Consolidating Smart Contracts with Behavioral Contracts

GUANNAN WEI, Purdue University, USA
DANNING XIE, Purdue University, USA
WUQI ZHANG, The Hong Kong University of Science and Technology, China and Purdue University, USA
YONGWEI YUAN, Purdue University, USA
ZHUO ZHANG*, Purdue University, USA

Ensuring the reliability of smart contracts is of vital importance due to the wide adoption of smart contract programs in decentralized financial applications. However, statically checking many rich properties of smart contract programs can be challenging. On the other hand, dynamic validation approaches have shown promise for widespread adoption in practice. Nevertheless, as part of the programming environment for smart contracts, existing dynamic validation approaches have not provided programmers with a notion to clearly articulate the interface between components, especially for addresses representing opaque contract instances. We argue that the "design-by-contract" approach should complement the development of smart contract programs. Unfortunately, there is limited linguistic support for it in existing smart contract languages.

In this paper, we design a Solidity language extension, ConSol, that supports behavioral contracts. ConSol provides programmers with a modular specification and monitoring system for both functional and latent address behaviors. The key capability of ConSol is to attach specifications to first-class addresses and monitor violations when invoking these addresses. We evaluate ConSol using 20 real-world cases, demonstrating its effectiveness in expressing critical conditions and preventing attacks. Additionally, we assess ConSol's efficiency and compare gas consumption with programs fixed with manually inserted assertions, showing that our approach introduces only marginal gas overhead. By separating specifications and implementations using behavioral contracts, ConSol assists programmers in writing more robust and readable smart contracts.

CCS Concepts: • **Software and its engineering** → *Software verification and validation*; *Software development techniques*; *Specification languages.*

Additional Key Words and Phrases: smart contracts, behavioral contracts, specification, runtime verification

ACM Reference Format:
Guannan Wei, Danning Xie, Wuqi Zhang, Yongwei Yuan, and Zhuo Zhang. 2024. Consolidating Smart Contracts with Behavioral Contracts. *Proc. ACM Program. Lang.* 8, PLDI, Article 186 (June 2024), 25 pages. https://doi.org/10.1145/3656416

1 INTRODUCTION

Smart contracts are programs to automate the execution of transactional agreements, ensuring that all involved parties align in their expectations of the outcome. Smart contract programs are "smart" as they *ensure the economic outcome of transactions* by running on decentralized blockchains, which implement a consensus protocol of immutable, distributed ledgers [56]. Since transactions cannot

*Corresponding author.

Authors' addresses: Guannan Wei, Purdue University, USA, wei220@purdue.edu; Danning Xie, Purdue University, USA, xie342@purdue.edu; Wuqi Zhang, The Hong Kong University of Science and Technology, Hong Kong, China and Purdue University, USA, wuqi.zhang@connect.ust.hk; Yongwei Yuan, Purdue University, USA, yuan311@purdue.edu; Zhuo Zhang, Purdue University, USA, zhan3299@purdue.edu.

ACM 2475-1421/2024/6-ART186
https://doi.org/10.1145/3656416

be amended or revoked once finalized on the blockchain, the reliability of smart contract programs is of vital importance, especially for those involving financial or monetary operations.

It is however ironic that despite the potential of smart contracts, *languages* writing smart contracts often lack adequate mechanisms to *ensure the computational outcomes* of smart contracts. This unfortunate reality results in an unsatisfactory quality of smart contract programs. Without proper means to ensure the quality of these programs, the aspiration to ensure the economic outcome of transactions remains nothing more than a mere wish. Undoubtedly, the past few years have witnessed several catastrophic failures of blockchain systems resulting from attacks on vulnerable contract programs [92]. One notorious instance is the DAO hack [29], where an unauthorized attacker exploited recursive function calls and the callback mechanism, leading to a staggering loss of $50 million. The fundamental issue at the core of this attack on deployed contracts was the failure to prevent reentrancy under certain critical conditions.

To prevent such attacks, programmers need to be aware of and specify those critical conditions. The execution can be allowed only when those conditions are met. However, we observe that existing popular smart contract languages, such as Solidity, do not provide an *expressive, effective, and convenient* means to *specify* and *enforce* contract behaviors. Although static verification techniques have been intensively investigated [4, 73, 83], they can be expensive to use in practice or become imprecise after making over-approximated assumptions for many rich properties that depend on dynamic information. Run-time validations are more flexible, but existing work [47] focuses on coarse-grained global invariants or monitoring predicates of primitive data, which do not provide convenient and effective ways to examine address values or higher-order functions carrying latent computational content. As common patterns used in developing smart contract programs, these callable addresses and higher-order functions often exhibit elusive vulnerabilities (Figure 1a for an example).

The lack of *sufficient linguistic means* to modularly specify and enforce subtle and critical behaviors not only lead to vulnerable contract programs but also discourages programmers from writing clean and maintainable code with higher-level abstractions. Instead, programmers are obliged to meticulously write verbose low-level code for defensive checks (e.g. with assertions). These checks are interspersed with the main business logic, leading to poor readability and maintainability (see an example in Figure 1a). Often worse, defensive checks are neglected, resulting in vulnerable code causing real monetary loss.

Contracts for Contracts. To address these issues, we argue that *behavioral software contracts* [53] should play a fundamental role in the development of reliable *smart contract* programs. As a metaphor, “behavioral contracts” specify assumptions and guarantees between software components, just as “smart contracts” specify assumptions and guarantees between business parties. The two notions of “contract” should and can complement each other.

Behavioral contracts are an expressive and convenient tool for programmers since assumptions (pre-conditions) and guarantees (post-conditions) are specified as executable specifications, written in the same programming language syntax. At run-time, violations of these conditions are monitored and reported. Pioneered by the Eiffel language [51], programmers have embraced the design-by-contract methodology [52, 53] to build high-assurance software in various languages with extensions of behavioral contracts, e.g., Java [14], C++ [16], Python [60], Haskell [91], Racket [35], Elixir [62], etc. Several studies [19, 52] have also shown that behavioral contracts can effectively support the design, development, testing, and debugging of software systems.

Unfortunately, such expressive tools for smart contract programmers are not yet available. For example, Solidity, as the most widely-used smart contract language, is just equipped with less expressive mechanisms such as low-level assertions and modifiers [76], which are verbose to express and enforce conditions of latent behaviors of address calls. In this paper, we develop ConSol[1], a practical specification and monitoring system for Solidity to assist in developing reliable, readable, and maintainable smart contract programs.

Specifying and Monitoring Functions and Addresses. ConSol allows programmers to specify and monitor both *function behaviors* and *address latent behaviors*. At the top level of function definitions, programmers can attach preconditions and postconditions to examine arguments, return values, and side effects of the target functions. These predicates can state conditions using any Solidity expression of type Boolean. When the function is invoked at runtime, these conditions regarding first-order values are dynamically examined by code generated with ConSol.

Other than primitive first-order data types such as integers, a more intricate data type in Solidity is *address*, which is the focus of this work. Similar to pointers in other low-level languages, addresses are represented by unsigned integers, referring to an account or contract instance on the blockchain. Unlike ordinary values, addresses carry latent computational contents, i.e., they embody other callable functions. Addresses are commonly used to implement callback functions in Solidity, thus functions taking addresses as arguments can have more latent behaviors that are not immediately obvious by just examining the function. It is both a recommended software engineering practice and a security concern to specify and check address behaviors when they are used as function arguments or return values.

In contrast to monitoring the specifications of first-order values, which can be performed by assertions as the prelude and epilogue of function calls, predicates of addresses cannot be checked in this way. To see the reason, consider an address x passed as an argument to a function $f(x)$, the latent arguments and return values of *address call* $x.g(e_1, \dots)$ are unknown at the time of f's invocation (see Figure 1 for a detailed example). It is in fact *undecidable* to check the properties of address x when calling f. Moreover, addresses are first-class citizens, i.e., they can be used as function arguments, returned from functions, or stored in storage. The flexible usage of addresses poses a challenge: if programmers have specified conditions for latent address calls, when and how should we soundly enforce the specifications of address values?

ConSol tackles this issue by borrowing ideas from behavioral contracts for higher-order functions [35], which have been supported in functional programming languages such as Racket. ConSol deploys a whole-program transformation that designates a new value representation for guarded addresses that can be attached with programmer-specified conditions. This new representation of guarded addresses is designed to be cheap and effective in practice. Although using a different representation for guarded addresses breaks many operations on addresses (e.g. checking equality for the same address attached with two different specifications), ConSol inserts code to unwrap guarded addresses to naked addresses before performing these operations. In this way, ConSol ensures that violations of guarded address calls within the current contract programs are monitored, regardless of how the guarded address value flows in the program. The persistent monitoring empowers programmers to write down the expected behaviors of address calls in a clean way, without interspersing low-level checks with business logic.

Effectiveness and Efficiency. To evaluate the effectiveness and efficiency of ConSol, we examine 20 real-world smart contract attacks (a total loss of \$154.32M) and their defenses (Section 6). Our results show that these defenses (i.e., fixes to bugs causing these losses) can be specified with a few lines of ConSol specifications and the generated programs are effective in preventing these attacks. Compared to manually inserted assertions, our approach is non-intrusive and improves readability.

[1] *Con*tract *Sol*idity, or, *Con*solidated *Sol*idity.

To improve the efficiency, ConSol implements optimizations to reduce the "gas" consumption induced by additional checking. Solidity programs run on the Ethereum Virtual Machine (EVM), and each program can only use a bounded amount of resources to execute, known as "gas". Since gas is a scarce resource to users, it is important that our approach is economic in its additional gas overhead. To evaluate the gas consumption of our approach, we patch the vulnerable contracts using both low-level assertions and ConSol specifications, respectively, and compare the gas fee increase induced by them. Results show that patching with ConSol is comparably efficient as using assertions, only costing at most $0.94 (avg. 0.207%) more transaction fees. We also evaluate the gas consumption of our approach compared to assertion checks that achieve the same effect on a dataset of contracts collected from the previous work by Li et al. [47]. Our results show that ConSol exhibits on average 0.29% higher overhead while offering significant enhancements in modularity, readability, and maintainability.

To summarize, ConSol inherits the design-by-contract approach and has the following features:

- *Non-intrusive*: it does not alter the behavior of Solidity programs and allows the programmers to write (partial) specifications only when necessary.
- *Effective*: it monitors violations of specified conditions for both top-level functions and address calls, regardless of how the address value flows in the program.
- *Expressive*: programmers can liberally write and enforce any computable property.
- *Efficient*: compiled smart contracts incur marginal runtime monitoring overhead.

