Slither: A Static Analysis Framework For Smart Contracts

Josselin Feist
Verimag
16 PUBLICATIONS **419** CITATIONS
SEE PROFILE

Alex Groce
Oregon State University
150 PUBLICATIONS **4,126** CITATIONS
SEE PROFILE

Slither: A Static Analysis Framework For Smart Contracts

Josselin Feist
Trail of Bits
New York, New York
josselin@trailofbits.com

Gustavo Grieco
Trail of Bits
New York, New York
gustavo.grieco@trailofbits.com

Alex Groce
Northern Arizona University
Flagstaff, Arizona
agroce@gmail.com

***Abstract*—This paper describes Slither, a static analysis framework designed to provide rich information about Ethereum smart contracts. It works by converting Solidity smart contracts into an intermediate representation called SlithIR. SlithIR uses Static Single Assignment (SSA) form and a reduced instruction set to ease implementation of analyses while preserving semantic information that would be lost in transforming Solidity to bytecode. Slither allows for the application of commonly used program analysis techniques like dataflow and taint tracking. Our framework has four main use cases: (1) automated detection of vulnerabilities, (2) automated detection of code optimization opportunities, (3) improvement of the user's understanding of the contracts, and (4) assistance with code review.**

In this paper, we present an overview of Slither, detail the design of its intermediate representation, and evaluate its capabilities on real-world contracts. We show that Slither's bug detection is fast, accurate, and outperforms other static analysis tools at finding issues in Ethereum smart contracts in terms of speed, robustness, and balance of detection and false positives. We compared tools using a large dataset of smart contracts and manually reviewed results for 1000 of the most used contracts.

I. Introduction

A growing number of industries are using blockchain platforms to perform trustless computation using smart contracts. Applications ranging from financial services to supply chains, and from logistics to healthcare are being developed to rely on blockchain technologies. One of the most popular underlying technologies is the Ethereum smart contact. These contracts are typically written in a high-level programming language called Solidity, then compiled to Ethereum Virtual Machine (EVM) assembly instructions for blockchain deployment. Often, the deployed smart contract's code is insecure: software vulnerabilities are regularly identified, and have been exploited by malicious actors, resulting in millions of dollars in damages and harm to the reputation of blockchain systems.

In the last few years, a variety of tools and frameworks to analyze and find vulnerabilities in Ethereum smart contracts were developed based on static and dynamic analysis. These tools are based on popular program testing techniques such as fuzzing, symbolic execution, taint tracking, and static analysis.

Static analysis is one of the most effective ways to detect potential issues in contracts. Typically, static analysis tools work by analyzing the Solidity source code or a disassembled version of the compiled contract. Then, they transform the contract code into an internal representation, more suitable for the analysis and detection of common security issues.

While modern compilers, such as clang, offer various APIs on top of which third-party analyzers can be built, the Solidity compiler fails to offer the same features. Ideally, a static analysis framework for Ethereum smart contracts should have the following properties:

1) **Correct level of abstraction**: if the framework is too abstract, it can be hard to introduce accurate semantics that capture common usage patterns. Conversely, if the framework is too narrowly focused on the detection of certain issues only, it can be difficult to add new detectors or analyses.
2) **Robustness**: it should parse and analyze real-world code without crashing.
3) **Performance**: analysis should be fast, even for large contracts, so as to integrate easily into development tools like IDEs, or into continuous integration checks that run at every commit.
4) **Accuracy**: it should allow for the development of detectors that find most potential issues while maintaining a low false positive rate. If the number of false positives is very high, it can require a manual audit of the entire contract, defeating its utility.
5) **Batteries included**: it should include a set of common analyses and issue detectors that are useful for most contracts. This will appeal both to security engineers looking to extend the framework and to code auditors looking for issues to report.

This paper introduces Slither, an open-source static analysis framework. Our tool provides rich information about Ethereum smart contracts and has the critical properties enumerated above. Slither uses its own intermediate representation, SlithIR, designed to make static analyses on Solidity code straightforward. It applies widely used program analysis techniques such as dataflow and taint tracking to extract and refine information. While Slither is built as a security-oriented static analysis framework, it is also used to enhance the user's understanding of smart contracts, assist in code reviews, and detect missing optimizations.

We summarize our contribution as follows:

- We present Slither, a framework for static analysis of

Solidity contracts.
- We detail the design of Slither's intermediate representation and analyzers.
- We evaluate and compare the performance, robustness, and accuracy of Slither on a large set of actively used smart contracts.

