An Empirical Study of Smart Contract Decompilers

Xia Liu Baojian Hua* Yang Wang* Zhizhong Pan
School of Software Engineering
University of Science and Technology of China
{liuxiaer, sg513127}@mail.ustc.edu.cn {bjhua, angyan}@ustc.edu.cn*

***Abstract*—Smart contract decompilers, converting smart contract bytecode into smart contract source code, have been used extensively in many scenarios such as binary code analysis, reverse engineering, and security studies. However, existing studies, as well as industrial engineering practices, all assumed that smart contract decompilers are reliable and trustworthy, to generate correct and semantically equivalent source code from binaries. Unfortunately, whether such an assumption truly holds in practice is still unknown.**

In this paper, we conduct, to the best of our knowledge, the *first* and most *comprehensive* large-scale empirical study of smart contract decompilers, to gain an understanding of the reliability, limitations, and remaining research challenges of state-of-the-art smart contract decompilation tools. We first designed and implemented a software prototype SOLINSIGHT, then used it to study 5 state-of-the-art smart contract decompilers. We obtained important findings and insights from empirical results, such as: 1) we proposed 3 root causes leading to decompiler failures; 2) we revealed 2 reasons hurting performance; 3) we identified 3 root causes affecting decompilation effectiveness; 4) we proposed a measurement metric for completeness; and 5) we investigated the resilience of contract decompilers against program transformations. We suggest that: 1) decompiler builders should enhance decompilers in terms of effectiveness, performance, and completeness; and 2) security researchers should select appropriate decompilers based on the suggestions in this study. We believe these findings and suggestions will help decompiler builders, contract developers, and security researchers, by providing better guidelines for contract decompiler studies.

***Index Terms*—Empirical study, Smart contracts, Decompilation**

I. INTRODUCTION

Smart contracts [1] are programs that are executed inside a peer-to-peer network such as Ethereum [2], where nobody has special authority over the execution, and have been widely used to implement functionalities such as tokens of value, ownership [3], voting [4], finance [5], management [6], healthcare [7], and the internet of things (IoT) [8]. Once smart contracts are written, they are compiled by contract compilers, into binary bytecode [9] and stored on blockchains for execution. As smart contracts are Turing-complete to allow nearly arbitrary business logic to be implemented, they enable autonomous management of cryptocurrency and have the potential to revolutionize future business applications.

Due to their ease of use and high monetary value, smart contracts have been an appealing target for attacks, with many vulnerabilities and security issues [10] being detected or reported. These attacks and security issues, including arithmetic overflows [11] [12], reentrancy [13], unauthorized access [14], external calls [15], unsafe type inference [16], costly loops, overpowered owners, among others, have led to considerable losses. For example, the infamous "TheDAO" reentrancy attack resulted in a loss of more than 50 million US dollars worth of Ether [17].

To address these issues, a significant amount of studies, as well as industrial engineering efforts, have been conducted on reverse engineering [18] [19] [20] [21] and security studies [22] [23] [24] [25] [26] [27]. As smart contracts are deployed in binary forms on blockchains, the first step for these studies are decompiling, using some decompilers, the smart contract binaries into functional equivalent source code, on which further program analysis can be performed. To this end, many contract decompilers (e.g., Porosity [18], Gigahorse [19], Erays [28], and Vandal [29]) have been developed, whose correctness and trustworthiness is indispensable for studies leveraging them.

Unfortunately, prior studies all have assumed that existing smart contract decompilers are trustworthy and reliable. However, whether such an assumption holds in practice is still unknown. To the best of our knowledge, there has not been a large-scale empirical investigation into smart contract decompilers including their effectiveness, efficiency, functional correctness, and resilience to common program transformations such as compiler optimizations and obfuscations [30] [31] [32]. We conjecture the reason for this situation may be the widespread misbelief that decompilation is a solved problem and thus existing decompiling techniques are already mature. However, there are still three key technical challenges remaining: first, decompilation, in theory, is an undecidable problem and thus difficult to solve [33]. Prior studies (e.g., x86 [34] or ARM [35] decompilation) have demonstrated that even mature decompilers like Ghidra [36], Objdump [37] [38] [39] may have serious deficiencies.

Second, smart contracts, as a novel programming paradigm, have special peculiarities that make decompilation challenging. For example, in contract binaries, statically-sized variables are laid out contiguously starting from address 0, and contiguous variables with sizes less than 23 B can be packed into a single 32 B storage slot [40]. As a result, even different orders of variable declarations generated by decompilers may lead to semantic discrepancies, making the decompilation results unreliable or even invalid.

* Corresponding authors.

Finally, code transformations, such as compiler optimization and obfuscation [41] [42] [43], might be used by smart contracts to improve efficiency or security. Although these transformations are effective, they, unfortunately, have negative impacts on decompilation. For example, the control flow flattening (`fla`), one important code obfuscation algorithm, creates a large number of fake control flows [44], which may defeat contract decompilers failing to identify such fake flows.

Therefore, in this study, we explore the following research questions that remain unanswered related to the decompilation of smart contracts. What are the success rates and effectiveness of these decompilers? What is the runtime cost of these decompilation tools? Are contract decompilers resilient to code transformations such as compiler optimizations and obfuscations? How do these decompilers behave on malicious smart contracts? Without answers to these research questions, decompiler tool builders might base their work on wrong assumptions and thus miss opportunities to improve these decompilers, smart contract developers cannot benefit from state-of-the-art, and security researchers and auditors might draw conclusions from potentially wrong source code inputs.

Our work. To bridge this gap, this paper presents the *first* and most *comprehensive* empirical study of smart contract decompilers. To this end, we first designed and implemented novel software tool prototype SOLINSIGHT to conduct this study. Second, we selected and created two datasets to perform the empirical study: a normal dataset containing 15,198 normal smart contracts, and a vulnerable dataset containing 350 buggy contracts. Third, we have selected five off-the-shelf widely used contract decompilers: Erays [28], Vandal [29], Gigahorse [45], Panoramix [46], and EthervmDec [47], then leveraged these state-of-the-art decompilers to produce source from contract binaries. Finally, we perform an empirical study in terms of success rates, failure factors, performance, effectiveness, completeness, and resilience.

We obtained important findings and insights from these empirical results, such as: 1) we investigated the success rates of these decompilers and proposed 2 failure factors and 3 root causes for failures; 2) we studied the performance of contract decompilers and revealed 2 root causes of inefficiency; 3) we proposed 3 key factors affecting the effectiveness of contract decompilers; 4) we studied the interface completeness of decompilers and presented a quantitative metric to measure it; and 5) we investigated the resilience of contract decompilers against program transformations such as compiler optimizations or malicious contracts and present quantitative results.

Our findings, tools, and suggestions have actionable implications for several audiences. Among others, they 1) help decompiler tool builders further improve decompilers, by increasing the success rates, speeding up decompilation, and enhancing completeness; 2) help contract developers to make more effective use of decompiler to perform code analysis or diagnose issues; and 3) help security researchers leverage state-of-the-art decompiler more effectively to conduct security studies such as reverse engineering and buggy code analysis.

Contributions. To the best of our knowledge, this is the first and most comprehensive empirical study of smart contract decompilers. To summarize, this work makes the following contributions:

- **Empirical study and tools.** We present the first empirical study of smart contract decompilation, with a novel software prototype SOLINSIGHT we created;
- **Findings, insights, and suggestions.** We present interesting findings and insights, as well as suggestions, based on the empirical results; and
- **Open source.** We make our implementations and empirical results available in the interest of open science, at `https://doi.org/10.5281/zenodo.7241605`.

