SIF: A Framework for Solidity Code Instrumentation and Analysis

Chao Peng
University of Edinburgh, UK
chao.peng@ed.ac.uk

Ajitha Rajan
University of Edinburgh, UK
arajan@staffmail.ed.ac.uk

ABSTRACT

Solidity is an object-oriented and high-level language for writing smart contracts which are used to execute, verify and enforce credible transactions on permissionless blockchains. In the last few years, analysis of vulnerabilities in smart contracts has raised considerable interest and numerous techniques have been proposed. Current techniques lack traceability in source code and have widely differing work flows. There is no single unifying framework for analysis, instrumentation, optimisation and code generation of Solidity contracts.

In this paper, we present SIF, a comprehensive framework for Solidity contract monitoring, instrumenting, and code generation. SIF provides support for Solidity contract developers and testers to build source level techniques for analysis, bug detection, coverage measurement, optimisations and code generation. We show feasibility and applicability of the framework using 51 real smart contracts deployed on the Ethereum network.

KEYWORDS

high level languages, software testing, program instrumentation, dynamic program analysis, testing tools

1 INTRODUCTION

Blockchains are the underlying technology for making online secure transactions using cryptocurrencies such as Bitcoins and Ethers. Facilitating, verifying and enforcing negotiations and credible transactions on blockchains is done by smart contracts which are written into lines of code by the buyer and seller using Turing-complete languages[5]. Solidity is a popular object-oriented and high-level language for implementing smart contracts[6, 14] which can be compiled to byte code for execution on the blockchain network. With the increased use of smart contracts across application domains, there is a crucial need for analysis and testing techniques that provide strong security guarantees.

1.1 Problem

Existing work on analysing Solidity contracts for security vulnerabilities is primarily based on analysing the byte code of smart contracts, or translating Solidity code to other languages or intermediate forms over which analysis is performed. These techniques, although effective in detecting certain vulnerability types, suffer from lack of traceability back to source code. Oyente [9], for instance, analyses the byte code from the Solidity compiler and successfully reports the presence of vulnerabilities. However, it lacks the ability to trace and localise the bug in the Solidity code. Zeus [7] translates smart contracts to LLVM bitcode but does not support complete Solidity syntax, including `throw` statements , `self-destructs`, virtual functions and assembly code blocks.

Traditional programming languages are all supported by a comprehensive framework for code instrumentation, monitoring, optimisation and code generation such as LLVM/Clang [8] for C/C++. This framework support is lacking for smart contracts. Such a framework would allow easy-to-use instrumentation and monitoring passes for contract code, support for code analysis and checking different types of vulnerabilities, code coverage measurement, code optimisations, and finally code generation.

1.2 Goal

Our aim is to provide Solidity developers and testers a framework, similar to the widely-used C/C++ instrumentation technique Clang LibTooling [3], to conveniently and effectively instrument and analyse Solidity code.

In order to achieve this goal, we design and implement SIF (Solidity Instrumentation Framework) as illustrated in Figure 1 which has the following capabilities:

(1) Provide an interface for users to query the AST of Solidity code. For instance, the user can count the number of loops with a specific condition in the code.
(2) Allow users to modify nodes in the AST. One example use case is to generate mutations of Solidity contacts.
(3) Generate Solidity code from the AST which reflects modifications made by the user.

Benefits. SIF targets Solidity contract developers and testers. The framework aims to provide a compiler infrastructure for monitoring, instrumenting, analysing, optimising and generating Solidity code. Users of SIF wil be able to easily build their own code analysis tools, working from the AST with existing support for querying, instrumentation and code generation.

We evaluate our framework by conducting two types of code instrumentations on smart contracts: inserting property check assertions and mutation testing by seeding faults. We run these two instrumentations on a collection of 51 real smart contracts which are deployed and running on the Ethereum network. Our experiment result shows that SIF is able to automatically traverse and modify smart contracts based on users' instructions.

2 RELATED WORK

In this section, we describe existing tools for Solidity code analysis and instrumentation.

