SCO: A Visual Smart Contract Obfuscation Tool

1. Introduction and Problem Statement

Our project aims to develop a visualized, highly integrated, all-in-one code obfuscation tool specifically for smart contract source code.

Through our project, we hope to address the current lack of dedicated obfuscation tools for Solidity code and solve the issue of scattered functionalities, such as visualized code structure analysis and security analysis.

2. Background and Related Work

With the rapid development of Blockchain 2.0 technology, an increasing number of smart contracts based on Ethereum's Solidity are emerging. At the same time, both developers and blockchain users are facing a growing demand for security tools tailored to Solidity, to ensure the safety of their assets [1].

Smart contract code written in Solidity must first be compiled into bytecode and an ABI (Application Binary Interface) using compilers like solc. The compiled bytecode is then sent to the Ethereum network through an Ethereum client. Upon receiving the deployed Ethereum bytecode, the Ethereum network executes it on the Ethereum Virtual Machine (EVM). The EVM sequentially executes the bytecode instructions, updating the contract state (e.g., variable updates) or triggering other operations (e.g., events or transfers) [2].

However, unlike machine code, the bytecode executed by smart contracts is susceptible to reverse engineering by attackers. This enables them to analyze vulnerabilities within the code and exploit them. Furthermore, due to the deterministic and immutable nature of smart contracts [2], developers cannot address existing vulnerabilities by releasing patches as they would in traditional software.

Therefore, developers need to enhance the security of their smart contracts through techniques like code obfuscation, which can thwart attackers' attempts to discover vulnerabilities via reverse engineering. However, our observations reveal that most existing solutions for Solidity code obfuscation remain experimental and lack industrial-grade tools. Moreover, there is no integrated software that consolidates the functionalities developers need.

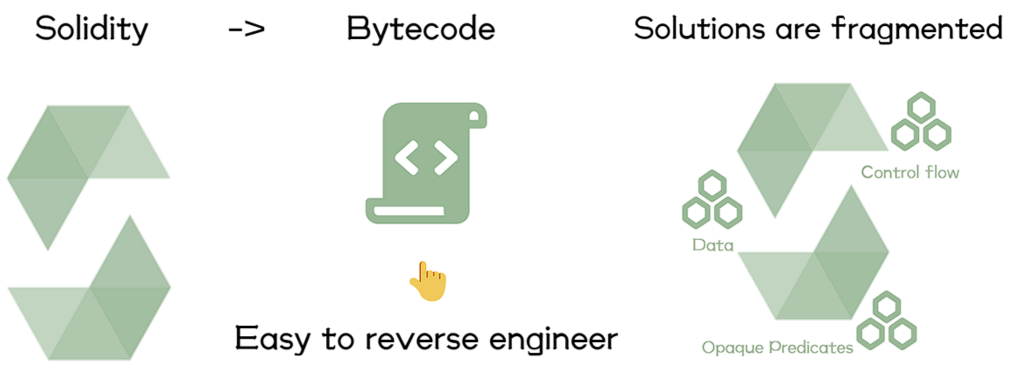


Figure 1 Security issues facing current smart contracts

Smart contract developers often spend significant time searching for an effective code obfuscation method, additional time identifying suitable security evaluation metrics, and must also contend with the complexity and poor usability of existing tools.

To address the aforementioned challenges shown in Figure 1, we have decided to develop an integrated, visualized obfuscation tool for smart contract source code that caters to the urgent needs of smart contract developers.

Main contributions of our project:

- **Consolidation of Best Practices:**

  Aggregate and implement state-of-the-art smart contract obfuscation and security evaluation techniques.
- **Integrated Software Design:**

  Develop software that combines code obfuscation, security evaluation, file management, and syntax tree visualization functionalities.
- **User-Friendly Interface:**

  Create a visualized interface that enables smart contract developers to quickly and effectively utilize our software.

3. Motivation

Our motivation is to develop a visualized, highly integrated, all-in-one code obfuscation tool to address the pressing needs of smart contract developers for enhanced contract security, shown in Figure 2. This tool resolves the challenges posed by the immaturity, fragmentation, and complexity of existing Solidity obfuscation technologies.

