Program Analysis for High-Value Smart Contract Vulnerabilities: Techniques and Insights

YANNIS SMARAGDAKIS, Dedaub and University of Athens,
NEVILLE GRECH, Dedaub and University of Malta,
SIFIS LAGOUVARDOS, Dedaub and University of Athens,
KONSTANTINOS TRIANTAFYLLOU, Dedaub and University of Athens,
ILIAS TSATIRIS, Dedaub,
YANNIS BOLLANOS, Dedaub,
TONY ROCCO VALENTINE, Dedaub and University of Malta

A widespread belief in the blockchain security community is that automated techniques are only good for detecting shallow bugs, typically of small value. In this paper, we present the techniques and insights that have led us to repeatable success in automatically discovering high-value smart contract vulnerabilities. Our vulnerability disclosures have yielded 10 bug bounties, for a total of over $3M, over high-profile deployed code, as well as hundreds of bugs detected in pre-deployment or under-audit code.

We argue that the elements of this surprising success are a) a very high-completeness static analysis approach that manages to maintain acceptable precision; b) domain knowledge, provided by experts or captured via statistical inference. We present novel techniques for automatically inferring domain knowledge from statistical analysis of a large corpus of deployed contracts, as well as discuss insights on the ideal precision and warning rate of a promising vulnerability detector. In contrast to academic literature in program analysis, which routinely expects false-positive rates below 50% for publishable results, we posit that a useful analysis for high-value real-world vulnerabilities will likely flag very few programs (under 1%) and will do so with a high false-positive rate (e.g., 95%, meaning that only one-of-twenty human inspections will yield an exploitable vulnerability).

1 INTRODUCTION

Smart contracts—autonomous programs deployed and operating on the blockchain—present a particularly interesting setting for software verification and validation approaches. The reason is stringent correctness requirements, largely arising due to the large and direct impact of possible contract errors. Smart contracts manage monetary assets, often in the many millions or billions of dollars. The most common domain for smart contracts deployed in the past few 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 would have a significant practical impact on smart contracts. This has arguably not been the case, however. Perez and Livshits [2021] conduct a thorough study of tens of thousands of Ethereum contracts reported vulnerable by six 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.

Presented at the Science of Blockchain Conference, 2023 (SBC'23), Stanford University, Aug. 2023

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 smart contract security analysis. A recent quote (one of many) 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]. Having a practical impact on deployed contract security via automated techniques seems like a formidable challenge.

In this paper, we present the techniques and insights that we have used to meet this challenge. Although human understanding is essential for finding many of the high-value vulnerabilities in deployed smart contracts, our work shows that automated analyses *can* pinpoint high-value vulnerabilities. The statement is more than a near-trivial existence proof: it is based on *repeatable* results, that have yielded hundreds of vulnerability reports, as well as ten independent bug bounties, for a total of over $3M, over DeFi protocols managing billions in assets.

The technical core of our approach consists of two main parts. The first is *symbolic value-flow (symvalic)* analysis: a high-precision/high-completeness *static analysis* approach, whose technical elements are described in detail in a recent publication [Smaragdakis et al. 2021]. The second part is a *corpus analysis*: a complementary (to the static analysis) *statistical analysis* of irregular or unusual code patterns, relative to a large number of other deployed smart contracts.

An interesting element concerns the interplay of symvalic (i.e., static) analysis and corpus (i.e., statistical) analysis. The vocabulary of the statistical analysis is directly informed by properties established by the static analysis. For instance, a common concept over which anomalous code patterns can be detected is that of *tainting*: e.g., the combined analysis may ask

> "can an untrusted user *taint* (i.e., control) the *n*-th argument of an external DeFi protocol call *in this contract*, when that argument is typically determined to be *untainted in many other contracts* that call the same DeFi protocol?"