Solidity byte code and transformation analysis. Some of the existing research and techniques on smart contract verification perform byte code analysis to check for potential vulnerabilities [9–11, 14]. Byte code is the compiled hexadecimal format of smart

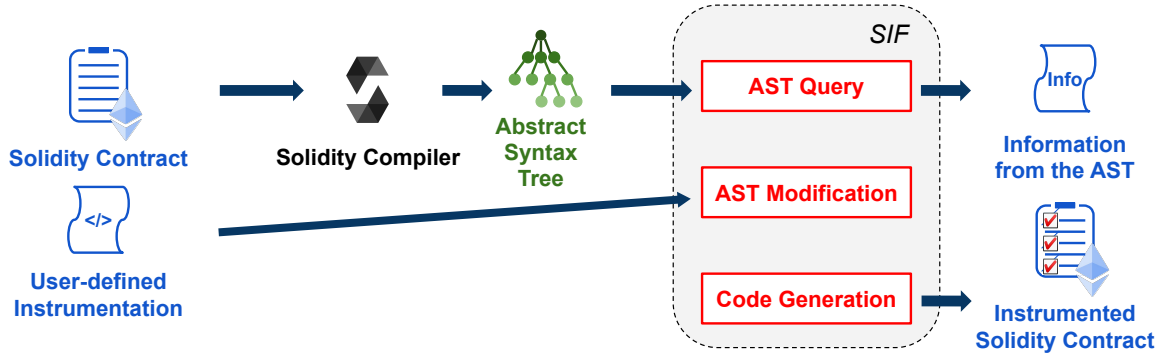

Figure 1: SIF work flow

contracts. Each unit of the byte code consists of the instruction representing a certain operation and if any, its operand. Although Smart contract verification on byte code level is sufficient to check for the presence of certain types of vulnerabilities, it is hard to trace the problem back to the Solidity source code level and aid the developer to fix it. Some other work translates the Solidity code to the F* languages or represents the code in intermediate forms including LLVM IR (Intermediate Representation) and XML format and performs code analysis on the intermediate form [4, 7, 13]. However, code conversion causes the loss of some code structures and features of Solidity and also suffers from the problem of mapping the detected vulnerabilities to the original source code.

Solidity code generation and decompilation. To the best of our knowledge, we are only aware of one tool, Soltar [1], that supports the translation of the AST back to the source code. However, Soltar is not maintained and does not support Solidity versions 0.4.3 onwards. Zeus [7] is able to translate Solidity code to LLVM bitcode but it does not support the complete syntax of Solidity. It is also worth noting that Zeus does not provide an interface for instrumentation and translation of intermediate form back to Solidity code. Porosity [2] and another Ethereum smart contract decompiler [12] are able to decompile the byte code back to the source code but they are not feasible for users who want to instrument smart contracts on the source level.

In the next section, We present a generic framework written in C++ for instrumenting the AST of Solidity code and translating the AST back to the native source code, supporting the entire Solidity syntax.

3 SIF OVERVIEW

SIF takes the AST of Solidity code produced by the Solidity compiler as input, accepts the user's instructions on AST queries and modifications to gather information and make changes to the AST, and generates Solidity code from the modified AST.

In this section, we describe details about our implementation of SIF which is available at https://github.com/chao-peng/SIF, and how its components fit together. We then present a brief user guide of SIF.

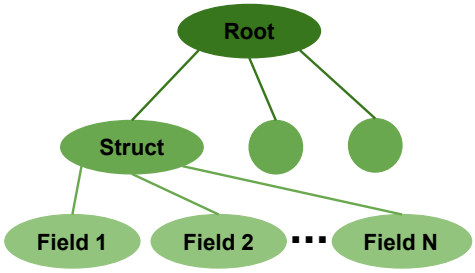

Figure 2: An AST example highlighting a struct definition

3.1 SIF Implementation

We implement our framework in C++. Key aspects of implementation are described as follows.

AST representation in C++ classes. The Solidity compiler generates the AST from the Solidity code in two formats: plain text with indentations to illustrate trees and JSON (JavaScript Object Notation) which is a structured data format. In an attempt to make our tool for Solidity code instrumentation generic and easy to use, we use C++ classes as our intermediate AST representation. We implement classes to represent different AST nodes, containing information captured from the AST in their data fields and providing methods to access and modify the data. Our tool traverses the Solidity AST and for each AST node, it instantiates a class of that node type with all the associated information. The C++ classes contain methods that allow information about the node to be retrieved easily and also modified, if necessary. For instance, a C++ class for an `struct definition` node has methods for getting and updating the struct name and maintaining its member fields.

