A Study of Inline Assembly in Solidity Smart Contracts

STEFANOS CHALIASOS, Imperial College London, United Kingdom
ARTHUR GERVAIS, Imperial College London, United Kingdom
BENJAMIN LIVSHITS, Imperial College London, United Kingdom

The Solidity programming language is the most widely used language for smart contract development. Improving smart contracts' correctness, security, and performance has been the driving force for research in vulnerability detection, program analysis, and compiler techniques for Solidity. Similar to system-level languages such as C, Solidity enables the embedding of low-level code in programs, in the form of inline assembly code. Developers use inline assembly for low-level optimizations, extending the Solidity language through libraries, and using blockchain-specific opcodes only available through inline assembly. Nevertheless, inline assembly fragments are not well understood by an average developer and can introduce security threats as well as affect the optimizations that can be applied to programs by the compiler; it also significantly limits the effectiveness of source code static analyzers that operate on the Solidity level. A better understanding of how inline assembly is used in practice could in turn increase the performance, security, and support for inline assembly in Solidity.

This paper presents a large-scale quantitative study of the use of inline assembly in 6.8M smart contracts deployed on the Ethereum blockchain. We find that 23% of the analyzed smart contracts contain inline assembly code, and that the use of inline assembly has become more widespread over time. We further performed a manual qualitative analysis for identifying usage patterns of inline assembly in Solidity smart contracts. Our findings are intended to help practitioners understand when they should use inline assembly and guide developers of Solidity tools in prioritizing which parts of inline assembly to implement first. Finally, the insights of this study could be used to enhance the Solidity language, improve the Solidity compiler, and to open up new research directions by driving future researchers to build appropriate methods and techniques for replacing inline assembly in Solidity programs when there is no real necessity to use it.

CCS Concepts: • **General and reference** → **Empirical studies**; • **Software and its engineering** → **Assembly languages**; **Language features**.

Additional Key Words and Phrases: Solidity, Smart Contracts, Inline Assembly, Empirical Studies

ACM Reference Format:
Stefanos Chaliasos, Arthur Gervais, and Benjamin Livshits. 2022. A Study of Inline Assembly in Solidity Smart Contracts. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 165 (October 2022), 27 pages. https://doi.org/10.1145/3563328

1 INTRODUCTION

Many programming languages provide inline assembly capabilities, i.e., assembly instructions embedded in a higher-level language. For example, in C, one can use `x86-64` inline assembly to access low-level assembly instructions (e.g., `rdtsc` instruction to read a timer). The presence of inline assembly in a language enables low-level programming, extending the language's functionality, and gives developers fine-grained control for performing specialized optimizations. Nevertheless, inline

Authors' addresses: Stefanos Chaliasos, Imperial College London, United Kingdom, s.chaliasos21@imperial.ac.uk; Arthur Gervais, Imperial College London, United Kingdom, arthur@gervais.cc; Benjamin Livshits, Imperial College London, United Kingdom, b.livshits@imperial.ac.uk.

2475-1421/2022/10-ART165
https://doi.org/10.1145/3563328

assembly thwarts the effectiveness of static analyzers and compilers (e.g., it reduces optimization opportunities). Rigger et al. [2018] investigated the impact and usage of inline assembly in C projects and discovered that, inline assembly is utilized by a sizable number of projects (15.6%) and, as such, tools should add support for it.

In Solidity, the primary programming language for implementing smart contracts for the Ethereum blockchain, one can interleave Solidity statements with inline assembly in a language close to the bytecode syntax of the Ethereum virtual machine. The primary motivation for enabling inline assembly statements in Solidity, is to provide developers more fine-grained control, which is especially useful when trying to enhance the language by writing libraries and performing specialized optimizations. The latter is especially critical in EVM-based blockchains (e.g., Ethereum), because most applications try to minimize gas costs (gas refers to the cost necessary to run instructions in EVM-based blockchains).

However, because smart contracts handle billions of USD, the risks of using inline assembly in Solidity can be daunting. Similar to other languages, inline assembly is inherently low-level in Solidity, and as such, it bypasses several essential safety features and checks provided by the language and the compiler. When using inline assembly, it is the developer's responsibility to ensure that the memory model is satisfied. If any assumption of the memory model is violated, it can lead to incorrect or undefined behavior that cannot easily be discovered by testing. Additionally, the compiler cannot generally perform optimizations when employing inline assembly because it cannot reason about the memory. Surprisingly, static analysis tools that operate on the source code of Solidity smart contracts (e.g., slither [Feist et al. 2019], Securify2 [Tsankov et al. 2018], and SmartCheck [Tikhomirov et al. 2018]), and not on the EVM-bytecode [Brent et al. 2020, 2018; Grech et al. 2018], only partially support inline assembly fragments (they typically just print a warning), which can lead to imprecise analyses.

In this work, we perform the first quantitative and qualitative study of the usage of inline assembly in Solidity smart contracts. Specifically, we aim to investigate how widespread the use of inline assembly in smart contracts is, why developers employ it, and obtain insights that would help (1) practitioners to understand when they should use inline assembly, (2) researchers and tool writers to design and implement approaches that would be able to analyze inline assembly, and (3) language designers and compiler developers to enhance Solidity and its compiler support of inline assembly. To this end, our study seeks answers to the following research questions.

RQ1 (Measuring Inline Assembly) How common is inline assembly in Solidity programs? How extensively do contracts that contain inline assembly use it? (Section 4.1)

RQ2 (Smart Contract Characteristics) What are the characteristics of contracts that use inline assembly? Do they differ from other contracts? (Section 4.2)

RQ3 (Evolution of Inline Assembly) How does the usage of inline assembly evolve through time? What is the percentage of contracts using inline assembly per compiler version? (Section 4.3)

RQ4 (A Taxonomy of Inline Assembly) What are the most frequently-used instructions? Which instructions are combined in frequently repeating fragments? (Section 4.4)

RQ5 (Usage of Inline Assembly) Why do developers use inline assembly in practice? What kind of functionalities do they employ inline assembly for? (Section 5)

To answer these questions, we compose a dataset of Solidity smart contracts deployed in the Ethereum blockchain with at least one transaction or token transfer recorded. We also gather additional metadata for each address containing a contract to perform holistic analyses. Our corpus comprises 12.4M contracts, of which 159.2K are unique contracts (i.e., there might be multiple deployments of the same contract). Starting with a large dataset, we then performed quantitative

analyses to answer our first four research questions. Then, we selected 170 unique inline assembly fragments from five different clusters, and we manually studied them to answer our last research question.

Contributions. Our work makes the following contributions:

- We present a method for collecting and assessing inline-assembly code fragments from Solidity smart contracts deployed in Ethereum. We also make available, to the best of our knowledge, the largest dataset of Solidity smart contracts containing 12.4M contracts.
- We provide a thorough study of the usage of inline assembly in real-world Solidity smart contracts. Our analysis measures inline assembly in smart contracts, studies the characteristics of contracts that use inline assembly, studies the evolution of inline assembly through time, and provides a taxonomy of inline assembly usage.
- By manually examining 170 inline assembly fragments, we introduce ten classes of inline assembly usage patterns in smart contracts.
- We enumerate the implications of our findings and discuss potential future directions for replacing inline assembly in smart contracts, as well as improving inline assembly support in source code analysis tools.

Summary of finding. Selected, representative findings of our study are: (1) 23% of smart contracts in our corpus use inline assembly, (2) contracts using inline assembly typically only include two fragments (2.08 on average) and 10.62 instructions, (3) contracts that contain assembly are larger than those that do not (161 vs. 40 lines of code) and have more transactions, (4) the percentage of contracts using inline assembly increases through time, (5) 48 out of the 85 available instructions are used by less than 1% of the contracts that employ inline assembly, (6) an inline assembly fragment typically uses instructions from at least three different instruction categories (e.g., arithmetic operations and system operations), (7) inline assembly is mainly used for implementing functionality not available in Solidity as well as for gas optimization using specific code patterns.

Availability The research artifact is available at https://github.com/StefanosChaliasos/solidity-inline-assembly [Chaliasos et al. 2022].

2 BACKGROUND

This section provides a short introduction to Ethereum, Ethereum Virtual Machine (EVM), and Solidity, the foremost programming language for writing smart contracts; lastly, we discuss inline assembly in Solidity smart contracts.

2.1 Ethereum, EVM and Solidity

A blockchain implements a distributed ledger that records transactions between parties in a verifiable way into blocks. New transactions are processed by connected nodes that add those transactions in blocks, and then cooperate in appending blocks into the blockchain through a consensus protocol [Nakamoto 2008]. Ethereum [Wood 2022] is the most prominent blockchain with smart contract capabilities [Szabo 1997]. In Ethereum, beyond accounts that are mere balances, there are specialized accounts containing balances, volatile and non-volatile storage, and code that can perform arbitrary computations. Ethereum smart contracts are written in EVM, a stack-based bytecode language, and are executed in Ethereum distributed virtual machine. Smart contracts are initialized and executed through transactions and remain immutable once deployed.

EVM is a quasi Turing-complete, stack-based, low-level intermediate representation (IR). EVM defines several opcodes, which perform standard stack operations like `XOR`, `AND`, `ADD`, and `SUB`. Additionally, the EVM also implements a number of blockchain-specific stack operations, such as `ADDRESS`, `BALANCE`, and `BLOCKHASH`. Ethereum applies an execution cost (gas) per opcode to prevent

adversarial actors from spamming its network. Practitioners usually prefer to write smart contracts in high-level languages, which are then compiled into the targeted low-level VM bytecode. The most prominent contract-oriented programming language for Ethereum is Solidity, a high-level language whose definition was influenced by Object-Oriented (OO) languages like Python, C++, and primarily JavaScript. Solidity is statically typed, supports inheritance, libraries, and complex user-defined types, among other features. Using Solidity, it is possible to write contracts for implementing Tokens (e.g., `ERC-20`), Decentralized Applications (dapp) such as Decentralized Exchanges (DEX), multi-signature wallets, and more.

2.2 Inline Assembly in Solidity

Inline assembly (typically introduced by a language-specific keyword, e.g., asm for C and C++) gives the ability to embed assembly language source code within a program written in a high-level language. Rigger et al. [2018] investigates the use of x86-64 inline assembly in C projects from GitHub and found that 15.6% of the analyzed projects contained inline assembly code. The majority of assembly fragments are mainly concerned with multicore semantics, performance optimization, and hardware control.