Finally, we present insights regarding the expected performance metrics of a program analysis for detecting high-value vulnerabilities. We argue that a useful analysis for complex vulnerabilities is unlikely to be very precise in terms of conventional metrics—and that is fine! A false-positive rate of 98% (1 of 50 warnings being truly exploitable) may be prohibitive for a top-tier academic publication, but it is perfectly expected when the domain of evaluation is not controlled benchmarks but actual, deployed, and scrutinized smart contracts. A warning rate of 0.5% (flagging one-of-every-200 contracts for a given vulnerability) may seem entirely unimpactful in terms of a research evaluation of the analysis benefit. ("99.5% of the programs are already correct, what is the need for an analysis to get this number to 100%?") But reality is quite the opposite: a 0.5% warning rate is entirely in the credible range for high-impact vulnerabilities.

2 BACKGROUND

Static Analysis. 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, practically, 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, is a fundamental conceptual challenge. Algorithmic design of static analyses has been trying to address this challenge for several decades, with thousands of different research innovations.

All static analysis design is, therefore, an effort to simultaneously achieve as much practical *precision* and *completeness* as possible. (The third dimension of analysis quality, *scalability* can mostly be ignored for the purposes of our presentation: we restrict our attention to algorithms that scale well, i.e., can give results in realistic time frames—e.g., in the hours—under current technology.) Generic definitions for the two terms can only be approximate, but, intuitively:

- an analysis is precise to the extent that the program behaviors it predicts are actually possible in program execution;
- an analysis is complete to the extent that the actually possible program executions are predicted.

Perfect precision and completeness is provably impossible, by standard undecidability results over program reasoning. Therefore a static analysis is by definition a game of approximation. Precision and completeness are typically measured experimentally (or one of them is guaranteed by-design, with the other measured experimentally). It should come as no surprise that an analysis picks what to approximate with more or less fidelity. For instance, the best known family of static analyses, *data-flow analyses* in conventional compilers, will commonly treat all code conditions as both satisfiable and negatable, without more complex reasoning.

Consider the example function below:[1]

```
function whichPaths(uint x) public (returns uint y) {
  y = 3;
  if (x % 2 != 0) {
    y++;
  }
  if (x % 4 != 0) {
    y = y * y;
  }
}
```

In this function, the actual possible values for return variable `y` are `3`, `16`, and `9`: it is not possible to satisfy the first conditional but not the second. A static analysis, however, may legitimately choose to consider all paths (i.e., all combinations of branches in the program control flow) to be realizable, sacrificing a bit of precision. Such an analysis could well predict that `4` (i.e., satisfying the first conditional but not the second) is a possible return value. This approximation loses precision in this example but gains the benefit of completeness when faced with other complex conditions. (Additionally, the approximation most likely is much more scalable than approaches that attempt full reasoning over symbolic conditions.)

Analysis of Smart Contracts. 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 likely 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

[1]We use the Solidity language for code examples. Solidity is dominant, accounting for more than 99% of deployed smart contracts on Ethereum, the largest programmable blockchain.

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. 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 Certora team 2017]). Our work is explicitly about *automated* techniques that apply universally, without needing human input (i.e., specifications) or per-smart-contract customization.

Our approach aims to be highly complete without sacrificing (much) precision. Completeness problems are often brought up in the domain of smart contracts. 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.

3 SYMBOLIC-VALUE-FLOW STATIC ANALYSIS

Symbolic-Value-Flow ("symvalic") analysis is the first weapon in our arsenal. We next present an informal overview, highlighting its design principles.

