SIF: A Framework for Smart Contract Native Code Analysis and Instrumentation

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

Sefa Akca
University of Edinburgh, UK
s.akca@sms.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 most recent years, smart contracts and blockchains have raised noticeable attention from the software testing community. However, due to the lack of a comprehensive source code analysis and instrumentation framework for Solidity, testing techniques including code coverage measurement and fault injection are still missing.

In this paper, we present SIF - a Solidity instrumentation framework which provides an interface for users to retrieve syntactic and semantic information from the source code and instrument the code based on the Abstract Syntax Tree (AST). We show feasibility and applicability of the framework by empirical evaluation using real smart contracts running on the Ethereum network.

KEYWORDS

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

ACM Reference Format:
Chao Peng, Sefa Akca, and Ajitha Rajan. 2019. SIF: A Framework for Smart Contract Native Code Analysis and Instrumentation. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 BACKGROUND AND MOTIVATION

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[7]. Solidity is a popular object-oriented and high-level language for implementing smart contracts[8, 16] which can be compiled to byte code for execution on the blockchain network. The increased popularity and adoption of smart contracts requires strong security guarantees, which attracted attention from both academia and industry. Existing work on verifying smart contracts for security and vulnerability checks is mainly based on analysing the byte code of smart contracts or translating Solidity programs to other languages or intermediate forms and perform analysis on the translated code. Either of these techniques discards some features of the native Solidity code and make it hard to trace the error back to the source code level. Moreover, testing techniques including fault injection and code coverage analysis are currently missing due to the lack of a comprehensive framework for analysing and instrumenting Solidity programs. To address this issue, we present in this paper, SIF (Solidity Instrumentation Framework) which has the following capabilities:

(1) Provide interfaces as per each Solidity code structure and keyword for users to retrieve syntactic and semantic information of the source code.
(2) Allow users to rewrite the source code by callback functions designed for every type of Abstract Syntax Tree (AST) nodes.

We evaluate our framework by analysing ASTs of smart contracts collected from the running blockchain network, translating ASTs back to smart contracts and comparing them with the original ones. In addition, we test our callback functions for code manipulation by checking the presence of the instrumented code manually.

Users. SIT's envisioned users are smart contract developers and especially researches with the need of analysing the native code of smart contracts and instrumenting the code for their various purpose. Our framework only requires the user to have a working knowledge of C++ and Solidity.

2 RELATED WORK

There exist numerous tools developed for source code analysis and instrumentation for generic or specifically targeted programming languages. However, summarising strengths and weaknesses of these techniques across different programming languages and application domains is beyond the scope of this paper. 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 [11–13, 16]. Byte code is the compiled hexadecimal format of smart 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 another language or represents the code in the XML format and performs code analysis on the intermediate form [6, 9, 15]. However, code conversion causes the loss of some code structures and

ISSTA 2019, 15–19 July, 2019, Beijing, China

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

features of Solidity and also suffers from the problem of mapping the detected vulnerabilities to the original source code.

Parsing Solidity Code. Existing tools to parse smart contracts [3–5] exploit the same idea. They traverse the Solidity code, identify language keywords and map the code to predefined JavaScript or Python classes. A set of these classes is able to represent the entire source code. However, [3] is not maintained and outdated and all of them do not provide an interface for us to instrument the 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, the tool is not maintained and does not support Solidity versions 0.4.3 onwards. Zeus [9] 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 [14] 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.

Moreover, the above AST parsers which are able to generate ASTs and the AST-based code generator cannot be combined together for code instrumentation simply because the structure of the AST used by these tools is different.

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

In this section, we present the main workflow of SIF and how its components fit together. We then describe details about our implementation and illustrate the usability of the framework with use cases with software testing purposes.

3.1 SIF Workflow

At the beginning, users need to fill in a callback function defined by our framework according to their needs and compile the functions with SIF. Once the desired functionality is implemented, the user needs to generate the AST using the Solidity compiler and feed it to SIF. SIF then traverses the AST and every time when it meets a node with the type highlighted in the callback function, it will execute instructions defined by the user e.g. print specific information or rewrite certain parts of the node. Finally, SIF produces the source code reflecting code modifications set by the user. If the user only writes statements to retrieve information from the AST, the unchanged source code is generated.

Phase 1: Callback function implementation. We provide a callback function named *visit* for users to query and instrument the Solidity code. The function has one parameter of the generic AST node type. Every time when SIF finishes gathering information of an AST node, the function is called by providing the AST node as the argument. The user can implement the function by checking if the currently visited node has the desired type as well as property, and make changes to that node. The framework provides adequate methods for querying and modifying all types of 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 *getCondition* 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.

Phase 2: AST generation. SIF relies on the AST generated by the Solidity compiler. Instead of starting from implementing a source code parser for Solidity, we take the existing functionality provided by the official Solidity compiler, which is surely able to handle all language features and code structures. The Solidity compiler generates AST from Solidity code in two formats: plain text and JavaScript Object Notation (JSON). In an effort to understand how the AST is organised in these two formats, we compiled a number of real contracts into both formats. We found that the AST formats, individually, lacked some information. For example, the JSON format does not have complete information on the *import* statement and the plain text format does not record certain function qualifiers including *public*, *pure*, *view*, among others. The information contained in both formats, however, were complementary, and when used together it was possible to retrieve the entire syntax and semantic information associated with the source code. We use both AST formats for our analysis in the phases that follow.