Outline. The rest of this paper is organized as follows. Section II introduces the background and motivations for this work. Section III presents the approach we used to perform this study. Section IV presents the empirical results we obtained, and answers to the research questions based on these results. Section V and VI discuss the implications of this work, and threats to validity, respectively. Section VII discusses the related work and Section VIII concludes.

II. BACKGROUND AND CHALLENGES

To be self-contained, this section presents necessary background knowledge and motivations for this work.

A. The Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) [48] is a distributed virtual machine that executes Ethereum smart contracts. EVM follows the Harvard architecture model [49] by separating code and data into different address spaces. These address spaces serve as different purposes: the code address space, which is immutable, contains smart contract code, the storage space stores global states, and the memory address space stores temporary data.

EVM is a stack machine with a maximum depth of 1024 stack items, with each item a 256-bit word, which was designed to facilitate the use of 256-bit passwords. EVM executes stack machine instructions, accessing or manipulating the topmost 16 items from the top of the stack simultaneously. During execution, the EVM maintains a transient memory as a word-addressable byte array, which will not persist between transactions.

A smart contract must be executed in the Ethereum network by every miner and every full node in the network to compute and verify the state before and after a block. Ethereum features the so-called *gas* to limit the execution time per smart contract and reward miners for executing smart contracts. Every EVM instruction requires a certain gas budget to execute. As a result, minimizing the gas required for executing a contract is important as it minimizes the cost of contract execution.

B. EVM Bytecode and Binaries

EVM bytecode is a binary machine language that EVM can execute. Every instruction contains a one-byte opcode, followed by a zero or more operands. For example, the push instruction encodes, in the instruction bytes, the constants, to

be pushed onto the stack by EVM. The Ethereum bytecode is Turing complete, whose main opcodes can be classified according to their functionalities, into the following categories: stack operations, arithmetic/compare/bitwise operations, environmental operations, memory operations, etc. [50] [51] [52].

Smart contracts are normally developed using high-level languages such as Solidity [53], as programming in EVM bytecode directly is tedious and error-prone. Contract sources are then compiled by contract compilers such as SOLC [54], into EVM bytecode, which is further installed and executed on blockchains.

Technically, a smart contract binary consists of three components: 1) deployment code, 2) runtime code, and 3) auxiliary data. First, the deployment code deploys smart contracts on Ethereum, which will be executed when a new contract is first created. It is used to check whether the function can be paid: if the function is marked payable, a client can send Ether to the smart contract by calling the function.

Second, the runtime code is stored on-chain to describe a smart contract. The runtime code does not include the constructors or constructor parameters of a contract, as they are irrelevant to contract executions.

Finally, the auxiliary data is a hash value, which can be used to fetch the metadata of the deployed contracts. It is mainly used for automatic interface generation, as well as for source code verification.

C. Challenges for Smart Contract Decompilation

A smart contract decompiler takes as input an EVM bytecode binary, reconstructs and outputs functionally equivalent contract source code such as Solidity.

Although it may not look difficult from a conceptual point of view, reconstructing source code from binaries is indeed a nontrivial and challenging task. To decompile a binary smart contract, the following 4 steps are generally needed: 1) decoding; 2) disassembly; 3) IR construction; and 4) source generation. First, contract binary files are decoded into instruction streams, along with other necessary information. This step is tedious and error-prone due to the diversity of language versions and the incompatibility of binary file format as well as the complex internal binary data structures.

Second, assembly programs are disassembled from the instruction stream. This task is challenging, as on the von Neumann architecture, it is even impossible to distinguish program code from data [33].

Third, intermediate representations (IRs) are reconstructed from the assembly programs. These IRs, ranging from abstract syntax trees to control-flow graphs, are challenging to build due to the absence of a high-level control structure information [19] [55].

Finally, source programs are generated from these IRs. This step is challenging as it needs to synthesize necessary source information which is generally absent in binaries, such as variable names or types [56].

To address these challenges, existing approaches for smart contract decompilation can be classified into two categories: 1) declarative static program analysis and 2) constraint solving. On one hand, the declarative static program analysis [19] encodes the target problem in a declarative manner based on logical formulas, which are further processed by logical inference engines such as Datalog [57].

On the other hand, the constraint-solving approach first builds directed graph from the program code with basic blocks as nodes, then leverages constraint solvers (e.g., Z3 [58]) to solve the constraints of the control flow graph. The solutions from such solvers guarantee that the values of each variable satisfy all constraints.

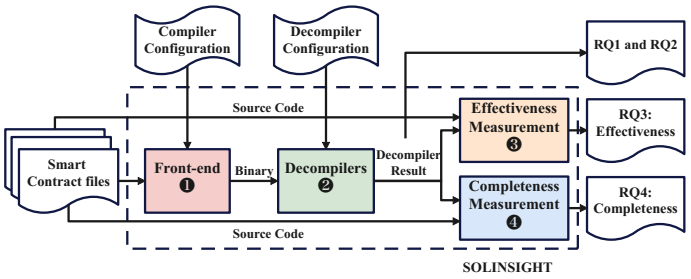


Fig. 1: SOLINSIGHT Architecture

III. APPROACH

This section presents our approach to conducting the empirical study. It is challenging to perform an empirical study of contract decompilation for a dataset with a large number of smart contracts, for two key reasons: 1) automation and 2) scalability. First, the study should be fully automated, otherwise it is difficult, if not impossible, to study large datasets with hundreds of thousands of smart contracts in a fully automatic manner; human analysis is only required to complement the analysis by manual code inspection. Second, this analysis should be scalable to analyze different decompilers, even potential ones in the future.

To this end, we have designed and implemented a software prototype SOLINSIGHT, with the goal to investigate research questions in an automated and scalable manner. We first present the architecture of SOLINSIGHT (Section III-A), then discuss the front-end module (Section III-B), the decompilation module (Section III-C), the effectiveness measurement module (Section III-D), and the completeness measurement module (Section III-E), respectively.

A. The Architecture

SOLINSIGHT is designed with one important principle of modularity and extensibility, so that it is straightforward to make modifications suitable for different needs, such as adding new contract datasets, experimenting with new smart contract compilers' optimizations, evaluating new contract decompilers, or studying new evaluation metrics.

Based on this design principle, we present, in Fig. 1, the architecture of SOLINSIGHT, consisting of four key modules. First, the front-end module (❶) takes as input smart contract

sources, compile them with respect to a user-supplied compiler configuration and outputs contract binaries.

Second, the decompilation module (❷) takes as input the generated binaries from the preceding module, decompiles them, and generates the contract sources according to a user-specified decompiler configuration. Decompilation results from this module are used to investigate the first and second research questions (**RQ1** and **RQ2**), that is, the decompilation success rates and performance.

Third, the effectiveness measurement module (❸) takes as input both an original smart contract source S and the decompiled source T for S, measures the effectiveness of decompilation by calculating the code similarity between S and T, and outputs a similarity score c. This score is further used to answer the third research question (**RQ3**), i.e., the decompilation quality and effectiveness.

Finally, the completeness measurement module (❹) takes as inputs both an original smart contract source S and the corresponding decompiled source T, and calculates a completeness score d for the function interfaces in them. This score is used to answer the fourth research question (**RQ4**).

In the following sections, we discuss the design and implementation of each module, respectively.

B. The Front-end

The front-end generates contract binaries by processing the input contract source files, follow these steps: 1) Code filtering: source code in the wild may contain duplicated or empty contracts, which are first filtered out before subsequent processing. 2) Source split: a single contract source file may contain multiple contracts which will be further compiled into multiple smart binaries. Thus, the front-end splits such source files into multiple files so that each file only contains one contract, which may simplify subsequent modules considerably. 3) Binary generation: the front-end compiles contract sources to corresponding binaries with official contract compilers such as `solc`, according to a user-supplied configuration. The configuration contains information controlling the compilation process, such as compiler options for optimization, and threshold to control compiling time.

