Informatics

Mining of Smart Contract Patterns

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Michael List, BSc
Matrikelnummer 01328836

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr.-Ing. Stefan Schulte
Mitwirkung: Avik Banerjee, MSc

Wien, 10. Oktober 2025

Michael List	Stefan Schulte

Informatics

Mining of Smart Contract Patterns

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Michael List, BSc
Registration Number 01328836

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr.-Ing. Stefan Schulte
Assistance: Avik Banerjee, MSc

Vienna, October 10, 2025

Michael List Stefan Schulte

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Michael List, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 10. Oktober 2025

Michael List

Acknowledgements

I would like to thank my supervisors, Prof. Stefan Schulte and Avik Banerjee, for their support and guidance throughout this thesis. Their time, advice, feedback, and (maybe most importantly) their patience were essential for getting this finished.

I am also very grateful to my partner, who was always there for me and kept me motivated when things became tough.

Thanks as well to my employer and my colleagues, who gave me the flexibility I needed and often had a kind word when I was trying to balance work and studies.

Finally, I want to thank my friends, many of whom studied alongside me and shared the ups and downs of this journey. Their encouragement and shared motivation made the process much easier and more fun.

And last but not least, I also want to immortalize our cat and our hamster in these pages for keeping me company during countless late nights of writing.

Kurzfassung

Ethereum ist seit Jahren die größte Smart-Contract-Blockchain und nach Bitcoin die zweitgrößte Blockchain-Plattform. Smart-Contracts, die als dezentrale Anwendungen beschrieben werden können, laufen auf einer gemeinsamen Rechenplattform, auf der alle Teilnehmer auf einer geteilten Codebasis arbeiten. Zur Absicherung ist es nötig, dass ein Konsens über die Ein- und Ausgaben aller Smart-Contracts geschaffen wird. Die Ausführung von Smart-Contract-Code verbraucht sogenannte Gas-Einheiten, die als eine Art Treibstoff betrachtet werden können. Gas-Einheiten zeigen den erforderlichen Rechenaufwand an und haben direkte Auswirkungen auf den realen Energieverbrauch. Daher sollten idealerweise alle Smart-Contracts so implementiert sein, dass sie möglichst wenig Gas-Einheiten verbrauchen. Derartige Codeoptimierungsansätze sind nicht trivial. Zum Zeitpunkt des Verfassens dieser Diplomarbeit gibt es bereits solche Mechanismen, welche teilweise direkt in den gängigen Compilern integriert sind. Solche Mechanismen basieren in der Regel auf festen Mustern, welche manuell beschrieben werden müssen und dann auf Smart-Contracts angewendet werden können.

In dieser Arbeit haben wir untersucht, ob klassische Verfahren zur Erkennung von Codeähnlichkeiten verwendet werden können, um Optimierungsmuster automatisch aus Quellcode-Repositories ableiten zu können. Zunächst haben wir einen Symbolic-Execution-Ansatz untersucht, welcher sich aufgrund von technischen Einschränkungen und der Abhängigkeit von veralteten Compiler-Versionen als ungeeignet erwies. Daraufhin haben wir einen Fingerprinting-Ansatz basierend auf Kontrollflussgraph-Blöcken gewählt. Mithilfe von Slither konnten wir Metriken wie Cyclomatic-Complexity, Fan-Out und Informationsfluss-Metriken extrahieren und anschließend Distanzen zwischen Codestücken berechnen, um mit den Ergebnissen potenzielle semantische Code-Klone zu erkennen.

Wir haben die Evaluierung unseres Ansatzes auf 1.200 manuell markierten Smart-Contracts aus einem Datensatz mit 160.000 Einträgen durchgeführt, was zu 574 Vergleichen führte und konnten eine korrigierte Genauigkeit von 88% für die Erkennung von semantischen Code-Äquivalenzen auf Blockebene erzielen. Für 1.300 Code-Paare haben wir zusätzlich eine Gasverbrauchsmessung durchgeführt, indem wir die Blöcke in generierte Smart-Contracts verpackt und auf einer lokalen Blockchain ausgeführt haben. Dabei konnten wir tatsächliche gasreduzierende Codeänderungen identifizieren. Trotz einiger wesentlichen Einschränkungen zeigt das, dass das Mining gasoptimiertem Codes aus versionierten Source-Code-Repositories mittels Code-Metriken möglich ist.

Abstract

For years, Ethereum has been the largest smart contract blockchain and the second-largest public blockchain platform, after Bitcoin. Smart contracts, which can be described as decentralized applications, are executed on a mutual computation platform, with all participants operating on a shared codebase. For validation purposes, consensus must be achieved on the inputs and outputs of all smart contract calls. The execution of smart contract code consumes gas units, which can be considered analogous to fuel. Gas units represent the computational effort expended and directly impact the real-world energy consumption. Therefore, all smart contracts should aim to consume as little gas as possible in an ideal scenario. Such code optimization approaches are not trivial, however. At the time of writing, mechanisms to achieve this already exist, as described in related work, and some are directly integrated into the most popular code compiler. Those mechanisms utilize fixed code patterns, which must be manually formulated and can then be applied to smart contracts.

In this thesis, we investigated whether classical code similarity detection methods can be used to identify optimization patterns from a repository of source code automatically. We first explored symbolic execution using Oyente, but due to technical limitations and its dependency on outdated compiler versions, the approach proved to be restricted in practice. We therefore shifted towards a Control Flow Graph (CFG) fingerprinting approach. Leveraging Slither, we extracted metrics including cyclomatic complexity, fan-out, and information flow metrics, and then computed distances between code fragments to detect potential semantic clones.

We evaluated our approach on 1,200 manually labeled contracts from a 160,000-contracts dataset, resulting in 574 comparisons. After a manual error correction, we achieved a precision of 88% for semantic equivalence computation at the CFG block level. For 1,300 code pairs we further measured gas consumption by deploying generated wrapper contracts to a local blockchain. We were able to identify actual gas-saving code changes, which demonstrates, besides some major limitations, that mining gas-optimized code from versioned code repositories is feasible by leveraging structural code metrics.

Contents

CHAPTER 1

Introduction

Blockchain technology started gaining popularity in 2008 when Satoshi Nakamoto introduced his proposal for a peer-to-peer electronic cash system to the public, which he named Bitcoin [Nak08].

Based on cryptographic hash functions and asynchronous cryptography (digital signatures), this was the first credible peer-to-peer cryptocurrency [TS16]. Designed without the need for a central financial institution, payments can be sent directly and electronically. Different from traditional currencies, Bitcoin operates on top of the entire transaction history of all accounts. Before Bitcoin, the problem of double spending prevented a trustable distributed payment solution. Double spending would allow using more money than there is available, both on the local account and globally. Without a central entity, no one could be certain that the same digitally signed incoming payment was not also sent to other payees. To overcome the necessity for a payment institution, Satoshi Nakamoto solved the problem of double spending by hashing sets (known as blocks) of transactions, and including a reference to the previous hash. This forms a chain of blocks, known as "blockchain". The blocks also contain a puzzle value, which is hard to compute and involves hashing the block multiple times. The result of the hash function would change as soon as any block content is changed. This serves as an integrity guarantee. Building on this idea, the concept behind Bitcoin is that the blockchain produced by a majority of total computing power will be considered the valid one, thereby providing credible traceability for every coin in circulation. Carrying over blocks into the blockchain this way is called Proof-of-Work (PoW) mining. As long as the majority jointly works on validating and extending the blockchain, its integrity is guaranteed [Nak08].

More blockchain-based cryptocurrencies have emerged by now. The second most relevant (measured by market capitalization[1]) is Ethereum, which was developed by Buterin et al. [But14], and is currently the largest smart contract-supporting blockchain platform.

[1]As of 2025-10-07 according to `https://coinmarketcap.com/`

Bitcoin was created for financial transactions, with a focus on coin ownership and the innovative blockchain technology. Ethereum aims to improve on Bitcoin by providing the ability to create and conclude arbitrary binding contracts. The Ethereum Virtual Machine (EVM) is Turing-complete. In the Ethereum world, smart contracts are programs that are deployed and executed on the blockchain as part of a transaction. As in Bitcoin, transactions are final by application of the consensus mechanism and cannot be changed. Gas units must be paid as part of transaction fees, and, intuitively, smart contract developers need to keep their gas costs low, i.e., need to aim for gas optimization, to minimize transaction fees and optimize the computational power consumed. With 128.33 to 131.00 billion gas used per day, gas fees oscillated between 314.61k USD and 2.64M USD for a single day in May 2025[2]. This indicates a significant savings potential for the gas optimization use case.

Although the fixed costs per instruction are known, it is non-trivial to use the optimum (i.e., the lowest possible) amount of gas during smart contract engineering. Additionally, the EVM rolls back transactions whenever their specified gas limit parameter has been exceeded. In such cases, the full gas costs up to the limit are charged. Setting the limit too high, however, increases the risk of long-running loops or external contract calls triggering unnecessarily high charges.

In recent years, there has been growing interest in researching the optimization of smart contract gas. For example, Brandstätter et al. [BSCB20] have examined static optimization strategies using fixed rules to reduce gas consumption and found optimization potential in over 6% of their sample set of smart contracts.

Some other approaches work on the bytecode level, irrespective of the high-level programming language used. For example, Chen et al. [CLZ+18] used fixed anti-pattern detection and replacement as early as 2018. Others work on more sophisticated ideas, as Nagele et al. [NS19], whose "ebso" increases gas efficiency by utilizing SMT solving.

Most of the Ethereum-related sources referenced in Chapter 3 have encountered the situation of observing non-optimized smart contracts in real-world use, which underscores the significance of this research. To the best of our knowledge, to date, research that mines existing Solidity code for gas-saving patterns is still in its infancy. Code mining is a technique that applies data mining methods to source code repositories, aiming to identify patterns or rules within the code. While other approaches, such as machine learning exemplified by Siamese neural networks [BES24], learn from annotated examples, code mining follows a white-box paradigm: the exact criteria for pattern extraction are predefined and transparent, unlike black-box models, whose internal reasoning remains opaque to the user. This difference was highlighted by Atzmueller et al. [AFKS24], who explain black-box vs. white-box models. Finding optimization rules through code mining could potentially improve the smart contract gas optimization situation, as rules would no longer need to be manually found and formalized. We can theorize that version-managed

[2] `https://ycharts.com/indicators/ethereum_gas_used_per_day` and `https://defillama.com/fees/ethereum`, accessed on June 28th, 2025

smart contracts already contain such optimizations, comparing code revisions to their respective previous revisions. Developers are continually working to enhance program efficiency, and numerous open-source smart contracts already exist. Program code is versioned in most cases nowadays, although it is not necessarily publicly available. The key aspect here is the extraction of the syntactical optimization changes that strictly preserve semantics. Identifying and generalizing such functionally similar syntactical patterns is not a trivial task.

In this work, we aim to develop a code mining approach to contribute to improving the situation towards optimal gas consumption. Using a large repository of versioned Solidity contracts, we design and evaluate two strategies: an initial symbolic-execution prototype with Oyente, and a final control-flow-based fingerprinting approach using Slither and CFG metrics. Our evaluation shows that the latter can successfully identify semantically equivalent code fragments and reveal their relative gas optimization, although several challenges remain.

The structure of this thesis is organized into seven chapters as follows:

1. **Introduction**: Provides an introduction about Ethereum, gas usage, and a problem overview.
2. **Background**: Covers blockchain essentials, focusing on an in-depth description of Ethereum and aspects relevant to this thesis.
3. **Related Work**: Provides an overview of related approaches, as well as gas optimization and code mining in the literature.
4. **Methodology**: Describes the research method used in detail.
5. **Implementation**: Describes the technical application of the approach.
6. **Evaluation**: Presents and discusses the findings, outcomes, and limitations of this thesis.
7. **Conclusion**: Reflects on implications of the results and outlines possible future work.

CHAPTER 2

Background

In this section, we will explore the foundational concepts of smart contracts and code optimization. Starting with a basic introduction to cryptocurrencies and blockchain technologies, we come to smart contracts in more detail and provide an overview of the concepts of code optimization and code mining.

2.1 Blockchain

Since the late 1990s, researchers have been seeking a decentralized digital currency solution[1]. Various problems remained unsolved until Satoshi Nakamoto introduced the first usable project in 2008. He combined existing partial solutions to a working Bitcoin idea [TS16].

In the following subchapters, we go into detail about the substantial technology that Bitcoin and subsequently developed cryptocurrencies – especially Ethereum – are built upon.

2.1.1 Digital Signatures

Analogous to handwritten signatures, digital signatures serve the purpose of authentication, data integrity, and non-repudiation [Nat23]. Based on public key cryptography, the core principle works on two types of keys. Anyone can use the public key to verify digitally signed data that has previously been signed using the associated (secret) private key [Sta14]. Different cryptosystems have been invented. Rivest–Shamir–Adleman (RSA) is based on the prime factorization problem, whereas ElGamal and Digital Signature Algorithm (DSA) are based on the discrete logarithm problem. Particular discrete logarithm-based cryptosystems, generally more efficient for the same key size, refer to computation in elliptic curves as Elliptic Curve Digital Signature Algorithm (ECDSA) [AZZZ19].

[1]as b-money[Dai98] in 1998, also [BMC+15]

Formally, digital signatures work as follows [Sta14]: Given are a plaintext P, signature S, forged signature S', secret key sk, public key pk, signing function $\text{sign}(SK, P)$, and verification function $\text{verify}(PK, P, S)$. $\text{sign}(SK, P)$ transforms the plaintext and secret key to a signature by applying the cryptographic signature function, whereas $\text{verify}(PK, P, S)$ applies the cryptographic verification function to the signature and public key, resulting in either verification success (1) or verification fail (0), depending on whether the verification function considers the signature as constructed by the plaintext and the corresponding secret key. Applying the function computes the following result: $\text{sign}(sk, P) = S$, $\text{verify}(pk, P, S) = 1$, $\text{verify}(pk, P, S') = 0$

2.1.2 Hash Functions

Hash functions generally take an infinitely large input and produce a fixed-size output, using reasonably small resources. The same output is produced for the same input [Sta14]. Considering well-designed hash functions, the following properties hold [NBF+16].

- **Collision-freeness**: For two different inputs, they are extremely unlikely to produce the same hash value, even if they differ by only 1 bit.
- **Hiding**: The input cannot be easily constructed from the output.
- **Puzzle-friendliness**: No input can be easily constructed, such that the output is predictable.

In general, hash functions are designed to be efficient; however, there exist ones, such as `bcrypt`, that are intentionally designed to be inefficient, for example, for hashing passwords, at least inefficient enough to compute a lot of them at once, i.e., to prevent brute-forcing password hashes.

For the following hash function property definitions, $H()$ is referred to as the hash function, I, I', $\ldots$ as inputs, O, O' as outputs. $||$ means concatenation. Min-entropy is a measure of unpredictability and focuses on the most likely outcome. Thus, high min-entropy means that the likelihood of the most probable value is very low. The input values should follow a probability distribution that is widely spread out to minimize the success of any guessing strategy, as mentioned in Cachin's dissertation [Cac97].

Collision-free

It is infeasible to find I, I' such that $I \neq I'$ and $H(I) = H(I')$.

Due to the hash function's nature (input produces fixed-size output), collisions exist because the number of possible outputs is limited, whereas infinitely different inputs are accepted.

The property can be measured by the amount of work needed to find a collision, which is, for hash functions with an output length of N, in general $2^{\frac{N}{2}}$ [Dan12]. An attack that takes advantage of hashes not fulfilling this property is the birthday attack, named after the birthday paradox from probability theory [MVOV96].

Hiding

Considering $H(I^U) = O^K$, when O^K is known, it is infeasible to reconstruct the unknown input I^U.

Naturally, this can only hold when the range of possible inputs is sufficiently large. For example, plain hash function usage is not sufficient to hide two possible input symbols $Heads$ and $Tails$ (an attacker would have to compute only two hashes to infer the input). This can be circumvented by using a "salt" value S, i.e. $H(I^U||S)$, when S is chosen from a high min-entropy input range (useful for e.g. password hashing hardening and commitment submission schemes) [NBF+16].

Puzzle-friendly

Considering $H(I^K||I^U) = O$, it is infeasible to find I^U in a way, such that O can be predetermined when I^K is from a high min-entropy input range.

In other words, a specific hash function fulfills this property when there is no strategy better than brute-forcing to solve such a puzzle [NBF+16].

Hash Algorithms

The Secure Hash Algorithm (SHA) family comprises some of the most significant hash algorithms currently in use. Succeeding the initial widely adopted hash functions Message Digest 5 (MD5) (designed in 1992) and SHA-1 (1995), SHA-2 (2001) and SHA-3 (2015) are widely used for data verification and digital signature purposes [DCD17].

SHA-3 has been designed as a successor to SHA-2 and provides data integrity and authentication properties, specifically the three necessary hash properties mentioned above. It employs a "sponge construction", which distinguishes it from its predecessors (MD5, SHA-1, and SHA-2 were based on the Merkle-Damgård construction, a design principle introduced independently by Merkle [Mer90] and Damgård [Dam90] in 1989)[AKDB10].

