification (e.g., [23,37,42,44,45,50]), we believe that the SmartACE approach is unique in its level of automation and its ability to handle parameterization in the number of contract users.

Automation. SmartACE is a fully-automated tool for inter-transactional verification, with optional user-guidance (i.e., user-provided interference invariants). Many other tools rely on semi-automated approaches, such as user-provided contract invariants (i.e., [23,50]) or predicate abstractions (i.e., [42]). Of the fully-automated tools (i.e., [37,44,45]), neither address the state explosion problem. Furthermore, [45] is designed for the harder problem of liveness checking, whereas [37,44] rely on less optimized model checking techniques than in SeaHorn.

Parameterization. SmartACE is based on the hypothesis that existing smart contract verifiers struggle to scale due to the impact of users on the size of the state space. This aligns with bug finding results for `Ownable`, `RefundEscrow`, and `Auction`. In the case of `Ownable`, user-data was not maintained, and as expected, the number of users had no noticeable impact on analysis time. In contrast, both `RefundEscrow` and `Auction` maintain user-data and are significantly impacted by the number of users. For BMC, analysis time increased from seconds to hours, whereas symbolic execution became infeasible. Greybox fuzzing was less impacted by the number of users, which likely reflects that greybox fuzzing is coverage-based, as opposed to symbolic.

Integration Challenges. Two major challenges were encountered while integrating SmartACE with Scribble. The first challenge came from `unchecked_sum`. When Scribble instruments `unchecked_sum`, extra ghost state is added such as `address[] keys` which is used to track all updated fields in the mapping. The

purpose of this ghost state is to support quantification, but it is not required for summation. However, this state is not supported by SmartACE, and also adds overhead for dynamic analysis. To support `unchecked_sum` in SmartACE, this state was manually removed. The second challenge came with formalizing the predicate: "*function* `fn` *has been called at least once.*" Formally, this predicate is expressed by `once(called(fn))`, and is supported by other smart contract verification tools such as [42,45]. However, these specifications are not supported by Scribble. As shown in Sect. 5.1, both `once` and `called` can be instrumented manually with flag variables. However, manual instrumentation is more error-prone than well-tested automated instrumentation. We conclude that SmartACE can integrate with Scribble, but that further improvements are needed for the integration to become seamless. Furthermore, these improvements would benefit all users of Scribble, as opposed to only SmartACE.

6 Related Work

Inter-transactional Verification. There are many tools for inter-transactional verification. Manual approaches, such as [6,19,27] provide proof-assistants for end-users to verify properties. These tools are versatile, but are also time consuming and are aimed at verification engineers rather than developers. Semi-automated approaches, such as [23,50], require end-users to manually provide contract invariants. In VerX [42], contract invariants are discovered automatically, but an end-user must provide an adequate predicate abstraction. Automated approaches, such as [37,44,45], do not offer solutions to parameterization, and instead rely on the underlying solvers to reduce symmetries.

Reusing Off-the-Shelf Tools. SmartACE is not the first smart contract analyzer to leverage existing analyzers for more widely used languages. For example, prior work has applied SeaHorn for gas estimation [36], and intra-transactional verification [2,26]. Other smart contract analyzers have reduced to Datalog for checking access control patterns [10], detecting gas exploits [17], and implementing general pattern checks [48]. In [49], SMACK is used to detect *non-deterministic payment bugs*. In [29], TLA+ is used to perform inter-transactional analysis with a reduced number of users. SmartACE is the first application of off-the-shelf tools to unbounded inter-transactional verification.

Bug Finding. There are multiple tools for smart contract symbolic execution (e.g., [32,35,39,47,55]) and fuzzing (e.g., [18,24,25,53]). A major challenge for such tools is finding deep violations across many transactions. In [55], a static analysis technique is introduced to eliminate uninteresting transaction sequences. In [47], a learning-based approach is used to train accurate fuzzers from symbolic execution. We suspect that SmartACE would also benefit from such techniques.

Parameterized Verification. Parameterized systems form a rich field of research, as outlined in [9]. In general, verifying a parameterized system is undecidable [3].

However, local bundle abstraction is an instance of PCMC [40], and is decidable (for finite-state systems) relative to an interference invariant. Furthermore, the discovery of interference invariants in SmartACE is an instance of [22]. Though this paper is restricted to safety properties, local bundle abstractions are known to extend to CTL* [41]. Abdualla et al. [1] propose a somewhat similar notion of *view abstraction* to abstract interfering processes in a network, though this abstraction has not been applied to smart contracts.

7 Conclusion

We presented SmartACE, a communication-aware smart contract framework with support for multiple off-the-shelf analyzers. The framework is based on parameterized smart contract verification, and can verify properties for arbitrarily many users. We reported on verifying two widely used smart contracts from the OpenZeppelin library. We then applied SmartACE to a simple open-bid auction to highlight limitations of existing smart contract analyzers, and how they are alleviated by SmartACE. We show that in practice, SmartACE is appropriate for fully-automated smart contract analysis.