Although combining the front-end with other modules is possible, the current design of SOLINSIGHT, from a software engineering perspective, has two key advantages: 1) it makes SOLINSIGHT feasible to process different types of contract sources and different compiler options; and 2) it is more effective by processing the peculiarities of the contract sources in an early stage, simplifying subsequent phases considerably.

C. Contract Binary Decompilation

The contract binary decompilation module decompiles contract binaries into corresponding sources, according to a user-supplied configuration. The configuration specifies configuration controlling the decompilers, such as decompiler options, decompiling timeout threshold, and output format.

The decompilation module leverages state-of-the-art decompilers. We have two key criteria for decompiler selection: 1) full automation; and 2) contract source generation. First, the selected decompiler should be fully automated, so that large datasets can be processed without human intervention. Second, the selected decompiler should generate explicit contract sources, as such sources are indispensable to investigating the quality and effectiveness of decompilers. To this end, 5 decompilers, as presented in TABLE I, have been selected and used in SOLINSIGHT. These decompilers, to the best of our knowledge, are comprehensive and represent the state-of-the-art of contract decompiling tools.

TABLE I: Decompilers leveraged by SOLINSIGHT

Name	License	Implementation Language	Open Source
Vandal	BSD-3-Clause	Python	✓
Erays	MIT	Python	✓
Panoramix	MIT	Python	✓
EthervmDec	NA	NA	✗
Gigahorse	BSD-3-Clause	Python	✓

Vandal [29] is a security analysis framework utilizing a logic-based approach for decompilation. Erays [28] is a reverse engineering tool generating pseudocode in three-address forms [59] suitable for manual analysis. Panoramix [46] is the official decompiler deployed on Etherscan.io [60]. Gigahorse [19] is a decompiler based on a logic-based (Datalog [57]) approach. EthervmDec [47] is a widely used online decompiler.

Although important and widely used decompilers were included in our study, some decompilers are omitted from TABLE I. Among them, Porosity [18], once the first decompiler on the Ethereum platform, is, unfortunately, no longer maintained and thus unusable. JEB [61], a commercial decompiler, is excluded due to our limited resources to explore proprietary software. Elipmoc [62], a relatively new decompiler, has no public sources available. However, the architecture of SOLINSIGHT (Fig. 1) is neutral to the specific decompilers selected and used. In addition, the modular design of SOLINSIGHT simplifies the incorporation of other decompilers as well.

D. Effectiveness Measurement

Technically, different contract decompilers might produce contract sources with different syntax but the same semantics and functionality. From the code auditing or reverse engineering perspective, the decompilation results, which are more similar to the original contract sources, are better and preferable, as they reflect the structures of the original sources more faithfully and thus make auditing results more accurate and reliable.

The effectiveness measurement module in SOLINSIGHT is designed to measure the effectiveness of decompilers by calculating the similarity between the contract source code and decompiled results. The similarity scores are further used to answer **RQ3** (Section IV-A). Decompilers with higher similarity scores are producing decompilation results with better quality.

To this end, the selection of appropriate effectiveness measurement algorithm is crucial in designing and implementing

this module. Although there have been a significant number of studies on similarity algorithms (e.g., fuzzy hashing [63], natural language processing [64] [27], and graph embeddings [65]), we have utilized Word2Vec [66], for two technical reasons. First, Word2Vec can produce more accurate results than fuzzy hashing [67], because it generates feature vectors in a context-sensitive manner. Second, Word2Vec is more efficient than graph embedding [68], as it makes use of feature vectors of low dimensions.

We deploy the Word2Vec [69] algorithm in SOLINSIGHT in three steps. 1) We build a custom Solidity parser for smart contracts. For contract source S_1 and the corresponding decompiler output source T_1, the parser extracts lexical, syntactical, and semantic information for tokens in the contracts S_1 and T_1. 2) We build a normalizer [66] reassembling the tokens to construct a fixed-dimension feature vector. As a result, the normalizer creates two feature vectors v_1 and v_2 for sources S_1 and T_1, respectively. 3) We calculate the similarity score between S_1 and T_1 with

$$Similarity(S_1, T_1) = 1 - \frac{Euclidean(v_1, v_2)}{||v_1|| + ||v_2||} \quad (1)$$

where $Euclidean(v_1, v_2)$ denotes the Euclidean distance measures the absolute distance between vectors v_1 and v_2, and $||v||$ represents the length of a vector v.

E. Completeness Measurement

A decompiler should identify the correct number of functions as well as the correct function interfaces, i.e., the argument types and corresponding return types. It should be noted that identifying function interface is different from recovering function source code, as function interfaces represent the contract application binary interfaces (ABIs) [53] facilitating function calls between contract binaries.

To this end, SOLINSIGHT incorporates a completeness measurement module to calculate a completeness score measuring to what degree function interfaces having been recovered by decompilers, by taking as inputs the generated sources from decompilers and corresponding original sources. The completeness scores are further used to answer **RQ4** (Section IV-A).

The calculation is performed in two steps. 1) We count the number of function interfaces x_1 for the contract source S_1. The number x_1 is also used as the ground truth. Similarly, we count the number of function interfaces x_2 for the decompiler-generated source T_1. 2) We utilize a mean value of ABIs

$$diff(S_1, T_1) = \frac{|x_1 - x_2|}{n} \quad (2)$$

to determine the difference between sources S_1 and T_1, where n is the number of all sources. The differential value $diff(S_1, T_1)$ represents the completeness of function interfaces, where smaller values indicate higher completeness (i.e., smaller differences).

F. Resilience Measurement

Prior studies (e.g., decompilation for C [56] [70], Java [71] or Android [72]) have demonstrated that malicious code may bring challenges and thus have negative impacts on decompilers [70] to reduce success rates because malicious code may leverage complex program transformations, such as obfuscation [32] [41] [44] or packers [73], which might defeat decompilers failing to deal with these transformations correctly. Meanwhile, even normal compiler optimizations may alter the target program in nontrivial ways, defeating decompilers.

Hence, we included, in SOLINSIGHT, a resilience measurement to evaluate to what degree SOLINSIGHT is resilient against malicious contracts or nontrivial program transformations such as compiler optimizations. The results from this measurement are used to answer **RQ5** and **RQ6** (Section IV-A).

Resilience measurement is performed by applying SOLINSIGHT to the normal dataset with compiler optimizations enabled, as well as a new malicious dataset we selected and created. On these two datasets, empirical studies are conducted to measure decompilation success rates, failure factors, performance, effectiveness, and so on.

IV. EMPIRICAL RESULTS

This section presents our empirical results by answering the research questions.

A. Research Questions

By presenting the empirical results, we mainly investigate the following research questions:

RQ1: Success rates and root cause analysis. What are the success rates of these decompilers? What are the root causes leading to decompilation failures?

RQ2: Performance. What is the average execution time of decompilers? Are they performant enough for practical usage?

RQ3: Effectiveness/Similarity. Are contract decompilers effective in generating high-quality sources? To what degree a decompiler can recover smart contract sources similar to original sources?

RQ4: Completeness. To what degree a decompiler can recover function (public and external) signatures for smart contracts?

RQ5: Compiler optimizations. Are contract decompilers resilient to contract compiler optimizations?

RQ6: Buggy contracts. Are contract decompilers effective in processing buggy contracts?

B. Evaluation Setup

All evaluations and measurements are performed on a server with one 20 physical Intel i7 core CPU and 64 GB RAM running Ubuntu 20.04.

TABLE II: Decompilation success rates, failures, failure factors for the 5 contract decompilers, on the normal dataset.