Contributions. This paper makes the following contributions:

- We introduce ConSol, an extension for Solidity that empowers programmers to specify and enforce higher-order behaviors. We demonstrate ConSol's design and use cases through extensive examples (Section 3).
- We present the core formalization of ConSol and its translation semantics (Section 4). We discuss its expressiveness, limitations, as well as soundness by characterizing the extent of effective monitoring, providing a notion of when and where programmers can trust ConSol.
- We implement ConSol as a compiler that translates annotated programs into ordinary Solidity programs (Section 5). We discuss optimizations that lead to marginal additional gas overhead.
- We examine 20 representative real-world attacks, showing the effectiveness of ConSol in defending most attacks meanwhile exhibiting better readability (Section 6).
- We evaluate the efficiency of ConSol-annotated programs compared to manually implemented assertions using 16 real-world attacks and a dataset from a previous study [47], showing that ConSol only introduces up to 0.29% more gas consumption (Section 7).

We discuss the motivation and challenges in Section 2 and discuss related work in Section 8. Our prototype implementation and experiment results are available at [89].

2 MOTIVATION AND CHALLENGES

In this section, we briefly introduce the basic concepts of blockchain and the Solidity, and discuss the challenges for designing a specification and monitoring system dedicated to Solidity.

Smart Contract & Solidity. Smart contracts are self-running programs running on a blockchain (e.g., Ethereum [15]) that enforces agreements without a third party. Actions on Ethereum are conducted through *transactions* invoking functions of smart contracts. Solidity is a widely used programming language for smart contracts with a syntax similar to Java. In Solidity, a **`contract`** is organized akin to a `class` in Java, containing executable functions and data fields that are persistent on the blockchain.

```
1 function getPrice(address chainklink) returns (uint256) {
2   (_, uint256 ethPrice, _, uint256 updatedAt, _) =
3     IChainlinkAggregator(chainlink).latestRoundData();
4   require(updatedAt > block.timestamp - 1 days);
5   require(ethPrice > 0);
6   uint256 price = ORACLE.getRate() * ethPrice;
7   require(price*0.95 < ORACLE.getLatestPrice() && price*1.05 > ORACLE.getLatestPrice());
8   return price;
9 }
```

(a) Source code with a readonly reentrancy vulnerability. Lines 7 is the fix with **require**, ensuring the price fluctuation within 5%.

```
 1 getPrice(chainlink) returns (price)
 2 ensures price*0.95 < ORACLE.getLatestPrice() && price*1.05 > ORACLE.getLatestPrice()
 3 where {
 4   IChainlinkAggregator(chainlink).latestRoundData() returns (_, answer, _, updatedAt, _)
 5   ensures updatedAt > block.timestamp − 1 days && answer > 0
 6 }
 9 function getPrice(address chainlink) returns (uint256) {
10   (_, uint256 ethPrice, _, _, _) = IChainlinkAggregator(chainlink).latestRoundData();
11   return ORACLE.getRate() * ethPrice;
12 }
```

(b) The fix with ConSol specifications (lines 1–6), decoupling the specification and business logic.

Fig. 1. Comparison of fixes for Sturdy (simplified) source code with assertions and ConSol specification with enhanced readability and maintainability (see Section 6 for detailed analysis).

Role of Addresses. Addresses in Solidity, as unique identifiers for contracts or accounts, could contain callable functions, providing a way to interact with other contracts and accounts. Functions taking addresses as arguments become *higher-order*, as they can induce latent behavior depending on the address arguments. For example, addresses can be used for implementing callbacks, allowing for functions to be passed between contracts and executed as part of an atomic transaction.

However, careless use of addresses can lead to elusive vulnerabilities [61]. Developers have to carefully design assertions for address calls, examining under what condition can the address be invoked and the results of the invocation can be accepted. Moreover, the open-world distributed execution of Ethereum exposes potentially untrusted adversarial parties without disclosed implementations. Thus, security checks must be enforced when invoking functions depending on addresses. However, there are several challenges in supporting a practical behavioral contract system for Solidity.

Challenge: Writing Modular and Readable Specifications. How do programmers write down specifications of addresses? The most straightforward way is to write low-level assertions. Figure 1a shows such an example, where `require` statements in lines 4-5 are used to check the post-conditions of the address call to `latestRoundData()`. However, using assertions has two major problems. First, assertions are coupled with a specific call and are not modularly defined. If there are multiple calls to the same address, assertions have to be duplicated, considered as a bad practice violating the "don't repeat yourself" principle [1]. Secondly, and perhaps more detrimentally, these assertions are often woven directly into business logic, which places a cognitive burden on programmers and thus, often worse, leads to negligence of critical checks. As a result, the readability and maintainability of the codebase are adversely impacted.

In Section 3, we discuss the design of ConSol and demonstrate how it can disentangle smart contract specifications from the implementation.

Challenge: Tracking and Enforcing Address Behaviors. Enforcing first-order behaviors for top-level functions is straightforward, e.g., using assertions or the `modifier` mechanism [77] attached to top-level functions. However, such mechanisms cannot be directly applied to address calls because addresses are first-class values that can be used as function arguments, return values, stored into mutable states, or escaped to external contracts. This flexibility presents the technical challenge of determining when and where to check and enforce address invocation specifications. Simple syntactic identification (as `modifiers`) at call sites would not work due to indirect value flows. In addition, `modifier` mechanisms cannot check postconditions since modifiers are only invoked before executing function bodies. Static flow analysis [73] could identify the target addresses of calls, but it can be imprecise and expensive. Existing dynamic validation approaches [20, 47] focus on contract-level global invariants, lacking fine-grained tracking of address behaviors and a modular way to specify behaviors among functions.

In Section 3.3, we demonstrate how ConSol tracks and monitors latent address behaviors.

Challenge: Gas Efficiency. Moreover, compiled smart contract programs running on the Ethereum Virtual Machine (EVM) consume a finite resource known as “gas”. As a unit of measurement for computational cost and storage on the EVM, gas is used to determine the fees associated with executing transactions and running smart contracts. The gas cost for executing operations on the EVM is designed to prevent abuse and ensure that the network remains secure and efficient.

Every instruction in the EVM consumes a certain amount of gas (i.e., gas units). These instructions include simple arithmetic, address calls, storage operations, etc. Gas costs of storage operations are temporally related: the initial load/store operation in EVM is the most expensive, while subsequent accesses incur lower costs. It is also worth noting that the gas cost of storage operations is usually the largest factor. Each unit of gas has a price, denominated in ether (ETH). Users specify the gas price that they are willing to pay for each gas unit when a transaction is created.

Given the characteristics of EVM’s cost model, it is crucial to be efficient when generating code to track and enforce runtime behaviors. In Section 5, we discuss the optimization used by ConSol to ensure minimal storage and gas overhead.

3 CONSOL BY EXAMPLES

ConSol is a non-breaking extension for Solidity that additionally provides means to specify and enforce behavioral contracts (we will often use *specifications* for behavioral contracts to avoid ambiguity). We now introduce the core features of ConSol with examples.

3.1 Contracts for First-Order Values

We begin with specifying preconditions and postconditions for functions involving first-order arguments. Hereafter, we write ConSol specifications in *italic serif style*, and ordinary Solidity program in `sans-serif`. Consider the following `swap` example that swaps the amount `in` of one token to another token according to the current exchange rate. Callers can specify the minimal amount `min_out` of the output token that they expect in case the exchange rate fluctuates.

```
swap(in, min_out) returns (out)
requires in > 0
ensures out >= min_out
function swap(uint in, uint min_out) returns (uint out) { ... }
```

The first three lines are ConSol specifications. The first line of the specification introduces bindings for function `swap`’s arguments *in*, *min_out* and the return value *out*. The ***requires***-clause specifies the precondition to call `swap`, and the ***ensures***-clause specifies the postcondition. Any occurrences of *in* in the conditions refer to the actual runtime argument value from call sites, and similarly, occurrences of *out* refer to the actual return value at runtime.

Note that in the specification, binding names do not have to match those in the function definition (although in the above example, they do). Similarly, types of arguments and returned values can often be inferred from the definition, and thus are omitted.

Syntactic Sugars. We can liberally omit any part of the argument or return value specification, e.g., the following specification omits the ***requires***-clause for `toWei` (converting Ether to Wei, 1 Ether = 10^{18} Wei):

toWei(x) ***returns*** *(_)* ***ensures*** *x < type(uint).max / 1e18*

```
function toWei(uint x) returns (uint) { ... }
```

The above specification is equivalent to the core form where the omitted part is simply the Boolean **`true`** expression:

toWei(x) ***returns*** *(_)* ***requires*** *true* ***ensures*** *x < type(uint).max / 1e18*

Dependent Contract. It is possible to write specifications where the postcondition depends on the arguments. In other words, the scope of argument bindings spans both the preconditions and postconditions. For example, we can write the following condition to specify monotonicity for a numeric function:

f(x) ***returns*** *(y)* ***ensures*** *y > x*

```
function f(int x) returns (int) { ... }
```

Any Expression is Allowed. When specifying the preconditions or postconditions, programmers are free to use any valid Solidity expression, including but not limited to function calls, memory operations, and built-in special variables carrying important transaction data (e.g. `msg.value`) or metadata (e.g. `block.timestamp`) whose values are only available at run-time.

For example, the following snippet examines the amount of Wei that is carried within `msg.value` of the transaction as the precondition of `buyTickets`:

buyTickets(n) ***requires*** *msg.value >= 1e15 ∗ n*

```
function buyTickets(int n) public payable { ... }
```

This liberty empowers programmers to check any computable properties depending on dynamic information, which may be beyond the capabilities of static verification.

So far, these "flat" specifications for first-order values are straightforward. They provide a systematic and readable way to write assertions wrapping around functions.

3.2 Contracts for Higher-Order Values

We next introduce ConSol's distinct feature, i.e., to specify and enforce rich conditions for addresses.

Addresses as Numeric Values. Resembling pointers in C/C++, addresses in Solidity are numerical values that can be compared or computed. Therefore, at the first approximation, the specifications of addresses are no different from other integer values. For instance, we can assert that the address argument value must be non-zero:

f(addr) ***requires*** *addr != 0*

```
function f(address payable addr) public { ... }
```

Addresses as Latent Computation. However, addresses in Solidity have richer higher-order behaviors: they represent deployed external Ethereum contracts that can be invoked. Suppose we have a `deposit` function, which takes an address argument `token` and transfers money from it. With

ConSol, programmers can specify the condition of such latent address calls using a ***where***-clause without the need to touch the function body:

```
deposit(token, amount) requires msg.sender == owner
where {
  IERC20(token).transferFrom(sender, addr, amount) returns (success)
  requires amount > 0 ensures success
}
function deposit(address token, uint amount) public {
  IERC20(token).transferFrom(...); // the call is now guarded
}
```

When `token` is considered an instance implementing the ERC20 token standard [87], we assert that the transfer must succeed and the transferred *amount* (the third argument) must be greater than zero. We also use ***requires***- and ***ensures***-clause syntax to specify preconditions and postconditions of address calls, same as in the top-level functional specifications.