Phase 3: Framework execution with plug-ins. With the callback function as a plug-in to the framework, SIF reads the AST of the original Solidity code and traverses it. Source code reflecting changes defined by the user in the plug-in is generated in this last stage.

3.2 SIF Implementation

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

AST representation in C++ classes. In an attempt to make our tool for Solidity code instrumentation generic and easy to use, we use the C++ classes as our intermediate AST representation. We implement classes, equivalent of the different AST nodes, containing information captured in both AST formats (text and JSON). 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 `if statement` node will have a method that can retrieve information on the condition expression for the then block and else block. Use of our instrumentation tool does not require the knowledge of JSON or the format and structure of the Solidity AST.

Code instrumentation. After analysing the AST generated by the Solidity compiler, all AST nodes are stored in C++ classes. SIF then traverses the AST starting from the root node in a depth-first fashion. The callback function (the plug-in) is called when it processes each node as mentioned in Section 3.1. If the plug-in has

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 curly 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 curly brace is added which indicates the end of the struct definition.

3.3 Example Use Cases

We briefly describe how to implement plug-ins with two use cases to utilise feasibility and general applicability of our framework. To the best of our knowledge, there exists no means of measuring code coverage for or seeding faults to Solidity programs. However, coverage measurement and fault injection are among the most fundamental and common techniques in the domain of software testing.

Code coverage report. If we want to check whether functions and code blocks in control flows are exercised by a given input, we can implement the callback function in a small number lines of code. The user only needs to write conditional statements to determine if the currently visited node is of any type among `function definition`, `if statement`, `for statement`, etc. For each node of function definitions and loops, we can simply ask SIF to add extra statements to the beginning of its body to record this flow has been entered. Similarly, extra statements can be added to branches of `if statements`. The user can then feed the AST of the smart contracts to SIF with the plug-in and it will produce an instrumented version with the inserted statements for control flows. Executing the instrumented version with inputs will log code coverage achieved by the given input.

Fault seeding. To seed vulnerabilities to smart contracts, we need to implement the callback function to modify the AST and seed a particular vulnerability by creating a new AST node containing the vulnerability. For instance, we can write a plug-in to inject a *Time Stamp Usage* fault, which has been identified as a well-known type of vulnerability [11, 13], to smart contracts. To do this, we can define a statement node containing the *Time Stamp Usage* fault and when a function definition node is visited, this statement is added to the function body. When the framework with this plug-in is applied to the AST of a smart contract, the generated code will contain this fault in its functions.

4 EVALUATION

5 CONCLUSION AND FUTURE WORK

SIF provides the capability to analyse the AST of Solidity smart contracts produced by the Solidity compiler and generate source code according to the AST and changes defined by the user, without any prior Solidity AST knowledge. The framework is promising in the aspect of testing techniques which requires source code instrumentation, including code 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 from and make changes to the AST and (3) generating Solidity source code reflecting changes back from the instrumented AST.

We consider the following future work to enhance our framework:

- ***AST matcher implementation.*** The idea is derived from the Clang Tooling framework [10]. Instead of writing conditional statements to determine if the node is the one we want to instrument, AST matchers provide a fancier way by calling *has* functions. For instance, a binary operation with the plus operator and integer literal 0 as the left hand operand can be accessed by calling `binaryOperator(hasOperatorName("+"), hasLHS(integerLiteral(equals(0))))`.
- ***Always up-to-date support.*** Our current implementation is based on the Solidity version 0.4.25 which is adequate for current running contracts on blockchain networks. However, the syntax of Solidity might change in the future. We will adopt the tool to support new features accordingly.
- ***AST generation integration.*** The user is required to feed the AST generated by the Solidity compiler to the framework. We plan to integrate this functionality as part of SIT and therefore the user can only provide the original source code as input and does not need to perform the extra step of AST generation.

REFERENCES

[1] 2016. Soltar. https://github.com/duaraghav8/soltar
[2] 2017. Porosity. https://github.com/comaeio/porosity
[3] 2017. Solidity Parser. https://github.com/ConsenSys/solidity-parser
[4] 2019. Python Solidity Parser. https://github.com/ConsenSys/python-solidity-parser
[5] 2019. Solidity Parser Antlr. https://github.com/federicobond/solidity-parser-antlr
[6] 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.
[7] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* (2014).
[8] Chris Dannen. 2017. *Introducing Ethereum and Solidity*. Springer.
[9] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing safety of smart contracts. NDSS.
[10] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.
[11] 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.
[12] MythX. 2019. Mythril Classic: Security analysis tool for Ethereum smart contracts. https://github.com/ConsenSys/mythril-classic
[13] Remix. 2018. Remix documentation. https://remix.readthedocs.io/en/latest/

[14] PNF Software. 2019. Ethereum Smart Contract Decompiler. https://www.pnfsoftware.com/blog/ethereum-smart-contract-decompiler/

[15] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. (2018).

[16] 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.