Ethereum uses the Keccak256 algorithm [Woo14], which is an earlier version of the NIST-standardized SHA3-256 [Nat15] and differs from the standard only by a minor padding change[2]. The hash function outputs are different and cannot be compared.

2.1.3 Hash Pointers and the Blockchain

Hash pointers refer to the hash of some data, which implicitly and unalterably points to that specific data. It can be used to create a link between data sets by including the

[2]Can be visualized in this library's source code `https://github.com/emn178/js-sha3/blob/c43f9d1d18f9bd220d77c05926122711cb41ad41/src/sha3.js#L23-L24`.

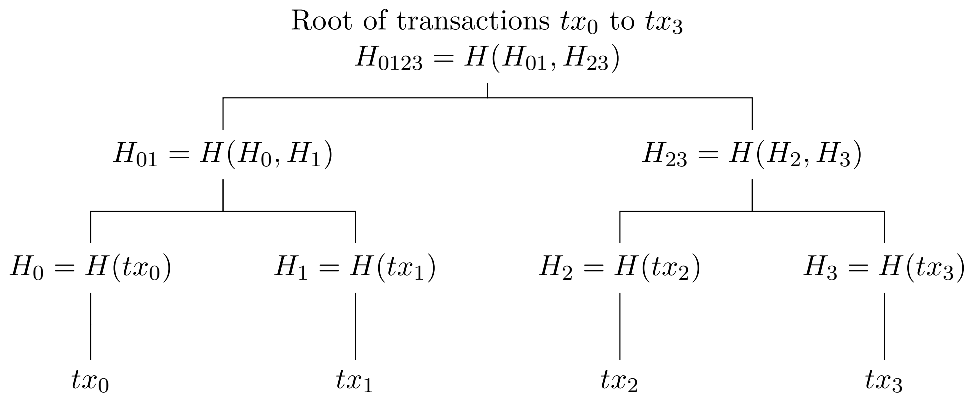


Figure 2.1: The Structure of a Merkle tree

hash of one data set within another. The second set will then be logically connected to the original state of the first one. This concept can be used to create a data structure, for example, by taking a linked list and using hash pointers within the data structure to link the elements. This effectively results in a "blockchain", with data packed into blocks, connected like a chain. A block points to its previous block using its hash, which effectively breaks the connection whenever a change occurs in an earlier block's data, thereby invalidating the blockchain.

This principle can also be applied to a binary tree. Merkle trees (also known as hash trees) are an efficient method for verifying the integrity of larger data structures. All nodes contain a hash of the data they point to, with the inner nodes pointing to their children. All leaf nodes are assigned to a data block and contain hashes of the respective data, and all non-leaf nodes contain hashes of their children. This allows the whole structure to be compared by checking only the top-level hash. Equally, a data block can be verified to belong to a specific Merkle tree by validating only $log(n)$ of n total tree nodes (which is ideally the tree height). Figure 2.1 visualizes the structure of a Merkle tree consisting of the four transactions tx_0 to tx_3. For their usage in blockchain cryptocurrencies, the transaction root, in this case H_{0123}, would be included within a data field in a block.

Bitcoin's blockchain is based on Merkle trees. The blocks contain transaction data. A user can verify that a specific transaction is in the blockchain by checking if the transaction hash is part of the block's Merkle tree hash [Nak08]. Ethereum uses Merkle-Patricia tries, a further developed data structure based on hash trees [Woo14].

2.1.4 Consensus Algorithms

Achieving decentralization when designing an electronic currency is a non-trivial task. To prevent double spending, the easiest solution is to rely on a central authority that validates all transactions and maintains a secure record of them. However, such centralization opposes the idea of a trustless system. Decentralized cryptocurrencies, however, employ consensus algorithms, which make sure that all nodes in the network agree on a single, tamper-proof transaction history – without the need for a trusted third party [TS16].

Different consensus mechanisms have been proposed to achieve this goal. The most well-known among these are PoW and Proof-of-Stake (PoS), currently used by Bitcoin and Ethereum, respectively, each offering distinct trade-offs in terms of security and scalability [XZLH20].

The concept of PoW, as currently used by Bitcoin, follows the principle of one-CPU-one-vote. It can achieve distributed consensus by probabilistically assigning incentives to honest nodes, based on their computational effort. While initially proposed as an email spam prevention mechanism [DN93], the consensus algorithm gained widespread popularity with the emergence of cryptocurrencies. In his manifesto, Nakamoto writes: "The system is secure as long as honest nodes collectively control more CPU power than any cooperating group of attacker nodes" [Nak08]. This highlights the PoW-based consensus core assumption.

To participate, nodes must solve a computationally intense puzzle, typically by finding a nonce that, hashed together with prepared data, produces a specific hash output. Depending on the requirements, the allowed output format can be adapted, which makes the difficulty tunable.

The goal is that no one can monopolize the distributed application, and the cost needed for participating aims to prevent exactly that. This leads to a concept where malicious actors need to gain substantial computational resources, accounting for over 50% of the computing power.

While PoW remains fundamental today in major blockchain systems, such as Bitcoin, its significant disadvantage is high energy consumption. Since the launch of Bitcoin, alternative consensus mechanisms have emerged – most notably PoS, which has been used in the Ethereum main blockchain since September 2022 and has reduced its energy consumption by around 99.95%[3]. PoS replaces computational effort with economic commitment. Participants can become validators by committing and locking a specific amount of cryptocurrency (32 ETH for Ethereum). In return, they earn rewards for proposing and validating blocks[4], similar to the mining incentives in PoW.

2.1.5 Bitcoin

Using the previously introduced concepts – such as hash functions, digital signatures, and distributed consensus mechanisms – Bitcoin was proposed by Satoshi Nakamoto in 2008. By introducing a PoW-based protocol, Nakamoto resolved the double-spending problem, thereby enabling the first decentralized digital currency [TS16].

Bitcoin transactions use a simple stack-based scripting language to define spending conditions. The payment logic is embedded within the "scriptPubKey" field inside

[3]`https://ethereum.org/en/roadmap/merge/`
[4]`https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/`

transactions, typically as a *pay-to-pub-key-hash* (P2PKH) script. Transactions intending to use the monetary output of an existing transaction must provide an unlocking script, defined in the “scriptSig” field. During transaction validation, both scripts are executed on the same stack, and the transaction is considered valid if the script execution is successful. While the scripting language supports 186 opcodes and enables basic programmable mechanisms such as multi-signature checks, timed locks, and conditional payments, it is not Turing-complete. As a result, Bitcoin does not support the execution of smart contracts [BMC+15].

2.2 Ethereum

> I happily played World of Warcraft during 2007-2010, but one day Blizzard removed the damage component from my beloved warlock’s Siphon Life spell. I cried myself to sleep, and on that day I realized what horrors centralized services can bring. I soon decided to quit.
>
> Vitalik Buterin [But]

In around 2010, Vitalik Buterin became disillusioned with central control when a privately operated game company unilaterally revoked certain in-game functionalities. This formative incident made him aware of broader vulnerabilities originating from “government regulation and corporate control”, and ultimately became a key motivation behind his vision of a decentralized platform. In late 2013, he published the concept of Ethereum [But14], and the first version went online in 2015.[5] Buterin and his team have continued to develop the distributed technology. Ethereum runs on the PoS consensus mechanism since “The Merge” in 2022[6].

Ethereum is a blockchain platform designed to support decentralized applications. In contrast to Bitcoin, where the primary focus lies on secure decentralized value transfer, with limited logic support, Ethereum supports smart contracts, which can be used to run Turing-complete programs. The execution environment is called EVM and can handle arbitrarily complex applications.

Transactions can contain not only monetary value, but also data input and bytecode. All operations, including smart contract bytecode, are executed on all full nodes to ensure consensus [Fou25] [Woo14].

Ethereum uses an account-based model, with the two types being: Externally-owned accounts (which can only be controlled using the corresponding private key) and contract accounts. Both account types use the same 20-byte address format and can interact with each other.

[5]2016 interview and backstory by Morgen Peck, see `https://www.wired.com/2016/06/the-uncanny-mind-that-built-ethereum/`

[6]Ethereum Foundation’s roadmap description, specifically “The Merge” `https://ethereum.org/en/roadmap/merge/`

Ethereum's base unit is Ether currency code (ETH), whereas its tiniest fraction, 1 Wei, equals 10^{-18} ETH, as seen in Table 2.1.

Table 2.1: Conversion between ETH, Gwei, and Wei

Unit	**Value in Wei**
1 ETH	1 000 000 000 000 000 000 (= 10^{18} Wei)
1 Gwei	1 000 000 000 (= 10^{9} Wei)
1 Wei	1 Wei

2.3 Smart Contracts

In this section, we go into the details of Ethereum smart contracts. While other blockchain platforms – such as Solana [SGW+23] – also provide support for Turing-complete code execution, this thesis focuses exclusively on Ethereum. Table 2.2 describes smart contract platform popularity[7] on GitHub. The EVM-based platform remains the most widely adopted and well-known smart contract platform, which is also well-supported by tools.

Table 2.2: GitHub search result count for relevant smart contract platforms as of June 30th, 2025

GitHub repository search text	**Search result count**
"Ethereum"	81.5k
"Fabric"	51.3k
"Solana"	44.8k
"Binance"	39.8k
"Waves"	38.5k
"NEM"	22k
"Stellar"	20.1k
"Cardano"	5.6k
'Corda"	2.2k

Buterin explained smart contracts in [But14] as "systems which automatically move digital assets according to arbitrary pre-specified rules". While technically, the contracts could do anything a conventional application can do, smart contracts are advantageous when digital money gets involved [BP17]. Tokens of different types can be created [MCT+23]; other applications include token escrow services [AK19], NFT assets [WLWC21], DAOs swap exchanges, interest management [JQW+23], loan providers, and numerous others [HLM+19].

[7]Relevant according to [HYL21] or part of the top3 (excluding Bitcoin) from Coingecko's Top Smart Contract Platform Coins by Market Cap, as of June 30th, 2025

Smart contracts deployments and executions take place within a transaction, and thus cannot be changed after the corresponding transaction becomes a part of the blockchain. Any Ethereum account (even contracts) can create and access contracts [Woo14]. For economic and technical reasons (e.g., to prevent infinite loops and account for resource utilization), all instructions in Ethereum smart contracts are assigned a price – so-called gas units. All consumed gas units correspond to a piece of the base currency Ether (ETH), where the exact price is variable [Fou25]. In May 2025, the average gas price oscillated between 1.60 and 10.61 Gwei[8]. Each EVM instruction (opcode) has a predefined gas cost value, and transaction calls need to contain a *gasLimit* and *gasPrice*. If execution exceeds the gas limit, the EVM rolls back the transaction and does not refund the spent gas. For both deploying and executing smart contracts in the form of EVM instructions, gas fees will be deducted from the Ethereum account that invokes the transaction [Woo14].

EVM bytecode does not have to be written by hand. The oldest and still most popular EVM programming language is Solidity, with Vyper, Yul, and Fe gaining traction[9]. In this thesis, we work with Solidity. An example of Solidity code can be seen in Listing 2.1.

Listing 2.1: Smart contract: Example contract from `https://solidity-by-example.org/variables/`

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

contract Variables {
    // State variables are stored on the blockchain.
    string public text = "Hello";
    uint256 public num = 123;

    function doSomething() public view {
        // Local variables are not saved to the blockchain.
        uint256 i = 456;

