EasyChair Preprint

№ 3332

SPrune: A Code Pruning Tool for Ethereum Solidity Contract Static Analysis

Zihan Zhou, Yan Xiong, Wenchao Huang and Lu Ma

May 4, 2020

SPrune: A Code Pruning Tool for Ethereum Solidity Contract Static Analysis

Zihan Zhou*, Yan Xiong*, Wenchao Huang*, Lu Ma†
*School of Computer Science and Technology
University of Science and Technology of China, Hefei, Anhui, PR China
†Beijing Institute of Remote Sensing, Beijing 122000, China
Email: zihzh@mail.ustc.edu.cn, {yxiong, huangwc}@ustc.edu.cn, syml1234@163.com

***Abstract*—Ethereum is a cryptographic currency system built on top of blockchain. It allows anyone to write smart contracts in high-level programming languages, solidity is the most popular and mature one. In the last few years, the use of smart contracts across domains has increased a lot, security analysis to detect the potential issues in contracts thus becomes crucial. Theorem proving is a formal method technique which mathematically prove the correctness of a design with respect to a mathematical formal specification, that can be applied to smart contracts' secure analysis. The successful implementation of a deduction calculs of theorem proving in an automated reasoning program requires the integration of search strategies that reduce the search space by pruning unnecessary deduction paths.**

This paper desribes SPrune, a code pruning tool designed to simplify static analysis for solidity contracts. It works by unfolding derived contracts based on the inheritance between contracts in one smart contract, and execute code pruning on the unfolded contract. Our tool allows for the application of static code pruning and provides facility for solidity contract developers and testers to trace and localize bugs in contracts.

***Index Terms*—Ethereum, Solidity, smart contracts, static analysis**

I. Introduction

Blockchains are underlying technologies for promising secure distributed computations even in the absence of trusted third parties. It is the decentralized nature of blockchain assures transactions using cryptocurrencies autonomously and truthfully executed. The most prominent and popular blockchain-based distributed computing platform is an open-source, public platform called Ethereum. It is also an operating system featuring smart contract functionality [1]. Developers write the smart contract in Turing-complete languages to develop a variety of applications across domains, including the financial industry, supply chain management, and government services.

Despite the increasing use of smart contracts in various domains, the deployed smart contract can be insecure. Bugs and vulnerabilities in smart contracts may lead to severe consequences. For instance, 150M were stolen from the popular DAO contract by exploiting the fallback function in the code that was exposed to reentrancy in June 2016, 30M were stolen from the widely-used Parity multi-signature wallet by exploiting the delegatecall and fallback function in the smart contract library for the multi-sig wallets in July 2017, and a few months later 280M were frozen due to a bug in the very same wallet. Today, Luu [2] estimate that 45 percent of smart contracts written in Ethereums programming language Solidity are vulnerable. Effective security analysis for smart contracts is thus crucial for the trust of the society in blockchain technologies and their widespread deployment.

Solidity is the most mature high-level language for writing smart contract [3], [4]. It allows developers to write smart contracts and compile to bytecode of the Ethereum virtual machine (EVM). Existing tools and frameworks designed to analyze smart contract were generally based on analyzing the bytecode of solidity contracts, or transform solidity code to other programming languages or intermediate representations [2], [4]–[6]. But most of these tools are based on functional verification and can only scratch the surface, as they work by verifying that the logic design conforms to the specification. When it comes to financial concerned contract, we want to be sure that our contract behaves correctly 100% of the time.

Formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics. It allows us to prove conclusively that certain error states can never occur. Theorem proving is a formal method for checking a infinite system. In theorem proving, systems are defined and specified by users in an appropriate mathematical logic. Important/critical properties of the system are verified by theorem provers. Theorem prover checks that whether a statement (goal) can be derived from a logical set of statements (axiom/hypothesis). It can model and verify any system that can be defined with the help of mathematical logics.

In this paper, we provide SPrune, a solidity code pruning tool to predigest smart contract for further verification. To make the smart contract suited to further modeling process, we first unfold derived contracts in one smart contract, using the inheritance between them. In solidity, inheritance is make a base contract that implement functionality and reuse it in future contracts. When a contract inherits from multiple contracts, only a single contract is created on the blockchain, and the code from all the base contracts is copied into the created contract. After unfolding the contracts, we unfold function calls with their respective definition to enhance the codes' clarity and make it less verbose.