Decompilers	Result		Failure Factors	
	Decompilation Failures	Success Rates	#Timeout	#Exception
Erays	9835	63.61%	6	9829
Vandal	85	99.69%	33	52
Panoramix	483	98.21%	479	4
Gigahorse	63	99.76%	63	0
EthervmDec	1	99.99%	0	1

C. Decompilers and Datasets

We first describe decompilers and datasets created and used in this study, which has been released in our open source.

Decompilers. We used 5 decompilers, as TABLE I presented, in our study: Erays [28], Vandal [29], Panoramix [46], EthervmDec [47] and Gigahorse [19]. These contract decompilers, to the best of our knowledge, represent state-of-the-art smart contract decompiling tools.

Datasets. To conduct the empirical study, we created two datasets: a normal dataset and a malicious dataset. We created the normal dataset with the following steps: 1) to guarantee the validity of contracts, that is, the contract sources should compile correctly, we first collected all the 175,230 smart contracts on Etherscan [60] which have been validated; 2) to speed up decompilation and facilitate manual code inspection when necessary, following the guidelines of Krejcie and Morgan [74], we created a sample that is representative of the smart contracts with a confidence level of 99.00% and a confidence interval of 1.0. To this end, we selected a sample of 15,198 contracts, then round the sample size up to 16,000 for convenience. The 16,000 contracts were compiled to produce 27,207 binaries, as there may be multiple contracts in a single source file. We created the second, i.e., the buggy dataset, by utilizing a publicly available vulnerable dataset [75], which has also been widely used by prior studies [76]. This vulnerable dataset contains 350 unique vulnerable contracts with 9369 bugs from 7 different bug types, which generate 737 contracts binaries after compilation.

D. RQ1: Success Rates and Root Cause Analysis

To answer **RQ1** by investigating decompilation success rates of decompilers, we first applied SOLINSIGHT to the normal contract dataset. During the study, we set up the timeout threshold to 120 seconds, which is 100X higher than the average decompilation time [19] [62].

TABLE II presents the empirical results. The first column lists all decompilers. The next 2 columns present decompilation failures as well as success rates. The last 2 columns present the failure factors. We classify failure factors into two main categories: timeouts and exceptions, that is, failures = timeouts + exceptions.

The empirical results give interesting findings and insights. First, except for Erays (with a success rate of 63.61%), the other 4 decompilers have relatively high success rates (all beyond 98.00%). Second, although Panoramix and Gigahorse have a high success rates (98.21% and 99.76%, respectively), they trigger timeouts or exceptions in decompiling many contracts (479 and 46, respectively). Third, EthervmDec triggers the fewest failures (with 0 timeout and just 1 exception), and thus has the highest success rate (99.99%), demonstrating its high decompiling quality.

We then conduct an analysis of the impact of file sizes and Solidity versions on success rates, as presented in Fig. 2. 1) For file sizes, we classify all contracts by file sizes, in intervals of every 5K bytes. This evaluation reveals that file sizes do affect success rates obviously for 3 decompilers: Vandal, Panoramix, and Erays. Especially, the success rates for Erays decrease from 89.00% to 8.45%. 2) We evaluated 5 solidity versions from 0.4 to 0.8, which are major versions in use. Among all decompilers, only Erays shows a significant decrease in success rate from 93% to 38%. We then manually inspected the source code of Erays, and found that Erays failed to recognize many new opcodes (e.g., the 0x1b opcode (SHL)) introduced in the latest versions of Solidity by throwing many exceptions.

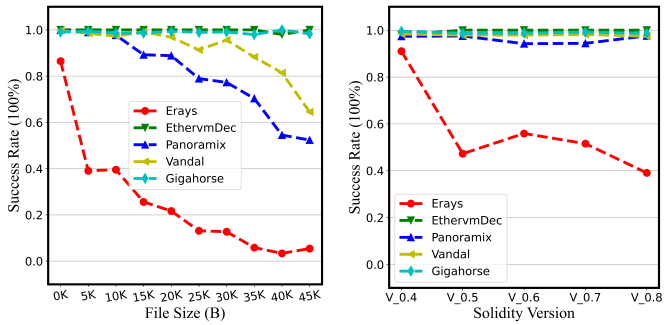
We then explore the root causes leading to decompilation failures, and identified 3 key reasons: 1) integer overflows; 2) incomplete instruction support; and 3) decompiler implementation defects. First, some decompilers do not handle integer overflows, which further triggers exceptions. For example, in the following sample code snippet, we have identified from Vandal [29], for large enough integer variable `b`, the expression `(cls.SIZE - b) * 8` will be negative, triggering decompiler exceptions.

```
1 # vandal/src/memtypes.py: line 389-392
2 @classmethod
3 def BYTE(cls, b: int, v: int) -> int:
4     """Return the b'th byte of v."""
5     return (v >> ((cls.SIZE - b) * 8)) & 0xFF
```

Second, some decompilers do not support the complete list of EVM opcodes, leading to many decompiling exceptions. For example, Erays [28], as we have observed, failed to recognize 14 EVM opcodes, which further lead to the 9829 exceptions in TABLE II. Our observation also explains the decrease in success rates in Fig. 2, as larger files or newer Solidity versions might have more new opcodes for the target decompiler (e.g., Erays) failed to recognize.

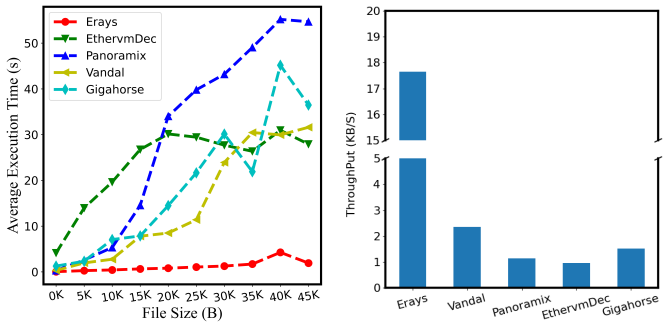
Third, some decompilers have implementation defects resulting in exceptions. For example, Vandal [29], as we have revealed, cannot handle null pointer correctly, leading to decompilation exceptions.

We then conduct a manual inspection of the root causes. To calculate the precision of the root causes, we formed a group with 3 graduate students who familiar with Solidity and decompilers to independently conduct a manual inspection of the source code of decompilers. Moreover, to ensure the reliability of the inspection results, we adopted the Cohen's Kappa statistic [77], which is frequently used to test inter-rater reliability. The inspection resulted in a Cohen's Kappa score



(a) Impact of File Sizes (b) Impact of Solidity Versions

Fig. 2: The impact of file sizes and Solidity versions on the decompiling success rates, respectively.



(a) Impact of file sizes (b) Throughput of decompiler

Fig. 3: The impact of file sizes on the decompiler performance, and throughputs (KB/s).

of 0.892, which indicates an "Almost Perfect" agreement. In the rare cases where the students disagreed, we conservatively judged these reports as false positives.

To this end, we explore the root causes for timeout and identify 2 key reasons: 1) exponentiation; and 2) inefficient keyword matching. First, for the exponentiation e^x, when x is large (typically, x is 256 bits for a password in EVM), we have observed that Python is slow to perform calculations further triggering timeouts. We speculate that this may be attributed to implementation defects in Python's mathematical library. Second, some decompilers (e.g., Panoramix [46]) utilized a sequential comparison algorithm for keyword matching, which is slow due to the inefficiency of strings (keywords) comparisons.

Summary: Except for Erays (63.61%), the other 4 contract decompilers: Vandal, Panoramix, Gigahorse, and EthervmDec, have high success rates (over 98%). Both file sizes and Solidity versions affect the success rates of the 3 decompilers considerably. We identified 3 root causes leading to decompiler exceptions: integer overflows, incomplete instruction support, and decompiler implementation defects. We have revealed 2 root causes for timeouts: exponentiation, and inefficient keyword matching.