Using ConSol, we have decoupled specifying and enforcing the conditions — there is no need to change the function body or manually insert checks around the calls, resulting in more readable and maintainable code.

Single Address, Multiple Callees. ConSol is expressive to specify conditions for multiple target functions associated with a single address value. This can be done by specifying conditions for different callable targets in the ***where***-clause. Consider function `f`'s address argument `addr`, the programmer can specify conditions for `addr`'s functions `transfer` and `trade`:

```
f(addr) where
  { addr.transfer(x) returns (success) requires x<=addr.balance }
  { addr.trade(msg, x) returns (success, rate) requires x>0 ensures success }
```

This flexibility allows programmers to effectively control and enforce distinct behaviors for different target functions at a finer granularity.

Call Option Arguments. Address calls in Solidity can take additional special arguments such as `value` and `gas`:

```
addr.call{value: 100, gas: 5000}(...);
```

Programmers can use ConSol to specify the conditions of these call option arguments by introducing additional bindings using the familiar syntax that is used in Solidity:

```
addr.call{value: v, gas: g}(msg, amount) requires v>=0 && v<=addr.balance
```

where the binding names `v` and `g` represent the actual Ether transfer value and gas value, whose names can be referenced in ***requires***/***ensures***-clauses.

Low-Level Calls. We have presented how ConSol can be used to specify and enforce conditions of high-level address calls, where signatures of the callees are available. Another form of calls in Solidity is *low-level calls*, where the callee's signature and arguments are encoded as raw bytes data:

```
bytes memory data = ...
(bool success, bytes memory result) = addr.call(data)
```

With ConSol and Solidity's `decode` functionality, programmers can enforce the behaviors of low-level calls by attaching specifications to the raw data. For example, by writing preconditions as a standalone function (recall that conditions can be any expression), we can examine the encoded signatures and arguments in a separate function and check it in the ***requires***-clause:

```
function checkPreCond(bytes memory data) returns (bool) {
  // check if data encodes ERC20 protocol:
  if (bytes4(data[:4]) != bytes4(ERC20SignatureData)) return false;
  (uint256 n, address a) = abi.decode(data[4:], (uint256, address));
  ... // check the actual arguments n and a
}
```
addr.call(data) ***requires*** *checkPreCond(data) // precondition of addr.call, postcondition omitted*

Higher-Order Functions. Similar to addresses, functions in Solidity are first-class values, i.e., they can be used as arguments of, or returned from other functions. However, as of Solidity 0.8.20, the support for first-class functions is limited to explicitly defined top-level functions. Writing anonymous functions (lambda expressions), nested functions, or named functions that capture variables from environments have not been supported. This restriction prohibits real closure values with lexical scopes (as familiarized by functional programmers). While our approach to monitoring addresses could be extended to functions, due to the restriction of Solidity, higher-order functions are rarely used in Solidity programs. Therefore, in this paper, we focus on the contracts and monitoring of address values, and leave the monitoring for higher-order functions as future work once Solidity has proper support for lambda expressions [33].

3.3 Persistent Monitoring

ConSol features persistent monitoring of address specifications, i.e., guarded addresses are first-class citizens too – passing them to other functions or returning them from functions preserve the attached conditions. These conditions will be checked when the address is called, even remotely from where the conditions were attached.

Passing Guarded Addresses. In the following example, the programmer has attached specifications to the public function `deposit` but not to function `actualDeposit`. The private function `actualDeposit` consumes the address passed from `deposit` and transfer an amount of money to the address.

deposit(token, amount)
where *{ IERC20(token).transferTo(addr, amt)* ***requires*** *amt > 10 }*
```
function deposit(address token, uint amount) public {
  address token2 = token;
  actualDeposit(token2, amount - 10);
}
function actualDeposit(address token2, uint amount) private {
  IERC20(token2).transferTo(.., amount); // call happens here
}
```
Although the condition for address `token` is specified for `deposit`, the call of `transferTo` happens remotely in `actualDeposit`, via indirect value flows. ConSol ensures the precondition of the address call (*amt > 10*) attached from `deposit` is preserved and checked in `actualDeposit`.

The function `actualDeposit` may be invoked through a different control-flow path. In such cases, different conditions may be checked, depending on the actual value of the argument `token2`.

Returning Guarded Addresses. Addresses attached with specifications can be returned from functions too. For example, a function can return a guarded address whose specifications were attached at a previous point, e.g., by other functions or at the time when the address is used as an argument. Consider this identity function, which requires its address argument to be monotonic when called:
```
interface IMono { function f(int) returns (int); }
```
id(addr1) ***returns*** *(addr2)*
where *{ IMono(addr1).f(x)* ***returns*** *(y)* ***requires*** *y >= x }*
```
function id(address addr1) returns (address) { return addr1; }
```

Then, when `addr1` is returned to the caller of `id`, the same condition is still attached. At a later point, when the returned address is invoked, the monotonic check is preserved and enforced.

It is possible to directly attach conditions to the returned address. Using the same "identity" function as an example, we can specify monotonicity of the returned address *addr2* (instead of *addr1*):

id(addr1) ***returns*** *(addr2)* ***where*** *{ I(addr2).f(x)* ***returns*** *(y)* ***requires*** *y>=x }*

Similarly, when later the received address is invoked, the monotonic check is preserved and enforced:

```
address c = id(...)
int y = IMono(c).f(x) // checks condition y >= x
```

It is also possible to attach different preconditions and postconditions to the same address value.

Allowing attaching persistent specifications to return addresses is a powerful notion. Addresses may flow through the business logic of contracts. ConSol liberates developers from meticulously tracking address flows and verbosely checking the critical conditions at every address call site. Consequently, developers can focus on the overall business logic, delivering code with improved readability and maintainability (see case studies in Section 6).

Additionally, a guarded address can be stored in storage, retrieved, and called later. When the call happens, ConSol preserves the checks as they are remotely specified.

Extent of Effective Monitoring. In Section 4.3, we discuss how ConSol implements persistent address specification monitoring using a whole program transformation. Nevertheless, there are cases where tracking specifications attached to addresses becomes impossible in a distributed setting, e.g., when a guarded address value is passed to another external contract program that is unknown to ConSol. In such cases, ConSol cannot monitor how the external contract program uses the address since our approach relies on a source-to-source program transformation. However, programmers can still trust ConSol within the scope of the annotated contract program with source code available. ConSol ensures effective and persistent monitoring of address specifications as long as the address calls happen within the current contract. In Section 4.5, we depict the boundary of effective monitoring after explaining the translation semantics.

4 FORMAL MODEL

To articulate how ConSol works, we present a core model λ_{ConSol}, modeling the essence of our approach in implementing behavioral contracts for Solidity. Modulo minor syntactic difference, λ_{ConSol} is entirely a subset of Solidity. This section presents λ_{ConSol}'s abstract syntax, statics, and translation semantics, which guide the actual implementation (Section 5). We also discuss expressiveness, limitations, as well correctness issues in this section.

4.1 Syntax

Figure 2 shows the abstract syntax of λ_{ConSol}, modeling the essential parts of Solidity.

Types. λ_{ConSol}'s type universe contains integers, unsigned integers, booleans, and addresses. Aiming for minimality, we omit compound data types such as mappings and structs. However, our presented formalization can be extended with mappings and structs too.

Top-Levels. At the top level, a contract C consists of field declarations and function definitions. These fields (type-and-identifier) declarations specify storage states, which are persistent data on the blockchain. As in other languages, a function definition consists of its name, parameters, return types, and its body. To model boundaries between multiple smart contracts, a function can be either `public` or `private`. A `public` function can be called by other contracts, and `private` functions can only be called within its defining contract. In λ_{ConSol}, a specification σ is attached to every function.

$$
\begin{array}{rl}
 & n \in \mathbb{Z} \qquad b \in \mathbb{B} \qquad x, y, f, \kappa \in \mathsf{Id} \\
\text{DataTypes } t, r & := \texttt{unit} \mid \texttt{int} \mid \texttt{uint} \mid \texttt{bool} \mid \texttt{address} \mid \ldots \\
\text{Type Decl } d & := t\ x \qquad \text{Values } v := \ n \mid b \\
\text{Assignable } a & := d \mid x \qquad \text{Target } \tau := \ f \mid \kappa(x).f \\
\text{Expressions } e & := x \mid v \mid e\ op\ e \mid f(e^*) \mid \kappa(e).f(e^*) \\
\text{Statements } s & := d \mid e \mid s; s \mid a = e \mid \texttt{return}\ e \\
 & \mid\ \texttt{if}\ (e)\ \{\ s\ \}\ \texttt{else}\ \{\ s\ \} \\
\text{Spec } \sigma & := \tau(x^*) : (y^*)\,\texttt{requires}\ e\ \texttt{ensures}\ e\ \texttt{where}\ \sigma^* \\
\text{Modifiers } m & := \texttt{public} \mid \texttt{private} \\
\text{Fun Decl } d_f & := \texttt{fun}\ f(d^*) : (r^*)\ m \\
\text{Fun Def } \mathcal{F} & := \sigma\ d_f\ \{\ s\ \} \\
\text{Contract } C & := \texttt{contract}\ \kappa\ \{\ d^*;\ \mathcal{F}^*\ \} \\
\text{Interface } \mathcal{I} & := \texttt{interface}\ \kappa\ \{\ d_f^*\ \}
\end{array}
$$

Fig. 2. The abstract syntax of λ_{ConSol}.