In Solidity, developers can embed inline assembly in a language close to the one of EVM called YUL. [1] YUL is an intermediate language that can be compiled to bytecode for different backends (e.g., EVM). The goal of YUL is to be readable while being a low-level language. The main difference with other assembly dialects is that it provides high-level control flow constructs, i.e., `if`, `switch`, and `for`. Hence, YUL does not support direct manipulation of the stack through `DUP`, `SWAP`, and `JUMP` instructions making it less error-prone. Note that other opcodes can be called from within YUL blocks as normal functions (e.g., `add(x, y)`). In the following, we will use the term *instruction* to refer to opcode calls and other YUL constructs, such as the switch statement. Furthermore, YUL is suitable for whole-program optimizations, and has been used by the Solidity compiler to perform more advanced optimizations in its latest releases. [2]

According to Solidity's documentation [Solidity 2022], inline assembly should be used when more fine-grained control is needed to implement libraries or perform operations not available on Solidity. Another use case might be for optimizations purposes when Solidity's optimizer fails to produce efficient code. Nevertheless, inline assembly is a way to access the Ethereum Virtual Machine at a low level. Therefore, it bypasses several essential safety features and checks of Solidity (e.g., memory safety) and should be used only when required.

```
1  contract Example {
2    function add(uint a, uint b) public
         view returns (uint) {
3      assembly {
4        let result := add(a, b)
5        mstore(0x0, result)
6        return(0x0, 32)
7      }
8    }
9  }
```

Fig. 1. A simple example of inline assembly in Solidity.

An inline assembly block is marked by `assembly { ... }`, where the code inside the curly braces is written in the YUL language. Inline assembly code can access local Solidity variables, while different inline assembly blocks share no namespace, i.e., it is not possible to call a YUL function or access a YUL variable defined in a different inline assembly block. Figure 1 presents a straightforward example where two numbers are passed in a function, then added together using inline assembly, and finally, the function returns the result. The program first calculates the sum of `a` and `b` using the `add` opcode, and saves it in a new variable called `result` (line 4). Then, in line 5, it stores the result in memory at

[1] https://docs.soliditylang.org/en/latest/yul.html
[2] https://docs.soliditylang.org/en/latest/internals/optimizer.html

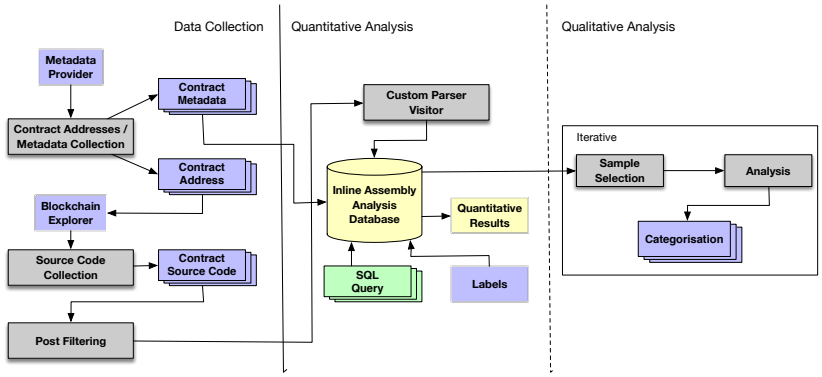

Fig. 2. The overview of our dataset creation and inline assembly analysis approach.

the address `0x0`. Finally, it returns 32 bytes from memory address `0x0`, which includes the value of the result variable. Although this is a very simple use of inline assembly, it is still more efficient than the respective Solidity code. In Section 5, we discuss more advanced use cases of inline assembly.

3 METHODOLOGY

First, we create a corpus of Solidity smart contracts deployed on Ethereum for which we can retrieve their source code (Section 3.1). Then, we present the implementation details of the quantitative analysis (Section 3.2), and we proceed with the methodology we used to access inline assembly usage qualitatively (Section 3.3). Note that the data collection and the quantitative analysis have been implemented as an automated pipeline, while we perform the qualitative analysis by manually examining inline assembly code fragments. Finally, we discuss the limitations and threats to the validity of our method (Section 3.4).

Our data collection and analysis approach is summarized in Figure 2. As a starting point, we use a blockchain metadata provider to collect Ethereum contract addresses. Then, we use a blockchain explorer (e.g., Etherscan [3]) to get the source code of smart contracts. In the next step (post-filtering), we perform the necessary data cleaning, where we process the results of the previous step and identify potentially duplicate smart contracts. The final outcome of this approach is a corpus consisting of a set of Ethereum smart contract addresses, metadata about those addresses (e.g., number of transactions), and the source code of contracts when available.

The resulting dataset is used as the input to our quantitative analysis (Section 3.2). To perform the quantitative analysis, we implemented a custom parser that identifies and analyzes inline assembly fragments and saves the results to a database (using SQlite3) of inline assembly fragments. The database contains information about each address, file, contract, inline assembly fragment, and assembly instruction analyzed. We then use this database to perform aggregate queries, for example, to determine how common is inline assembly in solidity contracts deployed on Ethereum. Note that we also have automated the quantitative analysis of our approach, and it is as simple as writing a SQL query to address potential new research questions.

The final step of our methodology is composed of the qualitative analysis. This analysis aims to understand why developers adopt inline assembly in practice. First, we filter smart contracts containing inline assembly using the database from the previous step to cluster smart contracts based on their characteristics (e.g., number of transactions and most duplicated contracts). Then we

[3] https://etherscanio.io

qualitatively analyze the inline assembly usage in iterations. Specifically, we first select a random sample of inline assembly fragments from all clusters, and then we manually study those fragments to identify use cases of inline assembly.

3.1 Data Collection

Our data collection approach consists of three steps, namely, *contract addresses and metadata collection*, *source code collection*, and *post-filtering*. To begin, we query Google Big Query (similar to Durieux et al. [2020]) to get the addresses of contracts deployed on Ethereum and retrieve specific metadata. Our study excludes contracts that have never been used, i.e., they have not any transaction or token transfer. Furthermore, for each contract address, we gather the following metadata: "transactions count", "unique callers", "token transfers count", "balance", "is erc20", "is erc721", and "block number".

After accumulating all the contract addresses, we proceed to the *source code collection* step. In this step, we use Etherscan and its API to retrieve the source code associated with an address. Specifically, Etherscan provides a source code verification feature, allowing users to upload the source code of a deployed smart contract, and then Etherscan compiles it and matches it with the code deployed on the blockchain. Note, however, that Etherscan does not have the source code for every contract. Therefore, at the end of this step, we obtained the Solidity files for each smart contract that has its source code available and verified in the Etherscan platform.

In the *post-filtering* step, having kept contract addresses and their source code when available, we proceed to (1) parse the output of Etherscan and save each Solidity contract in a single file in the filesystem, [4] and (2) detect possible duplicates. Although for answering most quantitative research questions, we are not going to filter out duplicate smart contracts, in the qualitative analysis, we want to filter them out to select unique code fragments more easily. To detect duplicates, we first remove all spaces and tabulations from contracts, and then compare their `SHA256` checksums. Finally, to reduce the processing time of the next steps of our approach, we filter projects using grep to detect if they contain the assembly keyword. Note that this approach would introduce false positives as the assembly keyword may exist only in comments. Nevertheless, the parser we implement (c.f. Section 3.2) can handle such false positives.

Figure 3 shows descriptive statistics of our data collection effort. We applied our approach from block 47,205 (7th of August, 2015) to block 14,339,876 (7th of March, 2022), a span of six years and four months. During this period, 49M smart contracts have been deployed, and 12.4M have at least one transaction or token transfer (25%). Furthermore, of these 12.4M, the $6.8M$ have verified source code uploaded in Etherscan, compromising of $722.6M$ lines of code. Finally, only 2% of the contracts are unique, meaning that $6.6M$ are duplicates.

3.2 Quantitative Analysis

The quantitative analysis phase of our approach is composed of three sub-steps: (1) analyzing Solidity smart contracts using a customized parser, (2) feeding the results to a database, and (3) executing queries to get quantitative results about inline assembly usage. First, we developed a customized parser performing a lightweight static analysis that identifies and analyzes inline assembly fragments. Our parser is implemented in Python (1.2K LOC), and is built on top of the `solidity_parser` [5] Python module, which in turn utilizes ANTRL4 [Parr and Quong 1995] to provide an abstract parser visitor for Solidity smart contracts. The customized parser processes

[4] Etherscan returns the result into JSONs. If a smart contract is compiled using multiple source files, Etherscan adds everything to a single JSON, and thus requires special processing.

[5] https://github.com/ConsenSys/python-solidity-parser

Start Block	47, 205 (7th of August, 2015)
End Block	14, 339, 876 (7th of March, 2022)
Total Contracts	49M
Contracts with at least one transaction	5.7M (12%)
Contracts with at least one token transfer	8.7M (18%)
Contracts without a transaction or token transfer	36.6M (75%)
Contracts with at least one transaction or token transfer	12.4M (25%)
Solidity source available	6.8M (55%)
Solidity source not available	5.6M (45%)
LOC	722.6M
Unique Solidity Contracts	159.2K (2%)
LOC	55.8M

Fig. 3. Statistics on the collection of Solidity smart contracts from the Ethereum blockhain.

inline assembly fragments to detect what opcodes and other YUL constructs each contract uses (i.e., control flow constructs). The output of our tool is a JSON containing details about smart contracts and inline assembly fragments.

Having processed all smart contracts for which we have obtained their source code and may contain inline assembly, the next step in our approach is to feed the data to a database. The database schema consists of the following tables: "NonAssemblyContractAddress", "AssemblyContractAddress", "Label", "AddressLabel", "SolidityFile", "Contract" (here, we refer to contracts as we would refer to classes in Java), "InlineAssemblyFragment", "Instruction", and "InstructionsPerFragment". [6] To populate the database, we use three data sources: the output of our parser, the metadata of contract addresses, and some external labels about Ethereum addresses.

The final step of the quantitative analysis is to answer all research questions of our study that can be answered quantitatively. To do so, we craft SQL queries that use our database and retrieve the data we need. The outcome of the quantitative analysis part is double-fold: first, we produce a database with thorough information about inline assembly in Solidity smart contracts, and secondly, we answer the quantitative research questions.

3.3 Qualitative Analysis

The smart contracts (deployed on the Ethereum blockchain) with at least one inline assembly fragment are 1.5M containing 5388 unique inline assembly fragments. Since we needed to analyze inline assembly fragments manually, it was not feasible to study every contract and fragment in the population. Therefore, we first cluster contracts based on their metadata to get samples from a broad spectrum. This way, we do not select contracts with only specific characteristics (e.g., a high number of transactions). Specifically, we carefully examined 170 unique inline assembly fragments to understand better why developers adopt inline assembly in practice. First, we selected the 70 most-deployed fragments, which account for 90% of all assembly usage in our corpus. Then, to create a more representative and comprehensive collection of fragments, we picked 100 more fragments from various clusters. We looked at 25 more fragments from each of the following four categories: (1) fragments from contracts with the most transactions, (2) fragments from contracts with the most unique callers, (3) fragments from the most duplicated contracts, and (4) 25 random fragments. To better understand the nature of the examined inline assembly fragments, cover a wide range of use cases, and reduce the possibility of getting biased, we chose to uniformly study fragments from the clusters rather than do it iteratively (i.e., first study all the fragments of one cluster and then continue with the fragments of the next cluster). Specifically, we randomly picked

[6]We refer the interested reader to our artifact for a complete description of the database schema (https://github.com/StefanosChaliasos/solidity-inline-assembly#database-description).