        // Here are some global variables
        uint256 timestamp = block.timestamp; // Current block timestamp
        address sender = msg.sender; // address of the caller
    }
}
```

When deploying a smart contract, the entire contract – compiled into bytecode – is submitted via a transaction. Storage operations are among the most expensive EVM opcodes. Therefore, contract creation is particularly gas-intensive because the bytecode is stored permanently in the blockchain, and the constructor is executed during this

[8]`https://ycharts.com/indicators/ethereum_average_gas_price`, accessed on June 28th, 2025

[9]`https://ethereum.org/en/developers/docs/smart-contracts/languages/`

process. Larger contracts incur higher gas costs, as they contain more bytecode; the cost increases linearly. The variable $G_{codedeposit}$, which represents the deployment cost per byte, is defined as 200 gas by the Ethereum Yellow Paper [Woo14]. Therefore, minimizing the contract size is a crucial consideration for gas-efficient deployment.

Function calls, however – such as an external account or a contract invoking another contract – do not necessarily modify the global state and can be relatively inexpensive [Woo14]. Functions marked as view or pure can even be executed off-chain – locally – which means free of charge[10]. However, when a state change is intended, submission using a transaction is required. Gas is consumed depending on the executed opcodes and is capped at the transaction's *gasLimit* [Woo14]. In some cases, computationally expensive logic can be executed off-chain; however, developers should avoid unnecessary state modifications, and the code should generally run as efficiently as possible in terms of gas usage.

The EVM is stack-based and operates on opcodes. It operates on three distinct data structures: Stack, memory, and storage. The stack is used for short-term operational cache, and access is cheap. Memory is a storage space that exists throughout the entire transaction execution, while storage refers to the contract's persistent state, which is directly stored on the blockchain [Woo14]. For a comparison of writing costs, see Table 2.3. It is essential to note that the stack is not as usable as the others; this column is provided for theoretical illustration purposes. Storage operations are the most expensive, as they modify or access the global state; however, under certain conditions, clearing values within storage can result in a partial gas refund. Efficiency in all storage types is crucial for optimal gas utilization.

Table 2.3: Gas costs for storing data in the EVM stack, memory, and storage

Data Size	**Stack (PUSH)**	**Memory (MSTORE)**	**Storage (SSTORE)**
1 byte	3 gas	$\sim$ 6 gas	5 000–20 000 gas
32 bytes	3 gas	$\sim$ 6 gas	5 000–20 000 gas
256 bytes	24 gas	$\sim$ 48 gas	40 000–160 000 gas

Successful transactions can alter the global state, transfer Ether, and emit events, among other effects. While an erroneous transaction is rolled back, the entire gas price is consumed by the EVM; for successful transactions, any unused gas is refunded to the transaction sender [Woo14].

2.4 Optimization

Code optimization is the process of modifying a program to enhance specific performance aspects (such as faster execution or reduced resource utilization), without altering its

[10]https://docs.soliditylang.org/en/v0.8.30/cheatsheet.html#modifiers

functional behavior. Generally, no program can be optimized simultaneously across all metrics. More often than not, the programmer must choose what to optimize, and this frequently leads to a degradation in other, sometimes overlooked metrics [BML+21] [GVB+25] [WO18] [KSK+18].

Programs can be optimized at different stages, from source code to compiler intermediate representation (IR), and also at the binary level. For example, compilers apply low-level optimizations, such as instruction scheduling, register allocation, and loop unrolling. Developers, however, will in general focus on high-level optimizations, such as selecting more efficient algorithms or data structures at the source-code level, to improve performance [GVB+25]. In this thesis, we look at optimizations on the source code level.

2.4.1 Smart Contract Optimization

In this thesis, we focus on gas-aware transformations of Ethereum smart contracts. As a background introduction to this topic, we are now addressing optimization on this more specialized level.

One of the most critical metrics for Smart contracts is gas consumption. Directly related to the monetary execution cost, optimizations in this area have gained attention.

Chen et al. [CXL+21] and Zou et al. [ZLK+21] claim that gas optimizations can negatively impact code readability. This means that writing highly readable Solidity code is challenging because it may increase gas costs. Similarly, gas efficiency decreases when using security-related safety checks, such as OpenZeppelin's SafeMath. Omitting such checks, however, can introduce vulnerabilities, as thoroughly analyzed by Gao et al. [GLL+19].

Besides achieving the lowest possible fees, gas-aware optimization also benefits Ethereum's gas limit, which exists at both the block level and transaction level. If a transaction's gas usage, or the sum of transactions in a block, exceeds this limit, the transaction will be excluded or reverted, leading to failed executions and delayed inclusion [YMRP19]. Due to these gas constraints, static analysis and code-mining tools (e.g., GASOL [ACG+20]) can help developers infer precise upper bounds on gas consumption and apply transformations that reduce per-call gas usage.

The levels at which gas optimizations can be achieved are defined as follows:

1. **Deployment-Level (Contract Architecture)**, where architectural patterns are used to minimize the gas cost of creating contracts, as the Minimal Proxy Contract proposed in ERC-1167 by Murray et al. [MWM18].

2. **Source Code-Level**, where anti-patterns, as loops or unused local variables, are detected and modified before compilation. Brandstätter et al. [BSCB20] analyze 25 classical source-level optimization rules (e.g. loop unrolling, constant folding, caching) directly in Solidity

3. **IR-Level (Intermediate Representation)**, where in Ethereum's coding stack, YUL was built to be "suitable for whole-program optimization"[11]. YUL is an intermediate language between bytecode and Solidity, supporting optimization modules such as *CommonSubexpressionEliminator* (replaces subsequent identical expressions with a reference to the same computed result) and *ExpressionSimplifier* (for constant folding and constant propagation)[12].

4. **Bytecode-Level (Compiler)**, where the default Solidity compiler *solc* applies built-in-optimizations, and also works like Nagele & Schett's `ebso` [NS19] focus on. They are transforming EVM code into SMT formulas, which are converted back to cheaper bytecode after being processed by SMT constraint solvers.

Regarding a complete overview of existing approaches to gas optimization, see the related work discussed in Chapter 3.

2.5 Program-Analysis Foundations

The optimization of smart contracts and the detection of structural code similarities eventually rely on an understanding of program behavior and control. The basis for describing this behavior is concepts from program analysis, such as CFGs and symbolic execution.

2.5.1 Control Flow Graphs

To reason about programs, a structural representation is needed, beyond raw source code. In the 1970s, Frances E. Allen formalized the CFG, a fundamental concept in compiler construction and static analysis. She introduced a systematic way to represent a program not as a sequence of instructions, but as a directed graph of basic execution units (basic blocks) [All70].

A *basic block* is a sequence of instructions with its first instruction as the only entry point and the last instruction as the only exit point. This also means that no branches or jumps occur inside the block. The *CFG* of a program or function is a directed graph $G = (B, E)$, where vertices represent basic blocks and edges represent the control flow path [All70].

2.5.2 Software Complexity Metrics

Clone detection and gas-efficiency analysis require numerical features that grade programs or program units. Following Oman and Hagemeister's maintainability dimensions classification – *control structure*, *information structure*, and *lexical / typography* [OH92] – we group the metrics relevant to this thesis as follows:

[11] `https://docs.soliditylang.org/en/latest/yul.html`

[12] Solidity documentation `https://docs.soliditylang.org/en/latest/internals/optimizer.html`, section Yul-Based Optimizer Module

Control Structure

- **McCabe's Cyclomatic Complexity**

$$\text{MC\_CABE}(s) = E - N + 2P \tag{2.1}$$

Computed from a program's CFG, this metric measures the number of possible execution paths. With E and N denoting the edges and vertices of the CFG, and P the number of connected components ($P = 1$ for basic blocks), McCabe states that "the cyclomatic number is equal to the maximum number of linearly independent circuits" [McC76].

- **Fan-in/Fan-out**
Counts the incoming and outgoing calls of a procedure. Additionally, Henry and Kafura include information-flow connections in their definition, namely the number of data structures from which the procedure reads or writes [HK81]. Some metrics-based research, such as Kontogiannis et al. [KDMM+96], omits this information-flow component.
High fan-in and fan-out indicate that a program part is coordinating with a large number of other procedures, which may have a significant impact on the overall program or system [HK81].

- **S-Complexity**

$$\text{S\_COMPLEXITY}(s) = \text{FAN\_OUT}(s)^2 \tag{2.2}$$

Defined by Kontogiannis et al. [KDMM+96] as the square of the fan-out metric.

Information Structure

- **D-Complexity**

$$\text{D\_COMPLEXITY}(s) = \frac{\text{GLOBALS}(s)}{\text{FAN\_OUT}(s) + 1} \tag{2.3}$$

A ratio that contrasts two different kinds of external dependence. GLOBALS(s) describes the data coupling to s's environment(global variables read or written), whereas FAN_OUT(s) counts the external control-flow coupling (calls to other procedures). The metric, therefore, captures the balance between data coupling and control coupling.

- **Albrecht's function point metric (Kontogiannis variant)**

$$\begin{aligned}\text{ALBRECHT}(s) = {} & p1 \times \text{VARS\_USED\_AND\_SET}(s) \\ & + p2 \times \text{GLOBAL\_VARS\_SET}(s) \\ & + p3 \times \text{USER\_INPUT}(s) \\ & + p4 \times \text{FILE\_INPUT}(s)\end{aligned} \tag{2.4}$$

Defined by Albrecht [Alb79] in the context of team productivity, the "Function Point" terminology can also be applied to software. Kontogiannis et al. [KDMM+96] utilize the idea of a weighted sum over used and set variables, as well as user and file inputs.

- **Henry–Kafura information-flow metric (Kontogiannis variant)**

$$\text{KAFURA}(s) = (\text{KAFURA}_{\text{IN}}(s) \times \text{KAFURA}_{\text{OUT}}(s))^2 \tag{2.5}$$

with $\text{KAFURA}_{\text{IN}}(s)$ as the total dataflow going to the program fragment s and $\text{KAFURA}_{\text{OUT}}(s)$ as the total dataflow coming from the program fragment s. This metric quantifies the amount of information flow (both local and global) for code pieces [HK81].

Typography, Naming, and Commenting

- **LOC / NLOC**
  Total and *normalized* lines of code (with blank and comment lines removed) provide a baseline size indicator.

- **Token Count**
  The number of lexical tokens – as returned by the compiler's first ("lexing") phase – for a given function. Tokens can include language keywords, identifiers, literals, operators, and punctuation. Because token count is insensitive to formatting choices, such as line breaks or very long lines, it provides a more robust measure than LOC.

2.5.3 Clone Detection

A clone is a pair or set of source code fragments that are identical or nearly identical, to the point that a maintainer can recognize them as "the same" with minimal effort. Clones arise from copying and pasting, manually retyping, or automatically generating code. They are a form of software redundancy that can amplify bugs and increase maintenance costs [Kos07].

According to the highly cited survey by Roy and Cordy, software clone research commonly distinguishes between the four categories [RC07] listed in Table 2.4.

2.5.4 Symbolic Execution

A clever way of analyzing a program exhaustively provides us with a solution to reason about *all* its inputs without trying them one by one.

Symbolic execution achieves this by running a program with *symbolic* inputs instead of concrete values [Kin76]. For each possible path, it generates a formula – the *path condition* – whose set of logical constraints must be satisfied for the path to be feasible.

Table 2.4: List of clone category types [RC07]

Type	Definition
I – Exact	Textually identical except for whitespace or comments.
II – Parameterised	Identical after systematic renaming of identifiers or literals.
III – Near-miss	Copy with statements added, removed, or reordered.
IV – Semantic	Same functionality implemented with different syntax/algorithms.

An SMT solver such as Z3 [dMB08] then instantiates the symbols with concrete values that satisfy the constraints, producing test inputs or counterexamples for a real run. Because the technique allows for exploring many paths in a single run, it can achieve high coverage and expose deep bugs that would be difficult to identify manually [PC13].

2.6 Code Mining

Code mining is a branch of Mining-Software-Repositories research that treats source code itself as a dataset. According to Hassan [Has08], MSR considers all data available in software repositories, including commits and issues, in addition to source code. In contrast, the term code mining generally refers to source code artifacts as the basis for mining. Khatoon et al. [KML11] describe source-code mining as a set of techniques that leverage "data-mining methods to determine patterns from source code". Used as a method to automate bug and clone detection, code mining can turn large code bases into quality improvements.

Extracting code patterns and applying them to other software can be effective because developers often write idiomatic code. Allamanis and Sutton [AS14] define that "an idiom is a syntactic fragment that recurs frequently across software projects and has a single semantic purpose". There exist collections and guides for idioms in specific programming languages, and the authors also present a method to mine such code idioms. We hypothesize that our work will identify optimizations for numerous code idioms, and since these idioms are frequently used, the resulting optimizations will be highly beneficial.

2.6.1 Mining Subjects

Jung et al. [JLW12] distinguish between several software repository mining subjects, all applicable to code mining: source code (text and tree), natural languages, graphs, and vectors.

- **Text** refers to pure source code literal processing.
- **Tree**-based mining is more sophisticated and can refer to Abstract Syntax Trees (ASTs).

- **Natural language** mining artifacts can come from source code comments or constant strings.
- **Graphs** direct to relationships as calls or dependencies as well as CFGs.
- **Vectors** are referred to by the authors as several key attributes such as LOC, cyclomatic complexity, and number of parameters.

2.6.2 Applications

Several areas of software engineering have been subject to code mining already:

- **Bug Detection**
  By understanding the rules for creating bug-free code, pattern mining techniques can be applied to identify and flag violations. In their study, Li and Zhou [LZ05] successfully detected multiple bugs using their "PR-Miner", some of which were even confirmed in the latest Linux version at the time.
- **Code-smell Detection**
  Palomba et al. [PBP+15] describe five code smells that signalize bad code quality. They leverage code mining at the source code repository level to detect those smells.
- **Refactoring Suggestions**
  Idiom mining learns recurring syntactic fragments. Once mined, these idioms can be used as refactoring templates or code idiom inserts inside an IDE. Allamanis and Sutton's HAGGIS system [AS14] was able to mine such idioms from public code repositories.
- **Performance Analysis**
  Zhao et al. [ZGH+24] proposed "an approach that predicts performance bugs by leveraging various historical data". They were able to extract performance-related patterns from 80 popular Java projects.

CHAPTER 3

Related Work

Efficient smart contract execution, particularly Ethereum, depends on the effective management of its resource unit *gas*. Gas optimization has been an active research topic over the last few years, as can be seen in Section 3.2, due to its direct relationship with transaction costs. Different approaches have already been explored to mitigate gas costs by enhancing the efficiency of smart contracts.

The application of code mining techniques leverages methods from data mining to analyze and enhance software. Code mining has proven to be a particularly beneficial tool in understanding code patterns [AS14], detecting inefficiencies [PBP$^+$15], and suggesting targeted improvements [PKW08].

In recent years, code mining and blockchain technology have emerged as subjects of academic exploration, specifically for detecting vulnerabilities in smart contracts [WFTW24], [SQLH23]. More recently, code mining methodologies have been explored to optimize gas usage [BSS25].

This chapter aims to provide context for the methodology presented in the subsequent chapter and thereby reviews the existing literature across four subjects: foundational code mining techniques, code mining approaches for smart contracts, gas optimization methods in smart contracts, and the application of code mining to smart contract gas optimization.

3.1 Code Mining Fundamentals

There have been several approaches to code mining, aiming at different objectives. As described in Section 2.6, code mining refers to the extraction of specific knowledge from software artifacts. Program knowledge extraction has been a subject of study for decades and remains an attractive area of research.

To the best of our knowledge, we can trace the roots of code mining back to a 1988 work by Harandi and Ning. Their PAT tool extracts higher-level information from raw source code, which can be considered an early form of code mining [HN88, HN90]. Although not widely cited, this pioneering work may have had an impact on subsequent research in clone detection, pattern mining, and repository mining studies.

Kontogiannis et al. introduce an approach for concept detection and code reuse by mining source code based on patterns derived from ASTs. As partially described in Section 2.5.2, the authors use five metrics that they generate for each statement, block, and function. The method is fully automatic and computationally efficient. Its code-to-code matching metrics approach is a cheaper, but less accurate way to produce results than its dynamic programming pattern matcher. However, the authors report that the metric comparison has a false positive rate of 39%, whereas the dynamic programming approach reduces the false positive (FP) rate to 10% [KDMM$^+$96].

XSnippet, proposed by Sahavechaphan and Claypool [SC06], is a developer-assistance tool that provides example code snippets by mining code from a large number of open-source projects. It operates in three stages, with the first being the construction of a code graph whose nodes represent types, objects, and methods, while the edges encode different relation types between them. The second stage performs the mining itself, using their so-called *BFSMINE* algorithm, which takes a query and walks through the code graph to enumerate every path (snippet) satisfying the query. A ranking heuristic is then utilized to order the resulting snippets. XSnippet performs well in the author's evaluation, which confirms that the system improves developer efficiency compared to other code-assistant systems.

Programmers often use frameworks or libraries that typically provide Application Programming Interfaces (APIs). Unfortunately, these APIs can lack comprehensive documentation. MAPO by Zhong et al.[ZXZ$^+$09] is, according to the authors, the first tool that mines API call patterns in the form of code snippets for developer assistance. Structured in three components, their *Code Analyzer* parses and prepares API calling source code. The subsequent *API Usage Miner* step first clusters similar API usage patterns by comparing the names used within the snippets. Then, by using a sequential pattern mining technique called the frequent subsequence miner, recurring patterns are extracted. Third, the *Recommender* evaluates and ranks the mined patterns to recommend the most relevant code snippets. Evaluation demonstrates that MAPO is effective at discovering meaningful usage patterns and at assisting developers in understanding and using API functionality [ZXZ$^+$09]. Concerning the same topic of insufficient API documentation, Wang et al. [WDZ$^+$13] propose "Usage Pattern Miner (UP-Miner)", a tool also designed to extract API usage patterns from source code repositories. Improving MAPO in terms of reducing redundant pattern occurrence, their approach first identifies frequent API call patterns through a mining process based on a frequent closed sequence mining algorithm. These results are then fed through two clustering passes to allow for identifying more specialized API usages. They demonstrate in their evaluation that UP-Miner produces sound output for developers and generates more accurate recommendations compared to

MAPO [WDZ+13].

Allamanis and Sutton have built HAGGIS, a code mining tool based on statistical natural language processing. While they claim that a simple search of frequently occurring code segments will not work for collecting code idioms, not least because idioms are not code clones, they perform a programming-language-agnostic AST fragment discovery. The authors argue that avoiding a deterministic CFG approach is an advantage for this use case, as it tends to demote recurring non-idiomatic code patterns, such as boilerplate fragments, from the output. Their extensive evaluation work conducted on open-source Java code repositories and snippets shows that the tool is indeed capable of mining code idioms [AS14].

Focusing on general applications rather than smart contracts, Palomba et al. address the problem of code smell mining. Their approach, HIST, aims to detect five predefined code smells by exploring source code changes over time (historical information), where evolving versions of code pieces are analyzed. Their evaluation shows that HIST achieves a precision of between 72% and 86% and, depending on the smell types, outperforms or performs as well as non-historical ("single snapshot") code smell mining tools. They also find that "in several cases" the smells flagged by HIST and single-snapshot miners have complementary detection rates, which suggests that the basis of a better technique is a combination of the two approach types. An additional evaluation study shows developers confirmed more than 75% of the smells reported by HIST as genuine design or implementation problems [PBP+15].

For change-history extraction, Palomba et al. leverage the MARKOS code analyzer component for method-level diffing, which is described by Bavota et al. [BCC+14]. Different works [PBP+15, LVBBC+13] employ this analyzer component to compare versioned code at the method level, even though MARKOS itself is not a code mining framework. The analyzer defines a fingerprint structure for functions, containing metrics as LOC, number of if/while/case/return statements, among others. According to Palomba et al., the underlying fingerprint leveraging heuristic detects 89% of method moves and even 98% of moves at the class level [PBP+15].

3.1.1 Code Mining for Smart Contracts

Samreen et al. [SA22] propose VOLCANO, which detects vulnerabilities in Ethereum smart contracts by utilizing code clone analysis. They are using the clone detection tool NiCad for analyzing Ethereum smart contracts. Using VOLCANO, contracts are compared to known vulnerable code snippets. The authors demonstrate that their vulnerability detection approach is successful and faster than existing methods.

Also aiming at vulnerability detection, Weiss et al. [WFTW24] propose two tools for analyzing code duplication and coverage in Ethereum smart contracts: Contract Code Coverage (CCC) and Contract Code Duplication (CCD). For CCC, they translate Solidity code into a Code Property Graph (CPG), which represents control (as in CFG) and data flow combined with the syntactical part (as in AST) in a unified graph. Known vulnerability patterns are formulated as graph queries beforehand and can then be detected by evaluating the specific CPG instances. CCD, on the other hand, targets the detection of code clones up to Type III (see Section 2.5.3). This is achieved by generating a normalized AST, which is then split into tokens and represented as a fingerprint using fuzzy hashing. Due to fuzzy hashing, the resulting fingerprint consists of many hash segments, such that a minor code change affects only a few segments, preserving the other digests. The edit distance is then computed and compared between N-grams of fingerprint pieces, and a formula determines a similarity score between the Solidity code pieces. Both CCC and CCD are being evaluated, with the result of outperforming existing tools. An evaluation with vulnerable code snippets supports the author's view that many deployed smart contracts contain copy-pasted code.

Although the work by Soud et al. [SQLH23] does not focus on code mining in the specific sense used in this thesis, the authors are mining several data sources for Solidity smart contract vulnerabilities. Focusing on building a vulnerability database, they introduce AutoMESC, an automated framework for mining and classifying Ethereum smart contract vulnerabilities and their corresponding fixes. AutoMESC integrates seven existing tools to generate vulnerability-fix pairs, along with metadata.

3.2 Smart Contract Gas Optimization

Research on smart contracts has primarily focused on security and vulnerability detection, with a primary emphasis on identifying and addressing bugs that can result in substantial financial losses, particularly following the 2016 DAO hack. In contrast – while crucial for reducing transaction costs – there was only limited initial research on optimizing gas consumption. Starting in 2017-2019, an increasing number of works have been published in this area. Smart contract optimization remains an area of active research.

Chen et al. have been among the first to focus specifically on gas optimization at the bytecode level [CLLZ17, CLZ$^+$18]. The researchers identify 24 distinct gas-costly anti-patterns at the EVM bytecode level and design GasReducer, a tool that automatically reduces gas consumption by matching and rewriting anti-patterns. By June 10, 2017, all 599,959 contracts deployed on Ethereum, as well as all related execution traces, have been processed by their tool. GasReducer has identified over 9.49 million deployment anti-pattern instances (wasting more than 2 billion gas units, which corresponds to roughly 643 USD at June 2017 prices[1]) and 557.57 million execution anti-pattern instances (wasting more than 7 billion gas units, which corresponds to roughly 2,250 USD) [CLZ$^+$18].

Also working at the EVM bytecode level, Nagele and Schett introduce their optimizer tool, ebso, in 2019. Built on the concept of super-optimization [Mas87], the "EVM Bytecode SuperOptimizer" converts a subset of the AST's execution state into SMT formulas and then utilizes the SMT solver Z3 [dMB08] to synthesize cheaper – but

[1]Assuming an average gas price of 25 Gwei and ETH/USD exchange rate of 321.80 USD in June 2017.

semantically equivalent – instructions. ebso is evaluated on 200 contracts originating from a gas optimization contest. In the 2743 CFG basic blocks analyzed, 19 optimizations are found, which partly result in non-intuitive bytecode sequence rewrites. The authors analyze all 24 anti-patterns from Chen et al. [CLZ+18]'s work, and for 19 of them, ebso independently identifies the optimized version. One major limitation of the paper is the lack of EVM memory support and the limitation to the basic block level.

Whereas ebso confines its super-optimizations to individual basic blocks, GASOL – created by Albert et al. [ACG+20] – supports transformations across multiple CFG blocks. Focusing on storage-related gas optimization (SLOAD/SSTORE), GASOL can detect and rewrite Solidity code to utilize a more cost-effective approach, which counters redundant storage access commands with intermediate memory caching. The authors determine that gas consumption can be reduced by between 20% and 49.45% for storage-intensive functions.

Correas et al. introduce a purely static analysis tool that inspects a contract's EVM bytecode and CFG to compute upper bounds. Using symbolic formulas, their profiler detects which parts of a program are the most gas-expensive and also contain vulnerabilities. Building on this profiling, the authors propose a Solidity code transformation process that replaces specific storage accesses with memory operations to reduce gas costs. In an evaluation of over 40,000 public smart-contract functions, it is found that 6.81% can be optimized using this tool [CGRD21].

