PhD-FSTM-2026-027
Faculty of Science, Technology and Medicine

Dissertation

Defence held on 10 February 2026 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Fatou Ndiaye MBODJI
Born on 29 December 1993 in Louga (Sénégal)

Trustworthy Software Engineering with Emerging Technologies

Dissertation Defense Committee

Dr. Tegawendé F. Bissyandé, Dissertation Supervisor
Professor, University of Luxembourg

Dr. Jacques Klein, Chairman
Professor, University of Luxembourg

Dr. Samuel OUYA, Member
Professor, University Cheick Anta Diop

Dr. Ousmane Thiaré, Member
Professor, University Gaston Berger

Dr. Gervais Mendy, Member
Professor, University Cheick Anta Diop

Abstract

Emerging technologies, such as Blockchain and Artificial Intelligence, are increasingly integrated into or interact with software systems, introducing new capabilities while also raising critical software engineering trustworthiness concerns. While prior research has largely assessed trustworthiness at the technology level, treating emerging technologies as monolithic entities, such approaches fail to capture how trust-related opportunities or issues originate from specific technological dimensions and their interaction with software engineering processes and artifacts. This thesis addresses this gap by adopting a dimension-aware, software engineering perspective on trustworthiness.

This dissertation investigates the software engineering of trustworthy systems that integrate or interact with emerging technologies, specifically Blockchain and Artificial Intelligence (AI). While these technologies introduce novel capabilities, they also generate complex trustworthiness challenges that cannot be fully addressed by technology-level assessments. Trust-related opportunities and risks often arise from specific technological dimensions and their interactions with software engineering processes and artifacts.

To this end, we propose the Trust in Depth (TID) framework, a software engineering conceptual and formal framework that characterizes trustworthiness as a context-dependent property emerging from the interaction between specific dimensions of emerging technologies and software engineering artifacts. TID enables systematic reasoning about how particular dimensions may reinforce, undermine, or ambiguously affect trust-related software engineering activities, and guides the identification of trust-related issues or opportunities as well as the design of appropriate engineering actions to leverage positive effects or mitigate risks. The thesis instantiates TID through a series of contributions spanning blockchain and AI enabled software systems.

The thesis instantiates TID through five interconnected contributions: Starting with security threats, we investigate cryptojacking in Android applications, where financial gains from cryptocurrency mining motivate malicious developers to embed unauthorized mining code. We expose their evasion strategies, distribution practices, and resource consumption patterns, providing actionable insights for designing detection mechanisms and improving security practices.

Moving to security analysis infrastructure, blockchain's immutability dimension creates a challenge: each new smart contract version receives a different address, breaking the connection between versions. This limits security research that relies on tracing contract evolution, such as studying vulnerability patterns or analyzing how bugs are fixed over time. We develop infrastructure that reconstructs these fragmented histories, enabling researchers to conduct version-based vulnerability

analysis and empirical studies of smart contract evolution.

Addressing comprehension barriers, the blockchain's transparency dimension makes all contract code publicly visible, but this visibility remains underutilized if stakeholders cannot understand what the code does. We leverage AI's reasoning capabilities to automatically evaluate and improve documentation quality, advancing contract comprehensibility to non-technical stakeholders.

Validating AI-based tools, since we rely on AI language models to evaluate documentation quality, we must first verify that these evaluators themselves are reliable. We empirically assess their performance and establish guidelines for integrating AI-based assessment tools into software engineering workflows.

Completing the cycle, we invert the relationship: just as AI advances trustworthy software engineering for systems involving blockchain, blockchain can enhance software engineering for systems involving AI. Poor dataset quality causes time loss through isolated cleaning efforts, errors in model outputs, and abandonment of low-quality datasets. Our SIEVE framework uses blockchain's immutability to anchor continuous, verifiable dataset quality assessments, enabling regulatory compliance and reducing waste in AI development pipelines.

Together, these contributions demonstrate how TID enables systematic analysis of trustworthiness across the blockchain-AI-software engineering nexus. We show how a specific technological dimension, mining, immutability, and transparency, generates both security vulnerabilities and research opportunities that require targeted software engineering interventions. By developing infrastructure for version reconstruction, leveraging AI reasoning for documentation evaluation, validating AI evaluators themselves, and using blockchain to anchor AI dataset quality, we illustrate how trustworthiness emerges from carefully engineered interactions between their dimensions and software engineering artifacts. This dimension centric approach enables practitioners and researchers to identify trust-related challenges early, design appropriate mitigations, and leverage technological synergies to build more trustworthy systems.