20 fragments from the first category and 5 from each other cluster in every iteration. This procedure was repeated five times. During these five iterations, we identified ten classes of inline assembly use. Our qualitative steps are in line with the best practices of previous work. In a similar study about eval usage in R, Goel et al. [2021] studied 113 eval uses, while previous work [Chaliasos et al. 2021] have been performed a manual analysis in iterations to reduce bias risks.

3.4 Threats to Validity

We use a standard methodology [Feldt and Magazinius 2010] to identify validity threats, which we mitigate where possible. This section discusses threats to internal and external validity.

Internal Validity. One potential threat to internal validity is associated with the selection criteria and representativeness of the inline assembly fragments on the qualitative analysis. We selected fragments from multiple clusters with different characteristics to minimize this threat. Another issue regarding the validity of the quantitative analysis is that there may exist an implementation bug somewhere in the codebase. We extensively tested the framework to mitigate this risk. Furthermore, the framework and the raw data will be publicly available for other researchers and potential users to check the validity of the results.

External Validity. A potential threat to the external validity is related to the fact that the set of smart contracts we have considered in this study may not be an accurate representation of all deployed smart contracts in Ethereum. Therefore, we attempt to reduce the selection bias by leveraging an extensive collection of real, reproducible smart contracts from Etherscan. However, Etherscan does not have 45% of the total contracts' source code. Nevertheless, we have created the largest dataset of Solidity smart contracts to the best of our knowledge. Third-party APIs like Google Big Query and Etherscan also compromise threats to external validity. An Ethereum archive node and a customized crawler would be a superior alternative to BigQuery. However, analyzing blockchain data for more than six years takes a long time. Unfortunately, there is no other method to get the source code of contracts than a blockchain explorer. To mitigate these concerns, we chose 100 random addresses and verified that both the BigQuery data and the source codes retrieved from Etherscan are correct.

4 QUANTITATIVELY STUDY INLINE ASSEMBLY ON SOLIDITY SMART CONTRACTS

This section presents the main findings of our quantitative analysis on smart contracts containing inline assembly in our corpus. We first measure how prevalent is inline assembly in smart contracts and how extensively it is used in those contracts (Section 4.1). Then, we study the characteristics of contracts that contain inline assembly and compare them with those that do not use inline assembly (Section 4.2). Following that, we examine the evolution of inline assembly usage through time (Section 4.3). Finally, we complete the qualitative analysis by performing a taxonomy of inline assembly based on the instruction usage (Section 4.4).

As noted in Section 3.1, all contracts in our dataset have at least one transaction or a token transfer, ensuring that all the contracts have been utilized in practice. Nevertheless, as more important contracts (i.e., contracts with more than 10 transactions) may differ from the rest of the contracts, we automated the quantitative analysis and re-ran the analyses for the following groups: contracts with more than 10 transactions, contracts with more than 10 token transfers, contracts with more than 10 transactions or 10 token transfers, and contracts with more than 10 transactions and 10 token transfers. There were no discernible differences between these categories regarding the insights gained from the research questions. However, for some specific sub-questions, the results for the contracts with more than 10 transactions deviate from the results of the complete dataset. In

Description	
Total Contracts	6,851,728
Total Contracts using Inline Assembly	1,583,177 (23%)
Total Unique Contracts	159,241
Total Unique Contracts using Inline Assembly	62,848 (39%)
Total Inline Assembly Fragments	3,427,424
Total Inline Assembly Fragments In Unique Contracts	176,961
Total Inline Assembly Unique Fragments	5388
Total Instructions	33,722,920

Fig. 4. Statistics about contracts using inline assembly, unique contracts using inline assembly, inline assembly fragments, and instructions.

Description	max	min	mean	median	std
Fragments per contract	81	1	2.16	2	3.57
Unique fragments per contract	53	1	2.08	2	2.93
Instructions per fragment	4013	1	9.94	8	9.44
Instructions per unique fragment	4013	1	10.62	4	63.52

Fig. 5. Statistics about inline assembly fragments per contract, unique inline assembly fragments per contract, and instructions per fragment.

these instances, at the end of each research question, we include a paragraph in which we describe the noteworthy differences for contracts with more than 10 transactions.

4.1 RQ1: Measuring Inline Assembly

This section reports on the frequency of inline assembly in Solidity smart contracts based on our quantitative analysis. Our corpus contains 6,851,728 contracts deployed in Ethereum, of which 1,583,177 (23%) contain inline assembly (c.f. Figure 4). These contracts contain a total of 3,427,424 inline assembly fragments. Furthermore, out of 159,241 distinct contacts 62,848 utilize inline assembly (39%) containing 176,961 inline assembly fragments. Notably, only 5388 of those fragments are unique, indicating that developers of smart contracts reuse the same inline assembly fragments across multiple smart contracts. The 3,427,424 fragments use 33,722,920 instructions in total, whereas the unique fragments use 56,262 instructions.

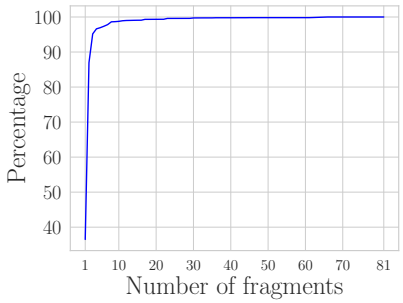

Fig. 6. Cumulative distribution of fragments per contract.

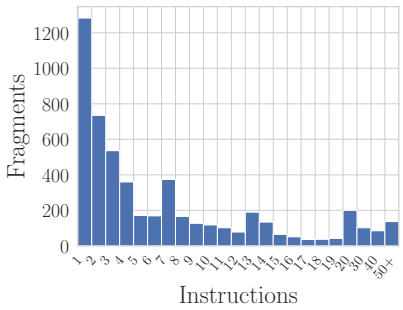

Fig. 7. Distribution of instructions per unique fragment.

Remark: In the following, we conducted the quantitative analyses without deduplication [7] to assess the impact and presence of inline assembly in contracts deployed in the Ethereum blockchain. This way, we can measure how widespread is inline assembly in deployed contracts (i.e., contracts executed in the Ethereum blockchain).

The percentage of projects with inline assembly is high (more than one of every five contracts uses inline assembly), which is surprising because many Solidity source code analysis tools are based on the assumption that inline assembly is rarely used [Feist et al. 2019; Tikhomirov et al. 2018; Tsankov et al. 2018]. These results are consistent with the study of Rigger et al. [2018], which indicates that 15.6% of C projects contain inline assembly. Nevertheless, in terms of density, inline assembly is orders of magnitudes more common than in C. Specifically, in Solidity smart contracts, there is one inline assembly fragment per 212 LOC, while in C projects, there is only one fragment per 40 KLOC of C code on average. This is mostly because smart contracts are often smaller than C

[7] Unless otherwise mentioned

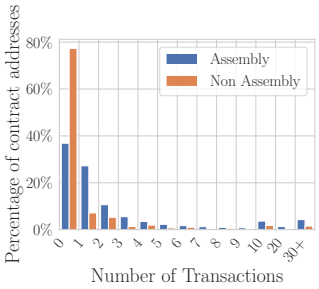

Fig. 8. Distribution of transaction number per contract.

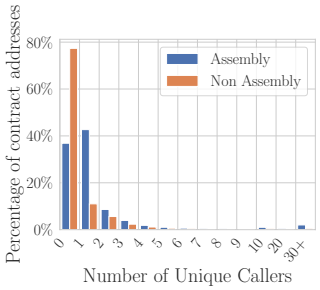

Fig. 9. Distribution of unique transaction callers per contract.

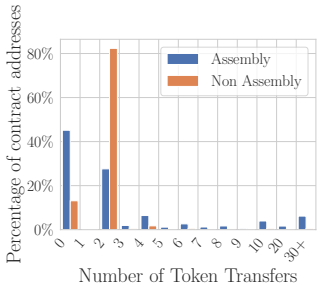

Fig. 10. Distribution of token transfers per contract.

projects (hundreds vs. thousands of LOC per project). Yet, it is surprising that in comparison with C projects, inline assembly is as much used in Solidity because there are more use-cases in C that inline assembly is the only way to perform certain operations (e.g., multicore programming, clock cycle counter, and other low-level operations). Hence, we manually study Solidity inline assembly fragments to reason why developers use inline assembly (c.f. Section 5).

Each contract contains on average 2.16 fragments and 2.08 unique fragments (c.f. Figure 5). We observe that although there is much duplication between contracts, there is minimal duplication within contracts. Figure 6 shows the cumulative distribution of fragments per contract. Notably, 54% of the contracts contain only one inline assembly fragment, and 95% contain at most 3. The aforementioned results suggest that practitioners prefer to use inline assembly for specific tasks in their programs.

To measure the number of instructions executed in an inline assembly fragment, we consider only unique fragments to get a better overview of the characteristics of the fragments. On average, a fragment contains 10.62 instructions with a median of 4. In some cases, there might be used as many instructions as 4K. Figure 7 illustrates the distribution of the number of instructions included in each fragment. Remarkably, more than half of the fragments (54%) use less than five instructions. Nevertheless, 30% of the fragments use more than ten instructions, while a non-negligible number of fragments (10%) use more than 20 instructions. The number of instructions utilized by a fragment reflects how simple, and thus how easy, an inline assembly fragment is to write and comprehend.

Discussion. Inline assembly is widely used in Solidity smart contracts deployed on the Ethereum blockchain. Nearly every fifth contract contains at least one inline assembly fragment. Nevertheless, the majority of contracts use inline assembly modestly (2.16 fragments on average). Furthermore, although fragments are typically small (having a median of 4), numerous contracts (30%) use many inline assembly instructions while performing complex operations. As a result, studying inline assembly in more depth is essential for understanding how developers use inline assembly, and what the implications of using inline assembly are in a blockchain environment.

Differences for contracts with more than 10 transactions. Notable, the use of inline assembly is much more widespread in contracts with more than 10 transactions (42% vs. 23%), highlighting the need for a deeper understanding of inline assembly fragments and improved tooling for analyzing inline assembly. Regarding the number of inline assembly fragments per deployed contract and the number of instructions per inline assembly fragment, contracts with more than 10 transactions have a slightly greater mean number of fragments (2.91 vs. 2.16), but almost the same number of instructions per fragment.