We evaluate SPrune on a collection of smart contracts that

are running on the Ethereum network. Our result shows that the utility is easy to use and could effectively cut down the size of solidity contract, while maintaining the necessary functionality.

II. Background Information

In this section, we provide a brief introduction to Ethereum platform, smart contracts and theorem proving.

A. Ethereum Platform

Ethereum is an open software platform based on the blockchain technology. The developers can implement, compile, test, deploy, and execute the centralized applications upon it [1]. Ethereum provides a decentralized virtual machine, the Ethereum Virtual Machine (EVM), which is designed to serve as a runtime environment for Ethereum smart contracts and can execute scripts using an international network of public nodes. Ethereum network is a distributed and decentralized network with permission-less untrusted peers.

Ethereum protocol moves far beyond just currency. Protocols and decentralized applications around decentralized file storage, decentralized computation and decentralized prediction markets, among dozens of other such concepts, have the potential to substantially increase the efficiency of the computational industry, and provide a massive boost to other peer-to-peer protocols by adding for the first time an economic layer.

B. Smart Contracts

Smart contracts in Ethereum are computer programs written in Turing-complete programming languages, the most mature and wide use one is Solidity [7]. Smart contracts are compiled to bytecode that is executable on the EVM. Ether is the second largest cryptocurrency in the world after Bitcoin, and smart contracts can be used to quickly build decentralized applications on the Ethereum platform. Users can use compatible programming languages to write any rules and functionalities, and the rules are encoded as smart contract to invoke an action, whenever it is required by users or other smart contracts. Smart contracts can be applied to various kinds of applications such as financial transactions, prediction markets and internet of things. Users invoke smart contracts by referring transactions toward the contract address [8].

C. Theorem Proving

Two most popular formal verification methods are model checking and theorem proving. In model checking, a finite model of the system is developed first, whose state space is then explored by the model checker to examine whether a desired property is satisfied in the model or not [9]. However, model checkers still face the state-space explosion problem [10]. Theorem proving on the other hand, can be used to handle infinite systems. In theorem proving, systems are defined and specified by users in an appropriate mathematical logic. Important/critical properties of the system are verified by theorem provers. Theorem prover checks that whether a statement (goal) can be derived from a logical set of statements (axiom/hypothesis). It can model and verify any system that can be defined with the help of mathematical logics.

III. Pruning Method

To improve the efficiency of applying theorem proving to smart contract verification, we proposed to cut down the size of contract and only retain codes that are necessary to further analysis. Considered solidity is the most mature high-level programming language for writing smart contract, we provided SPrune, a solidity code pruning tool to predigest smart contract for further verification, so as to reduce the search space of deduction before further modeling process.

Our solution completes in 3 steps. Step 1 focuses on using the inheritance between contracts to unfold contracts in a smart contract. Step 2 unfold calls of function with their respective definition. Step 3 executes code pruning and retain codes that are necessary to further analysis. We will discuss each of these steps in more detail in the rest of this Section.

A. Unfolding of Contracts

Solidity supports multiple types of inheritance, including multiple inheritance. Solidity copies the base contracts into the derived contract and a single contract is created with inheritance. A single address is generated that is shared between contracts in a parent-child relationship. To simplify further code pruning process, we need to unfold derived contracts based on the inheritance between contracts in one smart contract. We use an example shown in Listing 1, containing a base contract and a derive contract, to illustrate the phase.

```
1  contract Base{
2      uint internal id;
3      function setValue(uint _value) external {
4          id = _value;
5      }
6  }
7
8  contract Derive is Base{
9      function getValue() external view returns(uint) {
10         return id;
11     }
12 }
```

Listing 1. Contract combining example

Inheritance marks several associated contracts with a parent-child relationship, the contract that inherits from another contract (the parent) is the child. All function calls are virtual in solidity, which means that the most derived function is called, except when the contract name is explicitly given or the super keyword is used.

```
1  contract Derive {
2      uint internal id;
3      function getValue() external view returns(uint) {
4          return id;
5      }
6      function setValue(uint _value) external {
7          id = _value;
8      }
9  }
```

Listing 2. Contract combining example

In this case, the parent and child correspond to the Derive contract and the Base contract, respectively. The unfolded contract is shown as Listing 2. Contract Derive inherits from Base, only Derive is created on the blockchain, and the code from the base contract is compiled into the Derive.

