



Symbolic Value-Flow Static Analysis: Deep, Precise, Complete Modeling of Ethereum Smart Contracts

YANNIS SMARAGDAKIS, University of Athens, Greece

NEVILLE GRECH, University of Malta, Malta

SIFIS LAGOUVARDOS, University of Athens, Greece

KONSTANTINOS TRIANTAFYLLOU, University of Athens, Greece

ILIAS TSATIRIS, University of Athens, Greece

We present a static analysis approach that combines concrete values and symbolic expressions. This symbolic value-flow (“symvalic”) analysis models program behavior with high precision, e.g., full path sensitivity. To achieve deep modeling of program semantics, the analysis relies on a symbiotic relationship between a traditional static analysis fixpoint computation and a symbolic solver: the solver does not merely receive a complex “path condition” to solve, but is instead invoked repeatedly (often tens or hundreds of thousands of times), in close cooperation with the flow computation of the analysis.

The result of the symvalic analysis architecture is a static modeling of program behavior that is much more complete than symbolic execution, much more precise than conventional static analysis, and domain-agnostic: no special-purpose definition of anti-patterns is necessary in order to compute violations of safety conditions with high precision.

We apply the analysis to the domain of Ethereum smart contracts. This domain represents a fundamental challenge for program analysis approaches: despite numerous publications, research work has not been effective at uncovering vulnerabilities of high real-world value.

In systematic comparison of symvalic analysis with past tools, we find significantly increased completeness (shown as 83-96% statement coverage and more true error reports) combined with much higher precision, as measured by rate of true positive reports. In terms of real-world impact, since the beginning of 2021, the analysis has resulted in the discovery and disclosure of several critical vulnerabilities, over funds in the many millions of dollars. Six separate bug bounties totaling over \$350K have been awarded for these disclosures.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → *General programming languages*; • **Security and privacy** → **Software and application security**.

Additional Key Words and Phrases: Program Analysis, Smart Contracts, Security, Ethereum, Blockchain

ACM Reference Format:

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, Article 163 (October 2021), 30 pages. <https://doi.org/10.1145/3485540>

Authors’ addresses: Yannis Smaragdakis, University of Athens, Greece, smaragd@di.uoa.gr; Neville Grech, University of Malta, Malta, me@nevillegrech.com; Sifis Lagouvardos, University of Athens, Greece, sifis.lag@di.uoa.gr; Konstantinos Triantafyllou, University of Athens, Greece, kotriant@di.uoa.gr; Ilias Tsatiris, University of Athens, Greece, i.tsatiris@di.uoa.gr.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2021 Copyright held by the owner/author(s).

2475-1421/2021/10-ART163

<https://doi.org/10.1145/3485540>

1 INTRODUCTION

Static analysis is distinguished among program analysis techniques (e.g., *testing* [Czech et al. 2016; Meyer 2008], *model checking* [Clarke et al. 1986; Jhala and Majumdar 2009], or *symbolic execution* [Baldoni et al. 2018; King 1976]) by its emphasis on *completeness*, i.e., its attempt to model all (or as many as possible) program behaviors. To achieve this goal under a realistic time budget, static analysis often has to sacrifice some *precision*: the analysis may consider combinations of values that may never appear in a real execution. Maintaining high precision, while achieving completeness and scalability, remains a formidable challenge.

In this work, we present an analysis that attempts to meet this challenge, under realistic domain assumptions. The analysis maintains high precision (closely analogous to that of model checking or full program execution) while achieving very high completeness, as measured in terms of coverage of program behaviors. For conciseness purposes, we give to the analysis architecture the name *symvalic analysis*, for “*symbolic+value-flow static analysis*”.

A symvalic analysis computes for each program variable a set of possible concrete values as well as symbolic expressions. The analysis is *path-sensitive*: values propagate to a variable at a program point only if they satisfy (modulo natural analysis over-approximation) the conditions under which the program point would be reachable. In order to satisfy conditions, the analysis needs to solve equalities and inequalities over both concrete and symbolic values. To do this, the analysis appeals to a symbolic solver and simplifier tightly integrated with the analysis core. At the same time, most of the analysis coverage of the program is done inexpensively, based on concrete values—e.g., the analysis tries selected small integers, large integers, and constants from the program text as values of variables at every external entry point.

The above description is perhaps reminiscent of *dynamic-symbolic* (a.k.a. *concolic*) execution [Cadar et al. 2008; Godefroid 2007; Godefroid et al. 2005; Sen and Agha 2006; Sen et al. 2005; Tillmann and de Halleux 2008; Tillmann and Schulte 2006]. Some of the insights are indeed similar. The use of concrete values whenever possible (and, conversely, the appeal to symbolic solving selectively) is responsible for the symvalic analysis scalability, much as it has been for dynamic-symbolic execution. However, symvalic analysis also has significant differences from dynamic-symbolic execution. Most notably, it is a static analysis and not full program execution. To model a program statement, symvalic analysis does not need to create a full context of values for every other program variable and (heap) storage location. Indeed, symvalic analysis treats concrete values highly efficiently, manipulating them using *set-at-a-time* reasoning, instead of individually. This is the *value-flow* aspect of symvalic analysis: sets of values are propagated using mass operations (table joins) using a standard fixpoint (Datalog) analysis engine.

We apply the idea of symvalic analysis to Ethereum smart contracts [Wood 2014]: a domain where high-precision, high-completeness analyses are in demand. Smart contracts are programs that handle monetary assets whose value occasionally rises to the many millions of dollars. Tens of security researchers pore over the code of these contracts daily. Discovering an important bug is a source of both fame and reward.

Conventional analysis techniques are rarely effective for high-value contracts: Perez and Livshits [2021] recently find that only 0.27% of the funds reported vulnerable by some of the most prominent research tools have truly been subsequently exploited. Simply put, program analysis can help with vulnerabilities in contracts that have not received much scrutiny, but rarely helps with high-value vulnerabilities that have not been found through other means.

Partly addressing this need, symvalic analysis has already enjoyed significant success in the analysis of smart contracts, with several high-value vulnerabilities discovered.

```

1 address admin; // set up at construction, not in contract code
2
3 function withdrawToken(IERC20 token, uint256 amount, address sendTo) external {
4     onlyAdmin();
5     uint256 adjusted = amount * 103 / 100;
6     if (amount >= 10000 && amount < 100000)
7         token.transfer(sendTo, adjusted); ...
8 }
9
10 function onlyAdmin() internal view {
11     require(msg.sender == admin, "only admin");
12 }

```

Fig. 1. Illustration of symvalic reasoning.

In brief, our work claims the following contributions:

- We introduce *symvalic* analysis: an unconventional static analysis design that combines completeness and precision.
 - Symvalic analysis employs a value-flow fixpoint computation in *close cooperation with symbolic reasoning*. This aspect is qualitatively different from past approaches that generate symbolic expressions and dispatch a solver to reason about them—e.g., in our setting the solver is as-if invoked hundreds of thousands or even millions of times in the course of an analysis.
 - Symvalic analysis recovers precision by employing *dependencies* between inferred values: a value is inferred conditionally upon a set of other value assignments. Several elements of this mechanism, such as the use of the values of storage reads as dependency elements, have no counterpart in the literature.
- The analysis is applied to the high-value domain of Ethereum smart contracts and exhibits significant benefits.
 - Extensive experiments against leading industrial tools show precision, scalability, and completeness advantages: 30% more true vulnerability reports, much greater precision, and 85% statement coverage on high-value contracts (vs. 49% for the closest competitor).
 - In actual deployment, symvalic analysis has been responsible for detecting several high-value vulnerabilities, for monetary amounts in the many millions to tens of millions of dollars. Funds directly rescued (i.e., via white-hat hacking) due to symvalic analysis reports reach \$6M. We have received 6 separate bug bounties for these disclosures, for a total value of \$350K.

Brief Illustration. For a preliminary illustration of symvalic analysis, consider the fragment of a smart contract shown in Figure 1. (The code is written in the Solidity language. Solidity is dominant, accounting for more than 99% of deployed Ethereum smart contracts.)

A static analysis may attempt to reason about the feasibility of calling `token.transfer` with a specific recipient (`sendTo`) and transfer amount (`adjusted`). There are two guards that dominate the execution of the transfer statement, split in separate functions.

First, the external caller of the `withdrawToken` function (i.e., an end-user or separate contract identified by `msg.sender`) should be the same as the value of storage variable `admin`. (Otherwise the `require` statement in line 11 of `onlyAdmin`, called on line 4, fails and the transaction reverts.) This alone precludes untrusted callers. Symvalic analysis captures this behavior by maintaining two symbolic values, «unprivileged-user» and its contrasting «owner». These are possible initial values for the special variable `msg.sender`. The «owner» value is also a possible initial value of storage

locations: a contract's owner has set up the contract's initial contents. Therefore, at the return point of the call `onlyAdmin()` (line 4, via line 12), symvalic analysis has maintained a single possibility for `msg.sender`: the value «owner». Value «unprivileged-user» is eliminated from consideration *at that program point*, since it cannot satisfy the condition `msg.sender == admin`. This showcases the path-sensitive nature of the analysis. Notice also that the condition is in a separate function: symvalic analysis is fully inter-procedural (and strongly context-sensitive, via the mechanism of *dependencies*—see below).

For the condition “`amount >= 10000 && amount < 100000`” in line 6, symvalic analysis will initially assign `amount` a set of possible values, consisting of “interesting” integers and symbols. This is because `amount` is an external argument. For instance, `amount` may be considered to hold values 42, 0, and «user-unique-value». The latter, symbolic, value plays the role of a free variable in symbolic evaluation. At the condition on `amount` on line 6, the two concrete values will be eliminated: the unified symbolic solver and simplifier will determine, e.g., that `42 >= 10000` is false. The same solver will be asked about the symbolic value: is it true that «user-unique-value» `>= 10000 && «user-unique-value» < 100000`? Since this value plays the role of a free variable, the solver will suggest a solution to the condition: 10000. This solution will become a fourth possibility for `amount`. At the token.`transfer` call in line 7, the only possible (concrete) value for `amount` is 10000.

Symvalic analysis will not perform full, detailed execution. Reasoning over the values of variables is done on a per-set basis. For instance, variable `adjusted` will be inferred to contain 4 different values after line 5: 0, 43, 10300 and «user-unique-value» * 103 / 100. Treating values by means of independent sets may be performant, but ignores actual execution conditions. Observe that variable `adjusted` is not changed after its initial assignment in line 5, nor filtered (i.e., used in conditions) elsewhere in the computation. Does that mean that the amount transferred can be any of (the concrete among) the above values—0, 43, 10300? The answer is no. Although symvalic analysis does set-at-a-time processing of values, it tries to achieve precision by maintaining *dependencies* over crucial variables. Variable `amount` is one such: it is an external input variable, hence other variables that depend on it maintain this dependency. Therefore, the analysis actually computes that the value 10300 for `adjusted` is the only one compatible with the value 10000 for `amount`, which is necessary for reaching that program point. More precisely, `adjusted` holds 10300 *dependent upon* `amount` holding 10000 and 10300 is the only inferred (concrete) transfer value.

Associating values with dependencies is a natural extension of *context sensitivity* in static analysis. With appropriate choices of program entities to serve as *root dependencies*, a symvalic analysis can maintain high precision while offering completeness and scalability.

2 BACKGROUND AND MOTIVATION

Before introducing symvalic analysis in full, it is useful to motivate it and illustrate its positioning in the crowded space of analysis techniques. Specifically, we ask “why are other analysis approaches (such as dynamic-symbolic execution or regular value-flow static analysis) typically not sufficient?”