Both Slither and our dataset are open source, so that others can validate and improve on our results. The rest of the paper is organized as follows. Section II presents related work. Section III introduces a high-level overview of Slither's architecture, and Section IV details the intermediate representation language SlithIR. In Section V, we explain how we performed an evaluation and comparison of Slither against state-of-the-art tools and discuss the results. Finally, Section VI summarizes the paper and discusses future work.

II. Related Work

Several frameworks have been created to allow users and security researchers to analyze Ethereum smart contracts and detect potential issues. They are based either on static or dynamic analysis.

A. Static Analysis

These tools rely on the analysis of the code without executing it to detect issues in the smart contract. *Securify* is one of these tools [27], developed by the SRI Systems Lab (ETH Zurich). It works at the bytecode level: first, it parses and decompiles the EVM bytecode, then it translates the resulting code to *semantic facts* using static analysis. Finally, it matches the facts with a list of predefined patterns to detect common issues. *Securify* is open-source. It was implemented using Java and stratified Datalog. *SmartCheck* is another static analysis tool [20], developed by SmartDec. It works by translating directly from the Solidity source code into an XML-based intermediate representation. It then checks the intermediate representation against *XPath* patterns to identify potential security, functional, operational, and development issues. It is also implemented using Java. *Solhint* is a tool for linting Solidity code [18], developed by ProtoFire. It aims to provide both security and style guide validations. It is open-source and it is implemented in NodeJS using the *SolidityJ* parser [2]. Other notable static analysis frameworks include Vandal [4] and EtherTrust [10].

GASPER [5] and GasReduce [6] use static analysis to detect potential optimizations in contracts, GASPER at a high-level (e.g. dead code) and GasReduce at the bytecode instruction pattern level. Both focus on dead code and loop optimizations, rather than, e.g., the could-be-declared-constant optimization we discuss below, so these approaches are largely orthogonal to Slither's optimization patterns.

B. Dynamic Analysis

These tools rely on execution of the contract, leveraging symbolic execution, taint tracking, and fuzzing to discover vulnerabilities. Oyente was one of the first [14] tools for analysis and detection of security issues in Ethereum smart contracts. It is developed by Melonport and its code is open-sourced [17]. Manticore is an open-source symbolic execution tool for analysis of Ethereum smart contracts and binaries created by Trail of Bits [24]. Echidna [25], another product of Trail of Bits, is a property-based testing tool designed for fuzzing Ethereum smart contracts. Mythril Classic is an open-source security analysis tool for Ethereum smart contracts created by ConsenSys [7]. It uses concolic analysis, taint analysis, and control flow checking to detect a variety of security vulnerabilities. Finally, TeEther is an automatic exploit generation tool for certain types of vulnerabilities in Ethereum smart contracts created by Krupp and Rossow [13]. Its source code is not available at the time of this writing, but its authors promised to make their tool open-source 90 days after its presentation at Usenix 2018.

III. Slither

Slither is a static analysis framework designed to provide granular information about smart contract code and the flexibility necessary to support many applications. The framework is currently used for the following:

- **Automated vulnerability detection**: a large variety of smart contract bugs can be detected without user intervention.
- **Automated optimization detection**: Slither detects code optimizations that the compiler misses [5].
- **Code understanding**: printers summarize and display contracts' information to aid in the study of the codebase.
- **Assisted code review**: through its API, a user can interact with Slither.

Slither analyzes contracts using static analysis in a multi-stage procedure. Slither takes as initial input the Solidity Abstract Syntax Tree (AST) generated by the Solidity compiler from the contract source code. In the first stage, Slither recovers important information such as the contract's inheritance graph, the control flow graph (CFG), and the list of expressions. Next, Slither transforms the entire code of the contract to SlithIR, its internal representation language. SlithIR uses static single assessment (SSA) to facilitate the computation of a variety of code analyses. During the third stage, actual code analysis, Slither computes a set of pre-defined analyses that provide enhanced information to the other modules. Figure 1 summarizes all these stages.