Figure 2 and the following pseudocode listing illustrate how the AST and the corresponding class are mapped.

```
class StructDefinitionNode : public ASTNode {
public:
  StructDefinitionNode() : ASTNode(); // Constructor
  string source_code(); // get source code of the struct
  string get_name(); // get the name
  void set_name(new_name); // update the name
  int num_field(); // get the number of fields
  void add_field(new_field); // add a new field
  void remove_field(index); // remove a field by index
```

```
  void update_field(index, new_field); // update a field
  ASTNode get_feild(index); // get a field by its index
private:
  string name; // name of the struct
  vector<ASTNode> fields; // list of struct fields
};
```

Code instrumentation. After representing the AST generated by the Solidity compiler using C++ classes, SIF traverses the AST starting from the root node in a depth-first fashion. A function containing instructions of code instrumentation defined by the user is called when it processes each node as mentioned in Section 3.2. If the instrumentation instructions have function calls to modify the node, fields of the corresponding node are changed and stored accordingly.

Solidity code generation from the AST. For each type of AST node class, we provide code templates that allow corresponding Solidity code to be generated. The following pseudocode listing illustrates the code template for a `StructDefinitionNode`.

```
string StructDefinitionNode::source_code() {
  string code = "struct " + name + "{";
  Iterate a list of element definitions {
    code += element_definition.source_code() + ";";
  }
  code = code + "}";
  return code;
}
```

In the above listing, the template generates Solidity code for a `struct definition` by first printing the `struct` keyword, followed by the name of the struct, a left brace indicating the start of struct element definitions. The template then iterates through the list of struct element definitions. Once an element definition is visited, the source code of that definition is appended to the struct code. When we reach the end of element definitions, a right closing brace is added, indicating the end of the struct definition.

3.2 Using SIF

SIF provides an interface in the form of a predefined callback function. The parameter of this function is the AST node that SIF is currently visiting. This function is defined by the user to query information from the AST node and make changes to the node by calling built-in functions provided by SIF. When SIF traverses the AST of a smart contract, it calls the function every time when it processes a new node. We illustrate how to use SIF step by step.

Step 1: Write instructions of code instrumentation. We provide a function named *visit* and the user can implement the function to check if the currently visited node has the desired property, such as if it is an expression statement, or if it is a loop with a certain condition, and make changes to that node. The framework provides adequate methods for querying and modifying all types of Solidity AST nodes.

For example, if the user wants to print out conditions of all for loops, they need to first determine if the currently visited node is of the type of `for statement`. If so, they can call the function *get_condition* to get the condition of that `for statement`. All types of AST nodes in our framework have a function called *source_code()* which returns the source code of themselves. Conditions of loops are also represented by AST nodes and the function *source_code()* can be called. The user can then write statements to print out the condition to standard output or files.

Step 2: AST generation. SIF relies on the AST generated by the Solidity compiler. Instead of starting from implementing a source code parser for Solidity to generate ASTs, we take the existing functionality provided by the official Solidity compiler, which is undoubtedly able to parse all language features and code structures. The user only needs to run the Solidity compiler to produce the AST of smart contracts and use them as inputs to SIF.

Step 3: Framework execution with user-defined instrumentation instructions. With the *visit* function containing instructions of code instrumentation provided by the user,SIF reads the AST of the original Solidity code and traverses it. Source code reflecting changes defined by the user is generated in this last stage.

4 EVALUATION

In this section, we show that SIF is able to effectively instrument smart contracts of different sizes and generate instrumented code for two common use cases: property check and fault seeding. Both of them rely on querying and modifying the AST and are able to cover all capabilities of the framework.

4.1 Experiment Setup

We collect from the Ethereum network 51 unique smart contracts with the number of lines of code ranging from 27 to 1293. Our dataset is available at https://github.com/chao-peng/SIF.

We select 5 types of vulnerabilities that can be injected to smart contracts by instrumenting the code to evaluate the fault seeding part, as presented in Table 1. For the evaluation of property check, we utilise both AST query and modification capabilities provided by SIF. The first 3 types of arithmetic faults can be caught by inserting assertion statements. Timestamp usage as well as transaction origin vulnerabilities have specific keywords in the AST and can be detected by querying the AST at syntax level. Details on the design of code instrumentation for these two use cases are discussed below.

4.2 Code Instrumentation of Use Cases

Property check. When SIF traverses AST nodes and when a node is visited, property check first determines whether the node type is susceptible to any of the vulnerabilities. For arithmetic vulnerabilities including division by zero, integer overflow and underflow, it makes queries to the AST and gather information of the node including its type, operator used and operands. For example, an expression node with arithmetic operations may be prone to an overflow/underflow error. In this case, SIF generates relevant assert statements on the result of the operations in the node that flags a vulnerability when the assert statement fails. The assert statement as an extra AST node is added after the node under analysis in the program control flow. For division by zero vulnerability, SIF inserts a pre condition, before the node under analysis with a division operator, that asserts the divisor expression is greater than zero. The detection of timestamp usage and transaction origin does not require modifications to the AST. We only ask SIF to check if the AST node has specific keywords which uses timestamps and transaction origins.