Comparison to Dynamic-Symbolic Execution. Symbolic execution [King 1976] is a solver-aided program analysis that has enjoyed practical success in recent years, in the form of dynamic-symbolic or (*concolic*, for *concrete-symbolic*) execution [Cadaru et al. 2008; Godefroid et al. 2005; Sen et al. 2005; Tillmann and de Halleux 2008]. The approach has similarities to symvalic analysis, however it is important to not lose sight of the foundational differences. Symvalic analysis is *not* program execution, but instead a static analysis approach. This means that the analysis is not obliged to fully respect program semantics. Instead, the analysis can readily over-approximate any aspect of program execution the analysis designer sees fit. For instance:

- The analysis can overapproximate control flow: it can consider some conditions to be both satisfiable and negatable without needing to produce specific values for them. (This is likely treatment for conditions that are dependent on complex external parameters—e.g., in the Ethereum smart contracts setting, on the amount or price of *gas* left¹ or the exact monetary *balance* of the current account.) Modeling such technical specifics down to concrete numbers is a bad fit for a *static analysis*, which tries to capture all possible behaviors.
- The analysis can overapproximate data flow: it can decide, e.g., during analyzing a loop, that a certain variable has a *top* value, and, thus, can refer to any element of an array. The analysis does not need to faithfully emulate the full iteration of the loop up to the point of reaching a certain array element—it just needs to estimate that it does so eventually. (The estimate can also be false, which may be reflected as a loss of *precision* in the analysis results.)
- The analysis can produce abstract models of the heap (or other shared storage). For instance, the analysis can use symbolic expressions as heap addresses, or can use abstraction (e.g., consider all addresses that are functions of the external contract caller to be a single abstract address).

As a result, symvalic analysis can cover substantially more behaviors than symbolic execution (at a theoretical precision cost, whose magnitude needs to be evaluated experimentally). This addresses the significant *completeness* problems that (dynamic-)symbolic execution exhibits in practice, even when it attempts to find a *single* execution to cover specific program branches. For instance, the Manticore symbolic execution framework [Mossberg et al. 2019] (one of the foremost for Ethereum smart contracts) “achieves on average 66% code coverage” [Trail of Bits 2020a]. This prompted the Trail of Bits security firm’s company account to tweet “*Why can’t a symbolic executor achieve 100% coverage in a teensy little smart contract?*” [Trail of Bits 2020b], capturing the frustration of consumers of analysis results.

For a taste of how simple code can baffle a symbolic execution engine, consider the following (slightly simplified) real-code excerpt from an Ethereum smart contract:

```
function safeBatchTransferFrom(address from, address to, uint256[] calldata ids,
    uint256[] calldata values) external {
    require(to != address(0x0));
    require(ids.length == values.length);
    require(from == msg.sender || operatorApproval[from][msg.sender]);
    for (uint256 i = 0; i < ids.length; ++i) {
        uint256 id = ids[i];
        uint256 value = values[i];
        balances[id][from] = balances[id][from].sub(value);
        balances[id][to] = value.add(balances[id][to]);
    }
}
```

To exercise the code, the symbolic execution engine needs to invent arrays of values for `ids` and `values`, as well as consistent contents for two storage (i.e., analogous to heap, but on the blockchain) structures. These structures, `balances` and `operatorApproval` are maps of maps and their contents are not free variables: they can only be set in whatever ways other code permits. Additionally, these contents have to include the current caller of the contract (`msg.sender`), which is also not a free variable for symbolic solving purposes: an attacker cannot spoof its address to match another. If any aspect of these constraints ends up too complex to satisfy (e.g., the solver cannot invent arrays

¹Gas is a quantity used to assign a real-world cost to Ethereum computation, to avoid abuse of the network. It has instruction-based accounting, therefore it is programmatically visible to the smart contract execution.

ids and values of the same length, or cannot invent contents of ids that also appear in the first level of balances) symbolic execution will fail to reach all statements of the above code.

A static analysis, such as symvalic analysis, can manage to cover the above code a lot more easily, by employing abstraction and over-approximation. For instance, the analysis design may be collapsing into a single abstract value the entire first level of mappings, as long as the key is set from externally supplied values. This means treating expressions such as `balances[id][from]` effectively as the much simpler `balances[«any»][from]`. Some of the fidelity of a full execution is lost, however the analysis designer accepts this abstraction by explicitly employing it when they think it will not affect precision—in this case, under the condition that the first-level key is externally tainted.

Despite the theoretical precision loss, as later experiments demonstrate, symvalic analysis offers greatly increased coverage combined with high precision.

Comparison to Value-Flow Static Analysis. Static analysis can model a lot more program behaviors than a full-detail program execution, but is still faced with formidable scalability and precision challenges. A *path-sensitive* whole-program analysis, such as the symvalic analysis proposed, has extremely heavy demands, in terms of symbolic solving to satisfy program conditions. Conventional static analysis (tracking the flow of sets of values via a fixpoint computation) typically fares very well in terms of scalability, and often achieves significant precision. Several such analyses have been proposed in the past for the Ethereum space [Grech et al. 2018; Tsankov et al. 2018].

Some of these analysis are even argued to be highly-precise, despite having a shallow understanding of the code—e.g., not modeling symbolic conditions for reaching a certain program statement. The reason for this precision, however, is the careful (but *ad hoc*) analysis design that directly attempts to recognize both known *unsafe* patterns and *countermeasures* for these patterns. Examples of this approach are the recent MadMax [Grech et al. 2018] and Ethainter tools [Brent et al. 2020]—we illustrate with the latter.

Ethainter claims an extremely high precision (i.e., true positive rate) of 82.5%. To achieve this rate, the analysis seeks a code pattern that is commonly employed in a wrong way. Consider the simple contract below:

```
contract Victim {
  mapping(address => bool) authorized;
  function init() public {
    authorized[msg.sender] = true;
  }
  function sensitive(address recipient) public {
    require(authorized[msg.sender]);
    selfdestruct(recipient);
  }
}
```

Function `sensitive` self-destroys the contract, sending all its assets to a recipient address. This operation needs to be *guarded*, in the Ethainter vocabulary. A guard is a condition that checks some property of `msg.sender`—the caller of the transaction. Additionally, if the guard looks up a data structure element with `msg.sender` as key, the data structure should not be tainted by a different operation. Violating either of these conditions yields a warning. In our example, a warning is emitted because there is a guard but it accesses a data structure (`authorized[msg.sender]`) that can be externally tainted (through `init`).

Both of these conditions can be detected with high precision, yielding high rates of true positives *when contracts are coded in a specific style*. However, the conditions are rather rigid and *ad hoc*. For

instance, Ethainter cannot recognize guarding via a `require` clause in a separate function, as in our earlier example of Figure 1: the analysis will issue a false-positive warning in this case.

In contrast, symvalic analysis knows nothing about `msg.sender` guards or data structures that are tainted. Phrasing the same question in symvalic analysis is a mere 2-line query, with no domain modeling, yet much greater generality. Effectively, we ask symvalic analysis “can the `selfdestruct` statement ever be invoked by transactions whose originator is «unprivileged-user» and with an argument that is a (user-settable) free variable?”

Thus, the advantage of the symvalic approach is that it is not tied to specific *unsafe* or *counter-measure* code patterns. The contract can be protecting (or failing to effectively protect) sensitive operations in any arbitrary way. Symvalic analysis will recognize both safe and unsafe uses, since it closely models the conditions that govern a statement’s execution. As a result, symvalic analysis can be both more precise and more complete than custom-designed analyses.

3 SYMBOLIC+VALUE-FLOW STATIC ANALYSIS

We next present an informal overview of symbolic+value-flow analysis, highlighting its design principles. Section 4 will later show a precise model of the analysis.

3.1 Overview

Symbolic+Value-Flow static analysis (“symvalic” analysis, for short) is much like a common inter-procedural data-flow analysis: a “value-flow” analysis, such as a points-to or a taint analysis. Being a value-flow analysis implies that every variable is statically assigned a finite set of values and a fixpoint computation grows the finite sets according to monotonic equations.

In symvalic analysis, the values can be both concrete (e.g., numbers) and entire symbolic expressions. For instance, consider again the earlier example function `sensitive`:

```
function sensitive(address recipient) public {
    require(authorized[msg.sender]);
    selfdestruct(recipient);
}
```

The analysis will consider a set of concrete and symbolic values for all external input variables. For instance, for numeric variables, the analysis considers small constants (0,1, and up to 3 constants under 256), large constants (to cause overflow), and a pseudo-random choice of constants from the program text.

For external inputs of “contract address” type, such as the `msg.sender` implicit argument, the analysis will consider values that include:

- constants in the contract text that resemble addresses (i.e., 160-bit integers)
- the symbolic values «owner» and «unprivileged-user».

Similarly, the values initially considered for the `recipient` argument include:

- constants in the contract text that resemble addresses
- the symbolic values «owner-unique-value» and «user-unique-value».

The difference between the symbolic values is that the former («owner» and «unprivileged-user») will be considered *bound-variables* for symbolic reasoning purposes: although we treat them symbolically, the caller cannot set them freely. In contrast, the latter («owner-unique-value» and «user-unique-value») are free variables and symbolic reasoning can propose concrete values for them, in order to solve constraints. Section 3.2 will describe in more detail how these values are *dependent*: the analysis will model separately the case of the current caller of the contract being «owner» and that of it being «unprivileged-user», and similarly for the values they supply to arguments.

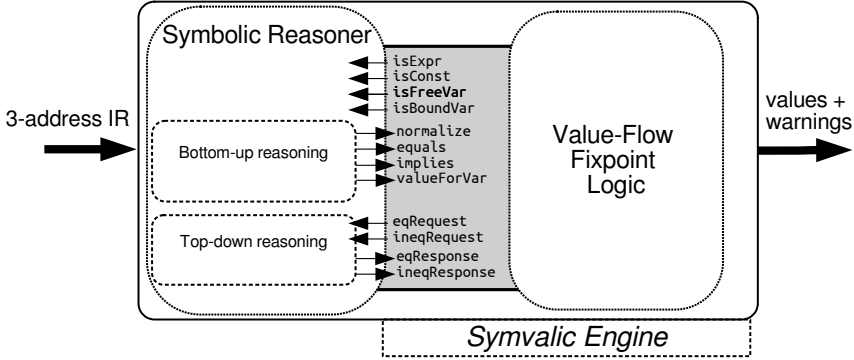


Fig. 2. Symvalic Analysis Architecture: API (slightly simplified) between symbolic reasoning and value propagation shown.

Symbolic values propagate and are used to form more complex expressions, whereas concrete values get constant-folded. For instance, in the example, the program expression `authorized[msg.sender]` is a lookup in a mapping structure on Ethereum storage (i.e., analogous to the heap in a conventional language). Due to the way Ethereum storage is organized, the expression is loading from the symbolic address `SHA3(«owner» ++ 0x0)`—SHA3 is the Keccak-256 hash function, ++ a byte array concatenation operator, and `0x0` (i.e., zero) the constant offset that identifies the authorized mapping among other attributes in storage. This symbolic expression is propagated by the analysis as a value for the corresponding local variable (unnamed in the source language but present in the intermediate representation).

The analysis proceeds via a close interaction of a value-flow fixpoint loop and a symbolic reasoner. Figure 2 shows the overall architecture, including the symbolic reasoning component and the value-flow component, as well as the interface between them and the two sub-components of symbolic reasoning: a bottom-up and a top-down component. The full interface and bottom-up/top-down distinction are elements of importance mostly for practical concerns, especially efficiency. Therefore we postpone the full discussion of such specifics to Section 5, preferring to concentrate on the high-level elements for now.

If viewed as an idealized reasoner over infinite expressions (i.e., without worrying about computational efficiency) the main products of the symbolic reasoner are:

- a predicate `normalize(expr, normExpr)` that returns for each expression its minimal equivalent form. This applies to both arithmetic and logical expressions;
- a predicate `implies(exprStrong, exprWeak)` that checks whether one logical expression implies another;
- a predicate `valueForVar(var, expr)` that proposes values (i.e., symbolic expressions) for free variable `var`, so that logical conditions get satisfied. The solver proposes such solutions in bulk, for any interesting logical expressions (the main case being equalities, which are usually much harder to satisfy than inequalities) and the analysis filters the solutions it chooses to truly try—e.g., if they satisfy an equality that was previously not satisfied.