Expression Translation $\mathsf{E}[\![\cdot]\!]$

$$
\begin{aligned}
\mathsf{E}[\![x]\!] &= x \\
\mathsf{E}[\![v]\!] &= wrap(v) \\
\mathsf{E}[\![e_1\ op\ e_2]\!] &= unwrap(\mathsf{E}[\![e_1]\!])\ op\ unwrap(\mathsf{E}[\![e_2]\!]) \\
\mathsf{E}[\![f(e, \ldots)]\!] &= f_{guard}(\mathsf{E}[\![e]\!], \ldots) \\
\mathsf{E}[\![\kappa(e_{addr}).f(e, \ldots)]\!] &= dispatch^{\kappa}_{f}(\mathsf{E}[\![e_{addr}]\!], \mathsf{E}[\![e]\!], \ldots)
\end{aligned}
$$

Statement Translation $\mathsf{S}[\![\cdot]\!]$

$$
\begin{array}{ll}
\mathsf{S}[\![t\ x]\!] = t^{\uparrow}\ x & \mathsf{S}[\![e]\!] = \mathsf{E}[\![e]\!] \\
\mathsf{S}[\![t\ x = e]\!] = t^{\uparrow}\ x = \mathsf{E}[\![e]\!] & \mathsf{S}[\![x = e]\!] = x = \mathsf{E}[\![e]\!] \\
\mathsf{S}[\![s_1\,;\ s_2]\!] = \mathsf{S}[\![s_1]\!]\,;\ \mathsf{S}[\![s_2]\!] & \mathsf{S}[\![\texttt{return}\ e]\!] = \texttt{return}\ \mathsf{E}[\![e]\!]
\end{array}
$$

$$
\mathsf{S}[\![\texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}]\!] = \texttt{if}\ (\mathsf{E}[\![e]\!])\ \{\ \mathsf{S}[\![s_1]\!]\ \}\ \texttt{else}\ \{\ \mathsf{S}[\![s_2]\!]\ \}
$$

Fig. 3. The translation semantics of λ_{ConSol} (statements and expressions).

Interfaces can be declared to specify the interaction between contracts. They contain function declarations and provide expected types of arguments and return values for address calls.

Statements and Expressions. The body of functions is simply a statement. A statement can be a type-and-identifier declaration (uninitialized), an expression (e.g. calling a function for its side effects), a composition of two statements, an assignment, a `return` statement, a `revert` statement, or a conditional statement. Both type-and-identifier declarations or identifiers can be used as assignable (i.e., left-hand side of assignments). The `revert` statement aborts the execution, reverting any changes made to the blockchain state.

An expression can be an identifier, a literal value (e.g., numbers), a binary operation, a function call, or an address call. Function calls over addresses have form $\kappa(e_1).f(e_2, \ldots)$, where e_1 is the target callee expression that yields an address value, f is the function name defined in the interface κ, and $e_2, \ldots$ denotes the arguments.

Function Translation $\mathsf{F}[\![\cdot]\!]$

```
$\mathsf{F}[\![f\,(x_1,\ldots)$ : $(y_1,\ldots)$ requires $e_1$ ensures $e_2$ where $(\sigma_1,\ldots)$
  fun $f\,(t_1\ x_1,\ldots)$ : $(r_1,\ldots)\ m$ { $s$ }$]\!]$ =
   fun $f\,(t_1\ x_1,\ldots)$ : $(r_1,\ldots)\ m$ {
     return $unwrap(f_{guard}(wrap(x_1),\ldots))$ }
   fun $f_{pre}(t_1^{\uparrow}\ x_1,\ldots)$ : () private { require($\mathsf{E}[\![e_1]\!]$) }
   fun $f_{post}(t_1^{\uparrow}\ x_1,\ldots,r_1^{\uparrow}\ y_1,\ldots)$ : () private { require($\mathsf{E}[\![e_1]\!]$) }
   fun $f_{guard}(t_1^{\uparrow}\ x_1,\ldots)$ : $(r_1^{\uparrow},\ldots)$ private {
     $f_{pre}(x_1,\ldots)$
     $attachSpec(x_1,\ldots,\sigma_1,\ldots)$
     $(r_1^{\uparrow}\ y_1,\ldots) = f_{worker}(x_1,\ldots)$
     $attachSpec(y_1,\ldots,\sigma_1,\ldots)$
     $f_{post}(x_1,\ldots,y_1,\ldots)$
     return $(y_1,\ldots)$ }
   fun $f_{worker}(t_1^{\uparrow}\ x_1,\ldots)$ : $(r_1^{\uparrow},\ldots)$ private { $\mathsf{S}[\![s]\!]$ }
```

Fig. 4. The translation semantics of $\lambda_{\textsc{ConSol}}$ (functions).

Specifications. Specifications are attached to top-level functions. Each specification σ designates a target, which is either a top-level function or a function call to addresses. When specified for addresses, an interface name is required to provide the signature of the corresponding target function. In addition, a specification introduces bindings for arguments and return values. As introduced in Section 3, the pre- and post-condition are denoted in the ***requires***-clause and ***ensures***-clause, respectively. In the ***where***-clause, programmers can specify the conditions for addresses appeared as arguments or return values, using the same syntax as that for top-level function specifications.

4.2 Static Semantics

The syntax permits arbitrary expressions to appear as pre- and post-conditions. However, not all possible expressions are valid executable specifications. For example, an arithmetic expression ***requires*** $x + 1$ is meaningless if used as a condition. The static semantics in the form of a *type system* concerns when a specification should be considered well-formed.

The general typing judgment takes the form of $\Gamma \vdash e : t$, where Γ is the typing environment, e is the program, and t is its type under Γ. For our purpose, the pre- and post-condition should be of type Boolean, and only use well-scoped variables. Since $\lambda_{\textsc{ConSol}}$ is a subset of Solidity and additionally introduces specification forms, Figure 5 only shows typing rules relevant to function and address specifications, and the typing for the rest of the language is omitted, which can be built atop other existing works [7, 26, 45, 69].

The specification typing judgment takes the form of $\Gamma \vdash \sigma : d_f$, where Γ is the typing environment, σ is the specification, and d_f is the declaration of the target function. The C-TOP rule examines contract well-formedness by checking if the specification attached to each function is well-typed. The F-SPEC rule checks if the annotated specification matches the actual function declaration. It also checks if pre-conditions, post-conditions, and address specifications are well-typed. Similarly, A-SPEC checks address specifications against the callee function defined in the declared interface.

Specification Typing

$$\frac{\begin{array}{c}\Gamma(x_{\text{addr}}) = \texttt{address} \qquad \Gamma' = \Gamma,\ \boldsymbol{x_i} : \boldsymbol{t_i} \qquad \Gamma'' = \Gamma',\ \boldsymbol{y_j} : \boldsymbol{r_j} \\ \Gamma' \vdash e_1 : \texttt{bool} \qquad \Gamma'' \vdash e_2 : \texttt{bool}\end{array}}{\begin{array}{c}\Gamma \vdash \kappa(x_{\text{addr}}).f\,(\boldsymbol{x_i})\ :\ (\boldsymbol{y_j})\ \texttt{requires}\ e_1\ \texttt{ensures}\ e_2\ : \\ \texttt{fun}\ f\,(\boldsymbol{t_i}\ \boldsymbol{x'_i})\ :\ (\boldsymbol{r_j})\ m\end{array}} \quad (\textsc{a-spec})$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma,\ \boldsymbol{x_i} : \boldsymbol{t_i} \qquad \Gamma'' = \Gamma',\ \boldsymbol{y_j} : \boldsymbol{r_j} \\ \Gamma' \vdash e_1 : \texttt{bool} \qquad \Gamma'' \vdash e_2 : \texttt{bool} \\ \forall k,\ \Gamma'' \vdash \sigma_k : d_{f_{\text{exn}}}\ \textit{s.t.}\ \text{target}(\sigma_k) = \kappa(x_{\text{addr}}).f_{\text{exn}}\ \wedge\ d_{f_{\text{exn}}} \in \kappa\end{array}}{\begin{array}{c}\Gamma \vdash f\,(\boldsymbol{x_i})\ :\ (\boldsymbol{y_j})\ \texttt{requires}\ e_1\ \texttt{ensures}\ e_2\ \texttt{where}\ \boldsymbol{\sigma_k}\ : \\ \texttt{fun}\ f\,(\boldsymbol{t_i}\ \boldsymbol{x'_i})\ :\ (\boldsymbol{r_j})\ m\end{array}} \quad (\textsc{f-spec})$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma,\ \text{id}(\boldsymbol{d_i}) : \text{type}(\boldsymbol{d_i}),\ \text{id}(\boldsymbol{\mathcal{F}_j}) : \text{decl}(\boldsymbol{\mathcal{F}_j}) \\ \forall j,\ \Gamma' \vdash \text{spec}(\mathcal{F}_j) : \text{decl}(\mathcal{F}_j)\end{array}}{\Gamma \vdash \texttt{contract}\ \kappa\ \{\ \boldsymbol{d_i};\ \boldsymbol{\mathcal{F}_j}\ \}} \quad (\textsc{c-top})$$

Fig. 5. Static semantics (excerpt) of $\lambda_{\textsc{ConSol}}$ specifications. Only checkings relevant to specifications are shown.

4.3 Translation Semantics

Figure 3 defines the semantics of specifications by transforming specification-annotated programs to ordinary programs. This process orchestrates and inserts assertions when appropriate. We use syntax-directed translation functions $\mathsf{F}[\![\cdot]\!]$, $\mathsf{S}[\![\cdot]\!]$, $\mathsf{E}[\![\cdot]\!]$ to define the translation of functions, statements, and expressions, respectively.

Translating Types. One of the key goals of ConSol is to provide persistent monitoring of address behaviors. To realize this, the translation designates a new type and value representation for a *guarded address*, so that (1) specification provenance is always kept along with the original address within the current contract, and (2) operations on the original address preserve their results.

Given an ordinary type t from untranslated programs, we use $t^{\uparrow}$ to denote its translated type. For example, `address` is the original type, and $\texttt{address}^{\uparrow}$ is the address type that can be attached with specifications. For primitive types other than `address`, $t^{\uparrow}$ can be equal to t. For compound data types such as arrays, $t^{\uparrow}$ should be recursively defined (the formalization has not modeled compound data types for brevity, but there is no technical difficulty to accommodate compound data types). Moreover, a pair of runtime functions *wrap*() and *unwrap*() is used to convert between values of type t and $t^{\uparrow}$, satisfying the following conditions:

$$\Gamma \vdash e : t \Leftrightarrow \Gamma \vdash \textit{wrap}(e) : t^{\uparrow}$$
$$\Gamma \vdash e : t^{\uparrow} \Leftrightarrow \Gamma \vdash \textit{unwrap}(e) : t$$
$$\textit{unwrap} \circ \textit{wrap} = \textit{id}$$

The conversions are useful when (1) performing primitive operations, and (2) communicating values across the boundary between ConSol's monitored world and the external unchecked world.