In her 2021 work, Li addresses gas estimation for loop functions in addition to storage-level and array gas optimization. She finds that a quarter of all contracts contain loop functions, making loop gas estimation crucial for preventing out-of-gas exceptions. However, she acknowledges that loop gas estimation is not an easy task. Li's approach to loop functions involves predicting transaction history, while the array optimization method she proposes involves eliminating costly bound checks [Li21].

Nelaturu et al. [NBLV21] develop a prototype that specifically targets loops. On average, this prototype trades off 5% of the increased deployment gas amount for a 21% decrease in execution gas consumption. The optimizations implement for three different loop patterns involve either moving code out of the loop or relocating computation away from repetitive storage access.

In [DSLV+22], Di Sorbo et al. identify 19 Solidity cost smells at the source code level by surveying existing literature and other sources. They introduce GasMet, which automatically detects these smells, allowing smart contract developers to optimize them. The tool parses Solidity code, statically computes gas-leak metrics, and can aid developers by highlighting and explaining the problematic portions of code. The author's approach is similar to the one presented by Brandstätter et al. [BSCB20], who identify 25 such gas optimization strategies in Solidity code, of which six can be applied automatically and non-intrusively. They develop the Python-Solidity-Optimizer, which statically scans Solidity code for optimization potential and automatically rewrites contracts using optimization rules.

Marchesi et al. [MMD+20] provide a manual for Solidity developers, which collects 24 patterns grouped into five categories. Aimed at developers, it enables optimization by manual code refactoring. Although their work does not include automatic optimization, it provides a solid summary of the Ethereum gas mechanism.

SmartCheck statically analyzes Solidity by first converting the source code into an XML-based intermediate representation and then running XPath queries to identify code smells, including two gas-inefficient patterns [TVI+18].

Zhang et al.'s SolidityCheck applies a purely syntactic approach to detect gas-inefficient code smells at the Solidity source level. For this, they leverage regular expressions and program instrumentation [ZXL19].

Susik and Nowotniak propose pattern-matching algorithms to optimize gas consumption during string processing. These algorithms, designed to be used within Solidity code and aiming to surpass existing Solidity libraries, demonstrate the potential to reduce gas usage compared to the widely used StringUtils library. The authors' approach shows a gas usage improvement of up to 22 times less compared to StringUtils, as evidenced by their evaluation [SN24].

Most of the referenced sources discuss findings of non-gas-optimized smart contracts in real-world applications or at least those that are publicly available.

To the best of our knowledge, only limited research has been published regarding code mining approaches for smart contract gas optimization.

Targeting gas efficiency directly, Banerjee et al. [BSS25] propose a code mining approach for smart contracts. Their method extracts contract functions as opcode sequences from previously generated CFGs and then estimates their gas cost. Using the neural network contextual word representation tool Word2Vec [MSC+13], functions are then compared pairwise. In the end, this enables developers to find a gas-efficient alternative with approximate functional similarity. Evaluated on a dataset of over 16,000 functions, this approach achieves an average gas reduction of 33–47%, which demonstrates its viability as an alternative to fixed-pattern optimization rules.

In summary, existing work on smart-contract optimization has focused mainly on static rule-based rewriting or representations that capture syntax rather than behavior. While these approaches achieve notable gas reductions, they depend on predefined transformation rules, computationally expensive analyses, or operate as opaque black-box models. Classical code mining, in contrast, has demonstrated strong scalability and efficiency on large datasets, yet, when combined with structural fingerprinting, it has not been adapted to the context of Solidity or gas consumption optimization. Consequently, no existing approach combines the semantic nature of code-mining techniques with the efficiency and transparency of structural fingerprinting – this gap serves as the main motivation for the work presented in this thesis.

CHAPTER 4

Methodology

This chapter provides a detailed description of the research method used in our thesis. Our work aims at extracting potential gas optimizations from versioned Solidity source code. After analyzing existing code and pattern mining techniques, we have worked on two different approaches to mining optimization rules.

Figure 4.1 provides a high-level overview of the methodology applied in this thesis. The approach aims to extract optimization knowledge from real-world Solidity code by mining differences between contract versions. By comparing semantically equivalent code segments that differ in gas cost, our method attempts to identify transformations that reduce gas consumption while maintaining contract behavior. On the left-hand side of

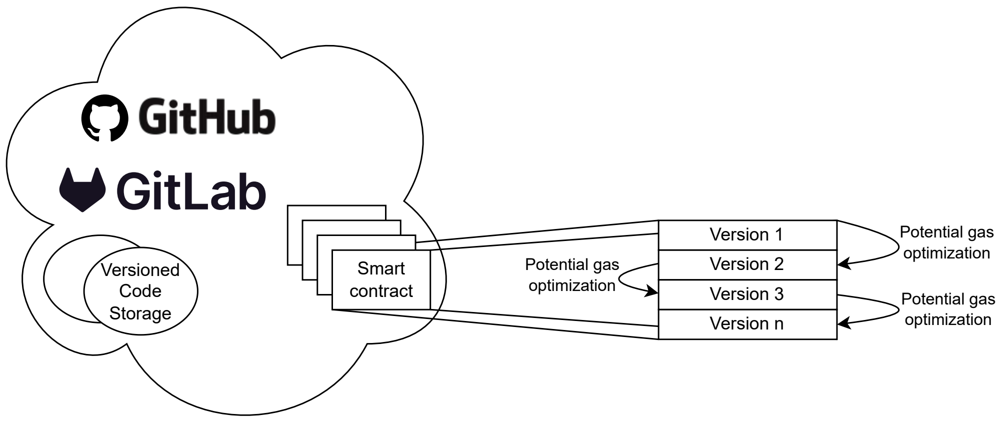


Figure 4.1: Optimization origin overview

the figure, version-controlled code repositories are shown as the source of smart contract versions. The right-hand side refers to multiple versions of a single smart contract. The arrows, pointing from version a to version $a + 1$, express that the change between the two versions contains a possible gas optimization. These transitions represent the core of our mining approach: identifying behavior-preserving changes that reduce gas cost.

The process starts with a large dataset of version-controlled Solidity smart contracts, which are processed and split into individual code segments, at the function or CFG block level. Features of the resulting segments are compared between versions. If a semantically equivalent but more gas-efficient version is found, the difference is considered a potential optimization.

The dataset which was used to develop and continually improve our approaches on consists of over 160,000 versioned smart contracts scraped from GitHub, as well as a small number of manually created contracts and implemented optimization patterns from [BSCB20, Bra20].

4.1 Research Workflow

In this section, the research workflow followed in this thesis is described, starting with an initial survey of related work and tools. During the literature review, the focus was set on pattern mining methods, as [PBP+15], code clone detection, as [JMSG07], static-analysis frameworks, as Slither, and symbolic execution tools, as [LCO+16]. During review, the first challenge that became obvious was the needed determination on the granularity level, as source code line, CFG block, or function.

For an initial concept, function-level fingerprints were computed to identify semantically similar code. Starting with fingerprint metrics similar to those used in [PBP+15], a prototype was implemented. However, this approach was found to be too inaccurate due to the function-level granularity. Additionally, alternative approaches were considered at this stage.

Focusing on program data flow, the symbolic execution tool *Oyente* was evaluated to produce symbolic traces that can serve as comparable fingerprints. As Oyente supports only limited, version-constrained source code and works best with bytecode, separate compilation was necessary. The Oyente implementation depends on EVM and solc, where it is constrained to a small number of versions due to recurring API changes.

Oyente internally uses symbolic variables for all EVM state fields, which were compared post-execution between contracts. Experiments showed that differing constants in `PUSH` instructions produced identical output, hinting at a limitation that needed to be addressed. This behavior meant that even semantically different code fragments could appear as equivalent in the comparison output, which reduces our comparison's reliability.

To overcome this issue, the comparison data model was enhanced with metadata related to the original source code. This resolved the limitation; however, this method depended on Oyente's internal source code compilation logic, which simplified the workflow but introduced compiler version constraints.

Because the resulting toolchain would have excluded over 90% of the dataset, and also, an internal Oyente bug led to incomplete basic block evaluation states, the Oyente approach was abandoned. As a direct replacement, *Manticore* was evaluated. Manticore is not optimized for the EVM, and although it is theoretically able to produce the needed output, its general-purpose design made it difficult to interpret the execution branch mappings and symbolic states. The relation between Solidity-level state and the symbolic representation remained unclear. In particular, while state values were basically available, their abstracted form and unusual ordering prevented an intuitive mapping with the contract code.

Therefore, we decided to opt for a CFG-metric approach. Continuing the initial idea of fingerprinting for similar code identification, code granularity was fixed at the CFG basic block level. Using *Slither* and custom Python embedding, metrics such as Cyclomatic Complexity were extracted for each basic block. Code fragments were then compared based on the Euclidean distance between metrics across different contract versions.

Two CFG blocks from different contract versions were considered equivalent if they had a sufficiently low distance. To measure their actual gas consumption, all extracted code snippets were compiled into runnable Solidity contracts to measure their actual gas consumption (for both deployment and execution) using Ganache and Web3.py. This allowed for checking whether one variant consumed less gas than the other – thereby identifying an optimization pattern.

4.2 Theoretical Foundations

4.2.1 Semantic Equivalence

A transformation from a code fragment f_1 to f_2 is considered a refactoring if the observable behavior of f_1 is preserved. Apart from program-language specifics, the result of both fragments must be observationally indistinguishable in any environment [HBT22, Fow18]. We consider f_1 and f_2 to be semantically equivalent when they differ only by a refactoring. Regarding Ethereum smart contracts, the observable behavior is determined by persistent state changes, namely storage, event logs, and Ether balance. Transient differences, such as memory, stack values, and volatile block data (e.g., block timestamps), are ignored, as they do not affect persistent contract state.

4.2.2 Gas Improvement

Ethereum contracts cause execution costs – gas. For gas-optimization mining, we require $\text{gas}(f_2) < \text{gas}(f_1)$, in addition to semantic equivalence, which means we are only interested in cheaper fragments. Gas needs to be measured from two perspectives: deployment and execution calls. $\text{gas}(f)$ therefore refers to $\text{gas}_{\text{deploy}}(f) + \text{gas}_{\text{call}}(f)$, which means the gas cost used for contracts plus the gas cost for a contract call. This represents the total cost paid by the contract owner and the caller and also allows optimization patterns that shift gas costs between contract deployment time and execution runtime.

4.2.3 Granularity

When comparing program fragments for equivalence, a level of granularity has to be chosen at which the comparison is performed. Granularity selection is a trade-off between precision and scalability. Table 4.1 summarizes the advantages and disadvantages of three obvious granularity choices.

Table 4.1: Granularity choices for Solidity code fragments

Level	Pros	Cons
Instruction (opcode)	Maximum precision; Direct gas accounting	Largest search space
CFG basic block	Metrics fast to compute; Captures coherent control and data flow; Remain stable across pragma versions	Ignores dependencies spanning across blocks
Whole function / contract	Full execution context (scope) available; Useful for macro optimizations	Too coarse for local optimizations; Tiny changes can break matches of otherwise identical code; Harder gas measurement due to control-flow constructs

4.2.4 Approaches To Equivalence

Although full semantic equivalence is undecidable in general [Ric53], we hypothesize that semantic similarity can be approximated. In this thesis, we explore two approaches to this problem, which are outlined in the following subsections.

Symbolic Execution

Symbolic execution formalizes program fragment behavior by treating input as symbolic variables, where each feasible program execution path produces logical formulas. Two fragments are semantically equivalent if the SMT solver finds no model that differs between the two – this condition can be checked by determining the satisfiability of their combined formulas. Tools like *Oyente* [LCO+16] apply this principle to EVM bytecode.

Its theoretical strength is high precision, whereas in practice, it has high analysis cost, with explored paths growing exponentially, as well as its tight coupling to specific EVM versions.

Structural Metrics

An alternative approach reduces code fragments to numbers. A signature is formed based on structural features, such as complexity and fragment length, that are encoded as numbers, such as cyclomatic complexity and fan-out. Fragments with similar signatures are candidates for semantical equivalence.

The process can be parallelized, scales linearly in time, and the computation of most metrics is version independent. Its weakness is the heuristic nature; equivalence cannot be proved. Also, false positives are impossible to avoid.

4.3 Dataset Requirements

To extract gas optimization patterns from existing Solidity code, a dataset of smart contracts with version histories is essential. The methodology applied in this thesis relies on the assumption that some historical changes made by developers were motivated by performance reasons. Provided this assumption holds, this allows to identify the source code changes leading to reduced gas consumption by observing how contracts evolve over time.

To support such mining, the dataset must meet the following requirements:

- **Multi-Version Smart Contracts**
  The dataset must contain multiple versions of the same contracts. Only with this temporal property code transformations can be detected systematically.

- **Solidity Source Code**
  Providing more context than bytecode and carrying the developer's intentions more naturally, our analysis operates on high-level source code. The dataset must provide parseable Solidity files, ideally with the `pragma solidity` declaration.

- **Compilable Code**
  Tooling such as *Oyente*, *Slither*, or *solc* requires fully compilable source code files such that control-flow information can be processed. Missing dependencies or unresolved imports are problematic.

- **Real-world Relevance**
  The dataset should include contracts that were either deployed or realistically written for deployment. Artificially generated code or educational samples are less valuable, although manually crafted contract optimizations could help during prototype evaluation.

Several data sources (e.g. Etherscan) do not fulfill these criteria, especially with respect to version tracking. To address this, we use a Solidity repository dataset that satisfies the requirements above, as described in Section 5.2.

4.4 Approach A - Symbolic Execution Design

4.4.1 Motivation

The first conceptual approach leveraged symbolic execution for determining semantic similarity between two code fragments. The core of this idea is that, when two different implementations generate the same state and output, they are semantically equivalent. In such a case, the lesser gas-consuming implementation can be preferred. Symbolic execution provides an advantage vs. structural comparison methods, as it can detect functionally equal implementations even for significant syntactical differences.

Symbolic execution runs produce a set of *path conditions*, describing the input conditions needed for a program path to be executed. When an identical set of path conditions is generated for two different code fragments, it can be assumed that both create the same output for all inputs.

4.4.2 Analysis Engine

For the implementation of this approach, we selected *Oyente* [LCO+16], a popular[1] open-source tool for symbolic execution of Ethereum bytecode. Oyente integrates the Z3 SMT solver [dMB08] and provides mapping to Solidity source code, CFG generation, stack, memory, and storage state, along with path conditions.

Due to this rich output and its specialization in Ethereum smart contracts, Oyente was considered a solid base for equivalence determination.

4.4.3 Planned Workflow

The approach concept was planned as follows:

1. Analysis with Oyente
2. Extraction of relevant output per contract, including a source mapping
3. Comparison on the function level
4. Gas measurement on equal results

[1]over 1300 stars and over 300 forks on `https://github.com/enzymefinance/oyente` as of August 8th, 2025

4.5 Approach B - Metric Analysis Design

4.5.1 Motivation

The next conceptual approach focused on a static, metric-based comparison of CFG basic blocks. The idea was to find semantic similarity between two code fragments by comparing numerical fingerprint vectors derived from their structural properties.

The concept is based on the work of Kontogiannis et al. [KDMM+96], who used metrics for code clone detection in software systems. Using this principle allows detecting functionally similar code, even when identifiers, formatting, or minor statement order differ. Similar enough fragments can then be gas measured and form a gas optimization pattern, in case one of them is cheaper.

4.5.2 Analysis Engine

Slither [FGG19], a static analysis framework for Solidity, was used. Slither can construct CFGs and provide detailed information about the input contract.

4.5.3 Planned Workflow

Phase 1

A method-level fingerprinting method inspired by [BCC+14] was evaluated, where functions were represented by simple structural metrics (e.g., number of statements and control-flow elements). Although this was simple to implement, early results showed that it was sensitive to minor structural changes. The design raised doubts about whether such metrics alone would be sufficient to capture semantic equivalence. For this reason, we later continued with a second phase based on CFG blocks and more expressive metrics.

Phase 2

When using CFG-based granularity, and the fingerprinting method from Kontogiannis et al. [KDMM+96], the design appeared much more robust. A set of metrics, including McCabe's cyclomatic complexity, fan-out, number of statements, and others, was computed. Details of these metrics can be found in Section 2.5.2. The metric vector for each CFG block forms a fingerprint, which serves as a comparison basis. The planned approach included the following steps:

1. CFG block extraction from input source codes
2. Metric computation per block
3. Fingerprint comparison using a distance function
4. Gas measurement for similar pairs, embedded inside a minimal contract structure.

4.6 Gas Measurement

Besides identifying semantically equal smart-contract fragments, it is necessary to measure whether one variant uses less gas than the other. Because gas usage affects deployment and invocation costs, the measurements must be reproducible and comparable.

The goal is to identify if two semantically equivalent code fragments differ regarding their gas cost. For each fragment, the following two gas consumption aspects are relevant:

- **Deployment gas** – the cost to deploy a contract containing the fragment
- **Execution gas** – the cost to invoke a contract part containing the fragment

Both aspects need to be measured, as optimizations could reduce deployment gas while increasing execution gas and vice versa.

Evaluation of each fragment is done in an isolated context, containing only the fragment itself and the minimum wrapper contract logic to make it executable. This is required to avoid interference with unrelated contract logic and to measure no unrelated code parts.

Real blockchain networks may contain changing block parameters, mining behavior, and protocol details. For exact measurement and to allow a reliable comparison, a deterministic and resettable execution environment is used. This ensures consistent results also for repetitions.

The gas measurement procedure for semantically equivalent fragment pairs consists of the following steps:

- Make the fragment deployable using a wrapper contract
- Deploy the contract and record the required gas
- Invoke the contract function with fixed inputs and record the required gas
- Compare the consumed gas of both fragments

CHAPTER 5

Implementation

This chapter is about the concrete realization of the approaches introduced in the previous chapter. While Chapter 4 focused on conceptual design decisions, the following sections describe the technical implementation.

5.1 Toolchain

5.1.1 Hardware and Base Setup

All experiments were conducted on a Lenovo Thinkpad T14s equipped with an AMD Ryzen Pro 4750U processor and 32GB of RAM. The host operating system was Windows 11, but all development and execution took place inside an Ubuntu Linux 20.04 distribution running under Windows Subsystem for Linux (WSL).

The implementation relied primarily on Python (version 3.8) with isolated virtual environments created via *venv*. This allowed each prototype to maintain its library dependencies without conflicts. For compilation, multiple versions of *solc* (Solidity compiler) had to be used, depending on the pragma version requirements of the contracts. A switching mechanism was used during the CFG approach, enabling automatic selection of a matching compiler version. Intermediate analysis results were generated in the Comma Separated Value file (CSV) format.

5.1.2 Symbolic Execution Environment (Oyente)

For our symbolic-execution prototype, we relied on Oyente version 0.2.7 from 2020, which corresponds to the latest version of the *enzymefinance/oyente* GitHub repository as of August 27th, 2025. To the best of our knowledge, this fork is the most actively maintained one. The original *oyente/oyente* repository has been inactive since 2017 and links to the version we use. We executed Oyente within a Python 3.8 environment, and used `venv` for virtual environments management. Oyente employs the Z3 SMT solver [dMB08] in the background; we have used version 4.8.7-4build1.

The Solidity smart contract compilation required the use of legacy versions. For switching compiler versions, we used *solc-select*, and we used *crytic_compile* 0.1.13 as a dependency for Oyente. While standalone compilation using *solc* (version 0.8.7) produced analyzable bytecode, this approach did not generate source maps within Oyente. Without source maps, Oyente was unable to account for constant values and produced identical symbolic outputs for contracts with differing *PUSH* instruction constants. This prevented meaningful equivalence detection, which we addressed by running Oyente with integrated compilation, which internally used crytic_compile. This worked only with *solc* 0.4.19 (2017) combined with the *geth* Ethereum Virtual Machine (EVM) 1.7.3 binary. Newer compiler releases triggered parsing errors within internal Oyente functionality due to a lack of compatibility.

5.1.3 Static Analysis Environment (Slither, CFG Metrics)

For the static-analysis-based approach, our toolchain was based on the Slither framework. We used version 0.9.3 from March 2023.

On top of Slither, we developed several Python scripts for analysis and comparison, including methods for handling contract comparisons across versions, metric computation, compiler switching, and code instrumentation. Together, these scripts automated the process of extracting CFG blocks, computing metrics, and recording differences between contract versions.

To handle the unbounded range of Solidity compiler support, we integrated the tool *solc-select*. Input for this environment consists of Solidity files, while the intermediate outputs are written as CSV files and diff patches. The final outputs consist of CSV files.

5.1.4 Measurement

For gas-cost analysis, we leveraged the local blockchain simulator Ganache, CLI version 7.9.2. We used Ganache's pre-defined configuration, which automatically generates test accounts. One of these accounts and the corresponding private key was then used as an argument for the measurement scripts.

The library *crytic_compile* (v0.1.13) was used for compiling the smart contracts, whereas the blockchain was accessed via *Web3.py* (v5.31.1). The measurement process outputs were stored as CSV files containing contract identifiers along with the gathered gas measurement information.

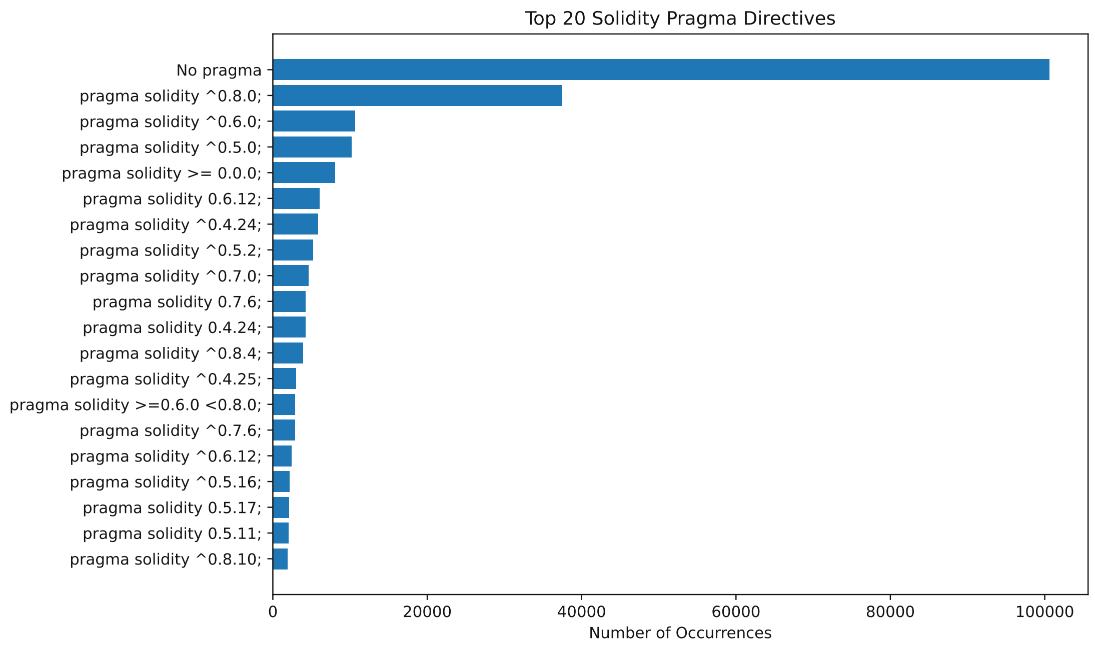


Figure 5.1: Top 20 most frequent `pragma solidity` directives in the dataset.

5.2 Dataset

We used a large dataset[1] of versioned Solidity source files [Egg22] in 2022. It was obtained from publicly accessible GitHub repositories with the goal of capturing the development history of smart contracts.

In total, the dataset contains 163,534 contracts, with each entry containing between 1 and 200 versions. Each version entry contains a contract file, representing the file snapshot for a specific Git commit. The distribution of version counts shows that 81% of all contracts (132,372) only contain one version and are therefore excluded from our usage. The remaining 31,162 contracts have multiple versions, whereas the average number of versions per contract is 5.21, and the median is 3.

The dataset spreads over a large range of `pragma solidity` version directives, with 627 distinct variants. The distribution of the top 20 directives is shown in Figure 5.1. This figure also visualizes that a predominantly large number of contracts do not contain a pragma version at all.

All version entries include a complete Solidity source code file; however, not all files can be successfully compiled due to missing imports, external libraries, or syntax errors. Nevertheless, the majority of source codes can be parsed and compiled without issues. The dataset covers a wide range of code types, from minimal helper contracts to more

[1]available at `scr.ide.tuhh.de`

complex multi-contract modules. Many of the entries refer to libraries, which we expect from open-source contracts from GitHub.

The complete dataset file size is approximately 2.6 GB in JSON format and is stored in 820 separate files to facilitate partial walkthroughs. The dataset allows us to observe actual developer modifications and serves as the basis for the evolution and evaluation of our mining process.

5.3 Attempt A - Symbolic Execution Prototype

We leveraged the symbolic execution tool Oyente to produce symbolic traces that can serve as comparable fingerprints. Oyente does not work with the latest Solidity version, but supports bytecode, meaning that we can compile the Solidity code separately to increase compatibility. Our goal was to extract symbolic execution traces from Oyente output, to match them as fingerprints between contract versions.

Internally, Oyente uses variables for all EVM state fields, as `Is`, `Ia`, `Iv`, memory, balance, and gas variables. We hypothesized that these variables provide a reasonable basis for comparing states between contracts and managed to output these variables in a parsable form.

Out of the box, Oyente does not provide an option to output internal state structures, which we need to be able to compare symbolic execution states. We have therefore adapted the Oyente program code and integrated a new flag: `-sci`. This flag captures and emits the internal Oyente symbolic execution states as a JSON file. For contracts between 5-20 lines of code, the output typically consists of 3.000-8.000 JSON lines. The output structure consists of the following four fields:

- **Blocks**, which refer to CFG basic blocks, modeled on instruction level. The instruction object contains the Oyente block ID, the instruction string, and symbolic representation values of stack, global state, memory, and the path conditions and variables state. All state representations are as of after the instruction execution.
- **Jumps**, whereby a basic block can be followed by three distinct jump types: Successor, left branch, and right branch. Each jump contains the source and destination block ID, as well as the jump type.
- **Vertices** are representing CFG basic blocks directly. They contain their block ID, vertex type (conditional, unconditional, and others), and all block instructions.
- **Edges**: Represent all possible jumps from every block. The block IDs are the jump targets.

We discovered that, when provided with bytecode input, *PUSH* instructions with varying constants yielded identical results. To work around this limitation and improve fingerprinting quality, we added a source code mapping to the output, which we could only obtain by compiling the contracts inside Oyente. This led to a version limitation on solc side, as newer versions would have required Oyente source code changes. Also, a limitation appeared during the comparison of specific contracts that differed only by the order of two commutative statements. We created two contract variants, one of which contained the two Solidity lines:

```
b = b + digit + 3;
b = 2 + b;
```

The second contract had the same lines in reverse order. From a semantic point of view, these variants must lead to the same end state, since both operations are commutative. However, Oyente produced different outputs at the end of the respective CFG basic blocks. We discovered that, sometimes, the framework does not evaluate all block instructions. In our experiment with the exact function body above, it processed only a subset of the EVM bytecode instructions, which was 7, respectively 15, out of 30 instructions. Oyente silently stopped before the block was complete. As a result, the output differed between the two versions, although the semantics should have matched. We did not investigate this issue further, which appeared to be an internal Oyente error.

The restriction to *solc* 0.4.19 means that a large part of the dataset (see Section 5.2) cannot be analyzed, whereas the error preventing whole block state generation means that we cannot reliably compare contract equality. Also, Oyente showed significant limitations in terms of usability and adaptability. We therefore decided that the fixed old versions were too big an issue to handle, and looked for other tools. Manticore, as a potential direct replacement for Oyente, was theoretically capable of producing the required output. However, as it is a general-purpose tool and not optimized for EVM, we were unable to understand the mapping to the execution branches within a reasonable timeframe. With regards to the disadvantages of this approach – as symbolic execution analysis is computationally expensive and therefore slower than static approaches – we decided to abandon this attempt and continue with a static fingerprinting approach, as described in the next section.

5.4 Attempt B - Basic Block Fingerprinting Prototype

For our initial concept of computing function-level fingerprints, we built a Python prototype based on *solidity_parser*[2]. This prototype applied fingerprint logic and processing on the statement level. After discarding the symbolic-execution-based Oyente approach, our focus shifted back to the original idea of static analysis, where we restarted prototyping with another framework.

The aim was to compare code fragments in a structured way using metric fingerprinting onto CFG structures. We generated the CFGs by leveraging the Slither framework. By

[2]https://github.com/ConsenSysDiligence/python-solidity-parser

default, Slither provides control flow information at the instruction level. While it exposes all expression and statement nodes for each function, it does not aggregate basic blocks. For this reason, we implemented a custom CFG generator to extract basic blocks on top of Slither. The algorithm is based on leader identification, as described in [ALSU07]. Nodes are processed from the function entry point. Each leader marks the start of a new basic block, and a block continues until the next leader starts. A node is considered a leader if it is the first node, the target of a branch, or if it follows a jump or branch.

We implemented the fingerprinting computation based on the idea of Kontogiannis et al.[KDMM$^+$96], as described in Section 2.5.2. The formulas had to be partially approximated to make them work with Solidity code. All metrics were computed at the node level, except for McCabe's Cyclomatic Complexity, which depends on context from the CFG. We chose the following implementation:

- **S-Complexity**: Defined as the square of fan-out. For the implementation, we used the number of outgoing calls, as provided by Slither for each node, and squared the value.
- **D-Complexity**: Computed as the number of global variables used or updated inside the node, divided by the fan-out plus one. Slither reports which variables are read, written and declared locally. We combined the counts of these sets, while ignoring duplicates. For fan-out, again the number of outgoing calls was used.
- **McCabe's Cyclomatic Complexity**: Implemented at the function level, we walked through the CFG nodes produced by Slither, and counted conditional statements. The final count plus one is returned.
- **Albrecht's function point metric**: We approximated this metric by using the variable and state-variable access information given by Slither. User and file inputs were set to zero, as there is no EVM equivalent.
- **Henry-Kafura information-flow metric**: We approximated this metric according to the data flow parameter Slither provides and used the square of the variable access information. Other data flow, such as block parameters or contract bytecode access, was ignored.

We have implemented the pipeline for Attempt B as a sequence of steps that transform raw Solidity code into comparable metric vectors and, at the end, CSV results. The process starts with .sol file extraction from our dataset (Section 5.2). After this, diff hunks are generated for contract repositories, from version to version. Each contract version is then provided individually to the analysis part. A helper script extracts the pragma version directive from each Solidity source code file and selects the corresponding compiler version before running Slither, via *solc-select*. This mechanism is crucial for both our dataset and real-world contract analysis, ensuring compatibility across heterogeneous source versions.

Within Slither, we invoke the compilation and analysis per contract version. After generating the basic block and computing the metrics, as described above, the results are written to CSV files, which form the primary artifact for comparison. Each row in the CSV contains metadata about the block along with the computed metrics, namely hash, type, Euclidean distance, Result, S-Complexity, D-Complexity, McCabe's Cyclomatic Complexity, Albrecht's function point metric, Henry-Kafura information flow metric, NodeType, Expression, Contract, Function, File, and solc-version. In parallel, the extracted basic blocks that are to be measured later are also written into compilable Solidity code snippets. This step ensures that the comparison results can be connected to gas measurements.

For comparing the potential equivalent code fragments, basic blocks are represented as numerical metric vectors. Our similarity check between two blocks then computes Euclidean distance between the vectors, which we implemented using *NumPy*. Whenever this distance was smaller than or equal to our predefined threshold, the two blocks were classified as equivalent. We have tested a range of thresholds, as explained in Subsection 6.2.3.

5.5 Gas Measurement

To evaluate whether code fragment pairs differ in gas consumption, we developed a toolchain that generates minimal Solidity contracts, compiles and deploys them to a local Ethereum test chain, and records the resulting gas usage.

Candidate code fragments are wrapped into Solidity contract skeletons. Generation starts with the pragma directive, which reflects the compiler version from our code, and is retrieved by accessing the solc version property of the first Solidity node. Following is a metadata comment, the contract declaration, and a single measurement function. The function header includes input parameters, one for each referenced external value. Compatible Solidity statements are then written into the function body.

As an example, the contract in Listing 5.1 is generated from a basic block consisting of 3 Slither nodes. The CFG excerpt containing the basic block is shown in Figure 5.2, with the example contract's basic block highlighted in yellow. The first ENDIF node, shown at line 7, is not rendered into Solidity code, as it represents an internal Slither node. The following two expression nodes are rendered normally. Lines 9-10 contain a variable assignment of a performed division, whereas the expression node in lines 12-13 corresponds to an addition assignment.

Listing 5.1: Smart contract: A basic block, to be gas-measured, embedded within our measurement contract structure