Inevitably, all three predicates are incomplete, i.e., will not capture all (infinite) true relationships.

The analysis appeals to these predicates throughout normal value propagation. Symbolic values are continuously *normalized* (i.e., simplified up to minimal form) and are also used to satisfy control-flow constraints, i.e., predicates in conditional statements. This makes the analysis path-sensitive:

the information pertaining to a program statement is derived to be compatible with the conditions necessary for reaching the statement. Mechanism-wise, this is a part of the *dependencies* machinery, described in Section 3.2.

The close interaction of value-flow and symbolic reasoning is a rather unconventional aspect of symvalic analysis. This is in contrast to explicit invocation of solvers in a static analysis [Krupp and Rossow 2018; Nikolić et al. 2018; Schneidewind et al. 2020]. It is common for a static analysis to collect a large condition and dispatch an external solver (e.g., Z3 [De Moura and Bjørner 2008]) to satisfy the condition. Symvalic analysis, however, invokes the symbolic solver a lot more regularly, to continuously simplify and solve conditions. Such appeals to the symbolic solver are essential, in the heart of the analysis, as the model of Section 4 will illustrate.

3.2 Precision and Dependencies

Symvalic analysis is a static analysis, aiming for completeness, i.e., covering *many* program behaviors. However the analysis is neither sound nor complete: it can both model unrealizable behaviors and miss actual behaviors. The analysis intends to strike a good balance between *completeness* and *precision*: to cover a lot more behaviors than symbolic execution, while still predicting behaviors that are realizable to a large degree.

To achieve precision, while keeping a set-based treatment, symvalic analysis associates variable-value tuples (as well as statements in the code) with *dependencies*. Every analysis inference of the form “variable v may have value (i.e., concrete or symbolic expression) e ” (as well as of the form “this statement is reachable”) is associated with *sets of mappings of variables to values* that have led the analysis to make the inference.

We distinguish two kinds of dependencies: *local* (reset per-function) and *transaction* (i.e., current-execution) dependencies. We write $v \rightarrow e \langle d^L; d^T \rangle$ to designate that variable v may hold value e under local dependencies d^L and transaction dependencies d^T . (When there is no need to distinguish local and transaction dependencies, we write $v \rightarrow e \langle D \rangle$.) To illustrate dependencies, consider a simple contract fragment:

```
contract Safe {
  address owner;    // set at construction
  mapping (address => uint) public balanceOf;
  function deposit(address to,uint amount) public {
    require(msg.sender == owner);
    uint curBalance = balanceOf[to];
    uint nextBalance = curBalance + amount*90/100;
    balanceOf[to] = nextBalance;
  }
}
```

Let us focus on the value of variable `nextBalance` during an invocation `deposit(0x42, 200)`.² For `nextBalance` to even have a value, its assignment statement needs to be reachable. Therefore the “`require(msg.sender == owner)`” statement needs to be satisfied. This forces the mapping $[sender \rightarrow \langle \text{owner} \rangle]$ to appear in the transaction dependencies. (The symbolic variable `sender` stands for the Solidity `msg.sender` expression and $\langle \text{owner} \rangle$ is a symbolic value generated to represent the contract that originally called the current contract’s constructor and, thus, set storage fields to initial values.)

²Contract addresses are 160-bit integers and generated randomly. We show a very unlikely short address (0x42) for the sake of conciseness.

Additionally, values of local variables will be parts of the dependencies. Assume that at some point the analysis has inferred that storage mapping `balanceOf[0x42]` has two possible values: 1 and 80, and then considers the call `deposit(0x42, 200)`.

The inferences for variable `nextBalance` will then be:

$\text{nextBalance} \rightarrow 181 \{ \{ [\text{to} \rightarrow 0x42], [\text{amount} \rightarrow 200], [\text{curBalance} \rightarrow 1] \}; \{ [\text{sender} \rightarrow \text{«owner»}] \} \}$
and

$\text{nextBalance} \rightarrow 260 \{ \{ [\text{to} \rightarrow 0x42], [\text{amount} \rightarrow 200], [\text{curBalance} \rightarrow 80] \}; \{ [\text{sender} \rightarrow \text{«owner»}] \} \}$.

The first of the two inferences can be read as “`nextBalance` is expected to hold the value 181 in executions where `to` holds value `0x42`, `amount` holds value 200, the most recent storage load into variable `curBalance` returned value 1, and the transaction initiator (i.e., external caller) is “owner””. The first three mappings make up the local dependencies for the inference, whereas the last is the transaction dependency.

Dependencies are *combined*, with an operator denoted \oplus . Combination of dependencies happens for every control-flow or data-flow join point: when two branches merge, or when two values are used as operands in the same operation. (Section 4 will make this fully precise—our current description is just for exposition.) Combining two sets of dependencies is a check for compatibility, to prevent mixing information from guaranteed-separate executions. Combining dependencies succeeds if there are no conflicting mappings for the same variable. For instance, the two inferences above cannot be combined: they conflict on the mapping for variable `curBalance`. If dependencies do not conflict (thus, combining them succeeds), combination is a mere pairwise union of the mappings sets.

3.3 Soundness and Completeness

Using the Safe contract example, we can illustrate why symvalic analysis chooses to be neither sound nor complete, i.e., why it accepts some incompleteness in order to be more precise, and why it accepts less-than-full precision (e.g., false positives) in order to be more complete or more scalable.³

Incompleteness. The last statement of function `deposit` stores `nextBalance` back into the persistent storage mapping `balanceOf[0x42]`. Therefore, new inferences are possible for variable `nextBalance`, which again cause new values to flow into `balanceOf[0x42]`, *ad infinitum*. Static analysis typically resolves this potentially infinite computation by overapproximation (e.g., a finite-height lattice that joins values into abstract values, at the expense of some loss of precision). In contrast, symbolic execution resolves non-termination (arising in the case of looping) by arbitrarily truncating execution traces (e.g., unrolling loops only a small number of times, or executing a fixed number of total instructions).

Symvalic analysis is agnostic regarding the handling of cyclic flow: both overapproximation and finite truncation are acceptable, per case. For instance, the current symvalic setting (described in Section 5) uses overapproximation for values from some sources (e.g., environment variables, such as gas remaining, current block number, miner of block). In most cases, however, the analysis favors

³Terminology note: “soundness” and “completeness” are duals. For an analysis that issues warnings, the absence of false positives can be termed “soundness”, if the analysis is viewed as a bug finder, or “completeness”, if the analysis is viewed as a correctness verifier. Conversely, the absence of false negatives (i.e., “no warning” implies “no problem exists”) can be termed “completeness”, if the analysis is viewed as a bug finder, and “soundness”, if the analysis is viewed as a correctness verifier. Since our analysis is neither sound nor complete, throughout the paper we prefer to use the terms “complete” and “precise”, because they are closer to intuitive meaning: “complete” means covering more behaviors (e.g., greater statement coverage, better statistics in revealing known bugs), whereas “precise” refers to modeled behaviors being realizable. This terminology is also easier to treat in quantifiable, non-binary terms. For instance, readers might be surprised to read an analysis being described as “more sound”, but not when it is being described as “more precise” or “more complete”.

concrete numeric values (such as those in `balanceOf[0x42]`), which yield other concrete values, up to a *finite* number of arithmetic operations. This treatment is one that explicitly favors precision over completeness: the analysis is not guaranteed to model all values arising in real executions. The advantage, however, is that the values that the analysis considers are *likely* to be realizable, i.e., precision is enhanced.

Precision. Dependencies can lend arbitrary precision to an analysis. Even fully concrete execution can, for instance, be simulated by maintaining in dependencies all dynamic variables of an execution. Conceptually, dependencies can be viewed as a generalization of *context-sensitivity* [Sharir and Pnueli 1981] in static analysis, or an instance of relational analysis (e.g., [Nielson et al. 1999, Sec.4.4.1], [Møller and Schwartzbach 2018, Sec.7]). Such mechanisms group together dynamic executions for uniform static treatment. Precision arises because executions mapped to different groups are never confused. The challenge, however, is to maintain sufficient precision without suffering from extreme lack of scalability (termed the *state explosion* problem). The state explosion problem is the bane of fully-precise program analysis approaches, such as concrete testing or concrete-state model checking. The number of possible value combinations rises exponentially, per set combinatorics. In the setting of symvalic analysis, if the dependencies mechanism were to keep full concrete state (i.e., what *precisely* are the contents of storage or variables in a simulated execution), the analysis would suffer tremendously in scalability. In practice, even dependencies on a handful of variables can render the analysis unscalable.

Symvalic analysis maintains a balance between precision and performance by computing dependencies only on a small subset of program variables. As a consequence, the analysis can produce warnings or values that are *imprecise*, i.e., do not correspond to actual executions.

These precision limits of dependencies in the analysis are as follows (with current defaults listed in parentheses, for concreteness):

- A bounded number of arguments of the current function, such as `to` and `amount` in the example. (Currently: up to 3 arguments are kept as local dependencies.)
- A bounded number of local variables that load values from shared memory (storage), such as `curBalance` in the example. (Currently: the first variable loading from storage per function is kept in local dependencies.)
- A bounded number of external arguments (i.e., arguments supplied at the original entry point of the transaction, by an external caller). (Currently: the first 2 arguments of the transaction entry point are kept in transaction dependencies. This case is not shown in the example. However, if we were to change the code to make function `deposit`—which is a transaction entry point—call a different, internal function, the values of `deposit`'s arguments, `to` and `amount`, would be kept in the transaction dependencies when analyzing the internal function.)
- The transaction's current caller (`msg.sender` in Solidity). (Currently: kept in transaction dependencies.)

4 ANALYSIS MODEL

Armed with the design insights of the previous section, we can next see symvalic analysis in full detail. The setting is simplified but fully representative of the interesting cases.

4.1 Input and Environment

Figure 3 shows a minimal input language that we will use for illustration. The domains of the analysis (and meta-variables used subsequently) comprise:

- $v, u, t \in V$, a set of variables,
- $\text{fun} \in F$, a set of functions,

<i>Instruction</i>	<i>Operand Types</i>	<i>Description</i>
$i : v = t \odot u$	$I \times V \times V \times V$	Binary operations
$i : v = \phi(u_i)$	$I \times V \times V^n$	Phi instructions
$i : v = \llbracket u \rrbracket$	$I \times V \times V$	Loads
$i : \llbracket v \rrbracket = u$	$I \times V \times V$	Stores
$i : \text{jumpif } v \text{ } j$	$I \times V \times I$	Conditional jumps
$i : \text{fun}(u_j)$	$I \times F \times V^n$	Calls

Fig. 3. Intermediate Representation instruction Set.

Notation	Description
DEF ($\text{fun}(\bar{a}):i$)	Function fun is defined with formal argument vector \bar{a} and first instruction i .
$i \xrightarrow{\text{next}} j$	instruction i has j as a possible next.
OK ($D_1 \oplus D_2 \dots$)	Dependencies combination is valid (no conflicting dependencies for same variable).
$v \rightarrow e \langle D \rangle$	Variable v may hold symbolic expression e under dependencies D .
$\llbracket a \rrbracket \Rightarrow e$	Storage location a (a symbolic expression) may have contents e .
$ i \langle D \rangle$ or $ i \langle d^L; d^T \rangle$	Instruction i is reachable with dependencies D . (Expanded: local deps. d^L , transaction deps. d^T .)
ORACLE (v) = e	The symbolic solver (or default logic) suggests value e for external arg./environment variable v .
NORMALIZE (e) = e_0	Expression e normalizes (simplifies) to e_0 .

Fig. 4. Definitions of notation used in the rules. The first two relations encode input program information and a helper function. The next three are the relations computed by the analysis. The last two are the interface with the symbolic solver.

- $i, j \in I$, a set of instruction labels,
- $n \in \mathbb{N}$, the set of natural numbers.

The simplified language omits (without loss of generality) features that can be translated away and their symvalic treatment is well-represented in that of others: there are no return values or explicit return statements, unary operators, private functions (all functions are public entry points), or environment-consulting operations.