E. RQ2: Performance

To answer **RQ2** by investigating the performance of decompilers, we applied SOLINSIGHT to the normal dataset (Section IV-C). Each contract is executed in 10 rounds to calculate an average running time. Fig. 3 presents the impact of file sizes on the decompiler performance and throughputs, respectively.

The empirical results give interesting findings and insights. First, except for Erays, the average execution time for the other 4 decompilers grow nearly linearly with file sizes. On average, these 4 decompilers can process 1KB of contract binaries per second. Second, Erays is the most efficient decompiler also with the highest throughput (17.64 KB per second), whereas EthervmDec is the least efficient decompiler with the lowest throughput (1.13 KB per second).

We then explored the root causes, based on a manual inspection of the decompilers' sources. This inspection revealed 3 key reasons. First, EthervmDec is an online decompiler that spent a lot time on network transfers, accounting for its low throughput. Second, while Erays might seem performant with the highest throughput, this result is inaccurate as it triggers many exceptions when processing large contracts (see TABLE II). Third, according to Fig. 3b, Panoramix's execution time grows rapidly when processing large files, as it spent more time on the processing function call parameters in large contracts.

Summary: Except for EthervmDec (a online decompiler), all other 4 decompilers are performant, and grow linearly with contract sizes. Except for Erays which triggers many exceptions, Vandal is the most efficient decompiler with the highest throughput.

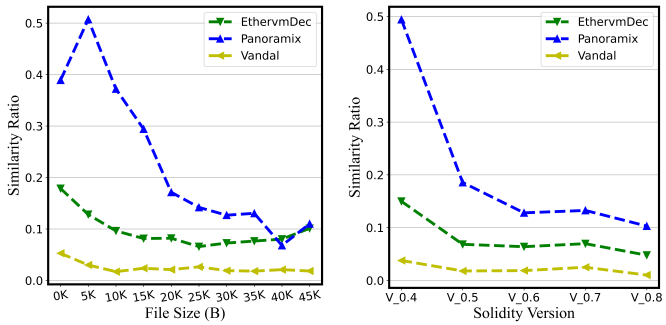
F. RQ3: Effectiveness/Similarity

To answer **RQ3** by investigating the effectiveness of decompilers, we applied SOLINSIGHT to the normal dataset to calculate the similarity scores between the sources generated by the decompilers with original sources.

We present, in Fig. 4, the similarity scores for three decompilers EtherevmDec, Panofamix, and Vandal, with respect to file size and Solidity version, respectively. The results for Erays and Gigahorse are omitted from this figure, as their similarity scores are nearly 0, due to their generation of three-address code instead of Solidity-like sources.

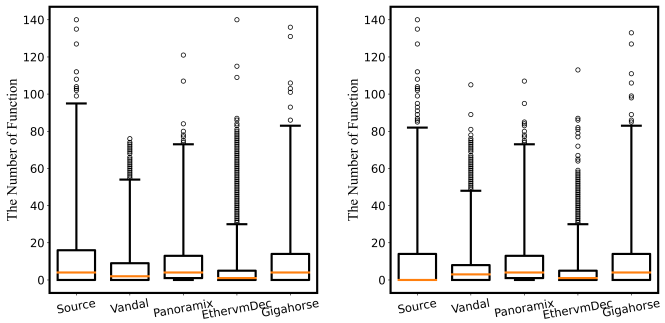
The empirical results give interesting findings and insights. First, Panoramix generates contract sources with the highest similarity scores (with a peak higher than 50%), whereas Vandal produces sources of the lowest scores (less than 5%). This result demonstrated that Panoramix, from a code similarity perspective, is of better quality.

Second, similarity scores decrease with respect to both file sizes and Solidity versions. To be specific, although Panoramix produces sources with higher scores, its scores decrease significantly, from 51% to 8% for file sizes, and from 50% to 10% for Solidity versions.



(a) Impact of file sizes (b) Impact of Solidity versions

Fig. 4: The impact of file sizes and Solidity versions on decompiler effectiveness/similarity, respectively.



(a) ABIs with optimizations off (b) ABIs with optimizations on

Fig. 5: The boxplots of ABI numbers for the normal dataset without and with compiler optimizations, respectively.

We then investigate the root causes for the similarity score decreases, based on manual code inspection. This investigation reveals 3 key reasons: first, larger files, as we observed, triggered more timeouts due to unrecognized opcodes (see also Fig. 2a), leading to similarity scores decrease. Second, we observed some decompilers (e.g., Panoramix) ignored all constructors during contract decompilation, resulting in low similarity scores. Third, some decompilers (e.g., Panoramix) make extensive use of stack slots instead of strings to represent the Solidity variable names. Such a lack of variable information leads to low similarities.

Summary: All state-of-the-art decompilers have low similarity scores normally less than 10% (with the peak less than 50%, for small files and old Solidity versions), indicating the low similarity of generated sources with original sources. Furthermore, similarity scores decrease with respect to file sizes and Solidity versions, leaving considerable space for future improvements of these tools.

G. RQ4: Completeness

To answer **RQ4** by investigating the function interface completeness, we applied SOLINSIGHT to the normal dataset to compute the completeness scores.

The boxplots in Fig. 5a present the distribution of ABIs numbers as well as maximums, minimums, and medians, for the original sources, as well as for decompilation results from 4 decompilers: Vandal, Panoramix, EthervmDec, and Gigahorse.

Empirical results indicate that Gigahorse is the most complete decompiler, whereas EhtervmDec is the least complete one. To further quantify this observation, we then make use of the $diff$ value (equation (2) in Section III), to determine which decompiler generates more complete results in term of function ABI numbers. TABLE III presents the results for this quantitative analysis, which gives interesting findings and insights.

First, Gigahorse is most complete by generating ABIs much closer to the original sources with the lowest $diff$ value of 1.6614, whereas EthervmDec generated the least complete ABIs with the highest $diff$ value of 8.9256. Recall that lower $diff$-values represent higher completeness [78] [79]. Second, all decompilers differ significantly from each other, which indicates the differences in their capabilities to recovering function ABIs.

TABLE III: Function interface completeness results for decompilers. Lower $diff$ values indicate better completeness results, that is, the generated results are closer to the ground truth.

Decompilers	Vandal	Panoramix	EthervmDec	Gigahorse
$diff$-value	4.6181	3.0736	8.9256	1.6614

To further explore the potential reasons, we perform a manual inspection of decompiler sources. This inspection revealed two key reasons. First, some decompilers, such as Vandal, only identified the entry point, end point, and the hash of signature but failed to identify function parameters, resulting in low completeness results. Second, some decompilers (e.g., Panoramix) failed to recognize function ABIs with more than 10 arguments, leading to decompilation failures and incompleteness.

Summary: Gigahorse has the smallest $diff$-value of 1.6614, indicating its relatively complete generation of functions ABIs. Overall, the $diff$-values of these decompilers vary widely, implying considerable improvement spaces in their capability to recognize function ABIs.

H. RQ5: Optimizations

To answer **RQ5** by investigating the impact of compiler optimizations on decompilers, we applied SOLINSIGHT to the normal dataset with compiler optimizations turned on or off.

TABLE IV presents empirical results of success rates and average execution time for the 5 decompilers, respectively. Each contract is executed for 10 rounds. First, the average execution time decreases, although insignificantly, for all 5 decompilers when compiler optimizations are turned on. Second, except for Panoramix (a decrease of 0.07%), success rates for the other 4 decompilers increased, the improvements are insignificant.

TABLE IV: Success rates and execution time for the normal dataset with compiler optimizations turned off or on.