We are called to be architects of the future, not its victims.

Richard Buckminster Fuller

Acknowledgements

My sincere thanks go to the members of my dissertation defense committee and my CET. I am deeply grateful to my supervisor, Prof. Tegawendé F. Bissyandé, for his guidance and support throughout this journey. I also thank my co-supervisor, Prof. Kui Liu, for his dedication and for providing me with strong foundations in research. My appreciation extends to the chairman, Prof. Jacques Klein, for his supervision within TruX and for his availability. I am equally thankful to Prof. Gervais Mendy and Prof. Samuel Ouya for their support before and during my PhD, to Prof. Ousmane Thiaré for kindly accepting to serve on my committee, and to Dr. El-Hacen Diallo for his support, our fruitful collaborations on some projects, and his role as an expert on the committee.

I am also grateful to my co-authors: Vinny Adjiby, Moustapha A. Diouf, Mariem C. Sougoufara, W. Christian Ouedraogo, Alioune Diallo and Jordan Samhi for the enriching collaborations we shared.

My sincere thanks go to the TruX Research Group and to all members of the Laboratoire LITA for the stimulating research environment and the collegial atmosphere.

I would also like to thank Ms. Fiona Levasseur and Ms. Jessica Giro for their responsiveness and support.

Finally, I express my deepest gratitude to my family, to whom I dedicate this dissertation.

Fatou Ndiaye MBODJI
University of Luxembourg
February 2026

Dedication

I dedicate this thesis first and foremost to my family.

To my parents, whom I am truly blessed and grateful to have; and in loving memory of my late paternal grandfather, whom I thank for founding this wonderful family that I am so fortunate to belong to, and of my late maternal grandmother. To every member of this family whether by blood, by marriage, or by former alliance, I express my deepest gratitude.

To the family I am building, especially my husband and my son, who give meaning and balance to my journey.

To my sibling, my cousins, and their spouses and children; to my uncles and aunts, including the two Yatma, Bassirou, Lahat, Pape Sidy, Diaba, Faty Mor, Amy Sandrine, and Fama Gueye; and to my family of the heart: my heart-grandmother, Mame Bintou Sow; my aunt and neighbor, Ndeye Sarr Mbodji; and Taa Aida Ndiaye.

I also dedicate this work to my classmates, teachers, and colleagues from Ndiàng, Sainte Marie, CEMT, NLLG, ESMT, ESP, LITA, and TruX, who have accompanied me at different stages of my academic and professional journey.

My gratitude extends to my colleagues at LuxWays and Thiezhu Sun and his family for their kindness and support during my maternity leave.

Finally, I dedicate this thesis to my colleagues at InnoCoffee.

Fatou Ndiaye MBODJI
University of Luxembourg
February 2026

Contents

List of Figures

List of Tables

1

Introduction

This chapter defines the scope and approach adopted throughout the dissertation. We begin by justifying the urgent need for trustworthiness in systems interacting with AI and blockchain. By analyzing specific motivating examples, we demonstrate why prevailing methods fall short of these challenges. In response, we present our evidence-based framework, synthesize the primary contributions, and provide a comprehensive roadmap for the chapters that follow

Contents

1.1 Thesis scope

Trustworthiness in software systems lacks a universally agreed-upon definition, with researchers holding different understandings of its constituent attributes [3]. Definitions vary depending on context [4], and what needs to be considered for engineering trustworthy systems remains unclear [5]. Furthermore, existing research lacks comprehensive trust models to support trust management across different decision points in software systems [6]. Rather than adopting a single definition, this thesis considers trustworthiness broadly, encompassing any attributes recognized through empirical practice or peer-reviewed research as contributing to stakeholder confidence in software systems including but not limited to security, reliability, transparency, privacy, fairness, and accountability.

The convergence of artificial intelligence, blockchain, and software engineering represents a turning point in technological development [7]. Artificial Intelligence (AI) refers to techniques that enable machines to perform tasks requiring human-like reasoning and decision-making. Blockchain is a decentralized ledger technology that allows secure, transparent, and tamper-resistant record-keeping without intermediaries. Software engineering (SE) provides systematic methods for designing, developing, testing, and maintaining complex software systems[8]. This thesis focuses on this field and investigates software engineering for trustworthy systems that integrate or interact with emerging technologies, specifically blockchain and AI. We identify how specific dimensions of these technologies (e.g., immutability, transparency, data-driven behavior, nondeterminism) affect software engineering activities, creating both challenges and opportunities for achieving trustworthiness. We develop software engineering techniques and tools to address these challenges and leverage these opportunities.