4.2 RQ2: Smart Contract Characteristics

The goal of this section is twofold. First, we analyze the source code and the metadata of our corpus to discuss the characteristics of the smart contracts containing inline assembly, and secondly, we compare the results with the respective characteristics of contracts that do not utilize inline assembly. Recall that the population of smart contracts containing inline assembly is 1,583,177, whereas our corpus includes 5,268,551 contracts that do not contain inline assembly; thus, the latter group may be more representative. The properties that we will utilize to examine the contracts are as follows: (1) number of transactions, (2) number of token transfers, (3) unique callers, (4) balances, (5) ERC token tag (i.e., fungible or Non-fungible Token (NFT)), (6) Etherscan labels, and (7) lines of code.

Figure 11 shows statistics for the characteristics of contracts that make use of inline assembly versus those that do not. Specifically, we have computed statistics for the number of transactions, token transfers, unique callers, balances, and lines of code. Contracts that include assembly have more transactions on average (145.72 vs. 97.16). Furthemore, more contracts without assembly have no transaction (c.f. Figure 8). The distribution of unique callers (c.f. Figure 9) follows the distribution of the number of transactions, which is reasonable as most contracts only have a few transactions. Hence, they only have a single caller (almost 40% for contracts using assembly) or less than five unique callers (99% of contracts without assembly). On the flip side, token transfers are more common in contracts that do not use inline assembly (c.f., Figure 10).

Description	max	min	mean	median	std
Non Assembly Transactions Number	126,721,373	0	97.16	0	60,702.88
Assembly Transactions Number	19,565,909	0	145.72	1	23,631.73
Non Assembly Unique Callers	18,282,428	0	16.42	0	8209.55
Assembly Unique Callers	5,657,480	0	37.45	1	5450.16
Non Assembly Token Transfers	37,174,087	0	26.42	2	16,539.01
Assembly Token Transfers	8,143,788	0	191.38	2	13,767.15
Non Assembly Balance	9,833,346	0	3.94	0	5347.19
Assembly Balance	572,709	0	3.01	0	879.55
Non Assembly LOC	9461	0	62.36	40	67.53
Assembly LOC	14,151	0	248.94	161	511.76

Fig. 11. General statistics about contracts containing inline assembly and those that do not.

Interestingly, there is a slightly higher percentage of contracts using inline assembly with many transactions. To investigate why that happens, we manually study the top contracts in terms of transaction numbers. Although the most popular contract addresses are stablecoins and other token contracts in both categories, contracts with inline assembly are more diverse. For instance, in contracts without assembly, eight of the ten contracts with most transactions are token contracts, and two are decentralized exchanges. In comparison, just four of the contracts that make use of assembly are token contracts, while the remaining contracts implement `Decentralized Finance` (DeFi) protocols (e.g., DEX and NFT marketplaces). Importantly, we also observed that contracts containing assembly, with more than 50000 transactions, use more inline assembly fragments on average (4.12 vs. 2.16) than when they have fewer transactions.

We detect no substantial difference in the balances of contracts between contracts involving inline assembly and others. (c.f. Figure 11). The former has a marginally higher mean balance because the population of contracts using inline assembly is smaller than the others, and in both categories, most contracts have zero or near-zero balance. Contracts with a vast balance are usually token bridges or Initial Coin Offering wallets (ICO wallets) used for funding purposes.

Non Assembly ERC-20	1.30%
Assembly ERC-20	0.09%
Non Assembly ERC-721	0.01%
Assembly ERC-721	0.02%

Fig. 12. Percentage of contracts that implement ERC-20 or ERC-721 standard.

Next, we categorized the contracts based on labels retrieved from Etherscan. In both categories, “Token Contracts” and “Old Contracts” are the most common labels. Interestingly, some popular DeFi protocols use inline assembly in their deployed contracts, while

others with similar functionality do not utilize inline assembly. The most prolific protocols that use inline assembly are Maker, bZx, Compound, and ENS, whereas some popular protocols that do not use inline assembly are Uniswap, Balancer, and Synthetix. Furthermore, we observe from both Etherscan labels and the metadata of our dataset that NFT contracts (i.e., ERC-721) tend to use inline assembly more frequently than ERC-20 tokens (c.f. Figure 12).

Finally, to get an insight into the complexity of Solidity smart contracts that use inline assembly, we measured the lines of code for the contracts in our dataset. Figure 11 presents general statistics about the lines of code of contracts. Notably, smart contracts containing inline assembly have a mean of 248.94 LOC, while other contracts contain 62.36 LOC on average. The main reason for this difference is that only a few contracts (10%) containing inline assembly are small (i.e., less than 50 lines). One possible explanation is that developers implementing small contracts do not typically require inline assembly. On the other side, there is a greater likelihood that large contracts will require assembly to conduct certain operations or optimize performance and gas consumption.

Discussion. The facts presented here indicate that there are little distinctions between contracts that use inline assembly and other contracts. However, an important point is that the usage of inline assembly appears to be a developer preference, as protocols with near identical functionalities (e.g., decentralised exchanges) choose whether or not to utilize it. Another important distinction is that contracts using inline assembly are often larger than average contracts. Additionally, NFT contracts are more likely to employ inline assembly than ERC-20 token contracts. Finally, when popular contracts (i.e., those with a large volume of transactions) employ inline assembly, they often include a greater number of inline assembly fragments than non-popular contracts (4.12 vs. 2.16).

Differences for contracts with more than 10 transactions. Regarding this research questions, there are no discernible variations between the contract characteristics of the entire dataset and those with more than 10 transactions. The primary reason for this is that contracts with greater values have a larger impact on both datasets. For certain features such as TVL, we notice a significant increase in both contracts with and without inline assembly. Specifically, the mean TVL rises from 3.94 to 103 for non-assembly contracts with more than 10 transactions, while it increases from 3.01 to 30.46 for contracts with inline assembly. Still, we observe that contracts without inline assembly have a higher TVL in both datasets. We also note similar differences in the number of token transfers and essentially no variation in the number of transactions, unique callers, and LOCs across the two datasets.

4.3 RQ3: Evolution of Inline Assembly

To gain a better understanding of how inline assembly usage has evolved over time, we counted the number of deployed contracts that include inline assembly in each block of our corpus. Furthermore, since developers may prefer to use specific versions of the compiler when using inline assembly, we also assess how many contracts that contain inline assembly fragments have been compiled with each compiler version.

Evolution through time. Figure 13 presents the percentage of smart contracts with published code in Etherscan that include at least one inline assembly fragment. We observe that developers started using inline assembly in late 2015, and since then, more and more contracts have included assembly fragments. The spikes and drops are difficult to explain because we only have a percentage of the contracts deployed in our corpus. For the drop in early 2017, we observed a large number of contracts with published code deployed in this period, and it happens that few of them include inline assembly. We also believe that the spike in 2017 happened because Solidity 0.4.12 was published

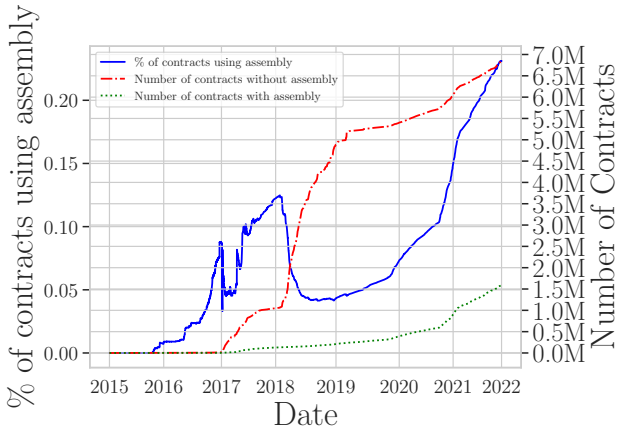

Fig. 13. Evolution of contracts using inline assembly through time.

in mid-2017. This version is the first with full support for YUL. Notably, the first comment in the release message mentions that *"This release introduces the AST export, solidifies inline assembly, …"*. In summary, we observe that contracts using inline assembly increase as the Solidity language and the Ethereum ecosystem evolve and mature.

Usage of inline assembly per compiler. We proceed by examining the number of contracts that use inline assembly per compiler version. We do not extract the compiler version from the source code, but rather retrieve it through Etherscan to determine the exact compiler version used to generate and deploy the smart contract. Figure 14 illustrates for each version the number of contracts that do not use inline assembly, those that use, and the percentage of contracts using inline assembly and been compiled with a particular version. The first compiler version that adds support for inline assembly is `v.0.3.1`. Nevertheless, only a negligible number of contracts compiled with this version use inline assembly. In contrast, more contracts use inline assembly for each of the subsequent compiler versions. Notably, there seems to be a preference to compile smart contracts containing inline assembly fragments using the latest compilers.

Major Version	Contracts Without Assembly	Contracts With Assembly	Percentage of Contracts With Assembly
0.1.x	13	0	0%
0.2.x	99	0	0%
0.3.x	4,168,244	37	0%
0.4.x	1,014,702	489,165	32%
0.5.x	39,071	219,269	84%
0.6.x	12,785	66,484	83%
0.7.x	7638	748,356	98%
0.8.x	22,340	59,866	72%

Fig. 14. Number of contracts compiled with each `solc` version.

Discussion. There is an evident trend that inline assembly usage in Solidity smart contracts increases over time. This result highlights the importance of inline assembly as a core component of smart contracts. Furthermore, there are some indications that developers use the latest compiler versions to compile contracts containing inline assembly while compiling other contracts with older versions. We leave it as future work to investigate why developers prefer to compile their contracts with older versions when there is no strict restriction (such as some inline assembly features available only in the latest versions).

Differences for contracts with more than 10 transactions. Similar to the preceding research question, we do not observe significant changes in the evolution of inline assembly utilization for contracts

with more than 10 transactions. The most significant difference is that the percentage of contracts employing inline assembly surpassed 10% percent in 2017, and although there is a dip after that point (similar to Figure 13), the increase is more steady than in Figure 13.

4.4 RQ4: A Taxonomy of Inline Assembly

The preceding sections provided a quantitative analysis of inline assembly usage in Solidity smart contracts; this section attempts to classify inline assembly instructions. To do so, we extend the categorization performed by Wood [2022] in Ethereum's yellow paper. Specifically, we reclassified some instructions (e.g., `stop` from Arithmetic Operations to System Operations), we split some categories (e.g., Stack, Memory, Storage, and Flow Operations). Finally, we include two YUL-specific categories (i.e., YUL Declarations and YUL Special Instructions).