Figure 2 Three objectives of our project

Three objectives of our project:

i. **All-in-One:**

   "All-in-One" signifies that we collect the state-of-the-art methods for smart contract obfuscation, security evaluation, and visualized development tools. We aim to build a centralized smart contract security management framework, offering developers a practical and effective one-stop solution.

ii. **Integration:**

   "Integration" means we package all our solutions into a standalone application. Users can install the application to perform centralized project management, code obfuscation, and security analysis directly, streamlining their workflow.

iii. **Visualization:**

"Visualization" refers to presenting our application through an elegant, visualized GUI. Users won't need to write code or execute commands to utilize our features. Instead, they can complete the entire process by simply clicking buttons on the interface, significantly reducing the learning curve for using our smart contract obfuscation tool.

4. Methodology

1. Desktop Application Framework

Using Windows desktop application development model Electron + Vue + Vite. Electron combines Chromium and Node.js, allowing developers to build desktop applications with a web development approach [3].

In the main process of the project, we include four core modules: **projectsHandler**, **projectsReader**, **obfuscator**, and **evaluator**.

➢ **projectsHandler** contains the following methods:

1. **importProject:** Creates a project folder in the project directory based on the selected source code file path. The source code is saved as original-code.sol, and a terminal subprocess is initiated to call the built-in solc compiler to generate an abstract syntax tree (AST) file (saved in JSON format). The terminal output is returned to the renderer process for processing.
2. **deleteProject:** Deletes the local project folder and all its contents recursively.
3. **openExplorer:** Opens a Windows File Explorer subprocess and navigates to the project folder directory.

➢ **projectsReader** contains the following methods:

1. **getProjects**: Scans all projects under the project folder path and returns an array of folder names and their contents to the renderer process for processing.
2. **loadInFile**: Reads the selected file's content and returns the file format information as a dictionary to the renderer process for processing.

➢ **obfuscator c**ontains the following method:

1. **obfuscateTargetProject**: Launches a terminal subprocess to call a code obfuscation tool packaged with pyinstaller, generating obfuscated code for the specified project's original-code.sol. The output is saved as output-code.sol. A terminal subprocess is also initiated to call the built-in solc compiler to generate an AST file (saved in JSON format). The terminal output is returned to the renderer process for processing.

➢ **evaluator** contains the following method:

1. **evaluateTargetFile**: Uses an open-source smart contract security analysis tool, **Solidity Code Metrics** [4], to calculate the specified file's lines of code, various types of comment lines, and a **Complexity Score** based on branches, loops, calls, and external interfaces. The results are sent to the renderer process for processing.

In the renderer process, our project includes seven core Vue components: **Nav**,

Editor, **Explorer**, **Files**, **Setting**, **JsonTreeGraph**, and **Terminal**.

- **Nav**: A top navigation bar component that provides entry points for file management, code obfuscation, and code evaluation features.
- **Editor**: A code view area component. It fetches the selected file and file type from the main process and dynamically adjusts rendering based on the file type using v-if. For .sol files, the text content is rendered with CSS styling. For AST files, the **JsonTreeGraph** component is used to visualize the abstract syntax tree.
- **Explorer**: A left-side navigation bar component that allows switching between file management and settings functionalities, invoking the **Files** and **Setting** components accordingly.
- **Files**: A file management component that communicates with the main process to refresh the file information in the project folder and render project details in the left navigation bar. It provides a user interface for selecting and deleting files.
- **Setting**: A settings management component with options rendered in the left navigation bar. Currently, it only supports switching between dark and light themes.
- **JsonTreeGraph**: Utilizes the powerful SVG editing tool **D3** to render JSON-formatted AST files, supporting drag-and-zoom functionality.
- **Terminal**: A terminal display component that retrieves the output information of subprocesses intercepted from the main process, including solc compilation messages, obfuscator outputs, evaluator outputs, and various error messages.

2. Code Obfuscation Scheme

During the writing process, in terms of code obfuscation schemes, it is essential to avoid reinventing the wheel. Therefore, we started from open-source schemes and gradually explored available tools, including obfuscation tools and tools for testing the obfuscation effect. However, we soon realized that Ethereum mainly controls "reading permissions" rather than "protective programming." As a result, most of the smart contract code, in order to gain the trust of users, after being obfuscated, defends against both attackers and users, which violates the principles of "clarity, transparency, and supervision." Thus, it is rather difficult to find a deeply considered smart contract obfuscation scheme on the Internet. We also examined some related schemes and got some inspiration. Therefore, we extracted some of the ideas and schemes [5] that we considered meaningful and formed two different ideas in the project.