Decompilers	Success Rates		Execution Time (s)	
	Opt. Off	Opt. On	Opt. Off	Opt. On
Erays	63.61%	63.92%	0.1180	0.1088
Vandal	99.69%	99.71%	3.0724	2.0272
Panoramix	98.21%	98.14%	5.9246	4.5212
Gigahorse	99.83%	99.86%	4.4486	3.8240
EthervmDec	99.99%	99.99%	10.7956	9.4011

Furthermore, as boxplots in Fig. 5b presented, the numbers of ABIs reduced with compiler optimizations turned on.

We further explored the potential root causes and identified 3 key reasons. First, smart contract compilers, with optimizations turned on, generate fewer opcodes in the binaries that decompilers failed to recognize, increasing success rates by triggering less decompiler exceptions. Second, some compiler optimizations (e.g., constant folding) optimize target programs by reducing the code sizes. Hence, decompiling the optimized binaries takes less time. Finally, compiler optimizations, such as function inline, reduce the number of functions by inline leaf functions.

Summary: Smart contract compiler optimizations increase success rates, reduce execution time, and decrease the number of ABIs recognized, due to the reductions of decompiling exceptions, binary code sizes, and numbers of functions. However, the differences in success rates, execution time, and numbers of ABIs with and without optimizations are insignificant.

I. RQ6: Buggy Contracts

To answer **RQ6** by investigating the resilience of decompilers against buggy contracts, we applied SOLINSIGHT to the buggy dataset consisting of 737 buggy binaries, with a timeout threshold 120 seconds (same as for **RQ1**).

TABLE V presents the empirical results of decompilation failures, success rates, and failure factors for the 5 decompilers. The empirical results give interesting findings and insights. First, except for Erays, success rates of the other 4 decompilers remain high for malicious contracts (beyond 96%). Among them, the success rates for Gigahorse and EthervmDec reach 100%, with all buggy contracts successfully decompiled. Second, compared with success rates for the normal dataset (TABLE II), the success rate differences are insignificant, which indicate that buggy contracts do not bring significant difficulties to decompilers. Finally, except for Erays, buggy contracts trigger fewer timeouts and exceptions (0 in most cases), than normal contracts do (TABLE II).

We further explored the root causes, which revealed 2 key reasons. First, we found that the average sizes of the buggy contracts (15.87 KB) are smaller than those of the normal contracts (30.27 KB). We speculate the smaller sizes are due to the fact that intentionally buggy functionalities take less space than normal ones. As a result, the decompilation of buggy contracts triggers fewer timeouts due to smaller code sizes. Second, we conduct a manual inspection of the source code of the buggy contracts but identified none of them were obfuscated or packed. Instead, these buggy contracts contain specific vulnerabilities, such as integer overflows, execution permission-related, or timestamp-related vulnerabilities. Fortunately, although these vulnerabilities bring serious security threats, they bring no challenges to decompilers.

TABLE V: Decompilation success rates, failures, and failure factors of the 5 decompilers, on the buggy dataset.

Decompilers	Results		Failure Factors	
	Decompilation Failures	Success Rates	#Timeouts	#Exceptions
Erays	359	51.28%	0	359
Vandal	1	99.86%	1	0
Panoramix	21	96.92%	21	0
Gigahorse	0	100.00%	0	0
EthervmDec	0	100.00%	0	0

Summary: Success rates of decompilers for buggy contracts increase, although insignificantly, compared to those for normal contracts because the relatively smaller code sizes of buggy contracts trigger fewer decompilation timeouts. Vulnerabilities in buggy contracts bring no challenges to decompilers.

V. IMPLICATIONS

This work presents the first and most comprehensive empirical studies of smart contract decompilers that have actionable implications for several audiences. This section discusses some implications of this work along with some important directions for future studies.

For decompiler builders. Smart decompilers are immature and still evolving rapidly. The results of this work provide decompiler builders with important insights into the state-of-the-art decompilers, in terms of success rates, performance, effectiveness, resilience, and so on. On the one hand, decompiler builders can leverage the insights proposed by this work to better improve the quality and reliability of current decompilers. For example, decompiler builders should pay special attention to the decompilation of function ABIs. On the other hand, decompiler builders might utilize the software prototype SOLINSIGHT we proposed as well as the datasets we created in this work to testify their improvements actually meet their expectations.

For smart contract developers. Smart contract developers make use of decompilers on a daily basis to decompile the binary for code analysis and bug diagnosis. The results and suggestions in this work can benefit smart contracts developers. On one hand, our empirical results demonstrated that some decompilers have high success rates, so contract developers should make use of these decompilers to aid in debugging or code auditing for binaries, or even can integrate them into the CI/DI pipelines. Furthermore, as Fig. 2 shows, developers should pay special attention to Solidity versions. On the other hand, our study results also demonstrated that ex-

isting decompilers might also trigger failures such as timeouts or exceptions, hence, developers might also need to deploy supplemental tools such as disassemblers to deal with such failures.

For security researchers. Contract decompilers are indispensable for security researchers to finish their tasks such as reverse engineering or vulnerability detection. The results of this work are important for security researchers. On the one hand, the results of this work demonstrated that existing decompilers are resilient against malicious contracts, so that security researchers might use these decompilers reliably. On the other hand, given the importance of decompilers in security studies, the results of this work do demonstrate the need to further improve the state of the art of decompilers, making the smart contract ecosystem healthier.

VI. Threats to Validity

As in any empirical study, there are threats to the validity of our work. We attempt to remove these threats where possible and mitigate the effect when removal is not possible.

Tools. In this work, we have used five decompilers to conduct this study. Although these decompilers are either official or most widely used and thus represent state-of-the-art, there may be other decompilers available (Section III). Furthermore, new decompilers might be developed in the future. Fortunately, the modular design of SOLINSIGHT makes it straightforward to testify to new decompilers. In the future, we plan to investigate other decompilers when they are available.

Datasets. In this work, we utilized a normal and a vulnerable dataset. As the normal dataset is randomly selected from Etherscan, and the vulnerable dataset is public and widely used, so the results are trustworthy. On the other hand, there are other datasets available. Fortunately, the architecture of SOLINSIGHT is neutral to any specific dataset used, so a new dataset can always be added without difficulties.

Errors in the Implementation. Most of our results are based on the SOLINSIGHT framework. Errors in the implementation could invalidate our findings. To mitigate this risk, we subjected all implementations to careful code reviews and tested them extensively.

Other Factors. In this work, we focus on optimization to study its impact on decompilers. On the other hand, there are other protection technologies such as obfuscation or packers. To the best of our knowledge, there have been no popular and widely used packers for smart contracts, but it is an interesting direction for future investigation.

VII. Related Work

There is a significant amount of research effort on decompilation. However, the work in this paper represents a novel contribution to this research field.

Decompilation Studies. Smart contract decompilation has been studied extensively. Grech et al. [19] proposed a decompiler from Ethereum virtual machine bytecode to program sources. Suiche et al. [18] proposed a decompiler from EVM bytecode into readable Solidity sources. Zhou et al. [28] proposed a reverse engineering tool for smart contracts. Albert et al. [80] proposed a tool translating EVM bytecode into a rule-based representation. Many open source decompilation tools [80] [81] [82] has also been proposed. However, they do not conduct large-scale empirical studies on contract decompilers.