B. Unfolding of Function Calls

To enhance clarity of the codes and facilitate further code processing, we unfold function calls with their respective definition after unfolded contracts in the same parent-child relationships. There shows three forms of function calls to unfold in our situation [11].

- Call functions defined in the same contract.
- Explicitly give contract name or use "super" keyword to call external function.
- Use the directive "using A for B" to attach functions from library A to type B, these functions will receive the object they are called on as their first parameter.

We use different templates to unfold the above forms of function call. In this section, we take the library function call in Listing 3 as example.

```
1  library libA {
2      function add(uint256 a, uint256 b) internal pure
           returns (uint256) {
3          uint256 c = a + b;
4          assert(c >= a);
5          return c;
6      }
7  }
8
9  contract C {
10     using libA for uint256;
11     function f(uint x, uint y) {
12         x = x.add(y);
13     }
14 }
```

Listing 3. Function call replacement example

For this form of call, we first check the object that called, if the object type is attached with a library, take the object as first parameter and replace the function call with function definition in corresponding library.

```
1  contract C {
2      function f(uint x, uint y) {
3          uint256 c = x + y;
4          assert(c >= x);
5          x = c;
6      }
7  }
```

Listing 4. Function call replacement example

C. Execution of Code Pruning

To reduce the search space in further theorem proving, the code that are unecessary to analysis should be removed. In this paper, we consider code that related to balance change as criterion.

```
1  contract C {
2      mapping (address => uint256) public balanceOf;
3      function senderTransfer(){
4          msg.sender.transfer(msg.value);
5      }
6      function subBalance(uint x) {
7          balanceOf[msg.sender]=balanceOf[msg.sender]-x;
8      }
9      function getBalance(){
10         return balanceOf[msg.sender];
11     }
12 }
```

Listing 5. Code pruning example

Take the contract in Listing 5 as an example, any code that involves transaction or balance operation need to be retained. After the unfolding of contract inheritance and function call, for each function that contains code that meet our criterion, it also contain the code that may affect the matching code. Thus, retain the functions which contain code that are relevant to criterion and prune the left ones can reduce the size of contract a lot, without affecting the functionality we concerned.

```
1  contract C {
2      mapping (address => uint256) public balanceOf;
3      function senderTransfer(){
4          msg.sender.transfer(msg.value);
5      }
6      function subBalance(uint x) {
7          balanceOf[msg.sender]=balanceOf[msg.sender]-x;
8      }
9  }
```

Listing 6. Code pruning example

In this case, call of "transfer" function and subtract operation to an element of "balanceOf" array meet our pruning criterion. Therefore we retained the functions that contained these code, and removed those that are unrelated to criterion. The pruned code of this example is shown as Listing 6.

IV. Evaluation

We evaluate efficiency, syntactic support in using our pruning tool over 100 unique Solidity contracts randomly chosen from the Ethereum network.

A. Solidity Language Support

How does SPrune support constructs in the Solidity syntax? To answer this question, we first inspect contract code in our dataset and the official Solidity documentation to check the range of syntactic structures present in the dataset. As the unfolding of function call is the most syntactic concerned phase in the working process of SPrune, we discuss supported types of function call in this section.

Function calls in Solidity can be of several types: internal, external, delegate, and calls to certain builtin functions. Internal function calls are simply jumps in the code of the current account. External calls cause a message to be sent over the Ethereum network, executing code on another account. Delegate calls exist to provide a functionality akin to shared libraries. That is, they allow code from another account to directly operate on the storage of calling account [11].

Since SPrune is built to facilitate static analysis, we only unfold internal calls and external calls in SPrune. For external calls, the code executed by outgoing external function calls may not be available, or not written in Solidity, thus we decide to focus on the calls to the known function in current contract file.

B. Syntactic Correctness

Does the output produced by SPrune satisfy syntactic correctness? In the case of predigestion for static analysis, we have to take the correctness of output into account. To validate this property, we ran SPrune on all smart contracts contained in our dataset. The outputs were checked through the solc compiler for syntactic correctness, no problems were found. Apart from this, we also test functions in the pruned contract on a browser-based Solidity realtime compiler named Remix to check whether the functions work as original. We ran the functions with the same input as we ran the original ones, the results appeared to be consistent.

To ensure the syntactic correctness of SPrune's output, we will use formal verification to verify it's correctness in our future work.