Type of vulnerability	Type of property assertion	Seeded Fault
Division by zero	Pre condition	Statement with division by zero is inserted.
Overflow	Post condition	Arithmetic operation resulting in overflow is inserted.
Underflow	Post condition	Arithmetic operation resulting in underflow is inserted.
Timestamp usage	AST Analysis	An expression using block timestamp is used in an assignment.
Transaction origin	AST Analysis	A code block containing a check using the value of tx.origin is inserted.

Table 1: Vulnerability types

Fault Seeding. For each type of vulnerability illustrated in Table 1, the instruction set creates a new code block containing that vulnerability and represent it using an AST node. In each run, SIF traverses the AST of a Solidity contract and inserts an extra AST node which is harmless to the original functionality and contains one type of vulnerability to the original AST. Solidity code is then generated from the modified AST, referred to as a mutated contract. We run SIF 5 times with the instruction set to produce all 5 types of vulnerabilities.

4.3 Experiment Result

We assess feasibility and effectiveness of the framework for AST traversing and instrumentation using the following criteria, (1) Solidity language support, (2) Performance of code instrumentation and (3) Extent of automation. We discuss each of these criteria below.

4.3.1 Solidity language support. The 51 smart contracts in our dataset contains a wide variety of solidity constructs. We manually checked against the Solidity documentation as well as the source code of the official Solidity compiler, and confirmed that our dataset covers all syntax of Solidity. Our technique was able to analyse the AST and generate instrumented code for all 51 smart contracts automatically. As mentioned in Section 2, the F* tool [4] does not support loop structures in Solidity. Three contracts in our dataset contain loops. In addition, self-destructs (appear in 2 subject programs), throw statements (appear in 10) and inline-assembly blocks (appear in 3) are not supported by the Zues analysis tool [7]. SIF fully supports the use of these constructs in Solidity contracts.

4.3.2 Performance of code instrumentation. SIF is proven to be able to query and modify the AST according to the instructions provided by the user and generate Solidity code from the modified AST. For all 51 smart contracts, SIF is able to automatically insert assertions to the original contracts and if the input causes applicable arithmetic exceptions, the assertions are able to report them. The vulnerability seeding tool allowed fully automatic insertion of different types of vulnerabilities in all 51 smart contracts. The mutated Solidity contracts were fully compatible with the Solidity compiler and the execution platform.

4.3.3 Extent of automation. In order to use SIF for AST analysis and instrumentation purposes, the user only needs to write a set of instructions of AST queries and modifications by calling functions provided by SIF to query information and make changes to the original code. After the instructions of code instrumentation are defined by the user, the tool is able to automatically traverse and instrument the AST with no need of further interactions.

5 CONCLUSION

SIF provides the capability of analysing the AST of Solidity smart contracts produced by the Solidity compiler and generating Solidity code back. The user does not need any Solidity AST knowledge. The framework is promising for contract analysis requiring code instrumentation, coverage measurement, fault seeding, event logging, etc. The empirical evaluation shows that our framework is capable of (1) automatically analysing the AST and representing it using C++ classes, (2) providing an interface for the user to retrieve information and make changes to the AST, and (3) generating Solidity code from the instrumented AST.

REFERENCES

[1] 2016. Soltar. https://github.com/duaraghav8/soltar
[2] 2017. Porosity. https://github.com/comaeio/porosity
[3] 2019. Clang LibTooling. https://clang.llvm.org/docs/LibTooling.html
[4] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 91–96.
[5] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* (2014).
[6] Chris Dannen. 2017. *Introducing Ethereum and Solidity*. Springer.
[7] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing safety of smart contracts. NDSS.
[8] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.
[9] 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*. ACM, 254–269.
[10] MythX. 2019. Mythril Classic: Security analysis tool for Ethereum smart contracts. https://github.com/ConsenSys/mythril-classic
[11] Remix. 2018. Remix documentation. https://remix.readthedocs.io/en/latest/
[12] PNF Software. 2019. Ethereum Smart Contract Decompiler. https://www.pnfsoftware.com/blog/ethereum-smart-contract-decompiler/
[13] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. (2018).
[14] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 67–82.