Smart Contract Analysis. A lot of research efforts are devoted to analyzing the security of smart contracts. Vivar et al. [83] proposed ESAF, a framework for detecting the vulnerabilities of smart contracts. Yang et al. [84] designed the first cross-platform security analysis tool. Jiao et al. [85] provided formal specifications to analyze and verify smart contracts. Ashouri Mohammadreza et al. [86] proposed a tool based on the combination of dynamic taint tracking and concolic testing, upon which security analysis can be performed. Brent et al. [29] proposed the security analysis framework for Ethereum smart contracts, in which the low-level Ethereum virtual machine bytecode can be converted into a semantic logical relation. Then they [23] further proposed a security analysis that uses data disinfection to check the information flow. Yang et al. [87] formally prove the security and reliability of smart contracts. Permenev et al. [88] proposed the first automatic validator which can be used to prove the functional properties of Ethereum smart contracts. Annenkov et al. [89] provided a method for writing dependent programs in Coq, and for semi-automatic testing verification. Hu et al. [25] proposed a tool SCSGuard using the automatic extraction of bytecode from the smart contracts. Ashizawa et al. [27] proposed a static analysis tool for vulnerability detection based on machine learning. However, these studies as well as analysis tools and frameworks do not discuss the reliabilities of the decompilers they leveraged but just assumed these decompilers are correctly implemented and thus trustworthy. On the contrary, in this work, we conducted the first empirical study of the smart contract decompilers, which is orthogonal to prior studies and thus complemented them.

VIII. Conclusion

In this work, we presented the first and most comprehensive study of smart contract decompilers. By designing and implementing a software prototype SOLINSIGHT, we proposed root causes leading to decompiler failures. We also revealed reasons for hurting performance. We identified root causes affecting decompilation effectiveness. And we investigated the resilience of contract decompilers against malicious contracts. We provided suggestions to decompiler builders, contract developers, and security researchers. A consideration of them can promote a healthier ecosystem for smart contracts.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work is partially supported by the National Natural Science Foundation of China (No.62072427, No.12227901), the Project of Stable Support for Youth Team in Basic Research Field, CAS (No.YSBR-005), Academic Leaders Cultivation Program, USTC.

REFERENCES

[1] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 Int. Workshop Blockchain Oriented Softw. Eng. IWBOSE*. IEEE, pp. 2–8.

[2] D. G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," p. 34.

[3] G. Pîrlea, A. Kumar, and I. Sergey, "Practical smart contract sharding with ownership and commutativity analysis," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implement.* ACM, pp. 1327–1341.

[4] P. McCorry, S. F. Shahandashti, and F. Hao, "A smart contract for boardroom voting with maximum voter privacy," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, A. Kiayias, Ed. Springer International Publishing, vol. 10322, pp. 357–375.

[5] F. Schär, "Decentralized finance: On blockchain- and smart contract-based financial markets," vol. 103, no. 2.

[6] A. Khatoon, "A blockchain-based smart contract system for healthcare management," vol. 9, no. 1, p. 94.

[7] H. L. Pham, T. H. Tran, and Y. Nakashima, "A secure remote healthcare system for hospital using blockchain smart contract," in *2018 IEEE Globecom Workshop GC Wkshps*. IEEE, pp. 1–6.

[8] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," vol. 4, pp. 2292–2303.

[9] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F.-Y. Wang, "An overview of smart contract: Architecture, applications, and future trends," in *2018 IEEE Intell. Veh. Symp. IV*. IEEE, pp. 108–113.

[10] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," p. 18.

[11] E. Lai and W. Luo, "Static analysis of integer overflow of smart contracts in ethereum," in *Proc. 2020 4th Int. Conf. Cryptogr. Secur. Priv.* ACM, pp. 110–115.

[12] J.-W. Liao, T.-T. Tsai, C.-K. He, and C.-W. Tien, "Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing," in *2019 Sixth Int. Conf. Internet Things Syst. Manag. Secur. IOTSMS*. IEEE, pp. 458–465.

[13] N. Fatima Samreen and M. H. Alalfi, "Reentrancy vulnerability identification in ethereum smart contracts," in *2020 IEEE Int. Workshop Blockchain Oriented Softw. Eng. IWBOSE*. IEEE, pp. 22–29.

[14] E. Yalon. Solidity top 10 common issues. Security Boulevard. [Online]. Available: https://securityboulevard.com/2020/05/solidity-top-10-common-issues/

[15] P. Praitheeshan, L. Pan, X. Zheng, A. Jolfaei, and R. Doss, "Solguard: Preventing external call issues in smart contract-based multi-agent robotic systems," vol. 579, pp. 150–166.

[16] M. Demir, M. Alalfi, O. Turetken, and A. Ferworn, "Security smells in smart contracts," in *2019 IEEE 19th Int. Conf. Softw. Qual. Reliab. Secur. Companion QRS-C*. IEEE, pp. 442–449.

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

[18] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," p. 30.

[19] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *2019 IEEEACM 41st Int. Conf. Softw. Eng. ICSE*. IEEE, pp. 1176–1186.

[20] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse, "Smartinspect: solidity smart contract inspector," in *2018 Int. Workshop Blockchain Oriented Softw. Eng. IWBOSE*. IEEE, pp. 9–18.

[21] T. Chen, Z. Li, X. Luo, X. Wang, T. Wang, Z. He, K. Fang, Y. Zhang, H. Zhu, H. Li, Y. Cheng, and X.-s. Zhang, "Sigrec: Automatic recovery of function signatures in smart contracts," pp. 1–1.

[22] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "Smartshield: Automatic smart contract protection made easy," in *2020 IEEE 27th Int. Conf. Softw. Anal. Evol. Reengineering SANER*. IEEE, pp. 23–34.

[23] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implement.* ACM, pp. 454–469.

[24] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Safevm: a safety verifier for ethereum smart contracts," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal.* ACM, pp. 386–389.

[25] H. Hu, Q. Bai, and Y. Xu, "Scsguard: Deep scam detection for ethereum smart contracts," in *IEEE INFOCOM 2022 - IEEE Conf. Comput. Commun. Workshop INFOCOM WKSHPS*. IEEE, pp. 1–6.

[26] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt, "Sok: Decentralized finance (defi)."

[27] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts," in *Proc. 3rd ACM Int. Symp. Blockchain Secure Crit. Infrastruct.* ACM, pp. 47–59.

[28] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: Reverse engineering ethereum's opaque smart contracts," pp. 1371–1385. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/zhou

[29] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts." [Online]. Available: http://arxiv.org/abs/1809.03981

[30] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *2010 Int. Conf. Broadband Wirel. Comput. Commun. Appl.* IEEE, pp. 297–300.

[31] D. Low, "Protecting java code via code obfuscation," vol. 4, no. 3, pp. 21–23.

[32] M. Zhang, P. Zhang, X. Luo, and F. Xiao, "Source code obfuscation for smart contracts," in *2020 27th Asia-Pac. Softw. Eng. Conf. APSEC*. IEEE, pp. 513–514.

[33] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *Machine Learning and Knowledge Discovery in Databases*, ser. Lecture Notes in Computer Science, D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, Eds. Springer Berlin Heidelberg, vol. 6913, pp. 522–536.

[34] D. Andriesse and X. Chen, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," p. 19.

[35] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An empirical study on arm disassembly tools," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.* ACM, pp. 401–414.

[36] Ghidra. [Online]. Available: https://ghidra-sre.org/

[37] objdump(1): info from object files - linux man page. [Online]. Available: https://linux.die.net/man/1/objdump

[38] G. Chen, Z. Qi, S. Huang, K. Ni, Y. Zheng, W. Binder, and H. Guan, "A refined decompiler to generate c code with high readability: A refined c decompiler," vol. 43, no. 11, pp. 1337–1358.

[39] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík, and A. Meduna, "Design of a retargetable decompiler for a static platform-independent malware analysis," in *Information Security and Assurance*, ser. Communications in Computer and Information Science, T.-h. Kim, H. Adeli, R. J. Robles, and M. Balitanas, Eds. Springer Berlin Heidelberg, vol. 200, pp. 72–86.