The first category contains arithmetic operations (Section 4.4.1). The second category of instructions includes comparison and bitwise logic operations (Section 4.4.2), and the third one contains hashing operations, i.e., keccak256 (Section 4.4.3). Following that, the instructions of the fourth category provide environmental information, for example, return information about the contract caller and the contract's code (Section 4.4.4). The fifth type of instructions contains information about the current block, such as the current block number and its timestamp (Section 4.4.5). The sixth category comprises low-level operations for manipulating the stack, the memory, and the storage (Section 4.4.6). Following are the flow operations (Section 4.4.7), which include opcodes for control flow (e.g., `JUMP`) and YUL control flow instructions (e.g., for-loop). Instructions for creating new contracts, communicating with other contracts, and manipulating the execution are included in the eighth category (Section 4.4.8). The ninth group is composed of the YUL declaration instructions for variables and functions (Section 4.4.9), while the instructions of the tenth category are used for logging operations (Section 4.4.10). Finally, although there are some additional opcodes for more general 'management' (e.g., moving values, pushing, and popping from the stack), such instructions are not used in inline assembly fragments. Similarly, certain more specialized YUL instructions (`datasize`, `dataoffset`, `datacopy`, `setimutable`, `loadimmutable`, `linkersymbol`, and `memoryguard`), which expose more advanced YUL features are rarely utilized in practice. The sole instruction we detected in only 12 contracts

Instruction	% Projects	Total Projects	Total Occurrences
`add`	80.22%	1,270,036	7,680,862
`sub`	7.75%	122,728	244,033
`exp`	1.78%	28,263	77,652
`div`	1.12%	17,688	125,294
`mul`	0.92%	14,639	116,783
`mod`	0.70%	11,107	20,820
`mulmod`	0.06%	942	111,854
`addmod`	0.00%	64	1084
`signextend`	0.00%	2	40
`sdiv`	0.00%	2	24
`smod`	0%	0	0

Fig. 15. Instructions for arithmetic operations.

	Instruction	% Projects	Total Projects	Total Occurrences
Comparison	`eq`	49.27%	780,014	1,618,655
	`gt`	1.88%	29,814	35,128
	`lt`	1.98%	31,296	115,250
	`slt`	0.04%	685	702
	`sgt`	0.00%	63	190
Bitwise Logic	`and`	68.24%	1,080,430	1,250,659
	`iszero`	7.94%	125,723	282,182
	`byte`	3.09%	48,854	69,582
	`or`	1.64%	25,915	48,334
	`shr`	2.04%	32,228	59,860
	`not`	1.31%	20,723	43,925
	`xor`	0.35%	5523	10,274
	`shl`	0.19%	3026	22,283
	`sar`	0.00%	21	36

Fig. 16. Comparison and Bitwise Logic Operations.

```
pos := mload(0x40)
mstore(0x40, add(pos, length))
```

Fig. 17. Arithmetic operations are commonly used with memory operations.

```
result := and(
  eq(mload(clone), mload(other)),
  eq(mload(add(clone, 0xd)), mload(add(other, 0
        xd)))
)
```

Fig. 18. Example where a combination of comparison and bitwise instructions to save a boolean value.

is `datasize`, which takes string literals as arguments and returns their size. It is worth noting that there may be several reasons for employing assembly code; thus, practitioners may include instructions from multiple categories within a single fragment. In Section 4.4.11, we discuss how inline assembly instructions are combined in the smart contracts of our corpus.

*4.4.1 Arithmetic Operations (*80.31*%).* Arithmetic operations (see Figure 15) are frequently used in inline assembly fragments. Notable add opcode is the second most common instruction in our dataset. Beyond performing optimizations such as in vector-reduction arithmetics (e.g., vector summation), arithmetic instructions are typically used along with memory instructions for memory management. For example, in the snippet of Figure 17 `add` is used to allocate some bytes in the memory. Interestingly, this appears to be quite common when using inline assembly in Solidity smart contracts, and the primary reason developers use the `add` instruction. In comparison, Rigger et al. [2018] observed no such pattern in C inline assembly fragments. Finally, one instructions (i.e., `smod`) was never used in any of our dataset's contracts.

*4.4.2 Comparison and Bitwise Logic Operations (*77.91*%).* Certain inline assembly fragments, mainly those larger in size, contained instructions to compare and perform bitwise operations (c.f. Figure 16). While the comparison instructions are often used in conjunction with control flow operations, several fragments use a combination of comparison and bitwise instructions to save a boolean value to a variable. For instance, in Figure 18, `mload` is used to retrieve two pairs of addresses, followed by `eq` to compare the pairs, and finally, `and` is used to apply logical conjunction. Additionally, a non-negligible number of contracts use the `iszero` instruction, which checks if the topmost element of the stack is 0. Finally, similarly to arithmetic operations, some instructions are barely used.

Instruction	% Projects	Total Projects	Total Occurrences
`keccak256`	0.96%	15,266	109,831
`sha3`	0.01%	172	1385

Fig. 19. Hashing operations.

	Instruction	% Projects	Total Projects	Total Occurrences
Current Environment	`gas`	23.51%	372,278	411,752
	`returndatasize`	16.91%	267,645	620,491
	`returndatacopy`	16.89%	267,442	289,248
	`calldatacopy`	15.09%	238,884	241,679
	`calldatasize`	15.15%	239,873	482,474
	`calldataload`	8.45%	133,803	231,687
	`chainid`	5.43%	85,926	87,881
	`address`	0.75%	11,931	12,087
	`codesize`	0.33%	5216	5216
	`codecopy`	0.33%	5216	5216
	`callvalue`	0.22%	3490	6009
	`caller`	0.17%	2677	2864
	`gasprice`	0.00%	9	17
	`origin`	0.00%	16	16
	`selfbalance`	0.00%	11	44
Account Information	`extcodecopy`	44.97%	712,027	712,070
	`extcodesize`	12.84%	203,304	226,917
	`extcodehash`	3.26%	51,645	51,771
	`balance`	0.56%	8823	8824

Fig. 20. Environmental Information.

*4.4.3 Hashing Operations (*0.97*%).* Only a few contracts use a hash function in inline assembly fragments. The fragments that perform hash operations are typically large and implement complex logic. For example, a common use case is to apply the `keccak256` instruction after retrieving some data from memory and then use that value in Solidity code. Note that `sha3` was used as an alias to `keccak256` until compiler version 0.5.0. Since then, only the `keccak256` instruction has existed. Finally, unlike Solidity, the hash function in inline assembly takes a byte range in memory rather than a string.

Instruction	% Projects	Total Projects	Total Occurrences
`timestamp`	0.85%	13,513	13,544
`number`	0.30%	4684	4697
`blockhash`	0.30%	4674	4682
`coidbase`	0.29%	4669	4671
`difficulty`	0.00%	1	1
`gaslimit`	0%	0	0

Fig. 21. Block information.

*4.4.4 Environmental Information (*80.86*%).* Numerous contracts take advantage of inline assembly instructions to query the blockchain and execution environment. This category is further divided into two subcategories: *Current Environment* and *Account Information* (c.f. Figure 20). The former

includes instructions that provide information about the current execution, such as returning the address of the currently executing account, whereas the latter contains instructions that return information for an arbitrary account address. Notably, instructions in this category are most frequently used in isolation, that is, they are not combined with instructions from other categories. One such instance is the use of `extcodesize` to determine whether an address is a contract or a regular account. Specifically, if an address is a contract, then `extcodesize` returns the size of the code on that address. Otherwise, it returns 0. Note that this procedure is not available in plain Solidity; consequently, determining if an address is a contract requires the use of inline assembly. Similarly, many inline assembly fragments use the `extcodecopy` instruction to obtain the code of another contract, despite that operation being available in Solidity (`_addr.code`). Another typical use case of instructions from this category is to compute the amount of available gas in the current execution (through the `gas` instruction), extract a certain amount, and pass the result to a message-call opcode (e.g., `staticcall`). Finally, the `returndatasize` and `returndatacopy` instructions, which access data from the prior environment, are extensively used.

4.4.5 Block Information (0.85%). Instructions in this category return information about the current block (see Figure 21). Notably, these instructions are sparsely utilized, and one of them has never been used in any contract in our corpus. The most common instruction in this group is `timestamp`, which returns the current block's timestamp. The primary application of this instruction is to retrieve its result and use it to produce a pseudo-random number. Finally, these instructions are used only in large fragments.

4.4.6 Stack, Memory, and Storage Operations (90.94%). The most frequently used instructions in inline assembly fragments are those that interact with the stack, memory, and storage. We further classify the instructions into three subcategories (c.f. Figure 22). Stack instructions are rarely used, and their application is limited to very specialized problems. On the flip side, memory instructions are extensively used in our dataset. Moreover, these instructions are typically combined with instructions from other categories. For instance, a typical use case is to use `mload` to load data from memory, then utilize that data in other procedures, and finally, either save the result back in the memory or use it directly for another operation. Storage instructions follow the same patterns as memory instructions but are less used due to their high cost.

	Instruction	% Projects	Total Projects	Total Occurrences
Stack	`pop`	0.24%	3859	7894
	`pc`	0%	0	0
Memory	`mload`	85.32%	1,350,842	6,659,754
	`mstore`	48.92%	774,457	4,653,053
	`mstore8`	0.29%	4595	37,482
	`msize`	0.26%	4139	4164
Storage	`sload`	8.22%	130,119	190,043
	`sstore`	3.17%	50,216	145,625

Fig. 22. Stack, Memory, and Storage Operations.

	Instruction	% Projects	Total Projects	Total Occurrences
YUL control flow	`switch`	21.35%	337,978	484,094
	`if`	4.76%	75,382	181,460
	`for`	1.78%	28,226	85,049
	`leave`	0%	0	0
Control-Flow Opcodes	`jumpi`	0.07%	1033	1354
	`jump`	0.01%	117	118
	`jumpdest`	0%	0	0

Fig. 23. Flow operations.

4.4.7 Flow Operations (24.17%). Control-flow-related instructions (c.f. Figure 23) are mostly confined to larger inline assembly fragments to facilitate branching. A less common use case is for performing loops (1.78%). Interestingly, the `switch` statement, which is unique to YUL (Solidity does not provide a `switch` statement), is the most frequently used control-flow construct. Note that the `if` statement can only be used for conditionally executing a code block, and no '`else`' block can be defined. Hence, the `switch` statement is more common in inline assembly fragments. Before the introduction of YUL, the control flow was achieved with `jump` instructions. However,

since the introduction of YUL, the `jump` instructions have been prohibited because they obfuscate the control flow.