During the implementation and evaluation of SmartACE, several challenges were encountered. At the implementation stage, we observed that many analyzers handle value selection and non-determinism using incompatible techniques. To overcome this incompatibility, we introduced libVerify to separate the details of an analyzer from the harness design. At the evaluation stage, we identified limitations in Scribble and suggested improvements. We also proposed manual solutions that can be used to circumvent the limitations of Scribble.

References

1. Abdulla, P., Haziza, F., Holík, L.: Parameterized verification through view abstraction. Int. J. Softw. Tools Technol. Transf. **18**(5), 495–516 (2015). https://doi.org/10.1007/s10009-015-0406-x
2. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: SAFEVM: a safety verifier for Ethereum smart contracts. In: Zhang, D., Møller, A. (eds.) Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, 15–19 July 2019, pp. 386–389. ACM (2019). https://doi.org/10.1145/3293882.3338999
3. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. **22**(6), 307–309 (1986). https://doi.org/10.1016/0020-0190(86)90071-2
4. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: Anderson, L., Coplien, J. (eds.) Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 1996), San Jose, California, USA, 6–10 October 1996, pp. 324–341. ACM (1996). https://doi.org/10.1145/236337.236371

5. Beyer, D., Lewerentz, C., Simon, F.: Impact of inheritance on metrics for size, coupling, and cohesion in object-oriented systems. In: Dumke, R., Abran, A. (eds.) IWSM 2000. LNCS, vol. 2006, pp. 1–17. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44704-0_1
6. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Murray, T.C., Stefan, D. (eds.) Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, 24 October 2016, pp. 91–96. ACM (2016). https://doi.org/10.1145/2993600.2993611
7. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
8. Bjørner, N., Gurfinkel, A.: Property directed polyhedral abstraction. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 263–281. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_15
9. Bloem, R., et al.: Decidability in parameterized verification. SIGACT News **47**(2), 53–64 (2016). https://doi.org/10.1145/2951860.2951873
10. Brent, L., Grech, N., Lagouvardos, S., Scholz, B., Smaragdakis, Y.: Ethainter: a smart contract security analyzer for composite vulnerabilities. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, 15–20 June 2020, pp. 454–469. ACM (2020). https://doi.org/10.1145/3385412.3385990
11. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, 8–10 December 2008, San Diego, California, USA, Proceedings, pp. 209–224. USENIX Association (2008)
12. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Juels, A., Wright, R.N., di Vimercati, S.D.C. (eds.) Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, 30 October–3 November 2006, pp. 322–335. ACM (2006). https://doi.org/10.1145/1180405.1180445
13. Carter, M., He, S., Whitaker, J., Rakamaric, Z., Emmi, M.: SMACK software verification toolchain. In: Dillon, L.K., Visser, W., Williams, L.A. (eds.) Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, 14–22 May 2016 - Companion Volume, pp. 589–592. ACM (2016). https://doi.org/10.1145/2889160.2889163
14. Chen, H., Pendleton, M., Njilla, L., Xu, S.: A survey on Ethereum systems security: vulnerabilities, attacks, and defenses. ACM Comput. Surv. **53**(3), 67:1–67:43 (2020). https://doi.org/10.1145/3391195
15. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
16. Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47, 587 Ethereum smart contracts. In: Rothermel, G., Bae, D. (eds.) ICSE 2020: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June-19 July 2020, pp. 530–541. ACM (2020). https://doi.org/10.1145/3377811.3380364

17. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: MadMax: surviving out-of-gas conditions in Ethereum smart contracts. Proc. ACM Program. Lang. **2**(OOPSLA), 116:1–116:27 (2018). https://doi.org/10.1145/3276486
18. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: Khurshid, S., Pasareanu, C.S. (eds.) ISSTA 2020: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, 18–22 July 2020, pp. 557–560. ACM (2020). https://doi.org/10.1145/3395363.3404366
19. 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
20. Grossman, S., et al.: Online detection of effectively callback free objects with applications to smart contracts. Proc. ACM Program. Lang. **2**(POPL), 48:1–48:28 (2018). https://doi.org/10.1145/3158136
21. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
22. Gurfinkel, A., Shoham, S., Meshman, Y.: SMT-based verification of parameterized systems. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, 13–18 November 2016, pp. 338–348. ACM (2016). https://doi.org/10.1145/2950290.2950330
23. Hajdu, Á., Jovanović, D.: SOLC-VERIFY: a modular verifier for solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) VSTTE 2019. LNCS, vol. 12031, pp. 161–179. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41600-3_11
24. He, J., Balunovic, M., Ambroladze, N., Tsankov, P., Vechev, M.T.: Learning to fuzz from symbolic execution with application to smart contracts. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, 11–15 November 2019, pp. 531–548. ACM (2019). https://doi.org/10.1145/3319535.3363230
25. Jiang, B., Liu, Y., Chan, W.K.: ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, 3–7 September 2018, pp. 259–269. ACM (2018). https://doi.org/10.1145/3238147.3238177
26. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, 18–21 February 2018. The Internet Society (2018)
27. Kasampalis, T., et al.: IELE: a rigorously designed language and tool ecosystem for the blockchain. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 593–610. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_35
28. Kildall, G.A.: A unified approach to global program optimization. In: Fischer, P.C., Ullman, J.D. (eds.) Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973, pp. 194–206. ACM Press (1973). https://doi.org/10.1145/512927.512945