- **Disassembly and storage based on syntax tree**

The first scheme, in terms of the concept, is to disassemble, distinguish, and preserve as much as possible with the help of the syntax tree. During the compilation process of the smart contract, if using integrated compilers commonly available in the market such as Remix, the generated intermediate syntax tree needs to be searched and copied manually, which is not conducive to the accuracy of our compilation. In some early compilers for smart contracts, an executable file named "solc.exe" for generating the syntax tree can be found. Although the results obtained after running on the Windows system and the Mac system are not exactly the same (due to some underlying reasons), it generates a compact JSON file syntax tree. Although this syntax tree has a considerable size (a few dozen lines of Solidity code can generate hundreds of lines), its saturated information provision offers a starting point for our obfuscation work.

However, a syntax tree file of a certain length also brings some difficulties to our disassembly work. Therefore, our approach is to conduct a saturated disassembly and storage (without backtracking) by combining the syntax tree with the source code of the smart contract. All the disassembled content should be stored as much as possible, and its function, structure, and whether it should be obfuscated need to be determined. It has to be mentioned that some problems emerged here. Since we encapsulated the entire source code with a complete large class and completed the override, when we found that the first two lines of declarations in the smart contract had an unexpected obfuscation that led to compilation failure, we could only protect it by matching the first word within the program, which also wasted a certain amount of time.

The work of saturably disassembling all the content is admittedly rather complex, because each part requires matching and correspondence. And in the final experiment, we still found that some parts of the code could not function properly. This is due to incorrect type matching and the problems caused by roughly merging the same types and performing replacements. For example:

```
mapping(uint256 => mapping(address => bool)) public votes;
```

Such directed variables do not make distinctions during declaration, which leads to a series of problems. And it is actually rather difficult to search for them in the AST.

By using command line arguments in main.py for reading, after reading in the source code and the syntax tree and conducting error exploration, we use the command line arguments to determine the degree of obfuscation required and specify the location of the output file (attempting to perform replacements on the source code to achieve a seamless process in the compiler, but it is not easy to debug). After obtaining the content, we determine the target obfuscation content, such as variables and function bodies, by comparing the two files (with a focus on searching the syntax tree). After exploring JS code obfuscators, we believe that obfuscating function bodies is of top priority because in the object-oriented programming process, the function body is the smallest independently operating unit. However, it turns out that this measure will cause some damage to the effectiveness of the smart contract. In short, we have attempted to implement it.

First, extract the function name, function body, parameter list, return function type, and other independent statements from the function definition line, and also store the positions of the independent statements in a list. Then, find the information about where the function is called and perform a match between the actual parameters and the formal parameters (to prevent issues with replacing literals). Obfuscate the extracted variable names together with the variable declarations outside the function. Here, we came across an interesting approach, that is: not caring whether the variable names are easy to read, simply replacing them with underscores of different lengths, which maximally protects the lookup of variable names. After testing, this scheme has brought almost destructive changes to the manual reading and function speculation of the obfuscated code.

Subsequently, we make changes to the function content. During the just-completed reading process, we counted the number of independent statements within the function and the process of function calls. We target the functions with "few calls and many statements" for inserting opaque predicates to minimize fuel waste (since it concerns money after all). After locking onto the functions where opaque predicates are to be inserted, we generate the number of useless arrays and useless indices to be added through random numbers and create these useless contents, randomly filling in some junk information to prevent detection by variable checks during operation. Then, using a certain number generated from these useless arrays as the root and a fixed number as the target, we select through random numbers how many steps it takes to change from the root to the target, and then generate longer or shorter arithmetic expressions through addition and multiplication. Use this arithmetic expression as the condition of an if-else statement to maintain calls and creations, and intersperse some independent statements with the true condition to execute the above steps, combine them into a new function body, and return it. Through the same arithmetic expression generation method, transform each constant integer in the source code (which generally represents important information such as the number of people and amounts of money) into a random seed and generate a new arithmetic expression.

Overall, this method is highly complex and comprehensive, but it cannot be regarded as a unique obfuscation scheme for smart contracts because it does not always take into account the characteristics of smart contracts. Moreover, our method for evaluating the obfuscation results is by comparing the syntax trees before and after. However, since our obfuscation scheme also utilizes the syntax tree, it is somewhat inappropriate for the referee to call the shots on its own.

- **Optimized for the characteristics of smart contracts**

We widely observed and summarized the cases of smart contracts and found that smart contracts have the following characteristics: The functions of smart contracts are not overly complex. Thus, typical smart contracts tend to have the features of a single file and a small number of lines. The key information that smart contracts need to protect is always related to trade. We should consider strengthening the protection of specific addresses, the number of people, transaction times, transaction amounts, etc. When obfuscating smart contracts, we need to be cautious not to generate excessive additional fuel waste.