3.1 Overview

Symbolic-Value-Flow static analysis 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 the example function below:

```
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`».

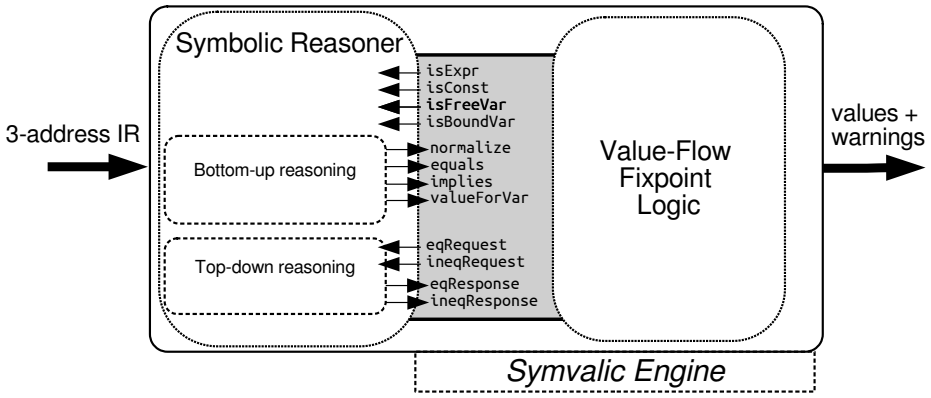

Fig. 1. Symvalic Analysis Architecture: API (slightly simplified) between symbolic reasoning and value propagation shown.

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.

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 1 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.

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.

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 most program behaviors, 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 $\mathsf{v} \rightarrow e\ \langle d^L; d^T \rangle$ 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 $\mathsf{v} \rightarrow e\ \langle D \rangle$.) 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` → «`owner`»] to appear in the transaction dependencies. (The

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

symbolic variable `sender` stands for the Solidity `msg.sender` expression and «`owner`» 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.)

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:

`nextBalance` → 181 ⟨ {[`to` → `0x42`], [`amount` → 200], [`curBalance` → 1] } ; {[`sender` → «`owner`»] } ⟩}

and

`nextBalance` → 260 ⟨ {[`to` → `0x42`], [`amount` → 200], [`curBalance` → 80] }; {[`sender` → «`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 ⊕. 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. 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 Precision 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.

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 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 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 CORPUS ANALYSIS

The symvalic analysis technology has been a great boon to our efforts to have repeatable high impact on smart contract security. However, the analysis by itself is just a way to get answers. An essential element of the approach is to know to ask the right questions. For instance, a very simple question may take the form "*is the first argument of a* `swap` *call tainted, i.e., controllable by an untrusted caller?*" That is, at the very least the analysis will need to know which arguments of external calls are sensitive and can lead to vulnerabilities. Other interesting questions concern missing information (e.g., missing a network identifier or a nonce when cryptographically signing messages), vulnerable code patterns (e.g., effects after a possibly-reentrant external call), and much more.

Formulating the right questions, i.e., writing symvalic analysis *clients*, requires significant domain expertise. Security experts, with deep knowledge of protocols and programming patterns in the DeFi space need to be involved. This effort needs to be ongoing, with new libraries and protocols (raising new requirements for integrating with them) being continuously introduced in the ecosystem.

Abstracting away for one step, the need for domain/security expertise is really a modularity enhancement. We need domain expertise so that the analysis can break up its effort in convenient single-contract chunks while having a summary of what *other* contracts expect in terms of their interactions with the current contract-under-analysis.

The second technique that we employ intends to address exactly this need. *Corpus analysis* is a summarization of the behavior of contracts and a subsequent statistical analysis. This enables extracting domain expertise without the help of human experts. For instance, it is not necessary for an expert to indicate that the first argument of a `swap` call is sensitive: if analysis of hundreds of past contracts shows that this argument is almost always *untainted* (i.e., only checked callers can supply it), the corpus analysis will establish this property as a question for the symvalic analysis of a contract, to report violations.

More specifically, corpus analysis is a summarization of contract behavior in two ways:

- For a contract itself, several key behavioral properties are summarized, so that the contract's callers can be checked in conjunction with these properties.
- For all the callers of a contract (more accurately, a specific API function), their "usual" behavior is summarized so that other callers can be checked for deviation from the common patterns.

Corpus analysis works in synergy with the static analysis of a contract: it both consumes the results of static analysis and produces results for the static analysis to consume.

Specifically, for each contract statically analyzed (via symvalic analysis), a wealth of behavioral summaries get produced. Some of the most important ones are listed below:

- Which functions can reach a `delegatecall` instruction (which is a common way to yield full control and, thus, a common vector for vulnerabilities).
- Which functions have which of their arguments control quantities with monetary significance (e.g., flowing to a money transfer source/destination/amount).
- Which functions perform state initialization.
- Which functions return values that can be manipulated by an external caller (i.e., values depending on storage state that any caller can affect—e.g., by depositing funds).
- Which functions allow reentrancy, i.e., yield control to a party supplied as a parameter. This does not indicate that the function itself suffers from a vulnerability: the function may not have state changes after the yielding of control, or the function may be checking that it gets called only by a trusted caller. However, *callers* of such a function may be vulnerable to reentrancy attacks.