C. Time Efficiency

Time efficiency is one of the most concerned properties when analyzing process tools. We present the composition of our dataset in Figure 1. Contracts with less than 1000 lines of code take 93 percent of the dataset, and those with less than 500 lines of code take 76 percent.

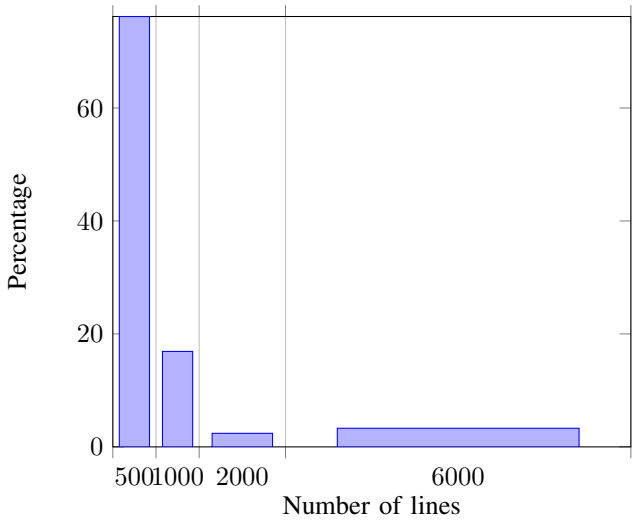

Fig. 1. Dataset composition

We randomly choose 100 contracts from the dataset with respect to the proportion and inspect the time efficiency of SPrune. The average time consuming for each set of contracts in using SPrune to prune smart contracts with different size are reported in Figure 2. SPrune finished pruning in 1.25 seconds for 80% of the contracts with less than 500 lines of code, 5 seconds for 90% of those with less than 1000 lines of code. Still, several contracts took times of the average to finish pruning. To understand why code pruning for these contracts took significantly longer, we profiled the SPrune phases and found that 95% of the time was spent on replacing the function call. The most time consuming part of SPrune's execution is the function call replacement phase, we use recursive queries to look for function definition, Thus the running time is mainly affected by the number of function calls in a contract.

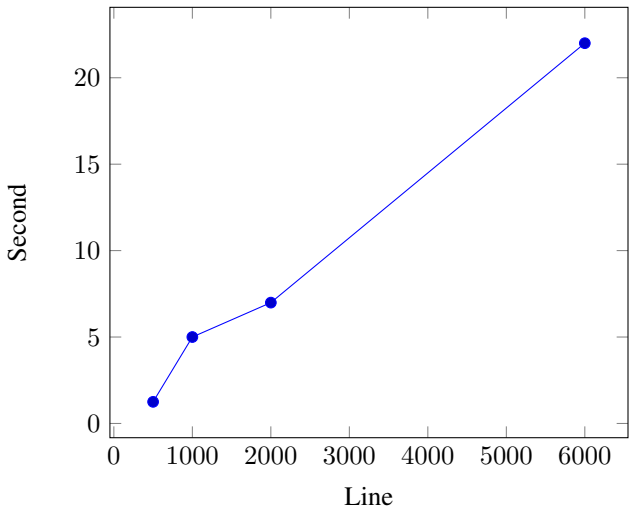

Fig. 2. Time consuming in each set of contracts

In our future work, we will find ways to optimize and accelerate function call replacements.

D. Pruning Efficiency

To validate that SPrune can facilitate static analysis effectively, we have used several solidity static analyzers to analyze the original contracts in our dataset and the pruned ones [12]. For each analyzer, we ran it over 100 contracts and their corresponding pruned contracts, and calculated the average ratio of time consuming. The result of the comparison is reported in Table I, P and O represent the execution time of pruned contracts and original contracts respectively.

TABLE I
PRUNING EFFICIENCY EVALUATION RESULT

Analyzer	Execution time(P) / Execution time(O)
SIF	21.24%
Smartcheck	88.83%
Slither	65.77%

SIF [13] is a comprehensive framework for solidity analysis, it takes both the source code of solidity contracts and the correspond AST as inputs and analyzes based on the AST of contracts. We used SIF to generate solidity code from the AST of the two sets of contracts. When using the ASTs of pruned contracts as input, the time it takes to generate code is 21.24% of the ASTs of original contracts. To verify the results of SIF, we checked the size of ASTs that generated by solc compiler and confirmed that the size of the ASTs of contracts could be cut down to 47.91% of the originals on average, which made the framework took far less time to generate code from the ASTs of the pruned contracts.