Based on its relatively short nature, I think the saturation scheme of "storing everything that should be stored" is only applicable to local execution, and it is feasible to scan the source code repeatedly. Since it is inconvenient to generate the syntax tree, only modifying the source code should also be a characteristic that it should possess. Drawing on the idea of data analysis, the first step should be to clean and format the smart contract code to ensure that it represents the "typical situation" in our analysis and has no abnormal features. Therefore, I choose to extract the declarations in the first two lines for preservation as the first step, delete all the comment content as the second step, and ensure that all line break symbols and indentations are in the appropriate positions (with the help of some VS Code plugins) as the third step.

Subsequently, regular expressions are used for repeated matching and storage. We use keywords matching to locate the positions of variable declarations (thanks to the mechanism of Solidity that doesn't allow intelligent declarations or direct usage). By extracting these declarations, we obtain the variable names and then search for them throughout the entire program. Regular expressions are used to ensure that they are indeed used for variables (such as the characteristics of the preceding and following characters). After that, we generate new variables (to prevent errors caused by different numbers of underscores and to make it easier to debug, we choose to generate random strings of the same length). We replace these variables one by one and leave spaces before and after them to prevent them from becoming unusable.

Similarly, we detect the keywords related to transactions and match the declarations of values, addresses, etc., as well as the positions of "return" in the relevant source code. We use the strengthening methods mentioned in Scheme One to lock them in an if-else branch or split them into two or three variables for separate storage and then combine them for use. For constant integers, we also perform operations to turn them into arithmetic expressions so as to prevent them from being directly read.

1. Code Security Evaluation Scheme

Integrating the excellent open-source smart contract security analysis tool **Solidity Code Metrics** [4], into an Electron project using npm, follow these steps:

1. **Install the Tool:** Use npm to install Solidity Code Metrics in Electron project:

```
npm install solidity-code-metrics
```

2. **Call in the Main Process:** Import the Solidity Code Metrics library in the Electron main process. Use the totals() data interface to calculate:
   - The number of lines of code in a specified file.
   - The number of comment lines of different types.
   - The Complexity Score based on branches, loops, calls, and external interfaces.
3. **Send Data to the Renderer Process:** Send the calculated metrics information from the main process to the renderer process for further processing or display using Electron's IPC communication.

5. Evaluation

We evaluate obfuscation from two perspectives:

1. **Functionality Consistency**
2. **Gas Consumption and Code Complexity**

For code complexity, we’ve already mentioned the scoring method. For gas consumption, we evaluate it in a real environment using the Remix virtual machine. The three types of gas costs are:

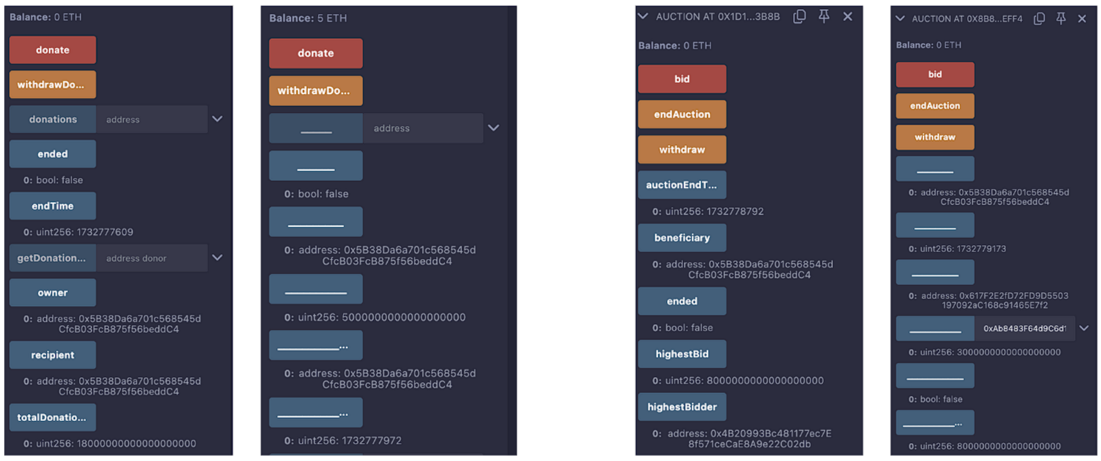
1. **Execution:** represents the gas consumed for actual code execution.
2. **Transaction Cost**: includes the execution cost and a fixed base fee.
3. **Gas Limit**

Writing to storage costs 20,000 gas, which is much larger than the other costs, shown in Table 1.
These relationships allow us to evaluate the impact of obfuscation in terms of cost, potency, and stealth.