4.4.8 System Operations (74.16%). This category contains instructions for executing calls, creating new contracts, halting the execution, as well as the `invalid` opcode that does nothing (see Figure 24). The most common instruction is `create2`. In comparison to other instructions from this category, `create2` and `create` are typically used in small fragments (less than three lines of code). Note that before Solidity version 0.7.0, `create2` was only available through inline assembly. `delegatecall` is used in large fragments because it requires some pre-call preparation and some further instructions to handle the call's outcome. Execution instructions are often used in conjunction with control flow instructions to revert a transaction. Furthermore, in most cases, the `return` instruction is used in fragments that also contain the `revert` instruction.

	Instruction	% Projects	Total Projects	Total Occurrences
Call	`delegatecall`	21.36%	338,214	340,921
	`call`	2.64%	41,770	49,813
	`staticcall`	1.39%	21,949	26,319
	`callcode`	0%	0	0
Create	`create2`	45.09%	713,871	714,477
	`create`	6.58%	104,130	104,446
Execution	`revert`	28.59%	452,665	686,586
	`return`	15.77%	249,679	290,419
	`stop`	0.15%	2319	2329
	`selfdestruct`	0.00%	4	7
Invalid	`invalid`	0.04%	640	1738

Fig. 24. System operations.

Instruction	% Projects	Total Projects	Total Occurrences
`let`	73.24%	1,159,452	3,579,730
`function`	0.00%	49	113

Fig. 25. YUL declarations.

Instruction	% Projects	Total Projects	Total Occurrences
`log1`	0.18%	2854	2869
`log4`	0.05%	809	820
`log3`	0.03%	548	569
`log0`	0.03%	539	539
`log2`	0.03%	541	541

Fig. 26. Logging Operations.

4.4.9 YUL Declarations (73.24%). Several contracts use the `let` instruction to declare a variable inside an inline assembly fragment, whereas only 49 contracts declare at least one `function` in a fragment (see Figure 25). `let` is typically used to store the result of a call instruction or save data retrieved by the memory. Functions are used in large fragments and typically perform complex arithmetic computations.

4.4.10 Logging Operations (0.20%). There are five log opcodes in EVM, the first one (i.e., `log0`) appends a log record without a topic, while the last one (i.e., `log4`) appends a log record with four topics (c.f. Figure 26). Normally, log instructions are used at the end of an inline assembly fragment to append a message in the logs. However, another use case is using a `switch` statement to determine which `log` command to use based on the topics it needs to write.

4.4.11 Instructions Usage and Combination. Figure 27 illustrates how instructions from different categories are combined in contracts. Note that the chart includes all categories that are utilized by more than 1% of our dataset's contracts and all combinations that are detected in more than 500 contracts. Notably, we observe that the majority of contracts incorporate instructions from numerous categories. The lone exception is the environmental information that more than 100,000 contracts use only instructions from this category. Another critical insight is that although memory operations, arithmetic operations, and comparison/bitwise logic instructions are the three of the four most prevalent types of instructions, they almost always paired with other instructions, and typically with instructions from more than one category. Furthermore, an sizable percentage of contracts (more than 180,000) utilize instructions from all categories.

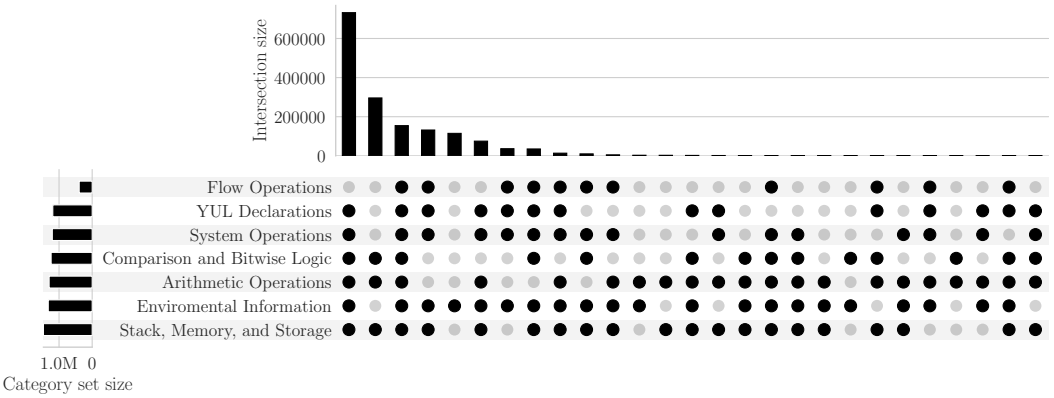

Fig. 27. Instruction usage and their combinations. Categories with • in the same column are combined in fragments, and the population of their intersection is displayed in the upper bar chart. The intersection of all categories except instructions from *Flow Operations* is the most commonly used combination in inline assembly fragments. Contrarily, *Environmental Information* opcodes are used most frequently without instructions from other categories. Note that the chart includes all categories that are utilized by more than 1% of our dataset's contracts and all combinations that are detected in more than 500 contracts.

Discussion. This Section's primary objectives were to (1) classify instructions, (2) quantify which instructions are most frequently utilized in practice, and (3) examine which instructions are combined in contracts using inline assembly. Only 37 out of 85 [8] instructions are used in our corpus by at least 1% of the contracts. One essential observation is that although inline assembly fragments have a median of only 4 instructions (c.f. Figure 5), they typically include instructions from at least three categories. The majority of fragments appear to focus on exploiting functionalities related to environmental information instructions, or manipulating/retrieving data in memory and storage. Furthermore, in this Section, we identify several prevalent inline assembly usage patterns. For instance, arithmetic operations are primarily used for memory management. Additionally, certain instruction providing environmental information are frequently used because they expose functionality that is unavailable in Solidity. The following Section investigates and discusses usage pattern in greater detail.

Differences for contracts with more than 10 transactions. In this research question, we observe some differences between contracts with more than 10 transactions and others. First, it is noteworthy that contracts with more than 10 transactions favor `extcodesize` (37.43%) and `extcodehash` (13.61%) over `extcodecopy` (4.61%), which is significantly different from the results of the entire dataset, where `extcodecopy` (44.97%) is the most frequently used opcode for retrieving account information. Another similar observation is that `create` is more prevalent than `create2` in the dataset of contracts with more than 10 transactions (12.40% vs. 6.58% and 6.77% vs. 45.09%). In addition, we notice variations in the proportion of contracts that employ instructions from particular categories. The two more significant changes are in the usage of 'Comparison and Bitwise Logic Operations' and 'Flow Operations'. Specifically, only 53.85% (vs. 77.91%) of contracts with more than 10 transactions use instructions from the former category, whereas we notice an increase from 24.17% to 37.82% for instructions from the 'Flow Operations' category. The combination of instructions is comparable

[8]Note that we count all opcodes and YUL instructions available in inline assembly. We do not include all specialized YUL instructions that do not appear in our corpus.

Class	Functionality not available in Solidity	Optimization	Complementary Fragment
Single Instruction Unavailable in Solidity	✓	✗	✗
Deserialization	✓	✓	✓
String, Bytes, and Math Utils	✓	✓	✗
Proxy Implementation	✓	✓	✗
Error Handling	✓	✓	✓
Call other Contract	✗	✓	✗
Optimizations for Cryptographic Primitives	✗	✓	✗
Handling Arbitrary Data	✓	✓	✓
Fine-Grained Memory and Storage Control	✓	✓	✓
Manual Inlining	✗	✓	✗

Fig. 28. Usage classes of inline assembly in Solidity contracts.

to the entire dataset, with the exception that the combination that includes 'Flow Operations' is the most common.

5 RQ5: USAGE OF INLINE ASSEMBLY

According to Solidity's official documentation, inline assembly is intended to used to enhance the Solidity language by writing libraries and performing optimizations when the compiler is inefficient to optimize code aggressively. To understand why developers utilize inline assembly in practice, we manually inspected 170 distinct inline assembly fragments. Firstly, we examined the 70 most-deployed fragments, which account for 90% of all inline assembly use in our corpus. Then, we further selected 100 more fragments from other clusters to have a more representative and complete dataset of fragments. Specifically, we studied 25 additional fragments from each of the following categories: (1) the contracts with the most transactions, (2) the contracts with the most unique callers, (3) contracts with the highest usage (i.e., duplicated), and (4) 25 random fragments. [9]

We present examples from ten classes from this batch (c.f. Figure 28). This classification is not exhaustive since there are 5388 unique fragments in our dataset, many of which do not fit in a particular category. Furthermore, we characterize the ten classes based on three main goals: fragments that provide functionality not available in Solidity, fragments used to perform optimizations, and complementary fragments used to support another inline assembly fragment. Note that one fragment might accomplish multiple tasks, and hence belongs to many classes. Finally, we will briefly discuss some additional use cases we identified during our manual analysis.

Single Instruction Unavailable in Solidity. One common use of inline assembly is employing a single opcode instruction to obtain information not available through Solidity code. For instance, the `extcodesize` opcode returns the size of a contract, if an address contains a contract, otherwise, it returns zero. It is impossible to find if an address is a contract through Solidity code; thus, developers have to use inline assembly (c.f Figure 29). Although in most cases, developers use Solidity to check if the return value is zero, occasionally, they compute that inside the assembly fragments using additional instructions. Another example is the `chainid` opcode that returns the id of the current chain. This opcode can be used to prevent replay attacks between different chains. [10] Note that since Solidity version 0.8.0 `chainid` is available in plain Solidity through `block.chainid`.

[9]Note that in case of overlap of one fragment from one category with one already studied from another category, we replaced the overlapped fragment with a different one from the same group.
[10]https://eips.ethereum.org/EIPS/eip-1344

```
1  assembly { size := extcodesize(account) }
```

Fig. 29. The `extcodesize` instruction provides functionality that is unavailable in Solidity.

```
1  assembly {
2      r := mload(add(signature, 32))
3      s := mload(add(signature, 64))
4      v := and(mload(add(signature, 65)), 255)
5  }
```

Fig. 30. Deserialize a signature using inline assembly.

```
1  assembly{
2      switch iszero(b)
3      case 0 {
4          res := div(a,b)
5          let loc := mload(0x40)
6          mstore(add(loc,0x20),res)
7          i := mload(add(loc,0x20))
8      }
9      default {
10         err := 1
11         i := 0
12     }
13 }
```

Fig. 31. Optimized safe division of two numbers using inline assembly

Deserialization. Inline assembly can be used to deserialize data efficiently. One frequent use case is when one needs to recover an elliptic key signature using the `ecrecover` [11] built-in function. The `ecrecover` function accepts three arguments: the first 32 bytes of a signature, the second 32 bytes of the signature, and the signature's final byte. The fragment of Figure 30 recovers these values from a variable called `signature`.