A. Built-in Code Analyses

- **Read/Write**
  Slither identifies the reads and the writes of variables. For each contract, function, or node of the control flow graph, it is possible to retrieve the variables read or written, and filter by the type of variables (local or state). For example, it is possible to know the state variables written from a specific function, or find which functions write to a given variable. Several detectors are based on this information, such as those for uninitialized variables and reentrancy.
- **Protected functions** A recurrent pattern in smart contract design is the use of ownership to protect access

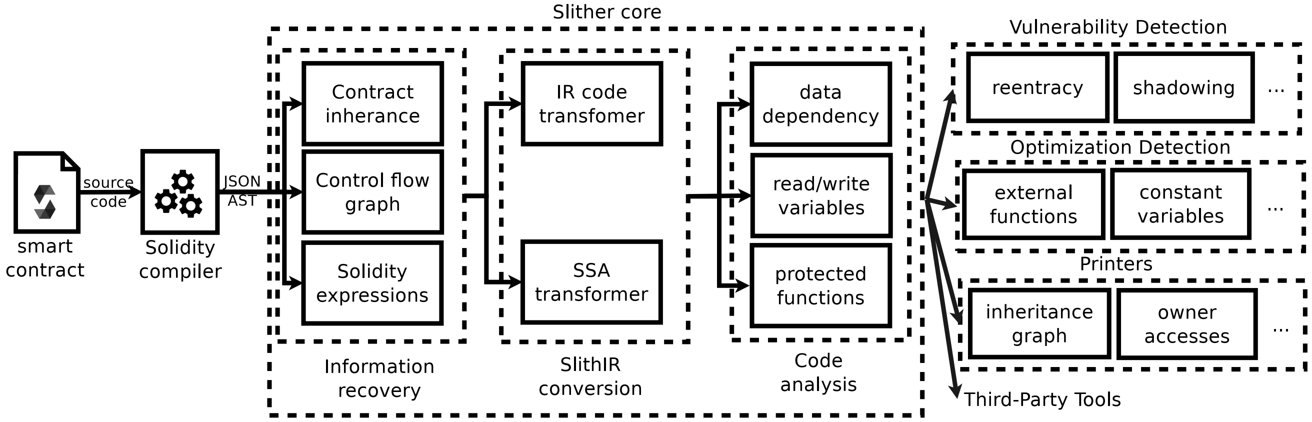

Fig. 1: Slither overview

to functions. The thinking behind this pattern is that a specific user (or set of users), called the *owner(s)*, can perform privileged operations. For instance, only an owner can mint new tokens. Modeling the protection of functions lowers the number of false positives. In order to detect unprotected functions, Slither checks for cases where (1) the function is not the constructor and (2) the address of the caller, *msg.sender*, is not directly used in a comparison. This heuristic can give false positives and false negatives, but our experience shows that it is effective in practice.

- **Data dependency analysis** Slither computes the data dependencies of all variables as a pre-stage using the SSA representation. The dependencies are first analyzed in the context of each function. Then a fixpoint is computed across all the functions of the contract, to determine whether there is a dependency in a multi-transaction context. Slither classifies some variables as *tainted*, which means that the variable is dependent on a user-controlled variable, and, therefore, can be influenced by the user. Finally, the data dependency takes into account the protected function heuristic. As a result, it is possible to select dependencies according to the user's privilege level. This granular information about data dependency is essential to writing accurate vulnerability detectors.

B. Automated Vulnerability Detection

The open source version of Slither contains more than twenty bug detectors, including:

- Shadowing: Solidity enables the shadowing of most of the contract's elements, such as local and state variables, functions, or events. Shadowed elements may not refer to the objects expected by the contract author.
- Uninitialized variables: Uninitialized state or local variables are a common source of errors in programming languages.
- Reentrancy: The reentrancy vulnerability (see Section V-A3) is well known.
- A variety of other known security issues, such as suicidal contracts, locked ether, or arbitrary sending of ether [15].

Closed-source detectors extend Slither's capabilities to detect more advanced bugs, such as race conditions, unprotected privileged functions, or incorrect token manipulations.

C. Automated Optimization Detection

Slither can detect code patterns that lead to costly code execution and deployment, a topic of considerable interest for smart contracts where there is a clear financial cost associated with inefficient code [5]. The framework includes detectors for:

- Variables that should be declared as constants. Constant variables are optimized by the compiler[1] , consume less gas, and do not take space in storage.
- Functions that should be declared as externals. External functions allow the compiler to optimize the code[2].

D. Code Understanding

Slither includes *printers*, which allow users to quickly understand what a contract does and how it is structured. Open source printers include:

- The exportation of different graph-based representations, including the inheritance graph, the control flow graph, and the call graph of each contract.
- A human-readable summary of the contracts, including the number of issues found and information about the code quality, such as cyclomatic complexity, or a minting restriction for ERC20 tokens.
- A summary of the authorization accesses and the variables that can be changed by the contract's owner.