1.1.1 Engineering Trustworthiness Matter in these Systems

Characterized by radical novelty, rapid growth, coherence, prominent impact, and inherent uncertainty [7], emerging technologies are increasingly being integrated into or interacting with contemporary systems[9]. Technologies such as blockchain, artificial intelligence, and the Internet of Things [10] are fundamentally transforming software engineering practices [11], necessitating a deeper understanding of the trustworthiness challenges they introduce [10]. While these technologies introduce new capabilities, they also fundamentally reshape software engineering processes, artifacts, and trustworthiness assumptions.

The emphasis on trustworthiness in this context stems from three key considerations. First, according to various research studies, trust assessment is fundamentally a decision-making problem [12, 13, 14]. Indeed, user acceptance of software systems depends on the trust [15] and therefore on verification and certification of their dependability properties, which form the foundation of trust [16]. Second, emerging technologies introduce distinct trustworthiness concerns that differ from traditional software systems [17, 18]. A tertiary research comparing trustworthy software engineering studies with trustworthy AI studies has revealed that specific considerations including AI model characteristics, attack vectors, and human-centered design approaches were found exclusively in trustworthy AI studies [19]. Third, despite ongoing research efforts, the trustworthiness of the complete system remains an unresolved challenge [5]. These factors collectively explain the research community's growing interest in the Trustworthiness of systems that Integrate or interact with Emerging

technologies. This thesis focuses on this field by specifically examining blockchain and AI as exemplars of emerging technologies that raise critical trustworthiness questions.

1.1.1.1 Novel Characteristics Require Novel Engineering Practices

Both blockchain and AI exhibit characteristics that fundamentally differ from traditional software systems, necessitating adaptations to established software engineering practices.

Blockchain-based software systems. Blockchain technology has attracted significant attention due to its paradigm of decentralizing trust, redistributing reliance from centralized intermediaries to distributed network consensus [20]. The technology's value proposition centers on enabling trustlessness, the ability to produce transactions without placing trust in any third party while delivering attributes such as transparency, immutability, and fault tolerance [21]. However, this paradigm presents a fundamental paradox: trust mechanisms must still be constructed upon what is theoretically a trust-free technological foundation [22]. The technology's effectiveness ultimately hinges on end-user trust, revealing that blockchain does not eliminate trust requirements but rather reconfigures them [23].

These unique characteristics challenge conventional software engineering practices across the entire development lifecycle [24]. Inadequate systematic development approaches have been identified as sources of critical failures: the MtGox attack (2014, $600M), Bitfinex breach (2016, $65M), and DAO attack (2016, $50-60M in Ether) [24], incidents attributed to "unruled and hurried software development" by researchers who advocate for novel engineering practices including new professional roles, enhanced security approaches, specialized modeling languages, and domain-specific metrics [25].

AI-based software systems. Artificial Intelligence represents another transformative technology, propelled by advances in massively parallel computing, extensive datasets, and sophisticated algorithms [26]. Despite its substantial benefits, AI introduces potential risks of adverse outcomes for individuals and society at large [27]. The unique characteristics of AI systems have generated extensive discourse on trustworthiness dimensions including privacy, fairness, and security [19]. While trustworthy AI shares certain quality attributes and practices with trustworthy software particularly in privacy, security, and testing domains, the connection between these fields remains underdeveloped [19]. Existing research identifies the need for context-specific trustworthiness solutions as a fundamental challenge in AI development [27], positioning trustworthy AI as an emerging research area requiring further investigation to strengthen reliability and trustworthiness [27]. These challenges are compounded by methodological tensions between AI and traditional software engineering. The conventional software engineering methodology, characterized by the Specify-Prove-Implement-Verify (SPIV) approach, assumes that problems can be completely circumscribed and surveyed [28]. However, this assumption proves inadequate for AI domains such as natural language processing, where problems resist complete specification. The failure to recognize this incompatibility has produced what some researchers characterize as a Kuhnian paradigm crisis in AI development [28], further emphasizing the need for novel engineering practices adapted to AI's unique characteristics.