- Which functions perform checked money transfer operations, i.e., examine the permissions of their caller. This implies that any caller of such a function, when propagating argument values to the function, should itself examine the permissions of *its* caller.
- Which functions perform guarded/unguarded (i.e., with checks as to the function's caller) external calls and to which other functions of external contracts.
- Which functions make external calls of monetary significance and whether (and which of) the arguments of such calls are tainted/untainted.

The above summaries get statistically analyzed and *re-imported* for further inference. Note the recursive nature of many of the above definitions. To produce results for tainted/untainted external calls of monetary significance, the analysis needs to first receive information as to *which* external calls have monetary significance. To produce functions that allow reentrancy, we need to know whether the external functions *they* call allow reentrancy.

The most useful statistical summarization results of corpus analysis are some of the simplest: e.g., whether a specific argument of an API is usually tainted/untainted; whether a given function is usually called only by trusted/checked callers or by anyone.

Feeding back into the static analysis, the results of corpus analysis help formulate the right analysis questions:

- Knowing that an external call allows reentrancy allows detecting *transitive reentrancy* attacks. It is worth emphasizing again that the external function itself may not be vulnerable, even though it yields control to a third party: the function may be checking its caller, or its code may have no permanent effects after yielding control. However, any *caller* of this function may be violating these properties, becoming transitively vulnerable.
- Knowing that a specific API parameter is typically (based on statistics of common usage patterns) *not* allowed to be controlled by untrusted callers can yield a high-precision warning if the current contract allows the same parameter to be tainted.
- Knowing that an external call is statistically only reached after authorization checks over the caller can yield a high-precision warning if the current contract has no such checks.

Corpus analysis has, thus, helped us scale up vulnerability detection by extracting domain expertise, instead of always relying on human experts. Key to the approach is the availability of hundreds of thousands of smart contracts already deployed on public blockchains. Secondarily the approach is enabled by the composable nature of DeFi protocols, which means that many different codebases use the same external protocols, so that common usage patterns can be ascertained.

5 DISCUSSION AND INSIGHTS

Automated program analysis tools have an extremely rich history in Computer Science, with entire research communities centered around program analysis. Security is often brought up as a major motivation for program analysis research. And yet, in one of the most critical security domains in existence, smart contract security, the impact of automated analysis tooling is repeatedly estimated to be close to zero. There are numerous possible explanations for this phenomenon, such as: the speed of development in smart contract engineering, which has not (up to now) allowed the time for research solutions to catch up; the immense, disproportionate (relative to other software engineering domains) investment on smart contract correctness that has attracted top human expertise to smart contract auditing, leaving little room for automated solutions; the fundamental limitations of fully automated solutions to software correctness, relative to, e.g., automation-assisted program verification, under the guidance/specification of human experts. (Although, on the latter point, a lot more can perhaps be said about the fundamental limitations of automation-assisted program verification.)

We would like to add to the discussion our observations on the contrast of academic research in program analysis (an area in which we have significant presence for over 15 years) and industrial impact. The discord between the two can be well-captured by the distance in answers to two questions.[3]

(1) What *warning rate* do you expect an impactful program analysis to have? I.e., out of 100 programs, how many do you expect to be flagged for vulnerabilities for an analysis to be useful?
(2) What precision, in terms of *false positive rate*, do you expect an impactful program analysis to have? I.e., out of 100 warnings, how many do you expect to indicate real vulnerabilities?

[3]The first author has repeatedly asked these questions in seminar talks at top Universities and research institutions, always with the same disparity between audience expectation and reality.

Researchers in program analysis typically give answers in the range of *10-30%* for the first question and *below 30%* for the second. Their expectation is that an impactful analysis will flag many contracts and will do so with very high precision.

Our own experience over random (de-duplicated) deployed smart contracts in active use is that realistic answers for the first question are *in the 1% or below* range and answers for the second question are in the *over-90%* range. That is, the most useful analyses flag very few contracts—around 1% or fewer—and it is fine if these warnings are often false positives. Anything outside these ranges is suspiciously off: it is probably a low-value analysis that is flagging too many contracts for "vulnerabilities" that human experts consider innocuous.