E. Assisted Code Review

Users can build third-party scripts and tools using Slither Python API[3]. A custom script can target contract-specific

[1] https://solidity.readthedocs.io/en/v0.5.2/contracts.html#constant-state-variables

[2] https://solidity.readthedocs.io/en/v0.5.2/contracts.html#visibility-and-getters

[3] https://github.com/crytic/slither/wiki/API-examples

needs. For example, a user can ensure that a specific variable is never tainted by the parameter of a given function, or that a function is reachable only via legitimate entry points. Third-party tools can use Slither internals to build more advanced analyses, such as symbolic execution on top of SlithIR, or a conversion from SlithIR to another intermediate representation, such as LLVM.

F. *Implementation*

Slither is implemented in 16K lines of Python 3 code and is available on GitHub[4]. It has support for continuous integration and developer toolboxes (such as Truffle and Remix). It has minimal dependencies and only relies on a recent version of the Solidity compiler to parse the AST of the contract under analysis.

IV. SlithIR

SlithIR is the hybrid intermediate representation used in Slither to represent Solidity code. Each node of the control flow graph can contain up to a single Solidity expression, which is converted to a set of SlithIR instructions. This representation makes implementing analyses easier, without losing the critical semantic information contained in the Solidity source code.

A. *Instruction Set*

SlithIR uses fewer than 40 instructions. It has no internal control flow representation and relies on Slither's control-flow graph structure (SlithIR code is associated with each node in the graph). The following gives a high-level description of some notable instructions; the complete descriptions is available at [26].

1) Notation: LV and RV respectively represent a variable that is assigned (left-value) and a variable that is read (right-value). A variable can be a Solidity variable or a temporary variable created by the intermediate representation.

2) Arithmetic Operations: are represented as either a Binary or Unary operator:

- `LV = RV BINARY RV`
- `LV = UNARY RV`

3) Mappings and Arrays:: Solidity allows the manipulation of mappings and arrays that are accessed through dereferencing. SlithIR uses a specific variable type, called REF (ReferenceVariable) to store the result of a dereferencing. The index operator allows dereferencing of a variable:

- `REF ← Variable [Index]`

4) Structures:: Access to a structure is done through the member operator:

- `REF ← Variable . Member`

```
using SafeMath for uint;
mapping(address => uint) balances;

function transfer(address to, uint val) public{
    balances[msg.sender] = balances[msg.sender].min(
        val);
    balances[to] = balances[to].add(val);
}
```

Fig. 2: Solidity code example

```
Function transfer(address,uint256)
  Solidity: balances[msg.sender] = balances[msg.sender].sub
      (val)
  SlithIR:
    REF_0(uint256) -> balances[msg.sender]
    REF_1(uint256) -> balances[msg.sender]
    TMP_1(uint256) = LIB_CALL SafeMath.sub(REF_1, val)
    REF_0 := TMP_1(uint256) // dereferencing
  Solidity: balances[to] = balances[to].add(val)
  SlithIR:
    REF_3(uint256) -> balances[to]
    REF_4(uint256) -> balances[to]
    TMP_3(uint256) = LIB_CALL, dest:SafeMath.add(REF_4, val
        )
    REF_3 := TMP_3(uint256) // dereferencing
```

Fig. 3: SlithIR code for Figure 2

5) Calls:: Slither gives in-depth information about calls and possesses nine call instructions:

- `LV = L_CALL Destination Function [ARG..]` (low-level Solidity call)
- `LV = H_CALL Destination Function [ARG..]` (high-level Solidity call)
- `LV = LIB_CALL Destination Function [ARG..]` (library call)
- `LV = S_CALL Function [ARG..]` (call to a inbuilt-Solidity function)
- `LV = I_CALL Function [ARG..]` (call to an internal function)
- `LV = DYN_CALL Variable [ARG..]` (call to an internal dynamic function)
- `LV = E_CALL Event [ARG..]` (event call)
- `LV = Send Destination` (Solidity *send*)
- `Transfer Destination` (Solidity $transfer$)

Some calls can have additional arguments, for example, `H_CALL`, `L_CALL`, Send and Transfer can have $Value$ associated to the call, representing the amount of ether for the transaction.

6) Additional Instructions: include `PUSH` for array manipulation, `CONVERT` for type conversion, and operators to manipulate tuples.

7) Example: Figure 3 is the SlithIR representation of the code showed in Figure 2.