Smartcheck [6] automatically checks for vulnerabilities and bad coding practices, it also highlights the vulnerability, gives

an explanation of the vulnerability, and a possible solution to avoid a particular security issue. It works by translating Solidity source code into an XML-based intermediate representation and checks it against the defined vulnerability criteria in intermediate representation terms. That makes it takes more analysis time than other tools, and the analysis time for the pruned contracts is 88.83% of the analysis time for originals.

Slither [14] can be used to detect code vulnerabilities and code optimization opportunities. Slither works by first generate the control flow graph and the list of expressions from the AST of contract, then transform the source code of smart contract to an intermediate representation, and perform actual code analysis in the third stage. The source code of contracts and the corresponding ASTs were used in two phases of analysis. When using Slither to analyze the two sets of contracts, the average analysis time of pruned contracts takes 65.77% of the original ones.

The result shows that use SPrune to prune smart contract code can smooth the way for further analysis, but the effectiveness depends on how the tools work.

E. Summary

Overall, our results indicate that SPrune works smoothly in reducing the size of smart contracts and facilitating further analysis. Going further, we see three relevant items for future work.

First, to ensure that SPrune produces contracts correctly, we will prove the correctness of the process formally in the future work. Second, it would be feasible to integrate SPrune with program slicing technique that provides fine-grained code slicing, a way to further facilitate theorem proving. Third, as there exists no pruning tool to facilitate smart contract analysis, we can leverage SPrune to improve existing solidity analysis, optimization and verification techniques. For example, SPrune can be used to reduce the time consuming of these tools or reduce search space for deduction calculs of theorem proving.

V. Conclusion and Future Work

We presented SPrune, a code pruning tool for Ethereum smart contracts, to facilitate theorem proving in further analysis. SPrune leverages the inherent nature in smart contracts that only the most derived contracts are deployed in the Ethereum network. Based on this insight, SPrune produces the contracts that actually deployed and prunes the code that are unnecessary to further theorem proving analysis. We used a dataset with 1803 contracts in our evaluation.

SPrune improved smart contract verification efficiency by reduce search space for theorem proving, there are also directions that we could take to improve it. For instance, SPrune is still a crude tool for smart contract pruning, we will look to prove the correctness of SPrune and bring slicing technique in to make SPrune fine-grained.

Acknowledgement

The research is supported by the National Key R&D Program of China 2018YFB0803400, 2018YFB2100300, National Natural Science Foundation of China under Grant No.61972369, No.61572453, No.61520106007, No.61572454, and the Fundamental Research Funds for the Central Universities, No. WK2150110009.

References

[1] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," https://github.com/ethereum/wiki/wiki/White-Paper, 2013.

[2] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[3] C. Dannen, *Introducing Ethereum and Solidity*. Apress, Berkeley, CA, 2017.

[4] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[5] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: A survey," *ArXiv*, vol. abs/1908.08605, 2019.

[6] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, May 2018.

[7] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F. Wang, "An overview of smart contract: Architecture, applications, and future trends," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, June 2018, pp. 108–113.

[8] P. Praitheeshan, L. Pan, J. Yu, J. K. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: A survey," *CoRR*, vol. abs/1908.08605, 2019. [Online]. Available: http://arxiv.org/abs/1908.08605

[9] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[10] Clarke, Edmund M. and Klieber, William and Nováček, Miloš and Zuliani, Paolo, *Model Checking and the State Explosion Problem*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[11] J. Zakrzewski, "Towards verification of ethereum smart contracts: A formalization of core of solidity," in *Verified Software. Theories, Tools, and Experiments*, R. Piskac and P. Rümmer, Eds. Cham: Springer International Publishing, 2018, pp. 229–247.

[12] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 51–78. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_4

[13] C. Peng, S. Akca, and A. Rajan, "SIF: A framework for solidity contract instrumentation and analysis," in *26th Asia-Pacific Software Engineering Conference, APSEC 2019, Putrajaya, Malaysia, December 2-5, 2019*. IEEE, 2019, pp. 466–473. [Online]. Available: https://doi.org/10.1109/APSEC48747.2019.00069

[14] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. IEEE / ACM, 2019, pp. 8–15. [Online]. Available: https://doi.org/10.1109/WETSEB.2019.00008