At this moment, we intentionally keep the notion $t^{\uparrow}$ abstract, i.e., not giving its concrete definition, since multiple representations with their wrap/unwrap functions could work. A naive way to represent a guarded address is to use a `struct` that stores both the original address and the *encoding* of the attached specifications. However, this scheme introduces additional storage overhead. In Section 5, we discuss the implementations and optimizations that do not introduce storage overhead for practical scenarios. Since the address type is pervasively used in the program, our approach is based on a whole-program translation.

Translating Functions. To translate an annotated function f, ConSol generates an interposition layer that orchestrates runtime checks (Figure 4). The main job is performed in f_{guard}, which checks pre- and post-conditions, attaches specifications (from the ***where***-clause) to address values, and invokes the actual function f_{worker}, whose body is recursively translated. We also rewrite call-sites of f so that f_{guard} is invoked. f_{pre} and f_{post} enforces the pre- and post-conditions respectively and aborts the call if a condition is violated.

Additionally, a function f with the same name and signature as the original function is generated. This is used only for external calls (if the f is public), which does not observe our translated value representation for addresses.

The attachment of specifications to addresses is performed by *attachSpec* at run-time, whose implementation depends on the representation of guarded addresses.

Translating Statements and Expressions. ConSol translates statements and expressions in a recursive fashion (Figure 3). Most cases of statement translation are straightforward; they recursively apply the translation to subcomponents. Declared types t are lifted to $t^{\uparrow}$ to carry specifications if necessary.

For binary operations, we first recursively translate the operands, and apply *unwrap* to the operands at run-time. Since the attachment of specifications to address values has changed the underlying value representation, this is necessary to ensure our translation preserves the results of these operations. For example, consider an address value x that has been attached with two different specifications, producing two different guarded address values x_1 and x_2. The programmer should expect that equality is preserved $x_1 == x_2$, which only holds after unwrapping. In general, *unwrap* should be applied when performing operations other than method calls over values.

Direct function calls are replaced with their guarded counterparts f_{guard} with arguments being translated recursively. Address calls $\kappa(e_{addr}).f$ are replaced with calls to $dispatch^{\kappa}_{f}$, which takes the address value e_{addr} in addition to the ordinary arguments as arguments. The *dispatch* function inspects the specification provenance carried along with the underlying address value and performs the corresponding pre-/post-condition checks before and after the address calls.

Runtime Facilities. Our translation works with a set of runtime functions, including *wrap* and *unwrap* that convert value representations, *attachSpec* that encodes specification provenance into the underlying guarded address values, and *dispatch* that decodes specifications and performs corresponding checked address calls. Note that *dispatch* is a family of functions that are specialized over the callee function f and its containing interface κ. Since *dispatch* makes address calls to external contract instances, it applies *unwrap* to arguments before interacting with worlds outside ConSol's monitoring boundary. In Section 5, we describe the implementation of *attachSpec* and *dispatch*.

4.4 Expressiveness and Limitations

ConSol aims to provide rich expressiveness thus allowing programmers to use the same host language to write down the pre/post-conditions as executable boolean expressions. For example, to specify the correctness of ERC20 contracts, the programmer needs to reason about the content of an entire mapping, which can be defined as a ghost state with a loop to compute the sum of a map/array. However, as a contract and monitoring system, what ConSol really provides is an interposition layer, which intercepts runtime events of interest and relays them to the monitoring system [30]. However, not all events in smart contracts can be intercepted with a reasonable cost, and the events we are interested in this work are function call/return behaviors (as in many other

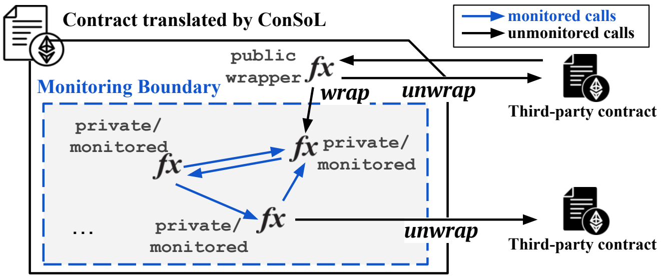

Fig. 6. Monitoring boundary of guarded address calls.

behavioral contract systems), with a more specific focus on calls made to first-class address values, which are important in smart contracts since address calls are the main way to communicate between multiple contract instances.

Although checking temporal properties is not our design goal in this paper, it is an interesting and useful category of behaviors especially for DeFi applications. In contrast to ordinary specifications that inspect a single event, "temporal" specifications inspect traces of events. Our current design does provide low-level mechanisms to encode temporal properties, which however can be inconvenient. For example, the programmer needs to manually declare state variables to capture/store the changes in the value of interest, which then can be checked in pre/post conditions. Integrating orthogonal work in temporal/trace contracts [31, 55] would be an interesting future extension for ConSol.

In terms of the obligations/guarantees of program components (client/supplier), another non-goal of our design is to enforce global invariants of smart contracts. These global invariants are usually expressed with quantifiers [47], but sometimes specify a different aspect of the specification. Additionally, inspecting local variables or intermediate results *within* a function is still useful and important, but a design-by-contract approach focusing on interfaces would not fit this purpose.

4.5 Correctness

Our translation preserves the behaviors of the original program, in the sense that if the ConSol-translated program does not raise ConSol-related errors, then we observe the same result and state on the original program. Our translation also ensures that violations will be detected (thus the execution will be reverted) *within the boundary of the current contract.* Figure 6 demonstrates the monitoring boundary of λ_{ConSol} with the blue dashed scope.

For top-level functions, it is straightforward to see that their invocations are relayed to their guarded versions, and thus are correctly monitored. For guarded addresses, there are three possible ways to unwrap the value (losing the attached specification) in our translation: (1) the address value is used in primitive operations (e.g. checking equality) within the current contract, (2) the address value is returned to other contract instances via a public function (Figure 4), and (3) the address value is passed as an argument (thus unwrapped by *dispatch*) to other contract instances. For case (1), the unwrap is benign and it does not hinder our guarantee to monitor potential violations. For cases (2) and (3), we say the guarded address has escaped the monitoring boundary (demonstrated in Figure 6), and there is no way to examine its behavior once the address is used in other closed-source smart contract programs. However, if the source code of these contracts is available, our approach can be indeed extended to multiple contracts, which would provide a guarantee to monitor address calls across multiple contracts.

Our formalization also translates user-provided condition expressions (Figure 4), which provides a stricter semantics (similar to the "picky" contract semantics [11]) to allow using guarded addresses to define conditions, which can capture more potential violations compared to "lax" contract semantics.

Table 1. Summary of studied cases. *LR* denotes LoC Reduced.

Project	Date	Loss ($)	Root Cause of Vulnerability	LR (%)
Qubit [17]	01-28-22	80M	Zero Address Function Call	15.38
TecraSpace [80]	02-04-22	63K	Any Token is Destroyed	50.00
Umbrella [86]	03-20-22	700K	Integer Over/Underflow	33.33
XCarnival [90]	06-26-22	3.87M	Infinite Number of Loans	26.32
BadGuys [65]	09-02-22	NFT	Missing Airdrop Eligibility Check	94.12
EFLeverVault [50]	10-14-22	1M	Business Logic Flaw	25.00
N00d [8]	10-26-22	29K	Reentrancy	11.11
Dexible [61]	02-17-23	1.5M	Arbitrary External Call	11.76
SushiSwap [72]	04-09-23	3.3M	Unchecked User Input	54.55
SwaposV2 [18]	04-16-23	468K	Erroneous Accounting	25.00
Unknown [81]	05-31-23	111K	Missing Slippage Check	30.00
Sturdy [9]	06-12-23	800K	Readonly Reentrancy	57.14
LEVUSDC [28]	06-15-23	105K	Access Control	33.33
AzukiDAO [70]	07-03-23	69K	Invalid Signature Verification	48.15
Bao [6]	07-04-23	46K	Inflation Manipulate	83.33
Miner [54]	02-15-24	466K	Lack of Validation	83.33
YearnFinance [5]	04-13-23	11.6M	Misconfiguration	-
ZunamiProtocol [44]	08-14-23	2M	Price Manipulation	-
KyberSwap [10]	11-22-23	48M	Precision Loss	-
Time [67]	12-06-23	188.9K	Arbitrary Address Spoofing Attack	-

5 OPTIMIZING GAS EFFICIENCY IN IMPLEMENTATION

Implementation. We implement ConSol as a preprocessor of Solidity programs annotated with ConSol's specifications. Given an input program, ConSol generates a Solidity program following the translation outlined in Section 4. The output programs can be compiled and deployed using off-the-shelf Solidity toolchains without any further modification. ConSol is implemented with `solc-typed-ast` [25], which can reify a modified AST back to a Solidity program.

Our implementation handles a few more Solidity features that are not covered by the formalization. For example, programmers can attach specifications to storage fields, which can be desugared to core forms. This is useful to produce guarded addresses at contract initialization time.

Optimizations. As discussed in Section 4.3, the translation presented in Figure 3 works with a set of runtime functions. These functions cooperate with the runtime representation of guarded addresses. We now discuss an optimized implementation that *does not incur additional storage overhead* in the generated code.

The optimization exploits the fact that the smallest unit for storage is 256 bits in the Ethereum Virtual Machine. In fact, storing or loading data that are smaller than 256 bits incurs the same gas overhead as that of 256 bits. Moreover, the size of an address is 160 bits, which spares an additional 96 bits that can be used to encode specifications up to 96 predicates. Therefore, we use **`uint256`** to represent guarded addresses, where the 96 most significant bits (MSBs) encode the attached specifications and the remaining 160 bits are preserved for the original address.

With this representation, our translation assigns numeric identities to possible conditions that will be attached to addresses. The correspondence between the assigned numeric identities and their runtime checking functions are emitted in the generated code, which is used for dispatching checks. Those runtime functions can be implemented straightforwardly: The *wrap* function coerces a **`uint160`** to **`uint256`**, and *unwrap* function discards the 96 MSBs of an **`uint256`** value. The *attachSpec* function only modifies the 96 MSB of a guarded address representation, i.e., sets the bit that represents the intended condition. The *dispatch* function can check which bits in the 96 MSBs of the guarded addresses are set, therefore delegating the actual validation to the runtime predicate function.

```
IERC20(token).transferFrom(_, _, _) returns (ret) requires ret
IERC20 public immutable token;
withdraw(n, user) requires canOperate(msg.sender, user) && balances[user] >= n
function withdraw(uint256 n, address user) {
  require(canOperate(msg.sender, user));
  balances[user] = balances[user] - n;
  bool success = token.transferFrom(address(this), user, n);
  require(success)
}
function deposit(uint256 n, address user) {
  bool success = token.transferFrom(user, address(this), n);
  require(success);
}
```

Fig. 7. Simplified code from Umbrella. The patched ConSol specification is highlighted in blue, eliminating the low-level assertions (in red). The additional condition to fix the bug is underlined.