Figure 4 introduces the relations that the analysis accepts as input, as well as those it computes. The analysis input consists of a set of instructions, linked into a control-flow graph, via relation $i \xrightarrow{\text{next}} j$ (over $I \times I$). Another straightforward input is **DEF**, which shows the type signature and initial instruction of a function. **OK** is notation to signify that a combination of dependencies is valid. (By convention, we omit **OK** in the antecedent of rules where the result of combining dependencies is used in the consequent: an invalid combination produces no result, hence a result implicitly establishes validity in the antecedent.)

The main concepts established by the analysis are:

- $v \rightarrow e \langle D \rangle$, discussed in Section 3.2
- $\llbracket a \rrbracket \Rightarrow e$: storage location a is inferred to hold value e —this is a global concept that transcends transactions and function invocations, hence no dependencies

- $|i| \langle D \rangle$: instruction i is reachable under dependencies D , where D can be expanded into $d^L; d^T$, when local and transaction dependencies need to be treated separately. The dependency combination operator is applied pointwise: $(d_1^L; d_1^T) \oplus (d_2^L; d_2^T) = d_1^L \oplus d_2^L; d_1^T \oplus d_2^T$.

Finally, the interaction between the main analysis and the symbolic solver, as well as the initial setup of the analysis, are encapsulated in relations **ORACLE**(v) and **NORMALIZE**(e). **NORMALIZE** captures the symbolic simplifier/reducer of expressions. It is used every time a new expression is formed (i.e., at a binary operator) to reduce it to its minimal form, hopefully a concrete value. Its interface is kept very simple in the model by hiding some of the real work behind **ORACLE**—which we can do by making **ORACLE** always “guess” the right values.

More specifically, **ORACLE** hides behind it two major elements. The first is straightforward: it supplies the initial (concrete and symbolic) values that the analysis will try, both as possible arguments to functions and as initial contents of storage. For instance, we mentioned earlier that the analysis will try to exercise the code with pre-defined constants, as well as constants found in the program text, as well as symbolic values, such as «owner» or «user». In the formalism, this aspect is captured by having **ORACLE** return the appropriate value for a function argument (to represent an external call). The second major role of **ORACLE** is to invent values for arguments that the solver uses to satisfy conditions. That is, **ORACLE**(v) omnisciently guesses values that should be given to external arguments so that a further expression e has **NORMALIZE**(e) = true or **NORMALIZE**(e) = false. These values are used to satisfy conditional jump conditions, and, thus, explore more branches.

4.2 Analysis

The above modeling of **ORACLE** yields an elegant specification of the analysis, since the interactions with the symbolic solver now go through a simple interface. Figure 5 shows the analysis model for the input language, as rules that will monotonically iterate until fixpoint. We next describe the elements and insights in detail.

- Rule **NEXT** propagates dependencies (both local and transaction, treated as a single D) from one instruction to the next, as long as there is no conditional control-flow transfer. ($*$ denotes a “don’t care” value.)
- Rules **JUMPIF-T** and **JUMPIF-F** handle dependencies transfer over conditional control-flow. There are two interesting aspects: First, for the next statement to be analyzed, the analysis must have inferred the value true (resp. false) for the condition variable. This ensures the path-sensitive nature of the analysis. Second, the rules showcase a pattern that also appears in most other rules: the (data-flow) dependencies of the variable (v) used in the statement have to be compatible with the (control-flow) dependencies of the statement itself. (Recall the convention that the check is implicit in the rule antecedent if the result of the combine operation appears in the rule consequent.)
- Rule **BINARYOP** shows the handling of expressions. Again, the dependencies of the variables need to be combined with those of the statement. **This is the only rule that constructs new expressions, as $e_t \odot e_u$.** It appeals to the symbolic solver, as discussed in the previous section. **NORMALIZE** minimizes the new expression (all the way down to a concrete value, if possible). Importantly, it may also refuse to yield anything if expression size or complexity limits are reached (as required to guarantee termination, per Section 3.3). In the full implementation, the expression construction “ $e_t \odot e_u$ ” is limited to a finite but rich universe of expressions. We bound this universe by running several iterations of a pre-analysis (essentially the full symvalic analysis but without dependencies), each time generating expression trees of up to size 4, simplifying

$$\begin{array}{c}
\text{(NEXT)} \quad \frac{|i| \langle D \rangle \quad i \xrightarrow{\text{next}} j \quad \neg(i : \text{jumpif} \ast \ast)}{|j| \langle D \rangle} \\
\\
\text{(JUMPIF-T)} \quad \frac{|i : \text{jumpif } v \text{ } j| \langle D_i \rangle \quad v \rightarrow \text{true} \langle D_v \rangle}{|j| \langle D_i \oplus D_v \rangle} \\
\\
\text{(JUMPIF-F)} \quad \frac{|i : \text{jumpif } v \text{ } j| \langle D_i \rangle \quad v \rightarrow \text{false} \langle D_v \rangle \quad i \xrightarrow{\text{next}} k \quad k \neq j}{|k| \langle D_i \oplus D_v \rangle} \\
\\
\text{(BINARYOP)} \quad \frac{|i : v = t \odot u| \langle D_i \rangle \quad t \rightarrow e_t \langle D_t \rangle \quad u \rightarrow e_u \langle D_u \rangle}{v \rightarrow \mathbf{NORMALIZE}(e_t \odot e_u) \langle D_i \oplus D_t \oplus D_u \rangle} \\
\\
\text{(PHI)} \quad \frac{|i : v = \phi(\dots u \dots)| \langle D_i \rangle \quad u \rightarrow e \langle D_u \rangle}{v \rightarrow e \langle D_i \oplus D_u \rangle} \\
\\
\text{(LOAD)} \quad \frac{|i : v = \llbracket u \rrbracket| \langle d_i^L; d_i^T \rangle \quad u \rightarrow e_u \langle d_u^L; d_u^T \rangle \quad \llbracket e_u \rrbracket \Rightarrow e}{v \rightarrow e \langle d_i^L \oplus d_u^L \oplus [v \rightarrow e]; d_i^T \oplus d_u^T \rangle} \\
\\
\text{(STORE)} \quad \frac{|i : \llbracket v \rrbracket = u| \langle D_i \rangle \quad v \rightarrow e_v \langle D_v \rangle \quad u \rightarrow e_u \langle D_u \rangle \quad \mathbf{OK}(D_i \oplus D_v \oplus D_u)}{\llbracket e_v \rrbracket \Rightarrow e_u} \\
\\
\text{(CALL)} \quad \frac{|i : \text{fun}(\bar{u})| \langle d_i^L; d_i^T \rangle \quad \forall j : u_j \rightarrow e_j \langle d_j^L; d_j^T \rangle, \mathbf{OK}(d_i^L \oplus d_j^L) \quad \mathbf{DEF}(\text{fun}(\bar{a}) : l)}{|l| \langle \bigoplus_k [a_k \rightarrow e_k]; \bigoplus_k d_k^T \oplus d_i^T \rangle \quad \forall k : a_k \rightarrow e_k \langle [a_k \rightarrow e_k]; \emptyset \rangle} \\
\\
\text{(EXTERNAL-ARGS)} \quad \frac{\mathbf{DEF}(\text{fun}(\bar{a}) : \ast) \quad \mathbf{ORACLE}(a_k) = e_k}{a_k \rightarrow e_k \langle [a_k \rightarrow e_k]; \emptyset \rangle} \\
\\
\text{(SENDER)} \quad \frac{\mathbf{ORACLE}(\text{sender}) = e}{\text{sender} \rightarrow e \langle \emptyset; [\text{sender} \rightarrow e] \rangle}
\end{array}$$

Fig. 5. Analysis rules. Suggestion: at first read, ignore dependencies (i.e., the parts in $\langle \dots \rangle$ and constraints on d and D variables) to see just straightforward value-flow and the interaction with symbolic reasoning (**NORMALIZE**, **ORACLE**).

them symbolically, and using the results as building blocks for expression trees of the next iteration.

- Rule PHI propagates unchanged expression values that originate in different control-flow paths. The rule has the conventional form for a may-analysis.
- Rule LOAD introduces new local dependencies, based on the variable that receives the value based on storage. The consequent of the rule shows both the expected inference, $v \rightarrow e$, and also that the same is inserted in the local dependencies. This may seem extraneous (though still correct) at this point: why should the analysis derive that v may be e under the condition that v is e ? However, this treatment ensures that any subsequent expressions that use the variable v will carry the $v \rightarrow e$ local dependency.

- Rule **STORE** complements **LOAD**, leading to the establishment of the contents of a storage location.
- Rule **CALL** is conceptually simpler than its daunting form may suggest: it checks that all actual arguments at a function call have values and dependencies compatible with each other and with reaching the statement. (The latter check is implicit, based on the big combine operations in the consequent.) There are two conclusions in the consequent of the rule. The first states that the dependencies for reaching the first instruction of the called function have to be the combination of dependencies for all arguments. The second merely gives values to each individual formal argument of the function, under near-trivial dependencies: the value of the argument itself. Just as in the **LOAD** rule, earlier, this is done so that any further use of the argument will be dependent on its value. The subtlety is that no such use can occur without also combining the *instruction* dependencies. (That is, the variables in the second conclusion in the consequent of the rule will also eventually receive all dependencies of the first conclusion.) As a result, the dependencies inside a called function really keep the full values of all its arguments.
- Rules **EXTERNAL-ARGS** and **SENDER** appeal to the **ORACLE** relation, as described in Section 4.1. The value the oracle supplies for sender is entered in the transactional dependencies.

The formalism, as captured in these rules, is in close correspondence with the full implementation of the analysis—even the subtleties mentioned in the rule explanations (e.g., in when dependencies get introduced) carry over from the actual implementation. Additionally, the formalism clearly illustrates the continuous appeal of the analysis to symbolic solving *in the course of* value propagation through a fixpoint computation.

5 SPECIFICS AND IMPLEMENTATION

We next discuss the specifics of the current symvalic analysis setup. Although, not germane to the precision of the analysis, the setup offers insights regarding its experimental behavior.

5.1 Technical Discussion

We implemented symbolic value-flow analysis for Ethereum smart contracts, over the IR exported by the Gigahorse decompiler [Grech et al. 2019] and the memory modeling library [Lagouvardos et al. 2020] built on top of it.⁴ Smart contracts offer an excellent platform for sophisticated static analysis due to their intensive security requirements, relatively small size, and modern emphasis (with several prominent tools in the space).

Symvalic analysis combines a fixpoint computation (per the monotonic rules of Section 4.2) and symbolic solving. Our current symvalic implementation uses a Datalog specification of the analysis and a symbolic reasoner that is also largely implemented in Datalog, for close integration with the static analysis fixpoint logic. The solver and analysis make minimal appeal to external, specialized C++ functors—for fast arithmetic, as well as data structures for efficient implementation of dependencies (Section 3.2) and the “combine” (\oplus) operator.

Before describing the approach, it is worth considering alternatives. Our original intent, and prototype, used an external solver (the Z3 [De Moura and Bjørner 2008] SMT solver) for symbolic reasoning. We found the approach less than desirable, given the number of appeals to the solver that symvalic analysis needs to perform. This is hardly surprising, considering that realistic symvalic analysis runs will appeal to the symbolic reasoner several tens-of-thousands of times (and up to millions). For comparison, traditional symbolic execution systems collect *path conditions* (i.e., the conjunction of all conditions that need to be satisfied to reach a statement) and dispatch a solver to satisfy them. As a result, such systems typically appeal to the solver a lot more rarely.

⁴Both tools were obtained from the public repository of the Gigahorse toolchain, available at <https://github.com/nevillegrech/gigahorse-toolchain>

For instance, Maian [Nikolić et al. 2018] sets a 10sec timeout for each invocation of Z3. Symvalic analysis performs thousands of inferences in this span of time.