String, Bytes, and Math Utils. Many libraries have been emerged to complement Solidity and provide functionalities that generally exist in a standard library. For instance, the 'String & Slice utility library', [12] which provides simple and advanced operations on strings, is one of the most frequently used libraries in Solidity smart contracts. We observe that many libraries performing operations on strings, bytes, or even arithmetic operations heavily use inline assembly. Furthermore, we also identify fragments where developers utilized inline assembly to perform optimizations. The fragment of Figure 31 efficiently performs a division between two numbers (i.e., a and b), while checking if the divisor equals zero.

Proxy Implementation. One of the most critical uses of inline assembly for many applications in the emerging DeFi field is implementing the proxy pattern. The proxy pattern allows creating a primary copy of a contract, then easily (and cheaply) creating clones with separate states. The deployed bytecode delegates all calls to the primary contract. Libraries typically provide three main functionalities (all implemented in inline assembly): a clone factory, a function to perform the proxy call, and finally, a function to check whether an address is a proxy contract. Figure 32 presents the assembly code for a clone factory. Line 4 loads the next free memory slot to store the clone contract data. Next, line 5 stores the bytecode for the clone contract based on the Minimal Proxy Standard. [13] The next line stores the address location of the implementation contract so that the proxy knows where to delegate the calls, while line 7 stores the rest of the bytecode. Finally, line 8 deploys the clone, and the result variable contains the address of the new proxy contract which is returned by the function.

Error Handling. Sometimes it is more convenient and efficient to perform error handling via inline assembly. The example of Figure 33 has been taken from OpenZeppelin's Address library, [14]

[11] https://docs.soliditylang.org/en/latest/units-and-global-variables.html#mathematical-and-cryptographic-functions
[12] https://github.com/Arachnid/solidity-stringutils
[13] https://eips.ethereum.org/EIPS/eip-1167
[14] https://github.com/OpenZeppelin/openzeppelin-contracts

```
1  function createClone(address target) internal
       returns (address result) {
2    bytes20 targetBytes = bytes20(target);
3    assembly {
4      let clone := mload(0x40)
5      mstore(clone, 0x3...)
6      mstore(add(clone, 0x14), targetBytes)
7      mstore(add(clone, 0x28), 0x5...)
8      result := create(0, clone, 0x37)
9    }
10 }
```

Fig. 32. Implementation of a clone factory.

```
1  ...
2  (bool success, bytes memory returndata) =
3    target.call{ value: weiValue }(data);
4  ...
5  if (! success) {
6    ...
7    assembly {
8      let returndata_size := mload(returndata)
9      revert(add(32, returndata), returndata_size)
10   }
11 }
```

Fig. 33. Error handling via inline assembly.

```
1  ...
2  bytes memory calldata = abi.encodeWithSelector(
3      IWallet(walletAddress).isValidSignature.selector, hash, signature);
4  assembly {
5    ...
6    let cdStart := add(calldata, 32)
7    let success := staticcall(gas, walletAddress, cdStart, mload(calldata), cdStart, 32)
8    // Error Handling
9  }
```

Fig. 34. Call a function from another contract efficiently using inline assembly.

and it uses inline assembly as it is the most suitable way to retrieve why the performed call was failed. Another common use case is to combine the `switch` instruction with the `revert` opcode to handle complex errors.

Call other Contract. Similar to the proxy implementation class, inline assembly is also used to perform efficient direct calls to other contracts. In the example of Figure 34 a static call verifies a signature using the logic defined in another contract (i.e., `IWalltet` contract). First, line 6 retrieves the pointer to the start of the input data (**calldata**), and then staticcall accepts the following arguments: the remaining gas, the address of the targeted contract, the pointer to the input, the length of the input, and again the pointer to the input where it will write the output. Other examples implemented using this pattern include address verification and delegating transactions.

Optimizations for Cryptographic Primitives. Implementation of cryptographic primitives is one of the primary operations practitioners try to optimize. To this end, they usually combine functionality from other classes, such as deserialization, with hashing functions or arithmetic instructions to efficiently implement cryptography algorithms. Figure 35 presents an example where given a hash of a specific transaction scheme, a salt value, an address, and some additional transaction data to be hashed; one can efficiently compute the hash of a transaction.

Handling Arbitrary Data. Inline assembly is used to handle arbitrary data, especially when implementing generic libraries. One example of this use is a library that provides generic function call logging by implementing a modifier that captures the function's first and second parameters and logs them in an event (c.f. Figure 36).

Fine-Grained Memory and Storage Control. Many inline assembly fragments contain only a few lines that manipulate either the memory or the storage, using `mload`, `mstore`, `sload`, and `sstore`. The goal of these fragments is to achieve fine-grained control over the memory. Typically, these

```
bytes32 schemaHash = EIP712_SCHEMA_HASH;
...
assembly {
  let memPtr := mload(64)
    mstore(memPtr, schemaHash)
    mstore(add(memPtr, 32), salt)
    mstore(add(memPtr, 64), and(signerAddr, 0xf...))
    mstore(add(memPtr, 96), dataHash)
    result := keccak256(memPtr, 128)
}
```

Fig. 35. Optimizing cryptographic operations using inline assembly.

```
modifier note {
  bytes32 foo;
  bytes32 bar;
  assembly {
    first_param := calldataload(4)
    second_param := calldataload(36)
  }
  emit LogNote(msg.sig, msg.sender, foo, bar,
        msg.value, msg.data);
}
```

Fig. 36. Modifier for enabling logging to generic function calls.

fragments are used as complementary to other inline assembly fragments. Nevertheless, this is still an essential class of inline assembly usage because it supports other operations while also providing control in the memory and the storage in a way that is not possible with plain Solidity.

Manual Inlining. A few fragments perform manual inlining using inline assembly, i.e., rather than using a library that implements certain functionality, they manually copy and paste only the assembly code into their code to reduce the gas costs. This class is mostly used for procedures from the 'String, Bytes, and Math Utils' class.

Other use cases. We further discuss some patterns we observed in only a few fragments in our corpus. One such use case is implementing timestamp-based locks using inline assembly instructions. Moreover, another inline assembly use that was interesting was implementing very specific functionality. For instance, one fragment creates a contract that can only be destroyed by another specific contract. Another example is a safer re-implementation of the built-in `ecrecover` function. One of the most intriguing fragments we studied was a highly optimized and convoluted code used to efficiently create and delete multiple contracts. Specifically, this contract [15] was introduced to make gas-efficient flash loans. Finally, we also observed a few fragments that combined multiple of the presented patterns to identify if a contract implements some specific interfaces.

Discussion. The results of our qualitative analysis suggest that developers use inline assembly for various reasons, from performing optimizations to implementing procedures that cannot be written in plain Solidity. One key difference between assembly fragments in library code and other smart contracts is that the former focus not only on optimizations but primarily on providing functionality that is not available to Solidity, whereas the latter mainly focuses on optimizations. While here we presented ten classes of inline assembly patterns, we observed that many fragments perform various operations, such as performing a contract call and then handle any possibly errors. Finally, we also notice that the number of fragments in a contract is usually a matter of the developer's taste because, in several cases, two fragments separated by one or two lines of solidity code could have been combined into a single inline assembly code block.

6 IMPLICATIONS AND DISCUSSION

We now discuss several implications of our work, and how our findings can serve as a basis for future research endeavors.

[15] https://etherscan.io/address/0x00000000454a11ca3a574738c0aab442b62d5d45#code

Prevalence of inline assembly in Solidity. In our corpus, 23% of the analyzed deployed contracts contain at least one inline assembly fragment, meaning that inline assembly is even more common in Solidity than in a system-level programming language such as C (15.6% [Rigger et al. 2018]). Furthermore, the trend of assembly usage has been upward since its introduction in late 2015. Interestingly, programs compiled with later versions of the Solidity compiler typically include inline assembly fragments more frequently than programs compiled with previous versions. We further analyzed usage patterns of inline assembly (Sections 4.4 and 5), and we observed that developers employ inline assembly for various reasons, from performing optimizations to using features not available in Solidity. Hence, we conclude that inline assembly is an integral part of the Solidity language, and its usage has become more and more popular.

Support for inline assembly. We believe that our results are essential to tool writers, language designers, and compiler developers. Firstly, most of the tools working on Solidity source typically provide limited support for inline assembly [Feist et al. 2019; Tikhomirov et al. 2018; Tsankov et al. 2018]. To this end, our study (1) highlights the importance of inline assembly in Solidity contracts, and (2) can guide tool developers in planning and prioritizing the implementation of instructions. Notably, only 37 out of 85 instructions are used by at least 1% of the contracts, and by implementing 43 instructions, more than 82% of the contracts that use inline assembly would be supported. Additionally, this study could be helpful to language designers, as it reveals where plain Solidity is inadequate to implement some functionality resulting in developers employing inline assembly. For example, a built-in function to reason if an address is a contract is a long-waited feature by smart contract developers. Finally, compiler writers could obtain feedback on which instructions and combinations are frequently used, giving them insights on what optimizations could benefit more programs.

Unnecessary use of inline assembly. In some cases, using inline assembly is inevitable because it is the only way to implement some functionalities. Additionally, we identified some patterns where developers employ inline assembly to perform some operations way more efficiently than using Solidity (e.g., *Call other Contract*). Nevertheless, the use of inline assembly comes with its disadvantages. Firstly, it bypasses several essential safety features and checks provided by the Solidity language. For instance, since Solidity 0.8.0, adding two `units` cannot result in an overflow, but this is not true if the addition is performed using inline assembly. Additionally, using inline assembly could result in less efficient bytecode because some optimizations cannot be applied in the presence of inline assembly fragments. As the Solidity compiler evolves and adds more efficient and aggressive optimizations, some uses of inline assembly would become rendundant. For instance, if the compiler optimises the code sufficiently, the *Manual Inlining* pattern appears superfluous. Another potential category that may become obsolete is the *Optimizations for Cryptographic Primitives*. Finally, as Solidity evolves, more functionalities become part of the languages, resulting in more obsolete inline assembly uses. For instance, the `chainid` opcode has been available in Solidity since version 0.8.0. Another example is the array slice function that was introduced in Solidity 0.6.0.

Inline assembly alternatives. As already mentioned, inline assembly could thwart the correctness of smart contracts or could even harm the efficiency of the generated bytecode. Another issue is that code auditing, a cornerstone in the development lifecycle of most important smart contract applications, becomes more complicated when inline assembly is present, as inline assembly code is typically more convoluted than Solidity code. For these reasons, it is essential to create tools for replacing inline assembly code with equivalent Solidity code. There is already a growing body of work that focuses on getting rid of inline assembly in C programs [Corteggiani et al. 2018;

Recoules et al. 2019] and replacing JavaScript's eval [Jensen et al. 2012; Meawad et al. 2012], which researchers can use as a starting point for building similar methodologies for inline assembly in Solidity. Finally, the Solidity team intends to develop a standard library for the Solidity language that will be distributed with the compiler. Parts of this library would be implemented in inline assembly, as it is preferable to implement low-level functionality in inline assembly rather than 'hide' it in the compiler's source code. Consequently, the goal is that client code would utilize inline assembly less frequently because the standard library and other libraries would provide the required functionality.