Of course, this approach is limited by the total number that can be squeezed into 96 bits. It is straightforward to see that the total number of conditions attached to all addresses cannot exceed 96. However, so far, we have not encountered real cases that need to use more than 96 address conditions. If there are more conditions specified by the user, a translation could make use of more bits to store the encoding of specifications, at the cost of more gas consumption.

6 CASE STUDIES

This section showcases ConSol's expressiveness and effectiveness using 20 real-world smart contract attacks ($154.32M total loss) and defenses. We first provide an overview of the case study, then delve into three representative cases, illustrating how ConSol specifications concisely and non-intrusively express security defenses, decoupling them from the primary business logic.

Benchmark Selection. We select a set of benchmarks and attacks from a reputable database of blockchain incidents [82], which is widely used in the literature [46, 92, 93] to study security issues in smart contracts. Our selection emphasizes the prevalence of the case in practice, the diversity of root causes, and the impact (e.g. value of loss) of these attacks. As a result, we select 20 past incidents with attack types and losses summarized in Table 1. The first three columns of the table present the project names, attack dates, and their corresponding financial losses.

Methodology. We analyze each attack, identify the root cause, and pinpoint the vulnerable function. We first evaluate whether it is possible to use assertions to patch the underlying vulnerability for each attack. If so, we implement the most appropriate patch using low-level assertions (as suggested by postmortem incident reports from third-party auditors [24, 85]), and then migrate the patch using ConSol. To validate the patches, we employ an Ethereum Archive Node [59] to replay all historical transactions on both patched contracts (assertion-based and ConSol-based), ensuing legitimate transactions succeed while blocking the malicious ones.

Results Overview. We conclude that 16 of the attacks (i.e., the upper part of Table 1) can be prevented using ConSol, while the remaining 4 attacks (i.e., the lower part in the table) cannot be prevented with assertions, thus ConSol cannot be used to defend them either. Column "LR" in Table 1 denotes the percentage of LoC reduced from migrating assertion-based patches to ConSol specifications, if applicable. On average 42.6% LoC in the assertion-patched functions can be expressed as specifications with ConSol. Since a substantial portion of these functions is dedicated

Table 2. Gas consumption on vulnerable contracts in $\mathcal{D}_2$, patched with assertions and ConSol. GFI: gas fee increase ($). GIR: gas increase ratio after patching to the original. Avg calculation excludes Qubit and TecraSpace.

Project	#Tx	by Assertions		by ConSol	
		GFI ($)	GIR (%)	GFI ($)	GIR (%)
Qubit [17]	0	-	-	-	-
TecraSpace [80]	4245	0.000	0.000	0.000	0.000
Umbrella [86]	58	0.001	0.111	0.001	0.015
XCarnival [90]	365	0.016	0.029	0.040	0.072
BadGuys [65]	950	0.003	0.096	0.005	0.166
EFLeverVault [50]	21	0.027	0.089	0.031	0.102
N00d [8]	111	0.009	0.547	0.009	0.571
Dexible [61]	54	0.126	0.230	0.178	0.324
SushiSwap [72]	202	0.007	0.099	0.007	0.106
SwaposV2 [18]	7	0.003	0.048	0.004	0.068
Unknown [81]	10	0.381	0.002	0.438	0.003
Sturdy [9]	23	0.940	1.126	0.941	1.128
LEVUSDC [28]	45	0.008	0.042	0.008	0.044
AzukiDAO [70]	2937	0.019	0.227	0.022	0.257
Bao [6]	15	0.001	0.005	0.002	0.018
Miner [54]	3922	0.002	0.007	0.011	0.030
Avg.	-	0.110	0.190	0.121	0.207

to validation, ConSol appears to be an ideal tool to decouple validations from business logic, hence improving readability and maintainability.

Four attacks cannot be defended with assertions for various reasons. For the YarnFinance [5] attack, the contract is not properly configured and deployed, so even assertions cannot be used to check if the contract is configured properly. The ZunamiProtocol [44] attack is caused by price manipulation, where the victim contract depends on the token price provided by external contracts, which are manipulable by attackers. The KyberSwap [10] contract improperly rounds down integer division calculations in its logic, thus causing a precision loss in the contract state. The vulnerability behind ZunamiProtocol and KyberSwap is caused by the design flaws of the contract and thus cannot be patched with assertions. The Time [67] attack is caused by a vulnerability in the Openzeppelin library used in the contract. The vulnerability lies in the upstream of the software supply chain and cannot be fixed in the attacked contract itself.

Case 1: Integer Underflow. We first use integer underflow, a notorious vulnerability type, as an example to demonstrate ConSol's effectiveness in expressing the defense. Figure 7 depicts a simplified code snippet from the Umbrella project. Specifically, the Umbrella project offers a staking service where users can stake and unstake their `tokens` through the `deposit` (lines 11-14) and `withdraw` (lines 5-10) functions, respectively. The `withdraw` function updates the `balances` of a user (line 7), transfers the staked tokens (line 8), and finally checks if the transfer is successful (line 9). We skip the details of the `deposit` function, except for its operation involving the transfer of tokens (lines 12-13), as well as several other functions operating on token transfers for simplicity.

Attack. The vulnerability resides at line 5, where the attacker attempts to withdraw a large quantity of their staked tokens. This action induces an integer underflow in `balance[user]-n`, leading to an anomalously large `balance[user]` for the attacker. Consequently, the attacker can transfer as many tokens as they wish.

ConSol-Patch. One effective patch to fix this vulnerability is to guard `withdraw` with a precondition `balance[user] >= n`. However, considering the number of security validations that must be placed around every invocation of `token.transferFrom`, there is a heightened risk that developers overlook these subtle checks for integer underflow. As depicted in Figure 7, ConSol's solution to this issue is to attach specifications to both function `withdraw` and calls of address `token` (line 1). As a result, there is no need to repeatedly write validation code checking the same condition in multiple locations, since the storage address is persistently guarded, thanks to our whole-program translation.

This strategy reduces duplicated assertions currently interspersed across various invocation sites. Moreover, by moving all validation logic from `withdraw` function to modularly defined specifications, it enhances the readability of both the business and validation logic.

Case 2: Readonly Reentrancy. Readonly reentrancy represents another notorious class of vulnerabilities. We use the Sturdy [34] project to illustrate how ConSol can enhance readability and potentially help prevent such vulnerabilities.

Attack. Figure 1a shows the simplified `getPrice` function from the Sturdy contract, which determines the price of the Sturdy token. The returned price of `getPrice` function is proportional to Ether's price, which ratio is informed by `ORACLE.getRate` (line 6). However, the return value of `ORACLE.getRate` (Figure 8) can be manipulated by attackers, exploiting the fact that `getRate` (line 1, Figure 8) returns a ratio of `this.balance / totalSupply`. The ratio can be manipulated by providing a malicious callback to the `withdraw` function (line 2-6, Figure 8), which first employs a low-level call to transfer the requested Ether amount back to `msg.sender`. Nevertheless, this low-level call allows `msg.sender` to callback `getPrice` before `withdraw` finishes its update to `totalSupply`. At this moment, the nominator of the ratio, `this.balance`, has been lowered by the Ether transfer, causing that `getRate` yields an abnormally lower value. This, in turn, leads to `getPrice` returning a deflated value. Given that smart contracts rely on precise token prices, such miscalculations due to this vulnerability can result in significant financial losses.

ConSol-Patch. The fix to this issue is to cross-validate `getPrice` with the latest Sturdy price, which can be accessed from `ORACLE.getLatestPrice`. In ConSol, this check can be concisely expressed by a post-condition (line 2 in Figure 1b).

Case 3: Refactoring Uniswap. We also conduct an empirical study to see how ConSol can be used in supporting smart contract development process. We report our experience in refactoring the Uniswap V2 [2] and V3 [3] protocols using ConSol.

Our focus is specifically on refactoring the `UniswapV2Pair` and `UniswapV3Pool` contracts while leaving other periphery and library contracts unchanged. The `UniswapV2Pair` and `UniswapV3Pool` contracts contained 10 and 20 primitive assertions in their bodies, respectively. Our goal is to lift these primitive assertions to ConSol specifications (for functions and addresses). Remarkably, we lift all primitive assertions in the `UniswapV2Pair` contract and 14 out of 20 (70%) in the `UniswapV3Pool` contract to ConSol specifications. Our investigation indicates that the high lifting rates are due to the practice of safe coding pattern of *Checks-Effects-Interactions* [75] in the existing Uniswap code base, which aligns closely with the design-by-contract principle, underscoring the practical potential of ConSol.

For the remaining six assertions in the UniswapV3Pool contract, we identified a common pattern involving effectful flow-sensitive properties, exemplified in the following snippet:

```
uint256 balance0Before = balance0();
IUniswapV3SwapCallback(msg.sender).uniswapV3SwapCallback(amount0, amount1, data);
require(balance0Before.add(uint256(amount0)) <= balance0(), 'IIA');
```

Since ConSol currently does not directly support temporal specifications, lifting these assertions, although doable, is not straightforward. We leave the support for temporal contracts as future work.

```
function getRate() { return this.balance / totalSupply; }
function withdraw() nonReentrancy {
  uint balance = this.balance;
  msg.sender.call{value: balanceOf[msg.sender]}();
  balanceOf[msg.sender] = 0;
  totalSupply -= amount * totalSupply / balance;
}
```

Fig. 8. The code snippet of ORACLE used by Sturdy.

Our initial experience in refactoring the Uniswap contracts demonstrates that the design-by-contract is a suitable approach to improving the development process for better clarity, aligned with previous studies [19, 52]. A large-scale empirical study remains valuable for future work.

Table 3. Average transaction gas consumption on contracts of dataset $\mathcal{D}_1$ [47]. GIR: gas increase ratio on ConSol-implemented contracts compared to the original. LR: percentage of lines in functions that are expressed in ConSol specifications.