```
1 pragma solidity 0.5.2;
2 // 3 nodes; from 800-999/088-62d9378193a03617342a7a92-06-
3 // contracts_common_lib_Merkle.sol-Merkle/Merkle-v01.sol
4 // / Merkle / checkMembership
```

```
contract Measurement {
  function a(uint256 i, uint256 index) public {
    // Node NodeType.ENDIF

    // Node NodeType.EXPRESSION
    index = index / 2;

    // Node NodeType.EXPRESSION
    i += 32;
  }
}
```

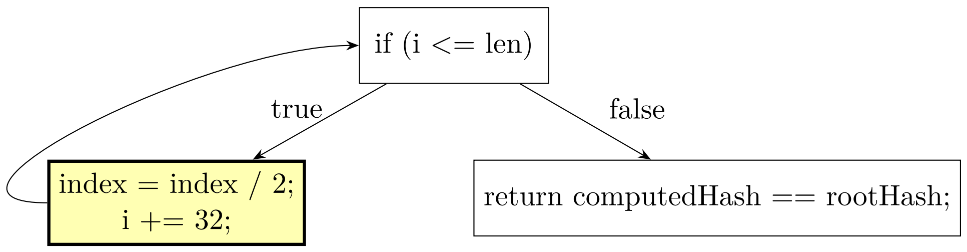


Figure 5.2: The CFG containing our basic block

We integrated the measurement contract generation directly into the metrics computation toolset, which allowed efficient reuse of the Slither CFG parsing information. The process consisted of three dedicated steps:

1. Compiling contracts to generate bytecode and ABI files.
2. Connecting to the Ganache blockchain, deploying contracts, executing selected functions, and recording deployment and execution gas usage.
3. Aggregating measured data with the metric results.

The generated Solidity files are compiled with Crytic Compile in the first step, producing the corresponding ABI and bytecode. *solc-select* is leveraged automatically for Solidity compiler version compatibility.

In the second step, deployment and execution are handled using *Web3.py* and a local Ganache Ethereum blockchain. For the input values our measurement contracts require, we set up fixed placeholder parameters, which are selected by required type, as defined in the generated ABI file. For each contract, we logged both the deployment and the execution gas.

The results of this process are – in the third step – printed to an intermediate CSV file, which is then combined with the metrics output for a unified CSV file. This unified file is the final process output and represents gas optimization potentials, block by block.

CHAPTER 6

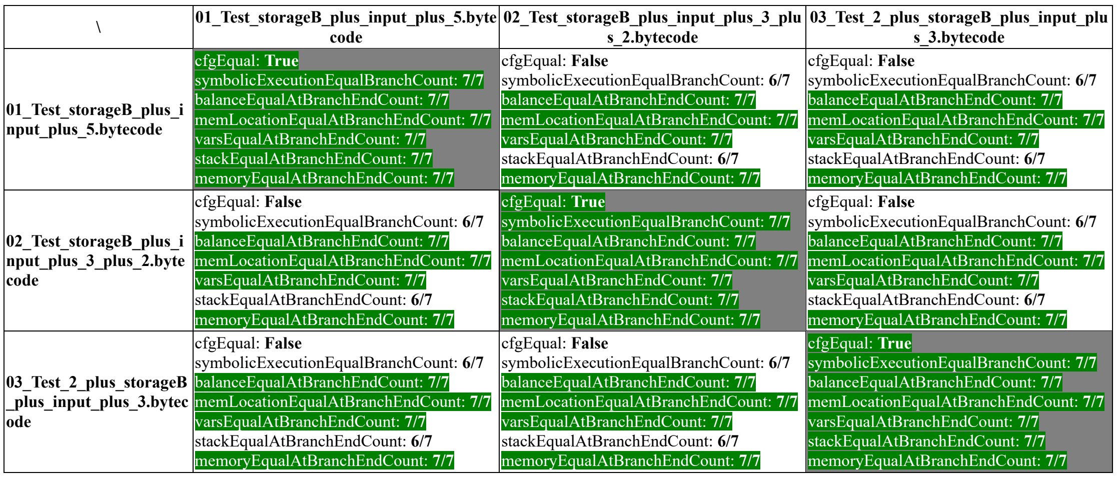
Evaluation

The evaluation aims to determine whether the proposed techniques can correctly identify gas consumption-optimized code fragments in Solidity. For this purpose, we used a two-stage evaluation approach. First, we measured the correctness of the semantically equal code detection by comparing our outputs to manually defined equality labels. Second, we validated the cost improvement of detected matches by compiling the code fragments into executable Solidity contracts and measuring their gas consumption by running them on a local Ethereum instance.

Our evaluation focuses on the CFG metric-based approach (Approach B), as we discontinued developing Approach A early on. The dataset used for evaluation consists of around 160k versioned contracts (as mentioned in Section 5.2) which allowed us to conduct both equality checks across versions and also the corresponding gas measurements. For the experiments measuring the accuracy of equivalence detection and false positive/negative analysis, we used a subset consisting of 1,200 contracts only, as they required to be labeled manually regarding actual semantic equivalence. In addition, we also used a subset of [BSCB20]'s optimization contract examples to test whether known optimizations could be detected.

6.1 Approach A

In this section, we evaluate our Oyente symbolic execution approach. The tool versions we have used are described in Section 5.1.2. Our objective regarding this evaluation was to determine whether symbolic execution traces could serve as fingerprints for approximating semantic equivalence between different versions of a contract. We expected that contracts differing only in small, potentially performance-optimizing instructions would be distinguishable from genuinely different logic.



\	01_Test_storageB_plus_input_plus_5.bytecode	02_Test_storageB_plus_input_plus_3_plus_2.bytecode	03_Test_2_plus_storageB_plus_input_plus_3.bytecode
01_Test_storageB_plus_input_plus_5.bytecode	cfgEqual: **True** symbolicExecutionEqualBranchCount: **7/7** balanceEqualAtBranchEndCount: **7/7** memLocationEqualAtBranchEndCount: **7/7** varsEqualAtBranchEndCount: **7/7** stackEqualAtBranchEndCount: **7/7** memoryEqualAtBranchEndCount: **7/7**	cfgEqual: **False** symbolicExecutionEqualBranchCount: **6/7** balanceEqualAtBranchEndCount: **7/7** memLocationEqualAtBranchEndCount: **7/7** varsEqualAtBranchEndCount: **7/7** stackEqualAtBranchEndCount: **6/7** memoryEqualAtBranchEndCount: **7/7**	cfgEqual: **False** symbolicExecutionEqualBranchCount: **6/7** balanceEqualAtBranchEndCount: **7/7** memLocationEqualAtBranchEndCount: **7/7** varsEqualAtBranchEndCount: **7/7** stackEqualAtBranchEndCount: **6/7** memoryEqualAtBranchEndCount: **7/7**
02_Test_storageB_plus_input_plus_3_plus_2.bytecode	cfgEqual: **False** symbolicExecutionEqualBranchCount: **6/7** balanceEqualAtBranchEndCount: **7/7** memLocationEqualAtBranchEndCount: **7/7** varsEqualAtBranchEndCount: **7/7** stackEqualAtBranchEndCount: **6/7** memoryEqualAtBranchEndCount: **7/7**	cfgEqual: **True** symbolicExecutionEqualBranchCount: **7/7** balanceEqualAtBranchEndCount: **7/7** memLocationEqualAtBranchEndCount: **7/7** varsEqualAtBranchEndCount: **7/7** stackEqualAtBranchEndCount: **7/7** memoryEqualAtBranchEndCount: **7/7**	cfgEqual: **False** symbolicExecutionEqualBranchCount: **6/7** balanceEqualAtBranchEndCount: **7/7** memLocationEqualAtBranchEndCount: **7/7** varsEqualAtBranchEndCount: **7/7** stackEqualAtBranchEndCount: **6/7** memoryEqualAtBranchEndCount: **7/7**
03_Test_2_plus_storageB_plus_input_plus_3.bytecode	cfgEqual: **False** symbolicExecutionEqualBranchCount: **6/7** balanceEqualAtBranchEndCount: **7/7** memLocationEqualAtBranchEndCount: **7/7** varsEqualAtBranchEndCount: **7/7** stackEqualAtBranchEndCount: **6/7** memoryEqualAtBranchEndCount: **7/7**	cfgEqual: **False** symbolicExecutionEqualBranchCount: **6/7** balanceEqualAtBranchEndCount: **7/7** memLocationEqualAtBranchEndCount: **7/7** varsEqualAtBranchEndCount: **7/7** stackEqualAtBranchEndCount: **6/7** memoryEqualAtBranchEndCount: **7/7**	cfgEqual: **True** symbolicExecutionEqualBranchCount: **7/7** balanceEqualAtBranchEndCount: **7/7** memLocationEqualAtBranchEndCount: **7/7** varsEqualAtBranchEndCount: **7/7** stackEqualAtBranchEndCount: **7/7** memoryEqualAtBranchEndCount: **7/7**

Figure 6.1: Analysis matrix for Oyente output comparison

To support the manual comparison process, we extended our prototype with an HTML matrix visualization output, as shown in Figure 6.1. In this representation, each row and column corresponds to a different contract. These may represent different versions of the same contract, or, as in this example, three distinct contracts that are slightly modified variants of a single original contract. Each cell shows the result of a set of comparison checks, referring to the four JSON output fields described in Section 5.3. We designed the matrix to contain multiple dimensions of equivalence:

- **CFG equal**: Represents whether all *Edges* and *Vertices* outputs are fully equal. As *Edges* only refer to the block IDs, this comparison is about control flow structure only, ignoring content. *Vertices*, however, contain instructions, and therefore this check includes a comparison of the CFG basic block contents on EVM instruction level.
- **Equal symbolic execution basic blocks**: Represents how many *Blocks* fully match for both contracts. For this comparison, all states and instructions of the respective blocks are compared, including block IDs.
- **Balance equal at end of basic block**: Describes how many *Blocks* have the same balance at the end of the execution path. Blocks to be compared are determined by matching block IDs on both sides.
- **Memory location equal at end of basic block**: Describes how many *Blocks* have the same memory locations at the end of the execution path. In Oyente, memory locations describe the byte position at which the next memory value will be stored.
- **Variables equal at end of basic block**: Describes how many *Blocks* have the same symbolic variable names at the end of the execution path.

- **Stack equal at end of basic block**: Describes how many *Blocks* have the same symbolic stack representation at the end of the execution path.
- **Memory equal at end of basic block**: Describes how many *Blocks* have the same symbolic memory state representation at the end of the execution path.

This visualization enabled us to highlight the detected differences between contracts, especially for inspecting patterns across versions. Diagonals naturally contain perfect matches, while off-diagonal entries show the fingerprint differences between the respective two contracts. Early experiments revealed that with Oyente, many of the symbolic states across versions can be matched. However, we observed a fundamental limitation, whereby contracts differing only in constant values in PUSH instructions generated identical traces. This was due to Oyente normalizing the constant values to some extent, which caused the distinct versions to appear equivalent. We observed this behavior in the matrix view, where all comparisons are green, meaning *CFG equal* and *Equal symbolic execution basic blocks* are entirely given. We solved this limitation by switching to Oyente-side Solidity compilation, which introduced the version limitation regarding solc, as described in Section 5.1.2.

A known limitation of Oyente was that it sometimes failed to evaluate all instructions within a CFG basic block. We have discussed this problem already in Section 5.3. Despite these issues, the evaluation provided valuable insights. We find that the symbolic state that Oyente keeps – balances, stack values, memory snapshots, among others – can in principle serve as a fingerprint for equivalence checking. However, Oyente's outdated dependency chain meant that only a small subset of our dataset could be processed.

6.2 Approach B

Our measurement setup toolchain for this approach is described in Section 5.1.4.

6.2.1 Diff-Based Methodology

To systematically label code fragments with the semantical equivalence property, we developed a patch-file-based annotation format. Instead of relying either on manual ad-hoc determinations or defining per-contract or per-line equivalence values in external files, we generated *unified diff* files for each version iteration of a contract. Each diff hunk was extended with a newly introduced header field, which we set to either 0 or 1, depending on whether the modified fragment preserved semantic equivalence. The following diff file snippet shows the position and content of this header field.

```
--- lao 2002-02-21 23:30:39.942229878 -0800
+++ tzu 2002-02-21 23:30:50.442260588 -0800
@@ -1,7 +1,6 @@ ext:semantic_diff=0
```