1.1.1.2 Bidirectional Opportunities Between Emerging Technologies

Beyond their individual challenges, AI and blockchain exhibit bidirectional relationships with software engineering and with each other[9]. Software engineering methods must adapt to develop trustworthy AI systems (SE4AI) and blockchain systems (SE4Blockchain), as discussed in previous paragraph. Conversely, AI methods can support blockchain development (AI4Blockchain) and general software engineering tasks (AI4SE), while blockchain properties can address AI engineering challenges (Blockchain4AI) and enhance software engineering practices broadly (Blockchain4SE) [17, 9].

AI4Blockchain: Leveraging AI to improve blockchain software engineering. AI capabilities can enhance software engineering practices for blockchain systems. AI-powered natural language processing can automatically generate documentation for smart contracts [29], transforming blockchain's raw code transparency into comprehensible artifacts accessible to auditors and non-technical stakeholders. Machine learning techniques can support vulnerability detection, behavioral analysis, and pattern recognition in smart contracts, addressing critical challenges in blockchain software verification and quality assurance.

Blockchain4AI: Leveraging blockchain to improve AI software engineering. Conversely, blockchain properties can address longstanding software engineering challenges in AI systems through several mechanisms. Blockchain's immutability and transparency enable verifiable, tamper-proof records of dataset provenance, training processes, and model versions. This addresses critical AI engineering concerns including data quality assurance, reproducible experiments, transparent model governance, and accountability areas where traditional software engineering approaches often fall short [9, 30, 31].

In summary, the novel characteristics of emerging technologies such as blockchain and AI necessitate substantial adaptations to software engineering practices. Examining how these technologies affect software engineering enables practitioners to develop the capabilities, skills, and practices necessary to build trustworthy systems in an evolving technological landscape [10].

Engineering Software with Emerging Technologies Matters

Emerging technologies pose distinct challenges and opportunities requiring adaptable software engineering approaches [32].The novel characteristics of blockchain (such as immutability, data redundancy, transparency) and AI (data-driven behavior, probabilistic outputs, opacity) require adaptations to software engineering practices.

1.1.2 Integration and Interaction: as mode of engagement

To comprehensively address trustworthiness, we examine trust formation in two distinct scenarios: software that explicitly *integrates* emerging technologies into its core functionality, and software that *interacts* with these technologies without explicit integration.

As an example of integration, smart contracts, programs deployed on blockchain inheriting properties such as immutability and cryptographic verifiability, can replace classical programs to enhance trust. In the context of loot boxes i.e. monetization mechanisms where players purchase randomized virtual rewards; smart contract

integration has been proposed to provide transparency [33]. They enable players to verify that displayed probabilities match those encoded in the contract, addressing opacity concerns regarding dynamic odds that may vary across players or time periods.

As an example of interaction, cryptojacking in Android applications [34] illustrates how software can interact with blockchain networks without explicitly claiming integration, using device resources illicitly for cryptocurrency mining. Another interaction example involves natural language processing tools based on AI that enhance software transparency through code summarization, enabling non-technical users to understand code [29]. This dual mode perspective ensures comprehensive coverage of trustworthiness dimensions related to emerging technologies.

Emerging technologies such as blockchain and AI fundamentally transform trustworthiness considerations in software systems. Their novel characteristics necessitate adaptations to conventional software engineering methodologies while simultaneously creating opportunities for mutual enhancement. The bidirectional relationships between these technologies and software engineering (AI4SE/SE4AI, Blockchain4SE/SE4Blockchain, AI4Blockchain/Blockchain4AI) reveal synergies that can address trustworthiness challenges through combined approaches. This thesis investigates both integration and interaction modes to develop engineering techniques and tools that leverage these opportunities while mitigating associated risks.

Thesis Scope

This thesis explores trustworthy software systems that integrate or interact with artificial intelligence and blockchain, through collaboration between software engineering, AI, and blockchain, with the goal of achieving system trustworthiness through engineering.

1.2 Motivation and Problem Statement

The illustrative examples presented below demonstrate that, for such an investigation, treating these technologies as inherently trustworthy or untrustworthy is insufficient. Instead, a fine-grained and evidence-based approach is necessary. Trust-building characteristics are most effective when interlinked to the underlying features of the technology itself [23].