7 RELATED WORK

Inline Assembly in C/C++. The closest study to our work is that conducted by Rigger et al. [2018] for inline assembly in C programs. The authors quantitatively and qualitatively analyzed the use of x86-64 inline assembly in 1,264 GitHub projects to mainly help developers of C-focused tools better understand how inline assembly is used in practice. Some of their key findings are as follows: (1) 15.6% of C projects contain inline assembly fragments, (2) inline assembly fragments usually consist of a single instruction, and more than 90% of the fragments contain less than ten instructions, (3) the majority of the projects use the same subset of instructions. Our study analyzed more than three orders of magnitude larger dataset. Inline assembly in Solidity is slightly more common than in C (23% [16] vs. 15.6% [17]), and fragments in Solidity contain more instructions than those in C (10.62 vs. 9.9). In a different spirit, Akshintala et al. [2019] measured instruction usage across the binaries of 9,337 C/C++ Debian packages. Following these studies, Recoules et al. [2019] developed TINA, the first automated and verification-friendly lifting technique turning inline assembly into semantically-equivalent C code. Similarly, Corteggiani et al. [2018] lift GNU inline assembly to semantically equivalent C code to verify mixed codes combining C and inline assembly. Furthermore, Recoules et al. [2021] propose RUSTINA for formally checking inline assembly compliance, with the ability to create patches and (optimization) refinements in some instances. Finally, beyond inline assembly researchers have highlighted the role of other non-standardized C language features that may reduce the ability of tools to analyze C code, such as linker scripts [Kell et al. 2016] and GCC builtins [Rigger et al. 2019].

Eval and Other Dynamic Features. Over the past decade, many works have studied the use and impact of eval and other dynamic features in various programming languages. Although eval differs in many ways from inline assembly, these studies have the same goals as our work, i.e., to study the use of a language component that hampers the efficiency of analysis tools and may introduce security risks.

Richards et al. [2010] performed the first study of the dynamic behaviour of JavaScript programs on 100 popular websites. Their results highlighted the need to study dynamic features of JavaScript further, and displayed the limitations of existing benchmarks that neglected specific language constructs such as eval. In a subsequent study, Richards et al. [2011] provided the first large scale study on JavaScript eval. Their dataset comprises of 10k popular websites and is analyzed through an instrumented web browser to gather execution traces. The outcome of this study was that 82% of the analyzed websites use eval for various purposes such as dynamic code loading, deserialization of JSON data, and lightweight meta-programming. Notably, many usages of eval could automatically be fixed by using more robust code [Jensen et al. 2012; Meawad et al. 2012]. In a recent study, Goel et al. [2021] analysed the use of eval in 15401 R packages and compared the results with previous

[16]Percentage of deployed contracts using inline assembly.
[17]Percentage of C projects using inline assembly.

work that analyzed eval in JavaScript programs. The authors argue that eval use is widespread in R programs, and is more pervasive and dangerous when compared to eval in JavaScript programs.

Empirical Studies on Solidity. Here we briefly present recent empirical studies on smart contracts. Bartoletti and Pompianu [2017] performed the first empirical analysis of smart contracts by manually analyzing the source code of 811 Solidity smart contracts obtained from Etherscan and classified them into five categories: financial, notary, game, wallet, and library contracts. Furthermore, they studied the distribution of transactions that interacted with the smart contracts per category and investigated various designed patterns employed in Solidity smart contracts. Similarly, we examined transaction data for contracts that contain inline assembly and compared them to contracts that do not contain inline assembly in our study. In a different spirit, Zou et al. [2019] conducted 20 semi-supervised interviews with Solidity developers and performed a survey on 232 practitioners to highlight the challenges in developing smart contracts in Ethereum. They identified five major fields that need improvement: security, debugging, limitations of Solidity, limitations of EVM, and Gas optimizations.

Analyzing the source code of smart contracts is a promising avenue to understand the properties of deployed smart contracts better. Similar to our work, numerous studies have crawled etherscan to get the source code of smart contracts and analyze them [Hegedűs 2019; Mariano et al. 2020; Pinna et al. 2019; Tonelli et al. 2018]. Tonelli et al. [2018] defined various metrics similar to those of C&K [Chidamber and Kemerer 1994] metrics in the OO world and compared their distributions with the metrics extracted from traditional software projects on more than 12 thousand smart contracts. Their results show that smart contracts metrics are typically more restricted than the corresponding metrics in traditional software systems. Hegedűs [2019] also defined and implemented C&K style software metrics and analyzed their distributions on 40 thousand smart contracts. In addition to complexity metrics (also studied by Tonelli et al. [2018]), they also studied coupling and inheritance metrics. Beyond source code characteristics, Pinna et al. [2019] performed a comprehensive empirical study of smart contracts deployed on Ethereum, studying on-chain data such as transaction numbers. In contrast to prior works, our study is performed on a much bigger dataset (millions instead of thousands of smart contracts). Additionally, we quantitative and qualitative study both on-chain data and smart contracts source code.

Finally, Mariano et al. [2020] studied syntactic and semantic characteristics of loops used in over 20,000 Solidity contracts to inform future research on program analysis for smart contracts. Similarly, we perform the first study on inline assembly usage in smart contracts to help researchers build appropriate techniques or adapt the existing ones to analyze Solidity inline-assembly.

8 CONCLUSION

We presented the first empirical study of inline assembly usage in a blockchain environment. Inline assembly is frequently used and has a variety of applications in Solidity smart contracts. Additionally, its utilisation continues to rise over time, and contracts with a broad range of characteristics employ it. The functionality enabled by inline assembly is essential for practitioners. Our analysis observed that libraries heavily depend on inline assembly, whereas independent contracts make use of inline assembly to optimise performance. Furthermore, inline assembly fragments typically combine different instructions and are larger than those seen in C. Notably, only 48 out of the 85 available instructions are used by more than 1% of the contracts that employ inline assembly. These observations go counter to the underlying intuition of static analysis techniques neglecting inline assembly.

We discussed several implications of our study's findings. Firstly, we emphasised the importance of improved support for inline assembly. The findings of this study can be used by researchers and Solidity tool developers to prioritise the implementation of specific instructions that will enable

their techniques to handle the majority of inline assembly fragments. Furthermore, Solidity might be extended via built-in functions to give developers certain functionality that is now absent (e.g., a function to return if an address is a contract). Finally, we believe that researchers will build upon our work's finding to develop approaches akin to C's inline assembly and JavaScript's eval for lifting inline assembly.

ACKNOWLEDGMENTS

We would like to thank the anonymous OOPSLA reviewers and Diomidis Spinellis for their insightful feedack on previous versions of the paper. We also want to thank Zhipeng Wang for his useful comments on our artifact. Finally, we want to thank Christian Reitwiessner for the insightful discussion about inline assembly's role in the Solidity ecosystem and, in particular, its usage for implementing a standard library.

REFERENCES

Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E. Porter. 2019. X86-64 Instruction Usage among C/C++ Applications. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '19)*. Association for Computing Machinery, New York, NY, USA, 68–79. https://doi.org/10.1145/3319647.3325833

Massimo Bartoletti and Livio Pompianu. 2017. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *International conference on financial cryptography and data security*. Springer, 494–509.

Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469.

Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *ArXiv* abs/1809.03981 (2018).

Stefanos Chaliasos, Arthur Gervais, and Benjamin Livshits. 2022. *Artifact: A Study of Inline Assembly in Solidity Smart Contracts.* https://doi.org/10.5281/zenodo.7071281

Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 123 (oct 2021), 30 pages. https://doi.org/10.1145/3485500

Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 309–326.

Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 530–541.

Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

Robert Feldt and Ana Magazinius. 2010. Validity threats in empirical software engineering research-an initial survey.. In *Seke*. 374–379.

Aviral Goel, Pierre Donat-Bouillud, Filip Křikava, Christoph M. Kirsch, and Jan Vitek. 2021. What We Eval in the Shadows: A Large-Scale Study of Eval in R Programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 125 (oct 2021), 23 pages. https://doi.org/10.1145/3485502

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

Péter Hegedűs. 2019. Towards analyzing the complexity landscape of solidity based ethereum smart contracts. *Technologies* 7, 1 (2019), 6.

Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the Eval That Men Do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 34–44. https://doi.org/10.1145/2338965.2336758

Stephen Kell, Dominic P. Mulligan, and Peter Sewell. 2016. The Missing Link: Explaining ELF Static Linking, Semantically. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA,

607–623. https://doi.org/10.1145/2983990.2983996

Benjamin Mariano, Yanju Chen, Yu Feng, Shuvendu K Lahiri, and Isil Dillig. 2020. Demystifying loops in smart contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 262–274.

Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. 2012. Eval Begone! Semi-Automated Removal of Eval from Javascript Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 607–620. https://doi.org/10.1145/2384616.2384660

Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf.

Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

Andrea Pinna, Simona Ibba, Gavina Baralla, Roberto Tonelli, and Michele Marchesi. 2019. A massive analysis of ethereum smart contracts empirical study and code metrics. *IEEE Access* 7 (2019), 78194–78213.

Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Matthieu Lemerre, Laurent Mounier, and Marie-Laure Potet. 2021. *Interface Compliance of Inline Assembly: Automatically Check, Patch and Refine.* IEEE Press, 1236–1247. https://doi.org/10.1109/ICSE43902.2021.00113

Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier, and Marie-Laure Potet. 2019. Get Rid of Inline Assembly through Verification-Oriented Lifting. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 577–589. https://doi.org/10.1109/ASE.2019.00060

Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–78.

Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1806596.1806598

Manuel Rigger, Stefan Marr, Bram Adams, and Hanspeter Mössenböck. 2019. Understanding GCC Builtins to Develop Better Tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 74–85. https://doi.org/10.1145/3338906.3338907

Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. 2018. An Analysis of X86-64 Inline Assembly in C Programs. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Williamsburg, VA, USA) *(VEE '18)*. Association for Computing Machinery, New York, NY, USA, 84–99. https://doi.org/10.1145/3186411.3186418

Solidity. 2022. Solidity Inline Assembly Documentation. https://docs.soliditylang.org/en/latest/assembly.html.

Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First monday* (1997).

Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 9–16.

Roberto Tonelli, Giuseppe Destefanis, Michele Marchesi, and Marco Ortu. 2018. Smart contracts software metrics: a first study. *arXiv preprint arXiv:1802.01517* (2018).

Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.

Gavin Wood. 2022. Ethereum: A secure decentralised generalised transaction ledger. (2022).

Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2084–2106.