```
-return "aaaaa"=="aaaaa" && 1==1
+return "aaaaa"=="aaaaa" && True
```

The annotated patch files were then processed automatically. For every contract and version transition, the processing tool extracted the changed code fragment for which the metric vectors were generated afterwards. This workflow enabled us to manage a large number of contract versions while maintaining track of their semantic status. It also set the foundation for the comparisons at both the node and block levels later on.

6.2.2 Node-Level Comparisons

In our first set of experiments, we attempted to compare code fragments at the instruction level, which corresponds to the node level in Slither. Instead of entire basic blocks, we grouped three consecutive statements inside a function, while ignoring loops and branches. The scenario was to determine whether small code fragments could already serve as fingerprints of semantic equivalence. We executed this run with an Euclidean distance threshold of 3.0 and a smaller subset of our dataset because the data required manual equivalence labeling. The threshold was chosen heuristically, after in early inspections, distances close to zero did not reliably indicate equivalence. So, we adopted a slightly higher value, where 3.0 appeared reasonable.

For this evaluation, we processed the first 1,200 contracts from our dataset. Only contracts that provided at least two versions were considered, which yielded around 6,500 version-to-version pairs of contracts. From these pairs, we filtered out those that did not compile and ran only code with the restriction of three nodes in a row. This resulted in 374 compilable triple-instruction comparison pairs. Out of these, our algorithm correctly identified 199 cases as equivalent or non-equivalent, which corresponds to an accuracy of 53.2%. The remaining 175 comparisons (46.8%) were misclassified. Given that a random classification would achieve a similar accuracy, this meant that the approach was ineffective.

Our interpretation of the result suggests that the primary reason for this poor performance was the excessively fine granularity level of the input data. The Euclidean distance often collapsed close to zero, producing arbitrary match decisions. Individual nodes or short node sequences do not capture the full structural properties and data dependencies, as intended by Kontogiannis et al. [KDMM^{+}96], who defined the metrics for full CFG basic blocks. This understanding encouraged us to move on to a block-level granularity.

6.2.3 CFG Block Level Comparisons

After the limited success at the node level, we adapted the CFG basic block comparison granularity, while staying with the previously chosen Euclidean distance threshold of 3.0 for comparability across experiments. We reprocessed the same subset of contracts from the dataset we used in the node-level analysis. In total, 574 block-to-block comparisons were conducted. The accuracy increased from 53% (node-level) to 59.4%. This indicates

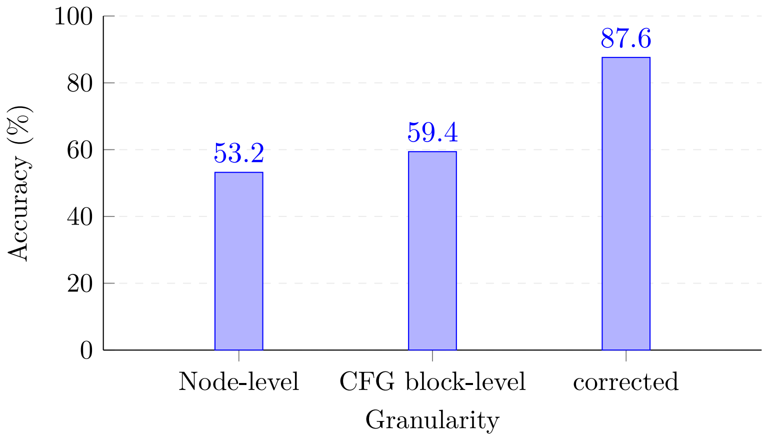


Figure 6.2: Accuracy comparison by granularity, including the corrected CFG block-level result

that the larger structural context of CFG basic blocks provides a more substantial base for deciding semantical equivalence when using metric fingerprinting. However, the absolute improvement remained small.

Error Analysis and Manual Label Correction

With the 59.4% accuracy value, we were looking to categorize false positive results. This inspection revealed substantial misclassifications regarding the resulting equivalence decisions. Out of the 574 block comparisons, 230 were detected as equivalent, although the underlying Solidity contracts were labeled as not similar, representing false positive results. We categorized 3 additional comparisons as false negatives.

To understand the cause of the false positives, we conducted a manual audit of all mismatches. A large part of the errors, 162 out of 233, originated from the patch-file semantic definition methodology. The diff-based segmentation often grouped large parts of code together, which we manually labeled as not semantically equivalent when judged as a whole. However, many of these large hunks contained multiple basic blocks that would individually have been equivalent. This labeling issue explains the large number of false positives. After correcting those mislabelings manually, the accuracy increased to 503 out of 574 correct matches, corresponding to a correctness of 87.6%. For future work, this problem should be addressed by refining the labeling granularity, e.g., by splitting the diff hunks into smaller fragments, before manually assigning equivalence labels.

The error distribution was highly skewed. As shown in Figure 6.3, almost all invalid cases were false positives, meaning our approach claimed semantic equivalence for actually differing code. Only three cases represented false negatives, i.e., blocks identified as different, even though manual classification showed semantic equivalence. This asymmetry indicates that the approach run with a Euclidean distance of 3.0 is hesitant in detecting differences, while overestimating equivalence for small distance values.

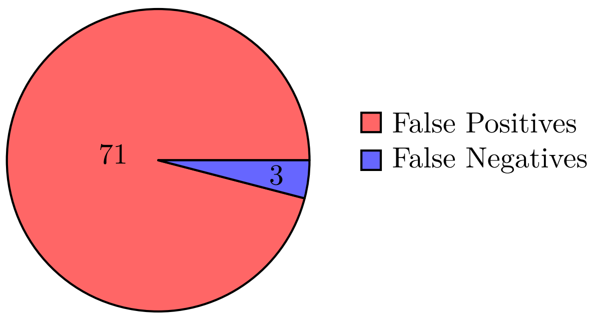


Figure 6.3: Distribution of misclassified matches, after error correction

We categorized the 71 false positives by the reason for their mismatch. 12 of them only differed in constants, while the other 59 entries included different function names called, differing parameters in function calls, additional assignment instructions, different variables used in assignment, additional variable declarations, different variables used in condition, different variable datatypes, and a combination of those. In Table 6.1, we show three false positive examples that arose during evaluation.

Table 6.1: False Positives in CFG-Based Matching

Metric Distance	**Semantic Difference**	**Reason for Misclassification**
0.0	Different function names in otherwise identical return statements (TransferERC23 vs. Transfer)	Identifier names are not included in the metric vector
0.0	Additional assignment in one version (extra field write for an object field, which has been written to before)	Number of assignments is not captured by the metrics
0.0	Variable declared with different data type (`uint256` vs. `uint32`)	Data type information is not represented in the metrics

We plotted the correctness change during the whole evaluation phase in Figure 6.2. The corrected accuracy shows that metric-based fingerprints on the CFG basic block level can produce meaningful semantic equivalence approximations. The error correction process highlights the need for a more precise diff labeling.

We have used CSV files for intermediate equality comparison output. Corresponding example entries are shown in Table 6.2 and Table 6.3. The first comparison produced an Euclidean distance of 0.0, which is due to identical metrics. It can be clearly seen that none of the used metrics captures the difference in the constant value in expression line 1. The comparison concludes with an EQUAL detection, as 0.0 is less than our threshold of 3.0. The second comparison, Table 6.3, consists of visually and value-wise very different code fragments. This leads to an Euclidean distance of 39.52, which formally validates our impression with the NOTEQ detection. All four occurring code fragments show an S-Complexity value of 0, which is because we do not have any calls to external code.

Table 6.2: CSV excerpt for Example 1 (Contract C, Function f).

	Code 1	Code 2	Result
S_COMPL.	0	0	
D_COMPL.	1.0	1.0	
MCCABE	1	1	
ALBRECHT	14	14	
KAFURA	1	1	
Expression L1	`y = uint8(0x12345678)`	`y = uint8(0x78)`	
Expression L2	`x = y`	`x = y`	
Eucl. dist.			0.0
Detection			EQUAL ($0.0 \leq 3.0$)

Table 6.3: CSV excerpt for Example 2 (Contract SimpleStorage, Functions dumbSetValue and setValueNotXDomain).

	Code 1	Code 2	Result
S_COMPL.	0	0	
D_COMPL.	2.0	5.0	
MCCABE	1	1	
ALBRECHT	14	37	
KAFURA	4	36	
Expression L1	`value = newValue`	`msgSender = msg.sender`	
Expression L2		`value = newValue`	
Expression L3		`totalCount++`	
Eucl. dist.			39.52
Detection			NOTEQ ($39.52 > 3.0$)

6.2.4 Euclidean Distance - Threshold

As written above, for the labeled dataset of 574 comparisons, we obtained an accuracy of about 88% with the default Euclidean distance threshold of 3.0. Given that only three false negatives occurred, further lowering the threshold below 3.0 would have eliminated too many valid matches. In the inverse, raising the threshold is expected to increase further the share of false positives, which was already over 95% (71 out of 74, see Figure 6.3).

To test this hypothesis at scale, we repeated the analysis with thresholds of 3.0 and 20.0 on a larger dataset subset (30k and 60k contracts, respectively). Since this data was mainly unlabeled, we cannot report accuracy or false positive/negative values; instead, we measured the absolute number of candidate matches. The measurements took between 10 and 20 hours per run. We received an equality detection *percentage of 91.5 for an Euclidean distance of 3.0* and *86.1% for the Euclidean distance 20.0* run. Due to compilation failures and solc-switching pipeline issues, the run with threshold 20.0 and 60k contracts produced only about half as many results as the earlier run with threshold 3.0 and 30k contracts. The numbers are therefore not directly comparable. We were unable to find an improvement in quality using manual spot-checks; in fact, we found no useful gas improvement pattern in either output.

6.2.5 Gas Measurement

In addition to semantic code equivalence measurements, we evaluated whether the detected code pairs also contain gas consumption differences. For this purpose, we generated minimal Solidity wrapper contracts that embed selected CFG basic blocks, and measured their gas consumption locally, as described in Section 5.5.

From the first 30,000 contracts in the dataset, we used those with at least two versions and were able to produce around 1,300 block comparisons with valid gas measurements. Roughly 50% of compilation attempts failed initially; this rate could be improved by working on the skeleton contract embedding. To further validate the approach, we tested four optimized contracts from Brandstätter et al. [Bra20]. We conducted these runs to test whether our pipeline can detect known optimizations. The results are shown in Table 6.4.

Table 6.4: Evaluation of Selected Optimizations from Brandstätter et al.

Optimization Rule	**Detection Result**	**Gas Outcome**	**Notes**
Logic1 (boolean simpl.)	✓ Match	Reduced	Correctly recognized as equivalent with lower gas cost.
Logic5 (boolean replacement with conditional)	× Not matched	N/A	Function-call handling unsupported in our CFG extraction.
SpaceForTimeRule1 (struct refactor multiplication execution)	× Failed	N/A	Code generation failed due to unsupported struct usage.
SpaceForTimeRule3 (caching storage into memory)	(✓) Match	Increased	Detected as equivalent, but gas cost increased due to variable declaration scope.

Out of the four optimizations, two were detected correctly. At the same time, two could not be reproduced due to missing support for specific Solidity features (function calls and structs) in our gas measurement pipeline. Interestingly, the SpaceForTimeRule3 optimization was detected as equivalent but showed increased gas usage, due to variable scoping at the contract skeleton block insertion. This highlights the sensitivity of gas cost to seemingly minor syntactical options. The gas evaluation demonstrated that gas computation is feasible with automatically generated contracts from basic blocks using a local test chain.

6.3 Discussion

The evaluation of both approaches shows a clear difference. While approach A ultimately cannot be used in practice for our purpose, our evaluation has shown that symbolic execution has the theoretical potential for extracting gas optimization patterns. Approach B, however, has proved to be capable of detecting some known optimizations. Although the accuracy reached only around 59% due to labeling granularity issues, manual inspection corrected this value to roughly 88%. The high share of false positives among the mismatches indicates that our structural fingerprinting approach is not always reliable in determining semantic equivalence. The false negative rate was comparatively low, showing that our method tends to overestimate equivalence instead of missing it.

Our gas measurement experiments demonstrated that the pipeline can produce valid findings. From the first 30,000 contracts in the dataset, we obtained about 1,300 block comparisons with measurable gas values. To increase confidence regarding correctness, we applied our method to four known optimization examples. Two were detected correctly, while the other two could not be reproduced. Interestingly, for the SpaceForTimeRule3 optimization, even though semantic equivalence was confirmed, the gas costs increased instead of decreasing, due to variable scoping effects in the generated wrapper contract.

Altogether, our results suggest that CFG-metric matching is in general feasible as a detection method for equivalence as well as for gas optimization. Compared to rule-based approaches such as the one proposed by Brandstätter et al. [BSCB20], our method lacks stability and predictability. At the same time, we are offering a more general framework that can be improved and extended. Regarding the research question on whether code mining can automatically identify gas-saving patterns, our evaluation partly confirms a "yes". The approach works in principle, but requires technical improvements before it can be used in practice.

6.4 Limitations

Several limitations restrict the generalizability of our findings. First, loops and branches have been ignored. At the node level, they were explicitly skipped, and at the CFG basic block level, their condition and body are naturally treated separately. As a result, optimizations that restructure control flow remain out of scope of our approaches. Additionally, optimizations that span across functions or involve inter-contract interactions cannot be observed or extracted with this methodology.

Second, regarding the basic block approach, some of the metric formulas had to be approximated. The original definitions by Kontogiannis et al.[KDMM$^+$96] were designed for general-purpose languages and not for Solidity. While we adapted them to a possible extent, this introduces a potential divergence from the theoretical definition.

Third, Slither does not natively provide CFG basic blocks. We therefore implemented our own basic-block detection based on leader-node detection. While this worked reliably in practice, it introduces an additional potential error source and a minor performance overhead.

Fourth, the gas measurements are incomplete. The automatically embedded block code contents are called with fixed input parameters, meaning that variations in input-dependent gas consumption are not fully captured. This approach, therefore, underestimates constant differences, conditional branches, and loops.

Fifth, only subsets of the dataset were evaluated in different experiments. While the full dataset consists of 150k contracts, we limited many runs due to runtime constraints or manual effort. Additionally, we exclusively analyzed open-source contracts. Many real-world deployed contracts are available only as bytecode, which prevents their inclusion.

Finally, our prototype does not yet support Solidity features such as structs or function calls within the generated measurement wrapper contracts. These limitations already caused known optimizations to fail during gas measurement.

CHAPTER 7

Conclusion

This chapter provides an overview and summary of our work, and also outlines the road ahead for future research.

The central question of this thesis was whether gas optimization patterns of smart contract code can be discovered automatically by mining version histories. Our motivation was that gas costs in Ethereum have direct economic impacts, whereby they are still at a state of optimization potential. Increasing gas efficiency, however, is a challenge developers are faced with, where, currently, producing gas-efficient code is just another non-functional requirement with, no perfect automatic solution yet. Instead of relying on predefined optimization rules, as in existing work, our focus was on determining whether optimizations could be mined directly from source code history.

We approached the research in two major phases. The first stage focused on a symbolic-execution approach using Oyente. We tried to identify semantic equivalence by comparing symbolic execution traces and ranking successful comparisons by gas consumption. In practice, however, toolchain limitations occurred. Oyente only supports outdated Solidity versions and stopped silently at incomplete symbolic execution states during our tests, which turned out to be unacceptable for our needs, without further modification.

Our second stage resembled a CFG-based static metric fingerprinting methodology, where we used Slither to extract metrics such as McCabe's cyclomatic complexity and fan-out at the CFG-block level. We then computed the Euclidean distance between the metric vectors to approximate code similarity. Early experiments at the statement level showed only 53% accuracy. After switching to basic blocks and applying a manual correction of mislabeled cases, the accuracy improved to approximately 88%.

In addition to the similarity detection, we validated code fragments with low metric distances for a difference in gas usage. For this, we generated minimal Solidity contracts for all measurement candidates and used a local Ganache blockchain to measure gas consumption on deployment and execution. While around half of the comparisons failed due to unsupported language features and incomplete code fragment embedding, around 1,300 worked and produced actual gas results. To understand the validity of our prototype, we also tested four static optimization examples. Here, our tool detected only one optimization. While two failed to produce a measurement, the fourth validation did produce a successful equivalence result. It showed a higher gas consumption on the improved version, due to variable scoping in our gas measurement template contract embedding.

Regarding whether semantically equivalent but more gas-efficient code fragments can be identified automatically, our CFG-metric-based approach demonstrates that this is possible in principle. With an accuracy of 88%, our tool was reliable at matching equivalent code fragments. Gas validation was able to confirm that some of these fragments consume less. However, both semantic similarity and gas measurement were limited to some extent, which only partially met the expectations of this thesis.

Our thesis still contains several areas for improvement and development. First, if the tool Oyente was to be modernized and improved, or a new symbolic engine focused on Solidity code was developed, the symbolic-execution approach could again become promising. Symbolic execution remains one of the most direct ways for proving semantic equivalence, after all. Second, our CFG metrics approach is operating at the basic-block level only, ignoring branching behavior, function calls, and inter-contract functionality. Extending the approach to a higher level of granularity would allow detection of optimizations involving more complex structures, such as the usage of calldata over memory for read-only array function arguments, which represents one known optimization that overlaps basic blocks.

Third, our gas measurement approach remains limited and should be improved to obtain more realistic and valid gas measurements. Future extensions should include support for additional Solidity features, in order to capture a wider range of gas optimizations. In cases where measurement compilation fails, a bytecode-level gas cost approximation would enable a trade-off between measurement accuracy and higher coverage rates. Additionally, while currently all parameter data types are supported, different variable scoping types should also be respected. Finally, the mocked test parameter inputs are fixed and should be varied to capture different loop and branch behavior. Fourth, the metric formulas we used have been crafted for general usage. In Solidity, multiple parameters had to be approximated or omitted from the computation. A more exact, EVM- or Solidity-specific metric definition could improve the approach in matching accuracy by reducing false positives. Fifth, a validation on-chain would improve evaluation quality.