We also explored resilience by attempting to reverse-engineer the bytecode to opcodes using disassembler tools. In practice, the obfuscated version was significantly more complex and longer.

Table 1 Operation and gas cost

Operation	Gas Cost
Addition (ADD)	3 Gas
Storage Write (SSTORE)	20,000 Gas
Storage Read (SLOAD)	800 Gas
Hashing (KECCAK256)	30 Gas + 6 Gas per byte



```
Balance: 0 ETH
0: bool: false
0: uint256: 1732777609
0: address: 0x5B38Da6a701c568545d
        CfcB03FcB875f56beddC4
0: address: 0x5B38Da6a701c568545d
        CfcB03FcB875f56beddC4
0: uint256: 1800000000000000000
```

```
Balance: 5 ETH
0: bool: false
0: address: 0x5B38Da6a701c568545d
        CfcB03FcB875f56beddC4
0: uint256: 5000000000000000000
0: address: 0x5B38Da6a701c568545d
        CfcB03FcB875f56beddC4
0: uint256: 1732777972
```

```
Balance: 0 ETH
0: uint256: 1732778792
0: address: 0x5B38Da6a701c568545d
        CfcB03FcB875f56beddC4
0: bool: false
0: uint256: 8000000000000000000
0: address: 0x4B20993Bc481177ec7E
        8f571ceCaE8A9e22C02db
```

```
Balance: 0 ETH
0: address: 0x5B38Da6a701c568545d
        CfcB03FcB875f56beddC4
0: uint256: 1732779173
0: address: 0x617F2E2fD72FD9D5503
        197092aC168c91465E7f2
0: uint256: 3000000000000000000
0: bool: false
0: uint256: 8000000000000000000
```

Figure 3 Original and obfuscated contract running info

Table 2 Auction of smart contract

	Auction	Auction out
Deploy	561353	913265

Bid	48278	49260
Endauction	39641	40568
Call	2447	2571

Table 3 Gas consumption and complexity score

	Original	Complexity	Out	Complexity
Charity	1429091	35	1848746	68
Auction	561353	30	913265	78
Donate	549304	27	801691	68
Vote	1187170	52	1714141	100
Average	931729	36	1319460	69

Our obfuscation method primarily changes variable names while keeping function names, shown in Figure 3. As shown in the images, although the layout may differ, the number of callable functions remains the same, and their functionality is unaffected. This means obfuscation does not impact the core functionality of the contract.

Table 2 shows that after obfuscation, the deployment gas cost increases significantly. The gas cost for executing functions remains nearly unchanged. We also tested other contracts and found similar results.

Finally, we analyzed deployment gas consumption across four contracts before and after obfuscation, calculating the average increase in gas cost and complexity. On average, the deployment gas cost increased by 41.6%, while code complexity rose by 92%, shown in Table 3.

6. Conclusions

In this project, we developed SCO, an integrated visual smart contract obfuscation tool that addresses the challenges of Solidity code obfuscation. By combining state-of-the-art obfuscation methods with a user-friendly interface, we provide developers with a practical and effective solution to enhance the security of their smart contracts.

Our obfuscation approach demonstrated significant improvements in code complexity and resilience against reverse engineering, as evident from the increased complexity scores and the extended bytecode length. While deployment gas costs increased by an average of 41.6%, the functionality of the smart contracts remained intact, and execution gas costs were minimally affected. This highlights the practicality of our obfuscation strategies in maintaining a balance between security and cost efficiency.

Reference

[1] Kalra S, Goel S, Dhawan M, et al. Zeus: analyzing safety of smart contracts[C]//Ndss. 2018: 1-12.
[2] Zou W, Lo D, Kochhar P S, et al. Smart contract development: Challenges and opportunities[J]. IEEE transactions on software engineering, 2019, 47(10): 2084-2106.
[3] GitHub - alex8088/electron-vite-boilerplate: Comprehensive and security Electron template (TypeScript + Vue3 + Vite).
[4] GitHub - Consensys/solidity-metrics: Solidity Code Metrics
[5] Zhang P, Yu Q, Xiao Y, et al. BiAn: smart contract source code obfuscation[J]. IEEE Transactions on Software Engineering, 2023.