Another tempting alternative is offered by the recent Formulog system [Bembenek et al. 2020]. Formulog combines Datalog and symbolic reasoning using Z3. However, the case studies that Formulog shines in support the earlier assessment: Formulog symbolic execution [Bembenek et al. 2020, Sec 5.4] is competitive with modern leading systems, such as CBMC [Clarke et al. 2004] and KLEE [Cadar et al. 2008]. Symbolic execution may cause a large number of appeals to the symbolic solver, however these are still an order-of-magnitude fewer than symvalic analysis: symbolic execution only has *one* path to maintain at any given time, whereas symvalic analysis maintains as many as possible, using sets of values and dependencies. Furthermore, value-flow analysis in Formulog [Bembenek et al. 2020, Sec 5.3] is around 7x slower than state-of-the-art fixpoint systems, making this an unattractive substrate for symvalic analysis. Still, an evolved implementation that will seamlessly combine Datalog rules and symbolic reasoning with high performance will be an ideal platform for symvalic analysis in the future.

5.2 Solver

As shown earlier in Figure 2, the symbolic reasoner has two parts: a bottom-up and a top-down reasoning component. The dual structure is a good fit for the implementation of the solver as a Datalog module.

Bottom-up reasoning is the main symbolic workhorse: it produces and memoizes the results of theorems applied to all symbolic expressions provided to the reasoner, as well as any others produced in the course of proving theorems. This is a super-exponential process, therefore the expressions supplied to the bottom-up reasoning component are typically of a small bounded size. (We currently do 4 rounds of simplification, each over expression trees of size at most 5.) Essentially, bottom-up reasoning proves *all* consequences of theorems (as long as they reduce the size of expressions) for expressions up to a certain size. As seen in Figure 2, the analysis initializes all reasoning by supplying the symbolic solver with expressions via predicates `isExpr`, `isConst`, `isFreeVar`, and `isBoundVar`. It receives the products of bottom-up reasoning in the form of relations `normalize` (highly analogous to **NORMALIZE** in Section 4), `equals`, `implies`, and `valueForVar`. The last of these offers suggestions to the (value-flow) analysis regarding extra values for free variables. The analysis logic is free to accept these suggestions and propagate them wherever free variables arise (e.g., values of arguments to public functions). The analysis can also reject such suggestions—e.g., if it has already covered all branches in a function with multiple values. (This two-step interaction of the value-flow analysis and the solver was simplified/abstracted into the **ORACLE** of Section 4.)

The top-down component is responsible for on-demand reasoning over expressions that are too big for bottom-up reasoning. Whereas bottom-up reasoning produces all theorems (i.e., reasoning consequences) for bounded expressions, top-down reasoning produces a bounded number of consequences (i.e., the result of a bounded number of reasoning steps) over unbounded expressions. The interface follows a request-response model, with the value-flow analysis sending the solver equality and inequality satisfiability requests (`eqRequest`/`ineqRequest`) and getting back responses (`eqResponse`/`ineqResponse`).

The reasoner implements a full collection of algebraic properties for arithmetic and boolean logic. Given the richness of operators in the Ethereum VM language (which includes 256-bit signed, unsigned, and modular arithmetic, several variants of shifts, etc.) we cannot make a full argument regarding the completeness of the algebraic rules. However, in practice, we have yet to find an expression that cannot be solved/simplified because of algebraic reasoning incompleteness. (In contrast, the aforementioned size limitations of bottom-up reasoning and inference-depth limitations of top-down reasoning *are* causes for incompleteness.)

6 SETTING AND REAL-WORLD APPLICATION

We next discuss the real-world setup and impact of symvalic analysis, also indirectly motivating and validating its design choices.

6.1 Ethereum Smart Contracts

The setting of Ethereum smart contracts is particularly interesting for software verification and validation approaches. The primary reason is the potential impact: smart contracts manage monetary assets, often in the many millions of dollars. The most common domain for smart contracts deployed in the past two years has been Decentralized Finance (DeFi), i.e., autonomous protocols that implement many of the functionalities of conventional finance (e.g., lending, exchanging, options trading), often in innovative ways.

With this much real-world value, one might expect that cutting-edge research techniques on program analysis and testing will have a significant practical impact on Ethereum smart contracts. This has not been the case, however. [Perez and Livshits \[2021\]](#) conduct a thorough study of tens of thousands of contracts reported vulnerable by six recent leading academic projects. They find that under 2% of the contracts and only 0.27% of the funds held in them have actually been exploited. A very small part of this impact is explainable by the tools having higher-than-reported false-positive (i.e., *imprecision*) rates. Instead, the main reason for this striking result is a strong bias in the sample: contracts that hold funds are very heavily scrutinized and much more likely to be false positives in the analysis. Even analyses with 90% precision (i.e., true-positive) rates in the overall contract population have extremely high false-positive rates in the subset of contracts that truly currently manage funds.

Therefore, it should come as no surprise that program analysis tools are not considered very valuable for Ethereum security analysis. A recent quote by a prominent security analyst in the Ethereum space captures the prevailing view: “*for an experienced contract author, it’s never the automated tooling that finds the bugs that kill them*” [[Konstantopoulos 2021](#)].

Thus, the domain of Ethereum smart contracts represents a high practical but also intellectual challenge for program analysis. The analysis should be extremely precise, so that the warnings for high-value contracts are true positives that humans may have missed, and yet fairly complete, since catching the easy “certain” cases is likely to yield no warnings for contracts that truly manage funds. Thankfully, there are elements of the domain that help. First and foremost, smart contracts are isolated from each other and coded defensively. This introduces a high degree of modularity: the contract can be analyzed mostly in isolation from others. Nearly anything that comes from the outside world is untrusted, unless either the data or the sender are vetted through specific mechanisms. Second, the contracts are of modest size: a deployed smart contract is at most of 24KB in binary size. This corresponds to at most a few thousand lines of code. The largest of the SmartBugs benchmarks that we will consider in our evaluation is under 2.5KLoC in Solidity source. Common sizes of “large” smart contracts are under 1KLoC, with another 1KLoC inherited or called in libraries.

The small size and (relative) modularity of smart contracts means that we can apply analysis techniques that are more ambitious (in terms of precision) than in a general-purpose language setting. Past work has used ambitious program reasoning techniques [[Albert et al. 2020c](#); [Grossman et al. 2017](#); [Permenev et al. 2020](#)] (indeed, even full program verification using proof assistants and off-line logics has been employed [[Hildenbrandt et al. 2018](#)]).

The design choice of high-precision is reflected in the mechanism of dependencies and the general path sensitivity of the symvalic analysis. In a general-purpose language setting (e.g., analyzing Java, C#, C++, etc.) path-sensitive analyses are rare and virtually never combined with whole-program

reasoning (i.e., a modeling of the global heap, such as our $\llbracket a \rrbracket \Rightarrow e$ “storage location contents” relation). The reason is scalability: whole-program analysis of globally shared structures can scale to applications of typical real-world sizes for general-purpose languages (in the hundreds of thousands of LoC) only when the analysis sacrifices path sensitivity, and often even flow sensitivity, context sensitivity, or field sensitivity.

6.2 Practical Impact

Symvalic analysis has been applied to all new contracts deployed on the Ethereum blockchain since the beginning of 2021. In the course of these months, the analysis has flagged numerous exploitable vulnerabilities [Dedaub 2021a,b,c,d; Primitive Finance 2021]. We have made several vulnerability disclosures, some of which resulted in major rescue efforts [Dedaub 2021a; Michales, Jonah 2021]. (The vulnerable services include 2 of the top-15 financial services on Ethereum, per current defipulse.com rankings.)

The total amount of vulnerable funds safeguarded is in the many millions, and potentially tens of millions, of dollars. The exact figure would have been readily computable for a black-hat hack, but is not computable for vulnerability disclosures because a) it greatly varies by time—e.g., in one case an extra \$900K became vulnerable two months after initial disclosure and rescue, because a customer exchanged some funds; b) in many cases there was no rescue operation because the vulnerability could be mitigated without a direct attack to vulnerable funds. In three cases, an attack (i.e., a white-hat hack, in collaboration with the vulnerable service) was necessary in order to rescue the vulnerable funds. The total funds *actually rescued* during these white-hat hacks amount to \$6M. We have received 6 separate bug bounties from these vulnerability disclosures, for a total of \$350K. Our vulnerability disclosures have been featured six times in “Week of Ethereum News”—the most popular weekly digest of the Ethereum space—in the 3 months since the beginning of the year. (The newsletter typically contains 1-5 items under the “security” heading every week.)

From a program analysis standpoint, all detected vulnerabilities have the same general structure: they correspond to warnings of the form “an *untrusted caller* C can reach argument X of a sensitive operation and supply parameter Y that is tainted” or “an untrusted caller can reach a sensitive operation (at all)”. That is, the vulnerability warnings typically query the main relations produced by the analysis: $X \rightarrow Y \{ *; [\text{sender} \rightarrow C] \}$, as well as $|i| \{ *; [\text{sender} \rightarrow C] \}$. The untrusted caller C corresponds to symbolic value «unprivileged-user», as seen in earlier examples, stored in the transactional dependencies of the symvalic analysis. The tainted parameter value Y is typically the symbolic value «user-unique-value», mentioned earlier, which designates that the value is completely unconstrained. The exact nature of the sensitive operation with argument X varies by vulnerability. For instance:

- *transferFrom* [Dedaub 2021a]: the contract is authorized to manipulate the funds of some accounts, and its code allows a direct transfer of funds from a tainted source to a tainted sink.
- *loan* [Dedaub 2021a]: similar to the above, the contract is authorized to manipulate funds, but the manipulation is limited to take place with specific kinds of funds, after a loan is received. Both the reachability of the code by an untrusted caller and the taintedness of the funds sink are essential.
- *swap* [Dedaub 2021c]: an exchange of funds from one token to another, (taking place after a loan and a liquidation of “shares”). The taintedness of the token being swapped and of the amount swapped are essential to the attack.
- *flashswap* [Primitive Finance 2021]: code executed upon an external service’s granting of a loan does not check that the loan parameters were as requested: the attacker can invoke the callback with tainted loan parameters (e.g., tainted token).

- *manipulated swap* [Dedaub 2021d]: the contract periodically converts its profits, and anyone can invoke this functionality. The attack is based on the reachability of this code by untrusted callers, and not on taintedness. The attacker calls the functionality exchanging the funds after having manipulated the online prices of the exchange service doing the conversion.

As can be discerned from the above descriptions, the attack point is buried deep in the code, under several complex conditions. One can, therefore, see why the precision (i.e., the path sensitivity) and completeness of symvalic analysis is important for the detection of the vulnerability. As we write in a vulnerability report [Dedaub 2021a]: “What made our symbolic-value flow analysis useful was not that it captured this instance but that it *avoided* warning about others that were *not* vulnerable. The analysis gave us just 27 warnings about such vulnerabilities out of the 40 thousand most-recently deployed contracts!”

7 CONTROLLED EVALUATION

Although symvalic analysis is continuously tested in the real world, this does not constitute a systematic evaluation, capable of demonstrating coverage, precision, and recall metrics, also in comparison to other tools. The symvalic analysis design is expected to offer both precision, and greater completeness compared to past static or hybrid analyses (or, alternatively, much greater completeness than a pure symbolic execution approach).

We next evaluate symvalic analysis experimentally in a controlled environment, against some of the best-known tools in the space of Ethereum smart contracts. To obtain ground truth for this systematic study, we consider the benchmarks of the recent SmartBugs work by Durieux et al. [2020]. SmartBugs offers a curated set of contracts, labeled with known vulnerabilities, and presents an empirical assessment of several security tools for Ethereum. The Mythril tool [Honig 2020], by Consensys, is the clear winner in the SmartBugs study, in terms of ability to detect vulnerabilities (with low rates of flagging un-tagged programs for vulnerabilities), and performs very well in terms of scalability. Furthermore, Mythril is a good comparable for symvalic analysis, since it is also a hybrid tool: it performs both symbolic execution and static analysis, and freely chooses between the approaches depending on the vulnerabilities it tries to detect.