Contract	BEC	USDT	ZRX	THETA	INB	HEDG	DAI	EKT	XIN	HOT	SWP	VOTE
Original	960,999	62,426	51,468	51,540	53,738	53,941	53,696	51,911	51,375	51,525	55,728	210,395
ConSol	965,478	62,910	51,468	51,777	53,986	54,110	53,865	52,307	51,609	51,773	55,886	210,543
GIR (%)	0.47	0.78	0.00	0.46	0.46	0.31	0.31	0.76	0.46	0.48	0.28	0.07
LR (%)	39.10	30.83	0.00	38.89	38.89	50.00	50.00	40.83	34.44	44.44	50.00	20.83

Contract	DOZ	MCHH	CC	CLV	LAND	CARDS	KB	TRINK	PACKS	BKC	EGG
Original	2,163,066	221,235	214,598	246,986	215,732	214,770	214,731	214,484	260,566	301,808	215,428
ConSol	2,166,247	221,526	215,082	247,471	216,201	215,123	215,079	214,965	260,920	302,199	215,791
GIR (%)	0.15	0.13	0.23	0.20	0.22	0.16	0.16	0.22	0.14	0.13	0.17
LR (%)	43.19	33.33	35.71	41.67	33.33	33.81	37.05	35.71	33.33	37.05	37.50

7 GAS EFFICIENCY

Although using ConSol could decouple the specifications from the business logic, it may introduce additional runtime overhead (i.e. gas consumption) due to the instrumentation. In this section, we show that using ConSol to monitor function specifications only induces marginal overhead.

Dataset. We use two datasets to evaluate the gas efficiency of ConSol. First, we leverage the dataset collected by Li et al. [47], which consists of 23 contracts in ERC20, ERC721, and ERC1202 standards ($\mathcal{D}_1$). Second, we leverage the contracts of the 16 real-world attacks collected in Section 6, which ConSol is applicable to defend ($\mathcal{D}_2$). We collect the historical transactions on Ethereum invoking the corresponding vulnerable contracts to evaluate the gas consumption of ConSol.

Baseline and Methodology. The baseline we compare ConSol with is the contracts with low-level assertions directly implemented as pre-/post-conditions in the function body or around address calls. Specifically, for dataset $\mathcal{D}_1$, the baseline is the original contracts collected by Li et al. [47]. Specifications are already implemented in the original contracts using `require` or `assert` statements. To evaluate the gas efficiency of ConSol, for each contract, we manually re-implement it by extracting the assertion-based pre-/post-conditions and expressing them in ConSol specifications, while preserving the original contract logic. We leverage the test cases (transactions) provided by Li et al. [47] to execute and evaluate the gas consumption of the baseline and ConSol. For dataset $\mathcal{D}_2$, we manually patch the vulnerable contracts using assertions and ConSol specifications (as in Section 6), respectively. We replay all historical transactions on Ethereum that cover the patched functions and measure the gas consumption of the baseline and ConSol-patched version.

Results. Table 3 shows the comparison of gas consumption on dataset $\mathcal{D}_1$. Rows *Original* and *ConSol* show the average transaction gas consumption on the original contracts and ConSol-patched contracts, respectively. The gas increase ratio of ConSol-patched contracts compared to the original contracts is given in the third row *GIR*. The results show that the overhead of ConSol specifications is marginal – only 0.29% more gas on average for each contract.

Table 2 shows the comparison of gas consumption on dataset $\mathcal{D}_2$. Column *GFI* and *GIR* show the average increased gas fee (in US dollars) and gas consumption increase ratio of transactions on the ConSol-patched contracts compared to the original vulnerable contracts. Columns *by Assertions* and *by ConSol* show the GFI and GIR on the baseline (assertion-patched contracts) and ConSol-patched contracts. Column *#Tx* gives the total number of historical transactions executed on the patched functions. Similar to the experiments on dataset $\mathcal{D}_1$, patching vulnerable contracts using ConSol specifications only has insignificant additional gas overhead (avg. 0.207%), corresponding to at most \$0.94 more transaction fees (Sturdy). Compared to the baseline, ConSol specifications induce a small increase in gas, thanks to the optimizations that avoid additional storage overhead in the generated code (Section 5). The overhead is induced by the additional private function calls generated in the translation of functions with ConSol specifications. Such overhead is insignificant since the private function calls are compiled into cheap JUMP instructions.

There are two special cases in Table 2: Qubit [17] and TecraSpace [80]. Qubit has no transactions replayed because the original vulnerable function was only called once in history and this invocation was the attack on the contract. Hence, we cannot measure the gas overhead of assertion-patched and ConSol-patched versions of Qubit. The ConSol-patched version of TecraSpace does not introduce any new assertions compared to the original vulnerable contract. Instead, one of the preconditions is modified to fix the vulnerability. The overhead of such a minor change is further eliminated by compilation optimization.

On dataset $\mathcal{D}_1$ we also report the percentage of lines in original functions that are expressed as ConSol specifications (Column *LR* in Table 3). Similar to the results in Table 1, a large portion of code in functions can be extracted as ConSol specifications. This indicates the ConSol is effective in separating specifications from business logic in functions.

8 RELATED WORK

Specification Languages and Behavioral Contracts. The Eiffel programming language pioneered the "design-by-contract" methodology [51, 53], advocating the idea of setting clear expectations between software components right at the outset of software development. Various languages have extensions with behavioral contracts, including but not limited to Java [14], C++ [16], Python [60], Haskell [91], Racket [35], Elixir [62], etc.

Findler and Felleisen [35] propose to extend the notion of behavioral contracts to higher-order functions, which has been implemented in Racket. ConSol borrows ideas from contracts for higher-order functions to monitor address specifications, given their similar higher-order essence. Strickland et al. [79] further refined the notion of higher-order contracts to chaperones and impersonators. A future direction of ConSol is to extend it with higher-order temporal/trace contracts [31, 55], which would be useful for checking temporal violations (e.g., reentrancy) in smart contracts.

In the context of smart contracts, Li et al. [47] allow users to specify global invariants using quantifiers. Scribble [68] is a specification language that facilitates property-based testing, fuzzing, and runtime validation of Solidity programs. A Scribble specification along with a Solidity program can be translated to assertions that check the specification. SmartPulse's SmartLTL [78] is a specification language for Solidity based on linear temporal logic. While ConSol does not directly specify global invariants or temporal invariants, it can be seen as a superset of what Scribble can describe with an additional focus on monitoring address calls/returns. The K-Framework [21] has been applied to model EVM semantics [43] and specifications of smart contracts (e.g. for ERC20 [66]).

Runtime Validation/Verification of Smart Contracts. Runtime validation of smart contracts is the most relevant direction to ConSol. Similar to ConSol that enforces pre-/post-conditions, Ellul and Pace [32] propose ContractLarva and violation resolution procedures in runtime verification when specifications are violated. ConSol is different from ContractLarva in that we provide finer-grained property checking with both function-level and persistent address call checks. Li et al. [47] propose Solythesis to translate user-specified contract-level invariants into solidity code via delta update and delta check. Chen et al. [20] propose a declarative smart contract language in the style of Datalog, which also inserts runtime checks after compilation. Both ContractLarva, Solythesis, and ConSol are source-to-source translation tools, that only rely on the standard Solidity runtime. As a quite different implementation strategy, there are also EVM modifications that can reduce the overhead of runtime checks [49, 64].

Static Verification of Smart Contracts. Orthogonal to ConSol, static analysis and verification of smart contract programs are intensively studied, and many of the static analysis techniques can be very effective for certain types of attacks. Tolmach et al. [84] survey the use of formal specification and verification techniques in securing smart contracts. Statically checked refinement types [23, 83] allow developers to write specifications as part of types. Bräm et al. [12] propose a specification methodology to capture the intended behaviors of contracts under development, as well as external unverified contracts. Grossman et al. [42]'s work detects callback-related vulnerabilities that ConSol can dynamically prevent. Recent studies have also focused on modeling Solidity semantics [7, 26, 45, 69], and analyze flaws and vulnerabilities using pre-defined oracles [4, 13, 37, 39, 48, 63, 73], focusing on aspects such as insecure payment [88], high-value vulnerabilities [74], and access control [38]. In contrast, ConSol as a programmer-oriented language extension, can be repurposed to express specifications for static verification. It would be interesting for future work to explore how ConSol's address specification can be statically or gradually verified (e.g. by adopting existing soft verification techniques [57, 58] for higher-order behavioral contracts).

Testing and Fuzzing of Smart Contracts. There has been a rich effort for smart contract testing and fuzzing. Foundry [36] provides a toolkit that includes Forge, which offers property-based testing and code coverage analysis. Echidna [40] is a property-based fuzzing tool that generates random transactions to test contract properties. Medusa [27] and echidna-parade [41] extend Echidna's fuzzing capabilities, with a Go-Ethereum-based fuzzer and distributed execution. Smartian [22] combines static and dynamic dataflow analysis for fuzzing smart contracts. ItyFuzz [71] is a snapshot-based fuzzer for smart contracts that focuses on state and dataflow analysis. Compared to fuzz testing, which detects vulnerabilities through (partially-)randomized inputs, ConSol provides a notion to specify expected properties, which can be used for both runtime enforcement and guiding testing and fuzzing, which is not explored in the current work.

9 CONCLUSION

In this paper, we propose a specification and monitoring system, ConSol, for the Solidity smart contract programming language. ConSol supports attaching and enforcing specifications for both top-level functions and address values. ConSol persistently monitors address calls via a whole-program transformation, which ensures any violation of address call conditions in the current contract scope is captured. We examine the effectiveness and gas efficiency using 20 real-world attacks. By replaying existing attack transactions, ConSol-patched contracts successfully defend the attack with only marginal additional gas consumption.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers of PLDI for their constructive comments and feedback.

REFERENCES

[1] 2000. Bookshelf - The Pragmatic Programmer: From Journeyman to Master, Introduction to the Team Software Process. *IEEE Softw.* 17, 6 (2000), 108–110.

[2] Hayden Adams, Noah Zinsmeister, and Dan Robinson. 2020. Uniswap v2 Core. (2020).

[3] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. 2021. Uniswap v3 core. *Tech. rep., Uniswap, Tech. Rep.* (2021).

[4] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020. Taming callbacks for smart contract modularity. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 209:1–209:30.

[5] Beosin Alert. 2023. https://twitter.com/BeosinAlert/status/1646481687445114881.

[6] Chickn Bao. 2023. Analysis and Response to the July 4th baoETH Exploit. https://medium.com/baomunity/analysis-and-response-to-the-july-4th-baoeth-exploit-3d60b886fcce.

[7] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. 2019. A Minimal Core Calculus for Solidity Contracts. In *DPM/CBT, ESORICS (Lecture Notes in Computer Science, Vol. 11737)*. Springer, 233–243.

[8] Block Sec. 2022. https://twitter.com/BlockSecTeam/status/1584959295829180416.