B. *SSA*

Static Single Assignment form [19] (SSA) is a representation commonly used in compilation and static analysis in general. It requires that each variable is assigned only one time and defined prior to its usage; e.g., `x = 3; y = x++; x`

[4] https://github.com/crytic/slither

```
struct MyStructure {
    uint val;
}

MyStructure private a;
MyStructure private b;

function increase(bool useb) external {
    MyStructure storage ref = a;
    if(useb){
        ref = b;
    }
    ref.val += 1;
}
```

Fig. 4: Smart contract using storage references

= y + x becomes $x_1 = 3; y_1 = x_1; x_2 = x_1 + 1; x_3 = y_1 + x_2$. One of the main advantages of SSA is to allow def-use chains to be easily computed, making data-dependency analyses straightforward. Using SSA form also enables more aggressive future analyses. For example, because gas limits force a bound on computational cost of contract execution, bounded model checking [1] using SAT/SMT is a natural fit for contract analysis. SlithIR's SSA representation plus graph-based control flow is very similar to the representation used in the successful CBMC [12] tool for C.

Slither stores two SlithIR versions: with and without SSA.

1) State Variables: A so-called ϕ function on a node indicates that a variable has multiple potential definitions and is a key element of SSA form [19]. One particularity of smart contracts is to heavily rely on state variables, which act as global variables. At the beginning of a function, the value of a state variable can be its initial value, or the value after the execution of any function. Additionally, an external call may re-enter, which can allow a state variable to change. As a result, a ϕ function is placed at the entry of each function, and after the external calls, for each state variable read by the function.

2) Alias Analysis: Solidity allows local variables to refer to state variables, as shown in Figure 4. Such variables are called *storage references*. A write operation to a local variable can, therefore, refer to a write to multiple state variables. Slither computes an alias analysis to find all the possible targets of the *storage reference* and gives the information to the SSA engine. As a result, ϕ functions can be correctly placed after the write to a storage reference. In the example in Figure 4, two ϕ functions will be placed at the expression $ref.val+=1$; to indicate that a and b are updated.

C. Comparison to Other Smart Contract Intermediate Representations and Discussion

Other intermediate representations for smart contracts have been proposed. Scilla [29] is the intermediate representation used by the Zilliqa blockchain. It is based on the notion of communicating automata and contains a translation to the Coq proof assistant to prove safety and liveness properties. While Scilla is a promising representation, and it does possess a translation from Solidity code, it is not clear if it would be able to represent the entire Solidity language or only a subset. Michelson [23] is the intermediate representation used in the Tezos blockchain. It is a stack-based representation, making it less suitable for static analysis than SlithIR. IELE [11] is the representation developed for the Cardano blockchain. IELE is modeled in the K framework, and can take advantage of K existing analyses. IELE works at a lower-level than SlithIR and is, therefore, complementary to it. For example, in IELE, state variables are represented through stores and reads to the storage memory rather than variables, which makes their analysis more complicated, while it allows a more accurate gas estimation. The Solidity authors are working on an intermediate representation called YUL [21], which is still a work in progress at the time of writing. While adding an intermediate representation within the Solidity compiler is a good step towards a safer language, YUL does not provide the same level of information as does SlithIR. For example, YUL does not possess an SSA representation, and is written as a list of nested expressions, making the construction of static analyses less straightforward.

SlithIR possesses some remaining limitations and has room for improvement. The principal omission is the lack of formal semantics, which would allow more rigorous analyses. Another limitation is that the representation is too high-level to accurately reflect low-level information, such as the gas computation.

V. Evaluation And Comparison to State-of-the-Art Tools

We performed an evaluation of three aspects of Slither: (1) vulnerability detection, (2) missing optimization detection, and (3) source-code exploration. Slither and the other state-of-the-art tools are evaluated using real-world contracts. Most of the state-of-the-art tools are focused on the detection of security issues, and therefore, (1) is the core focus of our evaluation.

A. Vulnerability Detection Evaluation

We compare Slither (release 0.5.0) with other open-source static analysis tools to detect vulnerabilities in Ethereum smart contracts: Securify (revision `37e2984`), SmartCheck (revision `4d3367a`) and Solhint (release 1.1.10). We decided to focus our evaluation exclusively on tools' reentrancy detectors, since reentrancy is one of the oldest, most well understood, and most dangerous security issues. This detector is available in all the tools we evaluated.

We do not consider tools based on dynamic analysis, such as Manticore, Oyente or Mythril, as they are known to have scalability issues and do not compete directly with static analysis tools.