1.2.1 Motivating Example 1: Blockchain Transparency as a Double-Edged Sword

Integrating blockchain technology into software systems can influence software trustworthiness by transferring intrinsic technological properties to software behavior. In particular, blockchain transparency enables developers, auditors, and automated verification tools to independently assess system correctness, data integrity, and adherence to expected workflows. From a software engineering perspective, this transparency supports auditability, traceability, and maintainability of critical artifacts.

While this example illustrates the trust-enhancing potential of blockchain transparency, it also provides a foundation for exploring its limitations. In certain contexts, such as business-critical or data-sensitive applications, transparency can introduce privacy, confidentiality, or compliance challenges that must be carefully managed through design, process, or tooling choices. This duality underscores the need for dimension-aware evaluation of emerging technology properties in software engineering.

1.2.1.1 Transparency as a Trust Enabler

In software engineering, transparency supports auditability, verifiability, and correctness of critical system operations. For instance, smart contract logic deployed on a blockchain can be independently inspected to ensure that business rules, access controls, or transaction workflows are faithfully implemented. This property strengthens trust in the software system by reinforcing attributes such as correctness, reliability, and auditability, thereby enhancing confidence in system behavior from both a development and operational perspective.

A prior study by Carvalho [33] illustrates this concept: blockchain transparency allows independent verification of probability distributions governing system processes. This demonstrates how transparency enables stakeholders, including developers and auditors, to validate that system logic behaves as intended. In this context, transparency directly reinforces trust attributes such as verifiability and fairness, illustrating how a technological dimension can enhance trustworthiness in practice

1.2.1.2 Transparency as a Trust Threat

The trust-enabling role of transparency does not generalize to all software contexts. Blockchain transparency results from design choices such as replicated storage and execution across network nodes. Consequently, all data stored on-chain including smart contract inputs and outputs, is visible to participants. While this supports auditability, it may also expose sensitive information or create compliance risks in business-oriented or data-sensitive applications. In such contexts, transparency can undermine trust attributes like privacy, confidentiality, and regulatory compliance.

Mitigation strategies, such as storing data off-chain, introduce trade-offs. Smart contracts cannot directly access off-chain data, limiting automation and reducing

some efficiency and trust guarantees typically associated with blockchain integration [35]. Privacy-preserving techniques such as zero-knowledge proofs (ZKPs), which allow verification of data properties without revealing the data itself, address these concerns [36]. These trade-offs demonstrate that transparency is not inherently trust-enhancing and must be carefully managed within software design, testing, and operational processes.

1.2.2 Motivating Example 2: Learning from Data as a Double-Edged Sword in Software Engineering

One of the intrinsic properties of all modern artificial intelligence systems is their ability to learn from data and extract complex patterns. This capability, often referred to as *learning from data* is key to establish trustworthy AI [27, 37]. In the context of software engineering, this intrinsic property directly affects software trustworthiness, as it can both strengthen and compromise stakeholder trust depending on how the AI is applied.

1.2.2.1 Learning as Trust Enhancement

In software engineering, AI systems can act as powerful assistants by learning from historical software artifacts such as code repositories, bug reports, or test suites. For example, an AI-based code recommendation tool can learn from previous development patterns and suggest relevant code snippets or fixes[38]. By providing accurate and contextually appropriate recommendations, the system demonstrates reliability and efficiency, reinforcing developers' trust in its suggestions. In this context, the intrinsic learning capability enables stakeholders to rely on the software system, directly enhancing trust attributes such as *performance*, *reliability*, and *adaptability*.

1.2.2.2 Learning as Trust Vulnerability

However, the same intrinsic property can also introduce trust risks in software engineering. A particularly illustrative example arises in the context of large language models (LLMs) fine-tuned for code completion or automated refactoring. Lopez et al. [39] highlight that inter-dataset code duplication i.e., code fragments appearing in both pre-training and fine-tuning datasets can artificially inflate performance and create misleading perceptions of model learning. In such cases, the model appears to have learned new programming patterns, while in reality it is largely reproducing memorized code. This can mislead developers into believing that the model provides novel solutions or insights. Furthermore, these duplications can inadvertently expose proprietary or sensitive code fragments, constituting a form of data leakage. Lopez et al. [39] also show that open-source models such as CODEBERT, GRAPHCODEBERT, and UNIXCODER may be particularly affected by this phenomenon, and that the choice of fine-tuning technique can accentuate the risk. In this scenario, the intrinsic learning capability of AI systems functions as a trust threat, undermining attributes such as *transparency*, *integrity*, *reliability*, and *privacy*.