[9] BlockSec. 2023. https://twitter.com/BlockSecTeam/status/1668084629654638592.

[10] BlockSec. 2023. https://blocksec.com/blog/yet-another-tragedy-of-precision-loss-an-in-depth-analysis-of-the-kyber-swap-incident-1.

[11] Matthias Blume and David A. McAllester. 2006. Sound and complete models of contracts. *J. Funct. Program.* 16, 4-5 (2006), 375–414.

[12] Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers. 2021. Rich specifications for Ethereum smart contract verification. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30.

[13] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *PLDI*. ACM, 454–469.

[14] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2005. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.* 7, 3 (2005), 212–232.

[15] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014), 2–1.

[16] Lorenzo Caminiti. 2023. *Boost.Contract.* https://www.boost.org/doc/libs/1_82_0/libs/contract/doc/html/index.html

[17] CertiK. 2023. https://certik.medium.com/qubit-bridge-collapse-exploited-to-the-tune-of-80-million-a7ab9068e1a0.

[18] CertiK Alert. 2023. https://twitter.com/CertiKAlert/status/1647530789947469825.

[19] Patrice Chalin. 2006. Are Practitioners Writing Contracts?. In *RODIN Book (Lecture Notes in Computer Science, Vol. 4157)*. Springer, 100–113.

[20] Haoxian Chen, Gerald Whitters, Mohammad Javad Amiri, Yuepeng Wang, and Boon Thau Loo. 2022. Declarative smart contracts. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 281–293.

[21] Xiaohong Chen and Grigore Rosu. 2019. $\mathbb{K}$ - A Semantic Framework for Programming Languages and Formal Analysis. In *SETSS (Lecture Notes in Computer Science, Vol. 12154)*. Springer, 122–158.

[22] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239.

[23] Michael Coblenz. 2017. Obsidian: A Safer Blockchain Programming Language. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 97–99. https://doi.org/10.1109/ICSE-C.2017.150

[24] Consensys. 2023. Consensys. https://www.consensys.net/.

[25] Consensys. 2023. solc-typed-ast. https://github.com/Consensys/solc-typed-ast.

[26] Silvia Crafa, Matteo Di Pirro, and Elena Zucca. 2019. Is Solidity Solid Enough?. In *Financial Cryptography Workshops (Lecture Notes in Computer Science, Vol. 11599)*. Springer, 138–153.

[27] crytic. 2024. medusa. https://github.com/crytic/medusa.

[28] Numen Cyber. 2023. https://twitter.com/numencyber/status/1669278694744150016?cxt=HHwWgMDS9Z2IvKouAAAA.

[29] Phil Daian. 2016. The analysis of the DAO exploit. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[30] Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. 2016. Oh Lord, please don't let contracts be misunderstood (functional pearl). In *ICFP*. ACM, 117–131.

[31] Tim Disney, Cormac Flanagan, and Jay McCarthy. 2011. Temporal higher-order contracts. In *ICFP*. ACM, 176–188.
[32] Joshua Ellul and Gordon J. Pace. 2018. Runtime Verification of Ethereum Smart Contracts. In *EDCC*. IEEE Computer Society, 158–163.
[33] Ethereum. 2023. Solidity Documentation. https://docs.soliditylang.org/en/v0.8.21/types.html.
[34] Sturdy Finance. 2023. Sturdy Finance. https://sturdy.finance/.
[35] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *ICFP*. ACM, 48–59.
[36] fountry-rs. 2024. Foundry. https://github.com/foundry-rs/foundry.
[37] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. eTainter: detecting gas-related vulnerabilities in smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 728–739.
[38] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. *Proc. ACM ICSE* (2023).
[39] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 116:1–116:27.
[40] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 557–560.
[41] Alex Groce and Gustavo Grieco. 2021. echidna-parade: A tool for diverse multicore smart contract fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 658–661.
[42] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.* 2, POPL (2018), 48:1–48:28.
[43] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *CSF*. IEEE Computer Society, 204–217.
[44] PeckShield Inc. 2023. https://twitter.com/peckshield/status/1690877589005778945.
[45] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanán, Yang Liu, and Jun Sun. 2020. Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. In *IEEE Symposium on Security and Privacy*. IEEE, 1695–1712.
[46] Ping Fan Ke and Ka Chung Boris Ng. 2022. Bank Error in Whose Favor? A Case Study of Decentralized Finance Misgovernance. In *ICIS*. Association for Information Systems.
[47] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing smart contract with runtime validation. In *PLDI*. ACM, 438–453.
[48] Zeqin Liao, Sicheng Hao, Yuhong Nan, and Zibin Zheng. 2023. SmartState: Detecting State-Reverting Vulnerabilities in Smart Contracts via Fine-Grained State-Dependency Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 980–991.
[49] Fuchen Ma, Ying Fu, Meng Ren, Mingzhe Wang, Yu Jiang, Kaixiang Zhang, Huizhong Li, and Xiang Shi. 2019. EVM*: From Offline Detection to Online Reinforcement for Ethereum Virtual Machine. In *SANER*. IEEE, 554–558.
[50] MevRefund. 2022. https://twitter.com/MevRefund/status/1580917351217627136.
[51] Bertrand Meyer. 1991. *Eiffel: The Language*. Prentice-Hall.
[52] Bertrand Meyer. 1997. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall.
[53] Bertrand Meyer. 1998. Design by Contract: The Eiffel Method. In *TOOLS (26)*. IEEE Computer Society, 446.
[54] Miner. 2023. https://twitter.com/minerercx/status/1757787864299934023.
[55] Cameron Moy and Matthias Felleisen. 2023. Trace contracts. *J. Funct. Program.* 33 (2023).
[56] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review* (2008).
[57] Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2018. Soft contract verification for higher-order stateful programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 51:1–51:30.
[58] Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft contract verification. In *ICFP*. ACM, 139–152.
[59] Paradigm. 2023. Reth: Modular, contributor-friendly and blazing-fast implementation of the Ethereum protocol. https://github.com/paradigmxyz/reth.
[60] Parquery. 2023. *icontract*. Accessed: July 13, 2023.
[61] PeckShield Inc. 2023. https://twitter.com/peckshield/status/1626493024879673344.
[62] Sergio Pérez, Luis Eduardo Bueso de Barrio, Ignacio Ballesteros, Ángel Herranz, Julio Mariño, Clara Benac Earle, and Lars-Åke Fredlund. 2022. Executable contracts for Elixir. In *Erlang Workshop*. ACM, 40–46.
[63] George Pîrlea, Amrit Kumar, and Ilya Sergey. 2021. Practical smart contract sharding with ownership and commutativity analysis. In *PLDI*. ACM, 1327–1341.

[64] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *NDSS*. The Internet Society.

[65] RugDoctorApe. 2023. https://twitter.com/RugDoctorApe/status/1565739119606890498.

[66] Runtime Verification. 2018. ERC20-Semantics: Formal semantics and verification properties for ERC20 smart contracts. https://github.com/runtimeverification/erc20-semantics. Accessed on 2024-03-24.

[67] S7iter. 2023. https://medium.com/@S7iter_/erc2771-multicall-arbitrary-address-spoofing-attack-analysis-and-recurrence-48c57fdb9a98.

[68] Scribble. 2024. *Scribble Documentation.* https://docs.scribble.codes/

[69] Ilya Sergey. 2021. *The Next 700 Smart Contract Languages.* Springer International Publishing, Cham, 69–94. https://doi.org/10.1007/978-3-031-01807-7_3

[70] SharkTeam. 2023. https://app.chainaegis.com/home/news/detail?contentId=341&lang=en-US.

[71] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis.* 322–333.

[72] SlowMist. 2023. https://twitter.com/SlowMist_Team/status/1644936375924584449.

[73] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. 2021. Symbolic value-flow static analysis: deep, precise, complete modeling of Ethereum smart contracts. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30.

[74] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Tony Rocco Valentine. 2023. Program Analysis for High-Value Smart Contract Vulnerabilities: Techniques and Insights. (2023).

[75] Solidity. 2023. https://docs.soliditylang.org/en/develop/security-considerations.html#use-the-checks-effects-interactions-pattern.

[76] Solidity. 2023. *Solidity Contracts: Function Modifiers.* https://docs.soliditylang.org/en/latest/contracts.html#function-modifiers Accessed on 2023-11-17.

[77] Solidity Developers. 2023. *Solidity Documentation: Function Modifiers.* Solidity. https://docs.soliditylang.org/en/v0.8.20/contracts.html#modifiers Accessed on July 20, 2023.

[78] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu K. Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *SP*. IEEE, 555–571.

[79] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and impersonators: run-time support for reasonable interposition. In *OOPSLA*. ACM, 943–962.

[80] SunWeb3Sec. 2022. https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/past/2022/README.md#20220204-tecraspace---any-token-is-destroyed.

[81] SunWeb3Sec. 2023. https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/ERC20TokenBank_exp.sol.

[82] SunWeb3Sec. 2023. DeFi Hacks Reproduce - Foundry. https://github.com/SunWeb3Sec/DeFiHackLabs.

[83] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. 2022. SolType: refinement types for arithmetic overflow in solidity. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29.

[84] Palina Tolmach, Yi Li, Shangwei Lin, Yang Liu, and Zengxiang Li. 2022. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 54, 7 (2022), 148:1–148:38.

[85] Trail of Bits. 2023. Trail of Bits. https://www.trailofbits.com/.

[86] Uno.Reinsure. 2022. Umbrella Network Hacked: $700K Lost. https://medium.com/uno-re/umbrella-network-hacked-700k-lost-97285b69e8c7.

[87] Fabian Vogelsteller and Vitalik Buterin. 2023. EIP-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20.

[88] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting nondeterministic payment bugs in Ethereum smart contracts. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 189:1–189:29.

[89] Guannan Wei, Danning Xie, Wuqi Zhang, Yongwei Yuan, and Zhuo Zhang. 2024. ConSol Artifact. https://github.com/Kraks/contract-for-contract/.

[90] XCarnival. 2023. https://twitter.com/XCarnival_Lab/status/1541226298399653888.

[91] Dana N. Xu, Simon L. Peyton Jones, and Koen Claessen. 2009. Static contract checking for Haskell. In *POPL*. ACM, 41–52.

[92] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying Exploitable Bugs in Smart Contracts. In *ICSE*. IEEE, 615–627.

[93] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2022. SoK: Decentralized Finance (DeFi) Attacks. *IACR Cryptol. ePrint Arch.* (2022), 1773.

Received 2023-11-16; accepted 2024-03-31