29. Kolb, J.: A languge-based approach to smart contract engineering. Ph.D. thesis, University of California at Berkeley, USA (2020)
30. Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., Saxena, P.: Exploiting the laws of order in smart contracts. In: Zhang, D., Møller, A. (eds.) Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, 15–19 July 2019, pp. 363–373. ACM (2019). https://doi.org/10.1145/3293882.3330560
31. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_2
32. Krupp, J., Rossow, C.: teEther: Gnawing at Ethereum to automatically exploit smart contracts. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, 15–17 August 2018, pp. 1317–1333. USENIX Association (2018)
33. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis & transformation. In: 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2004), 20–24 March 2004, San Jose, CA, USA, pp. 75–88. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665
34. LibFuzzer–A library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html
35. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016, pp. 254–269. ACM (2016). https://doi.org/10.1145/2976749.2978309
36. Marescotti, M., Blicha, M., Hyvärinen, A.E.J., Asadi, S., Sharygina, N.: Computing exact worst-case gas consumption for smart contracts. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 450–465. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_33
37. Marescotti, M., Otoni, R., Alt, L., Eugster, P., Hyvärinen, A.E.J., Sharygina, N.: Accurate smart contract verification through direct modelling. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12478, pp. 178–194. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61467-6_12
38. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12), 32–44 (1990). https://doi.org/10.1145/96267.96279
39. Mossberg, M., et al.: Manticore: a user-friendly symbolic execution framework for binaries and smart contracts. In: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, 11–15 November 2019, pp. 1186–1189. IEEE (2019). https://doi.org/10.1109/ASE.2019.00133
40. Namjoshi, K.S., Trefler, R.J.: Parameterized compositional model checking. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 589–606. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_39
41. Namjoshi, K.S., Trefler, R.J.: Symmetry reduction for the local mu-calculus. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 379–395. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_22
42. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.T.: VerX: safety verification of smart contracts. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, 18–21 May 2020, pp. 1661–1677. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00024

43. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982). https://doi.org/10.1007/3-540-11494-7_22
44. So, S., Lee, M., Park, J., Lee, H., Oh, H.: VeriSmart: a highly precise safety verifier for Ethereum smart contracts. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, 18–21 May 2020, pp. 1678–1694. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00032
45. Stephens, J., Ferles, K., Mariano, B., Lahiri, S., Dillig, I.: SmartPulse: automated checking of temporal properties in smart contracts. In: 42nd IEEE Symposium on Security and Privacy. IEEE (2021)
46. Szabo, N.: Smart contracts: building blocks for digital markets (1996)
47. Torres, C.F., Schütte, J., State, R.: Osiris: hunting for integer bugs in Ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, 03–07 December 2018, pp. 664–676. ACM (2018). https://doi.org/10.1145/3274694.3274737
48. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: practical security analysis of smart contracts. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, 15–19 October 2018, pp. 67–82. ACM (2018). https://doi.org/10.1145/3243734.3243780
49. Wang, S., Zhang, C., Su, Z.: Detecting nondeterministic payment bugs in Ethereum smart contracts. Proc. ACM Program. Lang. **3**(OOPSLA), 189:1–189:29 (2019). https://doi.org/10.1145/3360615
50. Wang, Y., et al.: Formal verification of workflow policies for smart contracts in azure blockchain. In: Chakraborty, S., Navas, J.A. (eds.) VSTTE 2019. LNCS, vol. 12031, pp. 87–106. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41600-3_7
51. Wesley, S., Christakis, M., Navas, J.A., Trefler, R.J., Wüstholz, V., Gurfinkel, A.: Compositional verification of smart contracts through communication abstraction (extended). CoRR abs/2107.08583 (2021)
52. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2014)
53. Wüstholz, V., Christakis, M.: Harvey: a greybox fuzzer for smart contracts. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) ESEC/FSE 2020: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, 8–13 November 2020, pp. 1398–1409. ACM (2020). https://doi.org/10.1145/3368089.3417064
54. Zalewski, M.: Technical whitepaper for AFL. http://lcamtuf.coredump.cx/afl/technical_details.txt
55. Zhang, W., Banescu, S., Pasos, L., Stewart, S.T., Ganesh, V.: MPro: combining static and symbolic analysis for scalable testing of smart contract. In: Wolter, K., Schieferdecker, I., Gallina, B., Cukier, M., Natella, R., Ivaki, N.R., Laranjeiro, N. (eds.) 30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, 28–31 October 2019, pp. 456–462. IEEE (2019). https://doi.org/10.1109/ISSRE.2019.00052
56. Zhang, Y., Zhang, J., Zhang, D., Mu, Y.: Survey of directed fuzzy technology. In: 2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS), pp. 696–699. IEEE (2018). https://doi.org/10.1109/ICSESS.2018.8663772