1) Experiments: We evaluate the static analysis tools using two experiments: in the first, we used two famous contracts vulnerable to reentrancy, DAO[16] and SpankChain[22], to verify if the tools can rediscover the issues actually exploited. In the second experiment, we used the 1000 most used contracts (those with the largest number of transactions),

```
mapping (address => uint) balances;
function withdrawBalance() public {
  if(!(msg.sender.call.value(balances[msg.
     sender])())) {
    throw;
  }
  balances[msg.sender] = 0;
}
```

Fig. 5: Reentrancy example

for which Etherscan [28] provides the source code. Several other evaluations in the literature [14], [15], [20] use a direct sample of contracts in the blockchain instead. However, we found that such a selection is biased in an unhelpful way, as most of the contracts deployed on Ethereum are essentially tests, lacking the complexity (and code quality) that widely used contracts have. As a result, most of these contracts are likely to contain trivial-to-detect bugs not reflecting real-world situations, which can skew results. A previous effort to compare Solidity analysis tools (both static and dynamic) proposed even more selective choice of contracts, based on Zeppelin audits, but this resulted in a set of only 28 contracts to analyze [8]; we were able to use a larger set by restricting the scope of comparison, rather than the contracts selected.

2) Metrics: The tools are evaluated on three metrics: performance, robustness, and accuracy. The performance metric indicates how fast a tool runs on each contract. It should include every step needed to analyze a smart contract. If a tool requires a contract to be compiled, the compilation time is included in this metric. The robustness metric indicates how often each tool fails to parse, decompile, or analyze a smart contract. Finally, the accuracy metric measures both number of contracts flagged and number of false positives detected.

3) Reentrancy Revisited: A function in a smart contract is said to be reentrant if it contains a call to an external contract that can be used to re-enter the function itself. A reentrancy can lead to a variety of exploitations, including loss of ether, when a state variable is changed only *after* an external call. This issue became famous during the DAO hack [16] and SpankChain[22].

Figure 5 shows a classic example of reentrancy. The exploitability of reentrancy issues is very well known, most vulnerability detection tools for Ethereum smart contracts issue a warning when they detect code like that in Figure 5. Reentrancy patterns are not uncommon, but in most of cases, the reentrancy is not severe for two reasons:

- It requires special privileges: only the owner can act maliciously.
- It is benign: the ”exploit” has the same effect as two successive calls.

During our evaluation, if a tool flags a benign reentrancy in a contract, it is considered a false positive.

4) Experiment 1: The last two columns of Table I show whether the DAO and SpankChain vulnerabilities were found by each tool. It is worth noting that several reentrancies are reported by the tools in both examples. However, we consider the result valid only if the tool can detect the actual reentrancy that was used during the exploitation of the contract. Slither is the only tool capable of finding these two real-world cases of reentrancy.

5) Experiment 2: For the second experiment, using a dataset of 1000 contracts, the tools were run on each contract with a timeout of 120 seconds, using only reentrancy detectors. We manually disabled other detection rules to avoid the introduction of bias in the measurements.

- **Performance:** The *Average execution time* and *Timed-out analyses* rows in Table I summarize performance results: Slither is the fastest tool, followed by Solhint, SmartCheck, and Securify. In our experiments, Slither was typically as fast as a simple linter.
- **Robustness**: The *Failed analyses* row in Table I summarizes robustness results: Slither is the most robust tool, followed by Solhint, SmartCheck, and Securify. Slither failed only for 0.1% of the contracts, while Solhint failed for around 1.2%. SmartCheck and Securify are less robust, failing 10.22% and 11.20% of the time, respectively.
- **Accuracy**: The *False positives*, *Flagged contracts* and *Detections per contract* rows in Table I summarizes accuracy results.
  Due to time constraints, we manually reviewed a random sample of at least 50 of the flagged contracts for each tool, rather than all potentially benign reentrancies.
  Our experiments show that Slither is the most accurate tool, with the lowest false positive rate: 10.9%, followed by Securify with 25%. SmartCheck and Solhint have high false positive rates of 73.6% and 91.3% respectively. Additionally, we include the number of contracts for which at least one reentrancy is detected (*flagged* contracts) and the number of findings on average per *flagged* contract. On one hand, SmartCheck flags a large number of contracts, confirming its high false positive rate. On the other hand, Securify flags a very small number of contracts, which indicates that the tool fails to find a number of true positives found by other tools.