Since symvalic analysis is close in spirit to symbolic execution, we also compare against the Manticore tool [Mossberg et al. 2019], by Trail-of-Bits. Manticore is a symbolic execution tool and one of the best-known industrial tools (together with Mythril) in the Ethereum space.

There are tens of security analysis tools in the Ethereum space. The SmartBugs study has already considered some of the most visible ones (including HoneyBadger [Torres et al. 2019], Maian [Nikolić et al. 2018], Osiris [Torres et al. 2018], Oyente [Luu et al. 2016], Securify [Tsankov et al. 2018], Slither [Feist et al. 2019], and Smartcheck [Tikhomirov et al. 2018]). Mythril outperforms all by a significant margin [Durieux et al. 2020], therefore, barring major recent upgrades, our conclusions (comparing with Mythril) should apply.

We “normalize” the SmartBugs benchmarks in the following ways:

- We eliminate or supplement benchmarks that exhibit a vulnerability but do not use the manipulated result, or otherwise prevent the vulnerability from really exhibiting itself. For instance, there were multiple benchmarks in which the vulnerable code was dead, removed during compilation by the Solidity compiler. These benchmarks are invalid for an automated evaluation of tools, so we added minimal code to ensure the vulnerability is real.
- We split and clone benchmarks so that each benchmark program exhibits vulnerabilities of exactly one type, removing non-labeled vulnerabilities present in the original SmartBugs benchmark. In this way, any extra (i.e., non-labeled) vulnerabilities are guaranteed, to be false positives. After this normalization, we remove contracts that exhibit the same vulnerability

in identical code patterns, so that each benchmark captures a different aspect of vulnerability detection. These are good properties that were not maintained in the original form of the programs.

Furthermore, since the SmartBugs benchmarks are unrealistically biased towards true vulnerable contracts (which are rare in practice), we complement our experiments with a second dataset, intending to demonstrate precision (i.e., low false-positive rates). This dataset consists of contracts that are *extremely unlikely to have serious vulnerabilities*. Specifically, these are all the contracts (a total of 166 unique bytecodes) that handle the 10,000 highest-value (by ETH balance) Ethereum blockchain accounts, as of early 2021.⁵ The least valuable of these contracts manages some \$350K, while the most valuable ones manage hundreds of millions.

The benchmark collection from our study is available as a public repository.⁶ We believe it constitutes an excellent evaluation suite for future studies, although the dramatic speed of evolution of real-world smart contracts will require continuous update for relevance.

The results of the controlled evaluation can be obtained using the publicly-available peer-reviewed version of the artifact [Smaragdakis et al. 2021] accompanying the paper.

7.1 Setup

We use the Souffle Datalog engine (v.2.0.2) on a machine with two Intel(R) Xeon(R) Gold 6136 CPUs 3.00GHz. (Each CPU has 12 cores, or 24 hardware threads. The machine has ample RAM for the tasks we run, at 768GB. We start up to 40 concurrent jobs and none of our experiments should show significant interference between jobs.) For all tools, timeout is set to 20mins (1200sec). This is a highly generous time allowance, intended to reveal the full ability of the tools for bug detection, regardless of their time costs.

We implemented symbolic clients for the top-5 (by number of vulnerable contracts) vulnerability categories from the SmartBugs repository [Various 2020].⁷ The clients build atop the output relations of Figure 4, such as “variable may hold value under dependencies”.

7.2 Ability to Detect Bugs

The normalized SmartBugs corpus contains 109 contracts with labeled vulnerabilities. We analyze all contracts in bytecode form, however Manticore is primarily usable with source code input,⁸ therefore we use source code for it.

Table 1 shows the recall (i.e., percentage of true vulnerabilities identified) of all tools on the curated dataset of 109 contracts. We evaluate symbolic analysis pessimally: First, we include the 16 contracts with vulnerabilities that symbolic analysis does not support, under the “Other*” category. (The numbers of these vulnerabilities are counted just like the rest, when higher is better, and not counted when lower is better, penalizing symbolic analysis for the omission of these clients.) Second, we include benchmarks even if we do not agree with them. For instance, in the *Access*

⁵Most high-value accounts are “wallets”, i.e., they don’t have code behind them. However, roughly 1-in-10 are handled by smart contracts. We deduplicate their code to get the 166 unique bytecodes to analyze.






















⁶<https://github.com/nevillegrech/gigahorse-benchmarks>

⁷We did not expend effort to develop clients for the last 5 of the vulnerabilities in the SmartBugs repository because a) these would be numerically insignificant for the experiment, accounting only for 16 out of 109 warnings in the benchmarks; b) we do not consider these clients to be promising for finding bugs in real-world contracts. E.g., the top-3 such vulnerabilities are “bad randomness”, “time manipulation”, and “front running”. These can be caught with specific code pattern-matching in simplistic benchmarks, but these well-known patterns never appear in real contracts any more. The full form of the vulnerability, however, is semantic and cannot be discerned reliably by static analysis without deep understanding of a program’s semantics, i.e., interaction with the programmer and programmer-supplied invariants.

⁸We expended much effort to run Manticore on bytecode input and got results that were significantly worse. In all experiments, we tuned Manticore according to instructions supplied by its authors, in direct communication.

Control category, our assessment is that symvalic analysis finds all true vulnerabilities, although the tables lists it as missing 6.⁹

Table 1. **Recall:** Percentage of true vulnerabilities detected by each tool. Also statement coverage, timeout/s/errors, average runtimes. **Fuller bars are better.**

Category	Manticore	Mythril	Symvalic
Access Control	3/16 19% 	11/16 69% 	10/16 63% 
Arithmetic Error	10/13 77% 	11/13 85% 	11/13 85% 
Denial of Service	0/6 0% 	0/6 0% 	4/6 67% 
Reentrancy	8/23 35% 	19/23 83% 	21/23 91% 
Unchecked Low-Level Calls	3/35 9% 	14/35 40% 	35/35 100% 
Other*	3/16 19% 	5/16 31% 	0/16 0% 
Total	27/109 25% 	60/109 55% 	81/109 74% 
Avg. Coverage	74%	86%	96%
Timeouts+Errors	54 (50%)	10 (9.17%)	0 (0%)
Avg. Runtime	231.1 seconds	107.3 seconds	2.9 seconds

Notably, Manticore times out for half the contracts and has an average running time of 185sec. (For symvalic analysis, the *maximum* running time over any of the benchmarks is 50sec.)

Qualitatively the tools justify their architectural characteristics. Manticore is the closest to a pure symbolic execution tool in the set, and exhibits low completeness: both the statement coverage metric and its percentage of vulnerabilities detected lag behind other tools—a result also observed by Durieux et al. [2020].

Mythril and the symvalic analysis both detect a significant number of the vulnerabilities. Symvalic analysis exhibits very high (original bytecode) statement coverage (96% vs. Mythril’s 86% and Manticore’s 74% but with numerous timeouts, for the more complex contracts). In total, symvalic analysis detects 80 of the 109 vulnerabilities, whereas Mythril detects 60 and Manticore 27 (including the Other* vulnerabilities, which symvalic analysis does not support). Additionally, the average running time of symvalic analysis is much lower than that of other tools (e.g., 2.9sec vs. 107.3sec for Mythril).




















Recall on its own means little. The message of symvalic analysis is that top-of-class recall can be achieved while combined with *unsurpassed precision*. Table 2 shows the percentage of labeled (in the curated dataset) vulnerabilities among the total flagged by every tool.

As can be seen in the table, Manticore is fully precise regarding its 24 labeled vulnerabilities. (This is commensurate with Manticore’s nature as a symbolic execution engine, exhibiting high precision, but one should also keep in mind its 50% timeout rate, which shows that the results only apply to the simplest contracts.) Mythril issues 104 reports to flag its 55 labeled contracts, while it times out on 10 out of the 109 contracts. Symvalic analysis reports just 95 vulnerabilities, of which 80 are true positives.

One may ask, what are the 15 false positives of symvalic analysis? Is the *dependencies* mechanism not sufficient to maintain precision relative to actual program execution? By inspection, we find that most of these false positives are not due to imprecise modeling of execution, but due to modeling

⁹Two of these are for an outdated vulnerability (checks over tx.origin instead of msg.sender, which is hardly indicative of a problem in modern contracts), two are over patterns that modern Solidity versions disallow (so they could only arise for old code, or for assembly code that explicitly implements a security hole), and two are calling a hard-coded external contract with caller-influenced values (a practice that is common and overwhelmingly not indicating a vulnerability).

Table 2. **Precision:** Percentage of vulnerabilities (per category) reported by each tool that truly have the corresponding vulnerability. The “Total” for Mythril and Manticore does not include the “Other*” category, so that the tools do not look worse merely on vulnerabilities that symbolic analysis does not support. **Fuller bars are better.**

Category	Manticore	Mythril	Symvalic
Access Control	3/3 100% 	11/19 58% 	10/13 77% 
Arithmetic Error	10/10 100% 	11/18 61% 	11/14 79% 
Denial of Service	0/0 -	0/23 0% 	4/10 40% 
Reentrancy	8/8 100% 	19/30 63% 	21/23 91% 
Unchecked Low-Level Calls	3/3 100% 	14/14 100% 	35/39 90% 
Other*	3/3 100% 	5/13 38% 	0/0 -
Total	24/24 100% 	55/104 53% 	81/99 82% 

of the external environment. For instance, the analysis cannot easily make assumptions about what values may be returned by an external call. For completely unknown calls, it will assume that the call can return anything. However, the call may have a specific protocol—e.g., only return a success flag for the current contract. Thus the analysis will model behaviors that will not be realizable in practice. (This can be remedied with complete models of the environment, or by taking into account dynamic behaviors in past transactions. We do not supply such information to any of the evaluated tools, nor is it clear that they can easily accept it. However, this is clearly a promising future direction.)

7.3 (Likely) False Warnings

































The curated dataset of the SmartBugs study gives an indication as to the ability of an analysis to find well-recognized vulnerabilities. However, real contracts deployed on the blockchain are a) more complex; b) much less likely to be vulnerable (and when they are, they are much less likely to hold Ether [Perez and Livshits 2021]).

Therefore we perform a second experiment with fully-realistic, ultra-high-value contracts: the 166 code bases (some, such as multisig wallets, instantiated tens or hundreds of times) that handle the 10,000 highest-value (by ETH balance) Ethereum blockchain accounts. Although there is no full ground truth on vulnerabilities available for these contracts, a better analysis is expected to a) scale well for them; b) yield few warnings in categories that have direct financial impact (i.e., Reentrancy, Access Control, or Arithmetic Error) since true vulnerabilities in these categories are extremely unlikely to exist.

Table 3 shows how the tools fare over the high-value contracts. Manticore could only apply to the 90 of the 102 contracts that have source code available, due to issues regarding their compilation. In these, it suffered 92% timeouts. In the three categories where warnings are *almost certainly false positives* (Reentrancy, Access Control, and Arithmetic Error), symvalic analysis issues just a small fraction of the warnings of Mythril. Overall, symvalic analysis issues just 61 vulnerability reports (13 in the “false positive” categories) over the 146 contracts that it analyzed, timing out for 20 contracts. The analysis exhibits high statement coverage (given the difficulty of the contracts) at 83%.