Besides the approach itself, the integration of a working pattern mining technique into practice is a necessity to improve the actual smart contract gas consumption. The patterns discovered could be included within IDE plug-ins. While this is one way for unconfirmed, approximated gas-improved clones, near-perfect semantic equivalent matches could be treated separately. For this category, integrating them into compiler optimizations would be both easy to use and effective. On a different level, mining and applying code improvements could also be performed continuously on public code repositories.

Overview of Generative AI Tools Used

- **Spelling and Grammar Assistance**: Suggestions for grammar, spelling, style and wording improvements were partly used and overtaken
  - Grammarly
  - DeepL Write
- **Idea Finding and Support**: Generative AI tools were used for brainstorming, discovering sources which could confirm or deny specific statements, exploring related thoughts of existing passages, and generating LaTeX code for tables and plots
  - OpenAI ChatGPT (models GPT-4o, GPT-o3, GPT-o4-mini, GPT-5)
    - Outputs from ChatGPT were exclusively used for the reasons listed above.
    - No AI-generated texts have been used directly or in a modified form.

List of Figures

List of Tables

Acronyms

API Application Programming Interface. 22, 28

AST Abstract Syntax Tree. 18, 22–24

CFG Control Flow Graph. xi, 3, 15, 16, 19, 23, 25, 26, 28–30, 32, 33, 35, 36, 38–48, 50–54, 57

CSV Comma Separated Value file. 35, 36, 40–42, 48

DSA Digital Signature Algorithm. 5

ECDSA Elliptic Curve Digital Signature Algorithm. 5

ETH Ether currency code. 11, 12

EVM Ethereum Virtual Machine. 2, 10–13, 15, 24, 25, 28–31, 36, 38–40, 44, 54

FP false positive. 22

MD5 Message Digest 5. 7

NIST National Institute of Standards and Technology. 7

PoS Proof-of-Stake. 9, 10

PoW Proof-of-Work. 1, 9

RSA Rivest–Shamir–Adleman. 5

SHA Secure Hash Algorithm. 7

SHA-1 Secure Hash Algorithm-1. 7

SHA-2 Secure Hash Algorithm-2. 7

SHA-3 Secure Hash Algorithm-3. 7

SMT Satisfiability Modulo Theories. 2, 15, 18, 24, 30, 32, 36

WSL Windows Subsystem for Linux. 35

Bibliography

[ACG+20] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Gasol: Gas analysis and optimization for ethereum smart contracts. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 118–125. Springer, 2020. doi:10.1007/978-3-030-45237-7_7.

[AFKS24] Martin Atzmueller, Johannes Fürnkranz, Tomáš Kliegr, and Ute Schmid. Explainable and interpretable machine learning and data mining. *Data Mining and Knowledge Discovery*, 38(5):2571–2595, 2024. doi:10.1007/s10618-024-01041-y.

[AK19] Aditya Asgaonkar and Bhaskar Krishnamachari. Solving the buyer and seller's dilemma: A dual-deposit escrow smart contract for provably cheat-proof delivery and payment for a digital good without a trusted mediator. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 262–267, 2019. doi:10.1109/BLOC.2019.8751482.

[AKDB10] Saif Al-Kuwari, James H Davenport, and Russell J Bradford. *Cryptographic Hash Functions: Recent Design Trends and Security Notions*, pages 133–150. Science Press of China, 2010. The 6th China International Conference on Information Security and Cryptology (Inscrypt 2010), 20-23 October 2010, Shanghai, China.

[Alb79] Allan J. Albrecht. Measuring application development productivity. In *Proceedings of IBM Applications Development Symposium*, pages 83–92, 1979.

[All70] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. doi:10.1145/390013.808479.

[ALSU07] Alfred Vaino Aho, Monica Sin-Ling Lam, Ravi Sethi, and Jeffrey David Ullman. *Compilers Principles, Techniques & Tools.* pearson Education, second edition, 2007.

[AS14] Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on*

Foundations of Software Engineering, FSE 2014, page 472–483, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2635868.2635901.

[AZZZ19] Mishall Al-Zubaidie, Zhongwei Zhang, and Ji Zhang. Efficient and secure ecdsa algorithm and its applications: A survey, 2019. doi:10.48550/arXiv.1902.10313.

[BCC+14] Gabriele Bavota, Alicja Ciemniewska, Ilknur Chulani, Antonio De Nigro, Massimiliano Di Penta, Davide Galletti, Roberto Galoppini, Thomas F. Gordon, Pawel Kedziora, Ilaria Lener, Francesco Torelli, Roberto Pratola, Juliusz Pukacki, Yacine Rebahi, and Sergio García Villalonga. The market for open source: An intelligent virtual open source marketplace. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 399–402, 2014. doi:10.1109/CSMR-WCRE.2014.6747204.

[BES24] Avik Banerjee, Carl Egge, and Stefan Schulte. Towards the optimization of gas usage of solidity smart contracts with code mining. In *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 365–367, 2024. doi:10.1109/ICBC59979.2024.10634345.

[BMC+15] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121, 2015. doi:10.1109/SP.2015.14.

[BML+21] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham LEGHETTAS, Kamel Abdous, Taha Arbaoui, Karima BENATCHBA, and Saman amarasinghe. A deep learning based cost model for automatic code optimization. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 181–193, 2021. URL: https://proceedings.mlsys.org/paper_files/paper/2021/file/d9387b6d643efb25132be36f7b908d96-Paper.pdf.

[BP17] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In *Financial Cryptography and Data Security*, pages 494–509, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-70278-0_31.

[Bra20] Tamara Brandstätter. *Optimization of solidity smart contracts*. TU Wien, Wien, 2020. Diploma Thesis. doi:10.34726/hss.2020.66465.

[BSCB20] Tamara Brandstätter, Stefan Schulte, Jürgen Cito, and Michael Borkowski. Characterizing efficiency optimizations in solidity smart contracts. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 281–290. IEEE, 2020. doi:10.1109/blockchain50366.2020.00042.

[BSS25] Avik Banerjee, Michael Sober, and Stefan Schulte. Towards solidity smart contract efficiency optimization through code mining. In *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*, SAC '25, page 348–357, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3672608.3707768.

[But] Vitalik Buterin. Short autobiography. https://web.archive.org/web/20220531114845/https://about.me/vitalik_buterin. archived at May 31th, 2022.

[But14] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *White paper*, 3(37):2–1, 2014. URL: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[Cac97] Christian Cachin. *Entropy measures and unconditional security in cryptography.* Doctoral thesis, ETH Zurich, Zurich, 1997. doi:10.3929/ethz-a-001806220.

[CGRD21] Jesús Correas, Pablo Gordillo, and Guillermo Román-Díez. Static profiling and optimization of ethereum smart contracts using resource analysis. *IEEE Access*, 9:25495–25507, 2021. doi:10.1109/ACCESS.2021.3057565.

[CLLZ17] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446. IEEE, 2017. doi:10.1109/SANER.2017.7884650.

[CLZ+18] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. Towards saving money in using smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '18, page 81–84, New York, NY, USA, 2018. IEEE, Association for Computing Machinery. doi:10.1145/3183399.3183420.

[CXL+21] Jiachi Chen, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges. *Empirical Software Engineering*, 26(6):117, 2021. doi:10.1007/s10664-021-10018-0.

[Dai98] Wei Dai. b-money. http://www.weidai.com/bmoney.txt, 1998.

[Dam90] Ivan Bjerre Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 416–427, New York, NY, 1990. Springer New York. doi:10.1007/0-387-34805-0_39.

[Dan12] Quynh Dang. Recommendation for applications using approved hash algorithms. Technical Report NIST Special Publication (SP) 800-107, Rev. 1, National Institute of Standards and Technology, Gaithersburg, MD, 2012. doi:10.6028/NIST.SP.800-107r1.

[DCD17] Santanu Debnath, Abir Chattopadhyay, and Subhamoy Dutta. Brief review on journey of secured hash algorithms. In *2017 4th International Conference on Opto-Electronics and Applied Optics (Optronix)*, pages 1–5, 2017. doi:10.1109/OPTRONIX.2017.8349971.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-78800-3_24.

[DN93] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology — CRYPTO' 92*, pages 139–147, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. doi:10.1007/3-540-48071-4_10.

[DSLV$^+$22] Andrea Di Sorbo, Sonia Laudanna, Anna Vacca, Corrado A Visaggio, and Gerardo Canfora. Profiling gas consumption in solidity smart contracts. *Journal of Systems and Software*, 186:111193, 2022. doi:10.1016/j.jss.2021.111193.

[Egg22] Carl Egge. *A Repository for Solidity Smart Contracts.* Hamburg University of Technology, Hamburg, 2022. Bachelor Thesis.

[FGG19] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15, 2019. doi:10.1109/WETSEB.2019.00008.

[Fou25] Ethereum Foundation. A next-generation smart contract and decentralized application platform. *Ethereum Whitepaper' updated version*, 2025. URL: https://web.archive.org/web/20250625082938/https://ethereum.org/en/whitepaper/.

[Fow18] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[GLL$^+$19] Jianbo Gao, Han Liu, Chao Liu, Qingshan Li, Zhi Guan, and Zhong Chen. Easyflow: Keep ethereum away from overflow. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 23–26, 2019. doi:10.1109/ICSE-Companion.2019.00029.

[GVB$^+$25] Jingzhi Gong, Vardan Voskanyan, Paul Brookes, Fan Wu, Wei Jie, Jie Xu, Rafail Giavrimis, Mike Basios, Leslie Kanthan, and Zheng Wang. Language models for code optimization: Survey, challenges and future directions, 2025. doi:10.48550/arXiv.2501.01277.

[Has08] Ahmed E. Hassan. The road ahead for mining software repositories. In *2008 Frontiers of Software Maintenance*, pages 48–57, 2008. doi:10.1109/FOSM.2008.4659248.

[HBT22] Dániel Horpácsi, Péter Bereczky, and Simon Thompson. Program equivalence in an untyped, call-by-value lambda calculus with uncurried recursive functions, 2022. doi:10.48550/arXiv.2208.14260.

[HK81] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, 1981. doi:10.1109/TSE.1981.231113.

[HLM$^+$19] Yining Hu, Madhusanka Liyanage, Ahsan Mansoor, Kanchana Thilakarathna, Guillaume Jourjon, and Aruna Seneviratne. Blockchain-based smart contracts - applications and challenges, 2019. doi:10.48550/arXiv.1810.04699.

[HN88] Mehdi T. Harandi and Jim Qun Ning. Pat: a knowledge-based program analysis tool. In *Proceedings. Conference on Software Maintenance, 1988.*, pages 312–318, 1988. doi:10.1109/ICSM.1988.10182.

[HN90] Mehdi T. Harandi and Jim Qun Ning. Knowledge-based program analysis. *IEEE Software*, 7(1):74–81, 1990. doi:10.1109/52.43052.

[HYL21] Tharaka Hewa, Mika Ylianttila, and Madhusanka Liyanage. Survey on blockchain based smart contracts: Applications, opportunities and challenges. *Journal of Network and Computer Applications*, 177:102857, 2021. doi:10.1016/j.jnca.2020.102857.

[JLW12] Woosung Jung, Eunjoo Lee, and Chisu Wu. A survey on mining software repositories. *IEICE TRANSACTIONS on Information*, E95-D(5):1384–1406, May 2012. doi:10.1587/transinf.E95.D.1384.

[JMSG07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, 2007. doi:10.1109/ICSE.2007.30.

[JQW$^+$23] Erya Jiang, Bo Qin, Qin Wang, Zhipeng Wang, Qianhong Wu, Jian Weng, Xinyu Li, Chenyang Wang, Yuhang Ding, and Yanran Zhang. Decentralized finance (defi): A survey, 2023. arXiv:2308.05282.

[KDMM+96] Kostas A. Kontogiannis, Renato De Mori, Ettore Merlo, Michael Galler, and Morris Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1):77–108, 1996. `doi:10.1007/BF00126960`.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. `doi:10.1145/360248.360252`.

[KML11] Shaheen Khatoon, Azhar Mahmood, and Guohui Li. An evaluation of source code mining techniques. In *2011 Eighth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, volume 3, pages 1929–1933, 2011. `doi:10.1109/FSKD.2011.6019877`.

[Kos07] Rainer Koschke. Survey of Research on Software Clones. In *Duplication, Redundancy, and Similarity in Software*, volume 6301 of *Dagstuhl Seminar Proceedings (DagSemProc)*, pages 1–24, Dagstuhl, Germany, 2007. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/DagSemProc.06301.13`.

[KSK+18] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. Tradeoffs in modeling performance of highly configurable software systems. *Software & Systems Modeling*, 18(3):2265–2283, February 2018. `doi:10.1007/s10270-018-0662-9`.

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery. oyente. `doi:10.1145/2976749.2978309`.

[Li21] Chunmiao Li. Gas estimation and optimization for smart contracts on ethereum. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1082–1086. IEEE, 2021. `doi:10.1109/ase51524.2021.9678932`.

[LVBBC+13] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: a threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 477–487, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2491411.2491428`.

[LZ05] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering*

Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, page 306–315, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1081706.1081755.

[Mas87] Henry Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, page 122–126, New York, NY, USA, 1987. Association for Computing Machinery. doi:10.1145/36206.36194.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. doi:10.1109/TSE.1976.233837.

[MCT+23] Oana Marin, Tudor Cioara, Liana Toderean, Dan Mitrea, and Ionut Anghel. Review of blockchain tokens creation and valuation. *Future Internet*, 15(12), 2023. doi:10.3390/fi15120382.

[Mer90] Ralph C. Merkle. One way hash functions and des. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 428–446, New York, NY, 1990. Springer New York. doi:10.1007/0-387-34805-0_40.

[MMD+20] Lodovica Marchesi, Michele Marchesi, Giuseppe Destefanis, Giulio Barabino, and Danilo Tigano. Design patterns for gas optimization in ethereum. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–15, 2020. doi:10.1109/IWBOSE50093.2020.9050163.

[MSC+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL: https://proceedings.neurips.cc/paper_files/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf.

[MVOV96] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC Press, Inc., USA, 1st edition, 1996. doi:10.1201/9780429466335.

[MWM18] Peter Murray, Nate Welch, and Joe Messerman. Erc-1167: Minimal proxy contract. *Ethereum Improvement Proposals, no. 1167*, 2018. URL: https://eips.ethereum.org/EIPS/eip-1167.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *White paper, published on a metzdowd.com mailing list*, 2008.

[Nat15] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical Report Federal Information Processing Standards Publications (FIPS) 202, U.S. Department of Commerce, Washington, D.C., 2015. doi:10.6028/NIST.FIPS.202.

[Nat23] National Institute of Standards and Technology. Digital Signature Standard (DSS). Technical Report Federal Information Processing Standards Publications (FIPS) 186-5, U.S. Department of Commerce, Washington, D.C., 2023. doi:10.6028/NIST.FIPS.186-5.

[NBF+16] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016. Chapter 1 accessible at https://assets.press.princeton.edu/chapters/s10908.pdf.

[NBLV21] Keerthi Nelaturu, Sidi Mohamed Beillahit, Fan Long, and Andreas Veneris. Smart contracts refinement for gas optimization. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 229–236. IEEE, 2021. doi:10.1109/brains52497.2021.9569819.

[NS19] Julian Nagele and Maria Schett. Blockchain superoptimizer. *Preproceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2019)*, pages 166–180, 10 2019. doi:10.48550/arXiv.2005.05912.

[OH92] Paul Oman and Jack R. Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–344, 1992. doi:10.1109/ICSM.1992.242525.

[PBP+15] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, 2015. doi:10.1109/TSE.2014.2372760.

[PC13] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 53–68, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-39799-8_3.

[PKW08] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, August 2008. doi:10.1007/s10664-008-9077-5.

[RC07] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. Technical Report 2007-541, School of Computing, Queen's University, 2007.

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358, March 1953. Dissertation; Origin of Rice's theorem. `doi:10.2307/1990888`.

[SA22] Noama Fatima Samreen and Manar H. Alalfi. Volcano: Detecting vulnerabilities of ethereum smart contracts using code clone analysis, 2022. `doi:10.48550/arXiv.2203.00769`.

[SC06] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: mining for sample code. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 413–430, New York, NY, USA, 2006. Association for Computing Machinery. `doi:10.1145/1167473.1167508`.

[SGW$^+$23] Sven Smolka, Jens-Rene Giesen, Pascal Winkler, Oussama Draissi, Lucas Davi, Ghassan Karame, and Klaus Pohl. Fuzz on the beach: Fuzzing solana smart contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1197–1211, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3576915.3623178`.

[SN24] Robert Susik and Robert Nowotniak. Pattern matching algorithms in blockchain for network fees reduction. *The Journal of Supercomputing*, 80(12):17741–17759, 2024. `doi:10.1007/s11227-024-06115-8`.

[SQLH23] Majd Soud, Ilham Qasse, Grischa Liebel, and Mohammad Hamdaqa. Automesc: Automatic framework for mining and classifying ethereum smart contract vulnerabilities and their fixes. In *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 410–417, 2023. `doi:10.1109/SEAA60479.2023.00068`.

[Sta14] William Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Education, sixth edition, 2014.

[TS16] Florian Tschorsch and Björn Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Communications Surveys & Tutorials*, 18(3):2084–2123, 2016. `doi:10.1109/COMST.2016.2535718`.

[TVI$^+$18] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: static analysis of ethereum smart contracts. In *Proceedings of the 1st*

International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB '18, page 9–16, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3194113.3194115.

[WDZ+13] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage api usage patterns from source code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 319–328, 2013. doi:10.1109/MSR.2013.6624045.

[WFTW24] Konrad Weiss, Christof Ferreira Torres, and Florian Wendland. Analyzing the impact of copying-and-pasting vulnerable solidity code snippets from question-and-answer websites. In *Proceedings of the 2024 ACM on Internet Measurement Conference*, IMC '24, page 713–730, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3646547.3688437.

[WLWC21] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. Non-fungible token (nft): Overview, evaluation, opportunities and challenges, 2021. doi:10.48550/arXiv.2105.07447.

[WO18] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018. doi:10.1109/JPROC.2018.2817118.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Technical report, Ethereum Foundation, 2014. Information at https://github.com/ethereum/yellowpaper, the Yellow Paper is out of date from April 2023 on (information retrieved on June 29th, 2025). URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[XZLH20] Yang Xiao, Ning Zhang, Wenjing Lou, and Y. Thomas Hou. A survey of distributed consensus protocols for blockchain networks. *IEEE Communications Surveys & Tutorials*, 22(2):1432–1465, 2020. doi:10.1109/COMST.2020.2969706.

[YMRP19] Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing ethereum's gas mechanism. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 310–319, 2019. doi:10.1109/EuroSPW.2019.00041.

[ZGH+24] Guoliang Zhao, Stefanos Georgiou, Safwat Hassan, Ying Zou, Derek Truong, and Toby Corbin. Enhancing performance bug prediction using performance code metrics. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 50–62, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3643991.3644920.

[ZLK+21] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, 47(10):2084–2106, 2021. doi:10.1109/TSE.2019.2942301.

[ZXL19] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. Soliditycheck : Quickly detecting smart contract problems through regular expressions, 2019. arXiv:1911.09425.

[ZXZ+09] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 318–343, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.