6) Discussion: In these experiments, Slither outperforms the other analyzers for detecting reentrancy by detecting real-world bugs while maintaining a low false positive rate. However, we should note that our experiments are limited in many ways. The false positives rate for Securify is based on a very low number of findings (8). Additionally, several contracts in our dataset share fragments of code, creating a bias in the results. For instance, the false positives from Slither result from only a few distinct functions that are duplicated across multiple contracts. Despite these limitations, the experiment shows that Slither is more accurate and mature than the other tools.

We also found that SmartCheck and Solhint have a large number of false positives due to a lack of in-depth understanding of Solidity. For example, the Solidity attribute *super*, which

		Slither	Securify	SmartCheck	Solhint
Accuracy	False positives	10.9%	25%	73.6%	91.3%
	Flagged contracts	112	8	793	81
	Detections per contract	3.17	2.12	10.22	2.16
Performance	Average execution time	0.79 ± 1	41.4 ± 46.3	10.9 ± 7.14	0.95 ± 0.35
	Timed out analyses	0%	20.4%	4%	0%
Robustness	Failed analyses	0.1%	11.2%	10.22%	1.2%
Reentrancy examples	DAO	✓	✗	✓	✗
	Spankchain	✓	✗	✗	✗

TABLE I: Summary of the evaluation results

allows calling an inherited function, is taken as an external call by these tools.

B. Optimization Detection Evaluation

To test the optimization capacity of Slither, we select the detector finding variables that should have been declared as constants. If a variable is declared constant, it will not take space in the storage of the contract, and will require fewer instructions when used. As a result, using constant variables when possible reduces the code size (and thus the cost to deploy the contracts), as well as the usage cost. To the best of our knowledge, Slither is the only available tool capable of finding this code optimization.

We applied the detector on two datasets: (1) the 1000 contracts selected for experiment 2 (Section V-A), and (2) 35.000 contracts from [28]. We found that 54% of the contracts from (1) and 56% from (2) contain one or more variables that could have been declared as constant. Some of these variables are never accessed and could have been entirely removed.

This result shows that Slither can efficiently be used to detect practical code optimization.

C. Code Understanding Comparison

Slither printers provide a variety of visual outputs to improve code understanding. The closest tool that provides similar features is Surya[5]. Surya parses the contract AST and offers a set of outputs. It only performs a syntactic analysis and does not provide any semantic analysis.

Table II shows a comparison of Slither printers with Surya features. Both tools provide similar features, except than Surya outputs a textual representation of the AST and the function call trace, while Slither yields information about variables read and written. Both tools can generate a report, but they differ in their contents. Surya shows contracts and functions name, as well as the mutability and modifiers calls, while Slither shows the number of bugs found, the code complexity (based on cyclomatic complexity), and provides high-level information for specific context based on a set of heuristics (such as the minting limitation in case of ERC20 tokens).

Slither includes the information provided by Surya, but can integrate more advanced information, as it has an in-depth understanding of the codebase.

D. Threats to Validity

The most important threat to validity is that our primary experiment is limited to reentrancy detection over a limited set of contracts. The measure we used to determine a meaningful set of contracts (number of transactions) is reasonable [3], but not standardized by the community. It was unclear how to use some very recently proposed frameworks for smart contract analysis, e.g. SmartAnvil [9], which were thus omitted from our experiments.

In order to make our results reproducible, and allow checking of our code and analysis methods, we have made our experimental setup available[6].

VI. Conclusions and Future Work

We presented Slither, an open source static analysis framework for smart contracts. Slither is fast, robust, accurate, and provides rich information about smart contracts. Our framework leverages SlithIR, an intermediate representation designed for practical static analysis on Solidity code. Slither is the only platform that can be used at the same time to find bugs, suggest code optimizations, increase code understanding for a given contract. It was built to be easily extended, and serve as a basis upon which third-party tools can be implemented.

We evaluated the bug finding capacity of Slither by comparing its performance on reentrancy bugs against other available state-of-the-art tools. We found that Slither outperforms the other tools in terms of performance, robustness, and accuracy.

There are several directions that we could take to improve Slither. As the tool is daily used during audits, a first improvement is the integration of new issue detectors. Optimization could be applied to SlithIR to generate more efficient code. Symbolic execution or bounded model checking could be built on top of our intermediate representation, allowing easy access to formal verification for bug detectors, and loop-bound based worst-case gas cost analyses. The Slither analyzer was originally designed to target Solidity code, but both the platform and the intermediate representation can be adapted for another smart contract language, such as Vyper. Finally, a translation from SlithIR to EVM or Ewasm bytecode would allow Slither to serve as a compiler.