Mythril issues 81 reports (reports for “Other*” not included) and fails to complete for 45 of the 166 contracts. However, Mythril still issues reports for contracts that time out. We list them in a separate column because they help illustrate that *timeouts are not random*: both the percentage of flagged contracts and the coverage grow once contracts that timeout are included, despite them

Table 3. Coverage, total timeouts/errors, average runtimes, and reports per vulnerability class for high-value contracts: all unique contract bytecodes that manage the top-10K Ethereum accounts. The denominator of fractions is the number of contracts analyzed by the given tool/configuration. The numerator may be larger, since multiple reports are possible for the same contract in this dataset. We exclude the “Other*” category from the Total for Manticore and Mythril, to avoid penalizing them for warnings that symvalic does not support. We include a separate column with Mythril warnings for contracts that were *partially* analyzed (i.e., timed out, but produced some results). **Empty bars are better, except for the “unchecked low-level calls” and “denial of service” categories (which include mostly/many true positives, respectively).**

Category	Manticore	Mythril	Mythril w. timeouts	Symvalic
Access Control	0/7 0% 	8/121 7% 	14/163 9% 	8/148 5% 
Arithmetic Error	0/7 0% 	54/121 45% 	74/163 45% 	3/148 2% 
Denial of Service	0/7 0% 	6/121 5% 	23/163 14% 	13/148 9% 
Reentrancy	0/7 0% 	12/121 10% 	34/163 21% 	2/148 1% 
Unchecked Low-Level Calls	0/7 0% 	1/121 1% 	1/163 1% 	37/148 25% 
Other*	4/7 57% 	22/121 18% 	42/163 26% 	0/148 0% 
Total	4/7 57% 	81/121 67% 	146/163 90% 	61/148 41% 
Avg. Coverage	57%	47%	49%	83%
Timeouts+Errors	83/90 92% 	45/166 27% 	3/166 2% 	18/166 11% 
Avg. Runtime	215.5 seconds	186.73 seconds	475.74 seconds	39.2 seconds

only being analyzed partially. This indicates that complex contracts lead both to more reports and to timeouts, and, in fact, timeouts hide extra Mythril reports. With partially-analyzed contracts included, Mythril issues 146 reports over just 163 contracts—122 of them in the three “false positive” categories. This is precisely the performance of past tools (issuing too many invalid warnings for the security inspector to inspect effectively) that Symvalic analysis aims to address. Furthermore, Mythril issues these reports while only covering 49% of the contracts’ statements.

Conversely, the “unchecked low-level calls” category indeed contains several true positives, even in deployed, high-value contracts. The analysis of all tools (as well as the labels of the curated dataset) define the vulnerability as lack of a check of a return value, regardless of the further implications this omission may have. Indeed, the vulnerability should be seen as more of a “bad smell”. To some extent, the same is true of the “denial of service” category: it may be true that, e.g., one of the signers of a multi-sig wallet can overflow a storage array, rendering the contract unusable. However, this may not be part of the valid threat model for the contract. The vulnerability should still be considered correctly reported, even though the contract is not third-party-exploitable. Symvalic analysis issues many warnings in these categories and a brief sampling reveals that most are true positives.

Summary. The experiments with the curated (SmartBugs) and high-value datasets confirm and quantify what has been our informal experience with symvalic analysis: it is an analysis that offers scalable, precise modeling combined with much greater program coverage than other precise (path-sensitive) static techniques, such as symbolic execution.

8 RELATED WORK

We have referred to directly related work extensively throughout the paper, therefore we next only discuss conceptual relatives that are not otherwise directly comparable.

General Program Analysis and Verification. Relational analysis techniques have been successfully applied in recent years, to tackle the problem of JavaScript static analysis, where precision is critical to getting a useful analysis. *Value partitioning* [Nielsen and Møller 2020], is an efficient trace partitioning [Rival and Mauborgne 2007] variant, where the analysis does not attempt to refine abstract states, but instead, abstract values. This approach manages to circumvent the expensive abstract state partitioning [Ko et al. 2017] or additional backwards analysis [Stein et al. 2019] that previous approaches required, while maintaining precision.

Symbolic execution has seen numerous variations that offer a different balance of scalability, completeness, and precision. *Compositional* symbolic execution [Anand et al. 2008; Godefroid 2007] has attempted to address scalability issues by use of summaries. *Steering* techniques [Li et al. 2013; Park et al. 2012] attempt to achieve higher coverage or depth, e.g., by prioritizing paths that are yet unexplored. Symvalic analysis has similar goals, but its coverage is only a small part of the story: as a static analysis, it explores many values at once and coverage is only incidental. At the same time, it may suffer from higher imprecision than symbolic execution techniques, since precision is limited by its current dependencies scheme.

Accordingly, symvalic analysis can be viewed as an attempt to address the state explosion problem. The model checking literature contains several approaches with similar goals, ranging from compositional *assume-guarantee reasoning* [Abadi and Lamport 1993] to symmetry reduction [Emerson and Sistla 1996; Norris Ip and Dill 1993], to partial order reduction [Flanagan and Godefroid 2005]. Symvalic analysis uses a very different scheme, due to both symbolic reasoning and its controlled sacrifice of precision. Conceptually, the combination of abstract interpretation and model checking (e.g., [Clarke et al. 1994]) has a similar flavor, but the actual mechanisms differ substantially.

Whitebox [Chipounov et al. 2011; Godefroid et al. 2008, 2012] and (later) greybox [Böhme et al. 2016; Various 2017; Wüstholtz and Christakis 2020a,b] fuzzing are approaches that use insights similar to those of symvalic analysis, in an effort to achieve coverage of a program under test, especially when the program makes use of very low level library code that is externally modelled. These approaches work by “fuzzing” an input, following the control flow of a program for concrete values, yet also potentially using constraint solvers to modify the input to follow alternative control flow. In the space of smart contracts, where the program is fully modeled, and bugs manifest themselves in several transactions, symvalic analysis can scale better and cover more program behaviors, since it is fundamentally a static analysis, overapproximating dynamic conditions and collapsing many paths.

Symvalic analysis is rather unconventional among analysis techniques, mainly in the ways described earlier: it uses symbolic expressions inside a full *static* analysis (not dynamic-symbolic execution), without delegating the solution of large expressions to an external constraint solver. There have been other combinations of symbolic expressions and static analyses, especially for custom analysis clients—e.g., Dudina and Belevantsev [2017] employ a symbolic static analysis for buffer overflow detection. In contrast, symvalic analysis is client-agnostic: symbolic expressions are used as regular values for *any* variable in the program text, without targeting specific kinds of expressions or specific program features. Furthermore, the mechanism of dependencies (Section 3.2) is key in the symvalic design, for purposes of precision.

There are certainly many more points in the static analysis design space and some can be compared for reference. SPLift [Bodden et al. 2013] shows a modular analysis for software product lines. This is almost at the opposite end of the spectrum from symvalic analysis: a very scalable analysis, but much less precise. The SPLift analysis is explicitly based on the IFDS framework, which means that it summarizes at the procedure boundary, thus losing precision to gain scalability. It is interesting to consider whether symvalic analysis could apply to large-scale software product

lines. This direction would certainly require significant work for a fruitful approach. For instance, symvalic analysis would likely apply to each Java file separately with the dependencies between different files modeled rather loosely (e.g., perhaps a single predicate to capture the configuration of the product line). Padhye and Khedker [2013] describe an analysis that achieves significant precision (flow- and context sensitivity) in a fairly general analysis setting. This may be a promising candidate for future combinations with symvalic analysis ideas, especially the use of dependencies as a data-flow-value context.

Program Analysis for Ethereum Smart Contracts. The small size and high value of Ethereum smart contracts has made them a suitable target for applying a variety of program analysis techniques, resulting in a plethora of academic [Brent et al. 2020; Grech et al. 2018; Kolluri et al. 2019; Lagouvardos et al. 2020; Luu et al. 2016; Nguyen et al. 2020; Nikolić et al. 2018; Torres et al. 2019; Tsankov et al. 2018] and industrial [Feist et al. 2019; Grieco et al. 2020; Honig 2020; Mossberg et al. 2019; The Certora team 2017] tools emerging. While most smart contract tools focus on vulnerability detection, past work has also focused on empirically identifying optimization opportunities [Chen et al. 2020], gas cost estimation [Albert et al. 2020a] using recurrence-relation theories or even superoptimization [Albert et al. 2020b] using SMT.

Tools [Hajdu and Jovanović 2020; Kolluri et al. 2019; Krupp and Rossow 2018; Luu et al. 2016; Mossberg et al. 2019; Nikolić et al. 2018] applying symbolic execution techniques have been very popular due to the precision of their reported warnings. (And also, a cynic might remark, their ease of implementation on a platform where the source language changes constantly and the low-level IR is extremely hard to analyze.) In Section 7 we evaluated our symvalic analysis against Mythril [Honig 2020] and Manticore [Mossberg et al. 2019], two state-of-the-art tools that employ symbolic execution techniques, with symvalic analysis having higher warning precision and statement coverage with vulnerability completeness comparable or superior to that of the best competitor.

Additionally, approaches [Albert et al. 2018; Brent et al. 2020, 2018; Feist et al. 2019; Grech et al. 2018; Lagouvardos et al. 2020; Tsankov et al. 2018] based on static analysis have seen success due to their high completeness and scalability. Even though conventional static analysis tools [Brent et al. 2020] have achieved high precision by tuning the analysis to common programming patterns found in Ethereum smart contracts, as we discussed in Section 2, symvalic analysis offers a more precise analysis while being agnostic to these specific program patterns.

Fuzzing-based tools [He et al. 2019; Jiang et al. 2018; Nguyen et al. 2020; Wüstholtz and Christakis 2020a,b] have also been successful in precisely reporting smart contract vulnerabilities. Notably, the recently presented Harvey fuzzer [Wüstholtz and Christakis 2020b] combines static analysis with fuzzing by using static analysis to guide a greybox fuzzer.

9 CONCLUSIONS

We presented symvalic analysis: a value-flow static analysis where the values can be symbolic expressions. The analysis achieves a rare balance between scalability, precision, and completeness—squeezed between the state explosion problem and imprecise approximation. For the setting of Ethereum smart contracts, the analysis yields high-value vulnerability warnings, which have already resulted in significant real-world impact, rescuing several millions in vulnerable funds.

We view symvalic analysis as an approach that holds promise for taming the state explosion problem in the general software verification and validation setting. Traditional execution-based approaches (including concrete execution, testing, dynamic-symbolic execution, and model checking) are *horizontal*: they keep an extremely detailed state about all variables/memory locations at every single point of modeling a program’s behavior. This results in the precision of the modeling but also in difficulties in scalability and behavior completeness. In contrast, value-flow static analysis

techniques are *vertical*: they keep a large set of values per-variable (or other program entity), but the values are rarely correlated—resulting in completeness but precision loss. Symbolic value-flow analysis keeps some of the horizontal and some of the vertical elements. It attempts to keep much of the completeness of static analysis, yet make it precise by a) employing concrete values and symbolic constraint solutions when necessary to satisfy program path conditions; b) connecting values together via the mechanism of dependencies. We hope that the technique will find further future adoption, helped by strong evidence of impact in the domain of Ethereum smart contracts.

ACKNOWLEDGMENTS

We gratefully acknowledge funding by the Hellenic Foundation for Research and Innovation (HFRI project DEAN-BLOCK).