[40] Miscellaneous — solidity 0.5.10 documentation. [Online]. Available: https://docs.soliditylang.org/en/v0.5.10/miscellaneous.html

[41] M. Ceccato, A. Capiluppi, P. Falcarin, and C. Boldyreff, "A large study on the effect of code obfuscation on the quality of java code," vol. 20, no. 6, pp. 1486–1524.

[42] T. Hou, H. Chen, and M. Tsai, "Three control flow obfuscation methods for java software," vol. 153, no. 2, p. 80.

[43] S. Qing, W. Zhi-yue, W. Wei-min, L. Jing-liang, and H. Zhi-wei, "Technique of source code obfuscation based on data flow and control flow tansformations," in *2012 7th Int. Conf. Comput. Sci. Educ. ICCSE*. IEEE, pp. 1093–1097.

[44] T. Laszlo and A. Kiss, "Obfuscating c++ programs via control flow flattening," p. 17.

[45] Bytecode decompilation - [object object]. [Online]. Available: https://library.dedaub.com/decompile

[46] palkeo. Panoramix. [Online]. Available: https://github.com/palkeo/panoramix

[47] Online solidity decompiler. [Online]. Available: https://ethervm.io/decompile

[48] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform." p. 36.

[49] B. Li and W.-f. Li, "Modeling and simulation of container terminal logistics systems using harvard architecture and agent-based computing," in *Proc. 2010 Winter Simul. Conf.* IEEE, pp. 3396–3410.

[50] S. Bistarelli, G. Mazzante, M. Micheletti, L. Mostarda, and F. Tiezzi, "Analysis of ethereum smart contracts and opcodes," in *Advanced Information Networking and Applications*, ser. Advances in Intelligent

Systems and Computing, L. Barolli, M. Takizawa, F. Xhafa, and T. Enokido, Eds. Springer International Publishing, vol. 926, pp. 546–558.
[51] A. Aldweesh, M. Alharby, and A. van Moorsel, "Performance benchmarking for ethereum opcodes," in *2018 IEEEACS 15th Int. Conf. Comput. Syst. Appl. AICCSA*. IEEE, pp. 1–2.
[52] R. Yang, T. Murray, P. Rimba, and U. Parampalli, "Empirically analyzing ethereum's gas mechanism," in *2019 IEEE Eur. Symp. Secur. Priv. Workshop EuroSPW*. IEEE, pp. 310–319.
[53] Solidity — solidity 0.8.9 documentation. [Online]. Available: https://docs.soliditylang.org/en/v0.8.9/
[54] A. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments*, ser. Lecture Notes in Computer Science, S. Chakraborty and J. A. Navas, Eds. Springer International Publishing, vol. 12031, pp. 161–179.
[55] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *2018 IEEE 25th Int. Conf. Softw. Anal. Evol. Reengineering SANER*. IEEE, pp. 346–356.
[56] A. Fokin, K. Troshina, and A. Chernov, "Reconstruction of class hierarchies for decompilation of c++ programs," in *2010 14th Eur. Conf. Softw. Maint. Reengineering*. IEEE, pp. 240–243.
[57] Datalog. Wikipedia. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Datalog&oldid=1104851396
[58] The z3 theorem prover. The Z3 Theorem Prover. [Online]. Available: https://github.com/Z3Prover/z3
[59] S. Pop, P. Clauss, A. Cohen, G.-A. Silber, and V. Loechner, "Fast recognition of scalar evolutions on three-address ssa code," p. 29.
[60] Ethereum (eth) blockchain explorer. [Online]. Available: https://etherscan.io/
[61] Jeb decompiler for ethereum - jeb decompiler by pnf software. [Online]. Available: https://www.pnfsoftware.com/jeb/evm
[62] N. Grech, S. Lagouvardos, I. Tsatiris, and Y. Smaragdakis, "Elipmoc: advanced decompilation of ethereum smart contracts," vol. 6, pp. 1–27.
[63] A. Lee and T. Atkison, "A comparison of fuzzy hashes: Evaluation, guidelines, and future suggestions," in *Proc. SouthEast Conf.* ACM, pp. 18–25.
[64] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symp. Secur. Priv. SP*. IEEE, pp. 472–489.
[65] M. Xu, "Understanding graph embedding methods and their applications," vol. 63, no. 4, pp. 825–853.
[66] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding," in *2019 IEEE Int. Conf. Softw. Maint. Evol. ICSME*. IEEE, pp. 394–397.
[67] H. Zhan, W. Zhou, X. Hu, Q. Cai, T. Zhang, and L. Yang, "A recommendation algorithm based on fuzzy clustering," in *2018 Int. Conf. Mach. Learn. Cybern. ICMLC*. IEEE, pp. 230–233.
[68] H. Cai, V. W. Zheng, and K. C.-C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," vol. 30, no. 9, pp. 1616–1637.
[69] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," p. 9.
[70] Z. Liu and S. Wang, "How far we have come: testing decompilation correctness of c decompilers," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.* ACM, pp. 475–487.
[71] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "Java decompiler diversity and its application to meta-decompilation," vol. 168, p. 110645.
[72] A. Desnos and G. Gueguen, "Android: From reversing to decompilation," p. 24.
[73] L. Xue, H. Zhou, X. Luo, Y. Zhou, Y. Shi, G. Gu, F. Zhang, and M. H. Au, "Happer: Unpacking android apps via a hardware-assisted approach," in *2021 IEEE Symp. Secur. Priv. SP*. IEEE, pp. 1641–1658.
[74] F. Kibuacha. How to determine sample size for a research study. GeoPoll. [Online]. Available: https://www.geopoll.com/blog/sample-size-research/
[75] Solidifi benchmark. [Online]. Available: https://github.com/smartbugs/SolidiFI-benchmark
[76] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection."
[77] M. L. McHugh, "Interrater reliability: the kappa statistic," vol. 22, no. 3, pp. 276–282.
[78] Statistics - t-distribution table. [Online]. Available: https://www.tutorialspoint.com/statistics/t_distribution_table.htm
[79] Paired sample t-test. Statistics Solutions. [Online]. Available: https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/paired-sample-t-test/
[80] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *Automated Technology for Verification and Analysis*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds. Springer International Publishing, vol. 11138, pp. 513–520.
[81] "Mythril," ConsenSys. [Online]. Available: https://github.com/ConsenSys/mythril
[82] "ethersplay," Crytic. [Online]. Available: https://github.com/crytic/ethersplay
[83] A. López Vivar, A. L. Sandoval Orozco, and L. J. García Villalba, "A security framework for ethereum smart contracts," vol. 172, pp. 119–129.
[84] Z. Yang, H. Liu, Y. Li, H. Zheng, L. Wang, and B. Chen, "Seraph: enabling cross-platform security analysis for evm and wasm smart contracts," in *Proc. ACMIEEE 42nd Int. Conf. Softw. Eng. Companion Proc.* ACM, pp. 21–24.
[85] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of solidity," in *2020 IEEE Symp. Secur. Priv. SP*. IEEE, pp. 1695–1712.
[86] M. Ashouri, "Etherolic: a practical security analyzer for smart contracts," in *Proc. 35th Annu. ACM Symp. Appl. Comput.* ACM, pp. 353–356.
[87] Z. Yang, H. Lei, and W. Qian, "A hybrid formal verification system in coq for ensuring the reliability and security of ethereum-based service smart contracts," vol. 8, pp. 21 411–21 436.
[88] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symp. Secur. Priv. SP*. IEEE, pp. 1661–1677.
[89] D. Annenkov, M. Milo, J. B. Nielsen, and B. Spitters, "Source code for paper: Extracting smart contracts tested and verified in coq."