[5] https://github.com/ConsenSys/surya

[6] https://gist.github.com/ggrieco-tob/748bca8a0d166e026ea717e225a4fbf9

	Slither	Surya
Contract Summary	*contract-summary*	*describe*
Functions Summary	*function-summary*	
Inheritance dependencies	*inheritance*	*dependencies*
Inheritance graph	*inheritance-graph*	*inheritance*
Call graph	*call-graph*	*graph*
AST		*parse*
Functions call trace		*ftrace*
Authorization and access summary	*vars-and-auth*	
Report	*human-summary*	*mdreport*

TABLE II: Comparison of Slither printers with Surya

References

[1] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 193–207 (1999)

[2] Bond, F.: Solidity grammar for antlr4. https://github.com/solidityj/solidity-antlr4 (2017)

[3] Bragagnolo, S.: On contract popularity analysis. https://github.com/smartanvil/smartanvil.github.io/blob/master/_posts/2018-03-14-on-contract-popularity-analysis.md

[4] Brent, L., et al.: Vandal: A scalable security analysis framework for smart contracts. CoRR **abs/1809.03981** (2018)

[5] Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on. pp. 442–446. IEEE (2017)

[6] Chen, T., Li, Z., Zhou, H., Chen, J., Luo, X., Li, X., Zhang, X.: Towards saving money in using smart contracts. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results. pp. 81–84. ICSE-NIER '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3183399.3183420, http://doi.acm.org/10.1145/3183399.3183420

[7] ConsenSys: Mythril: a security analysis tool for ethereum smart contracts. https://github.com/ConsenSys/mythril-classic (2017)

[8] Dika, A.: Ethereum Smart Contracts: Security Vulnerabilities and Security Tools. Master's thesis, NTNU (2017)

[9] Ducasse, S., Rocha, H., Bragagnolo, S., Denker, M., Francomme, C.: Smartanvil: Open-source tool suite for smart contract analysis. Tech. Rep. hal-01940287, HAL (2019)

[10] Grishchenko, I., Maffei, M., Schneidewind, C.: Ethertrust: Sound static analysis of ethereum bytecode (2018)

[11] Kasampalis, T., et al.: Iele: An intermediate-level blockchain language designed and implemented using formal semantics. Tech. Rep. http://hdl.handle.net/2142/100320 (2018)

[12] Kroening, D., Clarke, E.M., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 168–176 (2004)

[13] Krupp, J., Rossow, C.: teether: Gnawing at ethereum to automatically exploit smart contracts. In: USENIX Security) (2018)

[14] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. CCS '16 (2016)

[15] Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: ACSAC (2018)

[16] Phil Daian : Analysis of the dao exploit. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/ (June 18, 2016 (acceded on Jan 10, 2019))

[17] Project, M.: Oyente: an analysis tool for smart contracts. https://github.com/melonproject/oyente (2017)

[18] Protofire: Solhint: an open source project for linting solidity code. https://protofire.github.io/solhint/ (2017)

[19] Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: POPL (1988)

[20] S., T., et al.: Smartcheck: Static analysis of ethereum smart contracts. WETSEB (2018)

[21] Solidity: Yul. https://solidity.readthedocs.io/en/v0.5.0/yul.html (Accessed on Jan 10, 2019)

[22] SpankChain: We got spanked: What we know so far. https://medium.com/spankchain/we-got-spanked-what-we-know-so-far-d5ed3a0f38fe (Oct 8, 2018 (acceded on Jan 10, 2019))

[23] Tezos: Michelson: the language of smart contracts in tezos. http://www.liquidity-lang.org/doc/reference/michelson.html (Accessed on Jan 10, 2019)

[24] Trail of Bits: Manticore: Symbolic execution for humans. https://github.com/trailofbits/manticore (2017)

[25] Trail of Bits: Echidna: Ethereum fuzz testing framework. https://github.com/crytic/echidna (2018)

[26] Trail of Bits: Slithir documentation. https://github.com/crytic/slither/wiki/SlithIR (2018)

[27] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Securify: Practical security analysis of smart contracts. CCS '18 (2018)

[28] Wagner, G.: Verified contracts synced from etherscan. https://github.com/thec00n/etherscan_verified_contracts (2018)

[29] Zilliqa: Scilla safe-by-design smart contract language. https://scilla-lang.org/ (Accessed on Jan 10, 2019)