REFERENCES

- Martín Abadi and Leslie Lamport. 1993. Composing Specifications. *ACM Trans. Program. Lang. Syst.* 15, 1 (Jan. 1993), 73–132. <https://doi.org/10.1145/151646.151649>
- Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2020a. GASOL: gas analysis and optimization for ethereum smart contracts. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 118–125.
- Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. 2018. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *Automated Technology for Verification and Analysis (ATVA)*. Springer International Publishing, 513–520.
- Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria A. Schett. 2020b. Synthesis of Super-Optimized Smart Contracts Using Max-SMT. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 177–200.
- Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020c. Taming Callbacks for Smart Contract Modularity. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 209 (Nov. 2020), 30 pages. <https://doi.org/10.1145/3428277>
- Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-Driven Compositional Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 367–381.
- Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. <https://doi.org/10.1145/3182657>
- Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-Based Static Analysis (Extended Version). *CoRR* abs/2009.08361 (2020). arXiv:2009.08361 <https://arxiv.org/abs/2009.08361>
- Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPL^{LIFT}: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) (*PLDI ’13*). Association for Computing Machinery, New York, NY, USA, 355–364. <https://doi.org/10.1145/2491956.2491976>
- Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) (*CCS ’16*). Association for Computing Machinery, New York, NY, USA, 1032–1043. <https://doi.org/10.1145/2976749.2978428>
- Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) (*PLDI 2020*). Association for Computing Machinery, New York, NY, USA, 454–469. <https://doi.org/10.1145/3385412.3385990>
- Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *CoRR* abs/1802.08660 (2018). arXiv:1809.03981 <https://arxiv.org/abs/1809.03981>
- Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 209–224.
- T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang. 2020. GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts. *IEEE Transactions on Emerging Topics in Computing* (2020), 1–1. <https://doi.org/10.1109/TETC.2020.2979019>

- Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 46. Association for Computing Machinery, New York, NY, USA, 265–278. <https://doi.org/10.1145/1961296.1950396>
- E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (April 1986), 244–263. <https://doi.org/10.1145/5397.5399>
- Edmund M. Clarke, Orna Grumberg, and David E. Long. 1994. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.* 16, 5 (Sept. 1994), 1512–1542. <https://doi.org/10.1145/186025.186051>
- Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15
- Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. 2016. Just test what you cannot verify!. In *Software Engineering 2016*, Jens Knoop and Uwe Zdun (Eds.). Gesellschaft für Informatik e.V., Bonn, 17–18.
- Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Budapest, Hungary) (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.
- Dedaub. 2021a. Ethereum Pawn Stars: '\$5.7M in hard assets? Best I can do is \$2.3M'. <https://medium.com/dedaub/ethereum-pawn-stars-5-7m-in-hard-assets-best-i-can-do-is-2-3m-b93604be503e>
- Dedaub. 2021b. Killing a Bad (Arbitrage) Bot ... to Save its Owners. <https://medium.com/dedaub/killing-a-bad-arbitrage-bot-f29e7e808c7d>
- Dedaub. 2021c. Look Ma', no source! Hacking a DeFi Service with No Source Code Available. <https://medium.com/dedaub/look-ma-no-source-hacking-a-defi-service-with-no-source-code-available-c40a6583f28f>
- Dedaub. 2021d. Yield Skimming: Forcing Bad Swaps on Yield Farming. https://medium.com/dedaub/yield-skimming-forcing-bad-swaps-on-yield-farming-397361fd7c72?source=friends_link&sk=d146b3640321f0a3ccc80540b54368ff
- I. Dudina and A. Belevantsev. 2017. Using static symbolic execution to detect buffer overflows. *Programming and Computer Software* 43 (2017), 277–288.
- Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (Seoul, South Korea) (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 530–541. <https://doi.org/10.1145/3377811.3380364>
- E. Emerson and A. Sistla. 1996. Symmetry and Model Checking. *Formal Methods in System Design* 9 (08 1996), 105–131. <https://doi.org/10.1007/BF00625970>
- Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. 2019 *IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)* (May 2019). <https://doi.org/10.1109/wetseb.2019.00008>
- Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) (POPL '05). Association for Computing Machinery, New York, NY, USA, 110–121. <https://doi.org/10.1145/1040305.1040315>
- Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 47–54.
- Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 213–223. <https://doi.org/10.1145/1065010.1065036>
- Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*, Vol. 8. 151–166. <https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/>
- Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft. *Commun. ACM* 10, 1 (Jan. 2012), 20–27. <https://doi.org/10.1145/2090147.2094081>
- Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) (ICSE '19). IEEE Press, Piscataway, NJ, USA, 1176–1186. <https://doi.org/10.1109/ICSE.2019.00120>
- 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 Programming Languages* 2, OOPSLA (Nov. 2018). <https://doi.org/10.1145/3276486>
- 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 (Virtual Event, USA) (*ISSTA 2020*). Association for Computing Machinery, New York, NY, USA, 557–560. <https://doi.org/10.1145/3395363.3404366>
- Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *Proc. ACM Program. Lang.* 2, POPL, Article 48 (Dec. 2017), 28 pages. <https://doi.org/10.1145/3158136>
- Ákos Hajdu and Dejan Jovanović. 2020. solc-verify: A Modular Verifier for Solidity Smart Contracts. In *Verified Software. Theories, Tools, and Experiments*, Supratik Chakraborty and Jorge A. Navas (Eds.). Springer International Publishing, Cham, 161–179.
- Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) (*CCS '19*). Association for Computing Machinery, New York, NY, USA, 531–548. <https://doi.org/10.1145/3319535.3363230>
- E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 204–217. <https://doi.org/10.1109/CSF.2018.00022>
- J.J. Honig. 2020. Incremental symbolic execution. <http://essay.utwente.nl/81526/>
- Ranjit Jhala and Rupak Majumdar. 2009. Software Model Checking. *ACM Comput. Surv.* 41, 4, Article 21 (Oct. 2009), 54 pages. <https://doi.org/10.1145/1592434.1592438>
- Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) (*ASE 2018*). Association for Computing Machinery, New York, NY, USA, 259–269. <https://doi.org/10.1145/3238147.3238177>
- James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. <https://doi.org/10.1145/360248.360252>
- Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. 2017. Weakly Sensitive Analysis for Unbounded Iteration over JavaScript Objects. In *Programming Languages and Systems*, Bor-Yuh Evan Chang (Ed.). Springer International Publishing, Cham, 148–168.
- Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2019. Exploiting the Laws of Order in Smart Contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) (*ISSTA 2019*). Association for Computing Machinery, New York, NY, USA, 363–373. <https://doi.org/10.1145/3293882.3330560>
- Georgios Konstantopoulos. 2021. [Informal public comment, July 15, 2021]. ETHSecurity Community Telegram channel.
- Johannes Krupp and Christian Rossow. 2018. TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) (*SEC'18*). USENIX Association, Berkeley, CA, USA, 1317–1333. <http://dl.acm.org/citation.cfm?id=3277203.3277303>
- Sifis Lagouvardos, Neville Grech, Ilias Tsatiris, and Yannis Smaragdakis. 2020. Precise Static Modeling of Ethereum “Memory”. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 190 (Nov. 2020), 26 pages. <https://doi.org/10.1145/3428258>
- You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) (*OOPSLA '13*). Association for Computing Machinery, New York, NY, USA, 19–32. <https://doi.org/10.1145/2509136.2509553>
- Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) (*CCS '16*). Association for Computing Machinery, New York, NY, USA, 254–269. <https://doi.org/10.1145/2976749.2978309>
- B. Meyer. 2008. Seven Principles of Software Testing. *Computer* 41, 8 (2008), 99–101. <https://doi.org/10.1109/MC.2008.306>
- Michales, Jonah. 2021. Inside the War Room That Saved Primitive Finance. <https://medium.com/immunefi/inside-the-war-room-that-saved-primitive-finance-6509e2188c86>
- Anders Møller and Michael I. Schwartzbach. 2018. Static Program Analysis. Department of Computer Science, Aarhus University, <http://cs.au.dk/~amoeller/spa/>, Accessed: 2020-11-20.
- M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 1186–1189. <https://doi.org/10.1109/ASE.2019.00133>
- Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. SFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) (*ICSE '20*). Association for Computing Machinery, New York, NY, USA, 778–788. <https://doi.org/10.1145/3377811.3380334>
- Benjamin Barslev Nielsen and Anders Møller. 2020. Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript. In *Proc. 34th European Conference on Object-Oriented Programming (ECOOP)*.
- Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*.

- Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) (ACSAC '18). ACM, New York, NY, USA, 653–663. <https://doi.org/10.1145/3274694.3274743>
- C. Norris Ip and David L. Dill. 1993. Better Verification Through Symmetry. In *Computer Hardware Description Languages and their Applications*, DAVID AGNEW, LUC CLAESEN, and RAUL CAMPOSANO (Eds.). North-Holland, Amsterdam, 97 – 111. <https://doi.org/10.1016/B978-0-444-81641-2.50012-5>
- Rohan Padhye and Uday P. Khedker. 2013. Interprocedural Data Flow Analysis in Soot Using Value Contexts. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis* (Seattle, Washington) (SOAP '13). Association for Computing Machinery, New York, NY, USA, 31–36. <https://doi.org/10.1145/2487568.2487569>
- Sangmin Park, B. M. Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. 2012. CarFast: Achieving Higher Statement Coverage Faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) (FSE '12). Association for Computing Machinery, New York, NY, USA, Article 35, 11 pages. <https://doi.org/10.1145/2393596.2393636>
- Daniel Perez and Ben Livshits. 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Vancouver, B.C. <https://www.usenix.org/conference/usenixsecurity21/presentation/perez>
- A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1661–1677. <https://doi.org/10.1109/SP40000.2020.00024>
- Primitive Finance. 2021. PrimitiveFi post-mortem analysis. <https://primitivefinance.medium.com/postmortem-on-the-primitive-finance-whitehack-of-february-21st-2021-17446c0f3122>
- Xavier Rival and Laurent Mauborgne. 2007. The Trace Partitioning Abstract Domain. *ACM Trans. Program. Lang. Syst.* 29, 5 (Aug. 2007), 26–es. <https://doi.org/10.1145/1275497.1275501>
- Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) (CCS '20). Association for Computing Machinery, New York, NY, USA, 621–640. <https://doi.org/10.1145/3372297.3417250>
- Koushik Sen and Gul Agha. 2006. Cute and jCute: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Proc. International Conference on Computer Aided Verification (CAV)*. Springer, 419–423.
- Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. In *Proc. 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 263–272.
- Micha Sharir and Amir Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program flow analysis: theory and applications*, Steven S. Muchnick and Neil D. Jones (Eds.). Prentice-Hall, Inc., Englewood Cliffs, NJ, Chapter 7, 189–233.
- Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsadiris. 2021. *Symbolic Value-Flow Static Analysis: Deep, Precise, Complete Modeling of Ethereum Smart Contracts (Artifact)*. <https://doi.org/10.5281/zenodo.5494813>
- Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. 2019. Static Analysis with Demand-Driven Value Refinement. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 140 (Oct. 2019), 29 pages. <https://doi.org/10.1145/3360566>
- The Certora team. 2017. The Certora Prover. <https://www.certora.com>
- Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain* (Gothenburg, Sweden) (WETSEB '18). Association for Computing Machinery, New York, NY, USA, 9–16. <https://doi.org/10.1145/3194113.3194115>
- Nikolai Tillmann and Jonathan de Halleux. 2008. Pex - White Box Test Generation for .Net. In *Proc. 2nd International Conference on Tests And Proofs (TAP)*. Springer, 134–153.
- Nikolai Tillmann and Wolfram Schulte. 2006. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. *IEEE Software* 23, 4 (2006), 38–47.
- Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) (ACSAC '18). Association for Computing Machinery, New York, NY, USA, 664–676. <https://doi.org/10.1145/3274694.3274737>
- Christof Ferreira Torres, Mathis Steichen, and Radu State. 2019. The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1591–1607. <https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira>
- Trail of Bits. 2020a. Trail of Bits comments on average coverage. <https://forum.openzeppelin.com/t/symbolic-execution/2158/3> Accessed: 2020-11-20.
- Trail of Bits. 2020b. Tweet on symbolic execution coverage. <https://twitter.com/trailofbits/status/1223386823084384256> Accessed: 2020-11-20.

- Petar Tsankov, Andrei Dan, Dana Drachler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) (CCS '18). ACM, New York, NY, USA, 67–82. <https://doi.org/10.1145/3243734.3243780>
- Various. 2017. libFuzzer – a library for coverage-guided fuzz testing. <https://llvm.org/docs/LibFuzzer.html>
- Various. 2020. SmartBugs: A Framework to Analyze Solidity Smart Contracts. <https://github.com/smartbugs/smartbugs>
- Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. <http://gavwood.com/paper.pdf>.
- Valentin Wüstholtz and Maria Christakis. 2020a. Harvey: A Greybox Fuzzer for Smart Contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) (ESEC/FSE 2020). Association for Computing Machinery, New York, NY, USA, 1398–1409. <https://doi.org/10.1145/3368089.3417064>
- Valentin Wüstholtz and Maria Christakis. 2020b. Targeted Greybox Fuzzing with Static Lookahead Analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) (ICSE '20). Association for Computing Machinery, New York, NY, USA, 789–800. <https://doi.org/10.1145/3377811.3380388>