The order-of-magnitude discrepancy between expectation and reality is not that surprising, if one considers the setting. Program analysis experts are used to having their work evaluated over injected vulnerabilities, systematic benchmark suites with bug examples, or at least less-scrutinized, error-prone code. In the reality of deployed smart contracts, end-to-end vulnerabilities are extremely rare. A 95% false-positive rate might not be enough for a top-tier publication, but in practice it is almost too good to be true! It means that a vulnerability inspector will find a true, actionable vulnerability for every 20 inspections.

Therefore, it is clear that we have two problems: one of wrong metrics and one of perverse incentives (the latter being largely a consequence of the former).

At first glance, it seems reasonable to consider an analysis with a 0.5% warning rate to be less-than-impactful. After all, "if 99.5% of programs are correct, what is the big deal about getting that number to 100%? The relative improvement is tiny." However, this exact rationale leads to discounting the most practically impactful techniques. Conversely, techniques that focus on shallow "vulnerabilities" score much higher than impactful techniques on both metrics.

We believe that program analysis metrics should adjust to the rarity of the phenomenon they seek to capture. More is not better, nor is less. Instead, closeness to reality or absolute end-to-end impact is the only truly better measure for estimating the value of a technique.

6 PRACTICAL IMPACT

Our analysis has been applied to all new contracts deployed on the Ethereum blockchain since the beginning of 2021. The analysis has flagged numerous exploitable vulnerabilities—e.g., [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 total funds under threat from these vulnerabilities well exceed a billion dollars. We have received ten independent bug bounties totalling over $3M.

From a program analysis standpoint, many 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: $\texttt{X} \rightarrow Y\ \langle *; [\texttt{sender} \rightarrow C]\rangle$, as well as $|i|\ \langle *; [\texttt{sender} \rightarrow C]\rangle$. 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.

- *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 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 RELATED WORK

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.)

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, 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; Wüstholz and Christakis 2020a,b] have also been successful in precisely reporting smart contract vulnerabilities. Notably, the recently presented Harvey fuzzer [Wüstholz and Christakis 2020b] combines static analysis with fuzzing by using static analysis to guide a greybox fuzzer.

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 apporach manages to circumvent the expensive abstract state partitioning [Ko et al. 2017] or additional backwards analysis [Stein et al. 2019] that previous apporaches 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; Wüstholz 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. SPLlift [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 SPLlift 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.

8 CONCLUSIONS

We have argued that automated program analysis techniques can have a significant impact on the security of smart contracts. This does not mean that program analysis can counter most smart contract threats, nor that positive results come easy. To have actual impact on security, we have needed to invent new program analysis techniques and combine them with significant domain expertise and state/environment information.

Apart from the technical elements (symvalic analysis and corpus analysis) of our approach, perhaps the most useful aspect is the understanding of the changing nature of analysis precision and report rates in a high-impact setting. Conventional research in program analysis seems tuned to catching "cheap" bugs: numerous and low-impact. Such an approach also makes for good metrics: a high warning rate and high precision (i.e., true-positive rate) are possible for cheap bugs, but entirely unrealistic over rare high-value vulnerabilities. Yet coming up with ways to quantify the success of "needle in a haystack" searches will be an important element of achieving real impact on blockchain (and not only) security.

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

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

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

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.

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, 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, 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

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 Tsatiris. 2021. Symbolic Value-Flow Static Analysis: Deep, Precise, Complete Modeling of Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 163 (oct 2021), 30 pages. https://doi.org/10.1145/3485540

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

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 Drachsler-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

Valentin Wüstholz 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 Wüstholz 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