1.2.3 Summary

These two contrasting examples illustrate a key insight for software engineering: a specific properties of emerging technologies can both strengthen and threaten software trustworthiness.

Blockchain transparency is considered even as a data protection/privacy factor that fosters initial distrust formation on blockchain[40], our example 1 shows that this technological dimension of the blockchain can either reinforce or undermine trustworthiness depending on the context and stakeholder expectations. Similarly, a single intrinsic AI property, learning from data, can simultaneously function as a trust enabler and a trust threat. Its impact on trustworthiness depends on the software context, the characteristics of the dataset, and the expectations of developers or end-users. Recognizing this duality is essential for systematically evaluating the trustworthiness of AI-assisted software systems

1.2.4 Problem Statement

Investigating software trustworthiness is inherently complex. There is no universally accepted definition of trust or trustworthiness in software engineering. Attributes considered essential for trust vary across studies, assessment outputs differ in format and granularity, and methods for evaluating trustworthiness are highly inconsistent. While some works focus on individual trust attributes, others propose comprehensive models [3]. However, systematically integrating these attributes into practical software engineering workflows remains a major challenge [14].

This lack of standardization directly impacts software engineering pratices for trustworthy systems. Engineers face uncertainty in selecting appropriate metrics, validating system behaviour, and justifying trust to stakeholders, which can lead to fragmented assessments, inconsistent practices, and potential biases.

This gap highlights a critical need for a dimension-aware, evidence-based framework that links software artifacts, engineering processes, and emerging technology properties to specific trustworthiness outcomes. The Trust in Depth (TID) methodology proposed in this thesis addresses this need by providing systematic guidance for analyzing, evaluating, and improving trustworthiness in software engineering contexts.

Takeaways: Need for a SE-oriented Approach

Blockchain and AI properties can simultaneously enhance and undermine different trust attributes, creating complexity for software engineering practices. A systematic, dimension-aware engineering approach is therefore essential to design, evaluate, and maintain trustworthy software systems.

1.3 Objectives and Proposal

This thesis pursues two interconnected objectives that address fundamental challenges in engineering trustworthiness in software systems integrating emerging technologies.

We aim to develop a dimension-aware analytical approach that captures the distinctive trustworthiness concerns introduced by emerging technologies. This objective responds to three persistent challenges: (i) the lack of standardized trustworthiness attributes, which complicates comparison and reuse across projects, (ii) the difficulty of integrating multiple trust-related characteristics in design, testing, and evaluation processes, leading to fragmented assessments, and (iii) the focus of prior work on individual attributes, which fails to capture interdependencies critical for engineering trustworthy systems.

Then, we aim to operationalize this approach within software engineering practices, to systematically analyze, evaluate, and mitigate trustworthiness risks in systems leveraging or interacting with blockchain and AI. By applying the approach across diverse contexts and technology combinations, the thesis establishes its practical utility, generalizability, and relevance to real-world software engineering workflows.

To achieve these objectives, we propose the *Trust in Depth* (TID) methodology, which links technological dimensions, software artifacts, and engineering processes to specific trustworthiness outcomes. TID is grounded in empirical evidence and peer-reviewed research, ensuring that only validated attributes and properties are considered. By providing a structured framework for reasoning about trustworthiness in design, implementation, and validation activities, TID facilitates interoperability with existing SE frameworks, supports benchmarking, and enables systematic interventions to reinforce or recover trust in software systems.

Thesis Objectives

This thesis pursues two interconnected objectives: (i) adapting software engineering practices to newly introduced trust challenges, and (ii) leveraging the distinctive capabilities of emerging technologies to reinforce trust in software systems.

1.4 Contributions

In this section, we summarize the contributions of this dissertation: the *Trust in Depth* approach and five empirical contributions that collectively operationalize it within software engineering activities such as security analysis, software evolution, documentation, and system validation. Each contribution addresses software trustworthiness by either exposing vulnerabilities introduced by specific dimensions of emerging technologies or by demonstrating how these dimensions can be leveraged to reinforce trust in concrete software artifacts and engineering workflows.

1.4.1 Mining Dimension: Cryptojacking Malware [34]

The first contribution examines how the mining dimension of blockchain technology exhibits dual trustworthiness implications depending on context. While mining serves a legitimate and essential purpose in maintaining blockchain network security and consensus, the same mechanism can undermine software trustworthiness when repurposed in unintended contexts. Through an empirical study of cryptojacking in the Android ecosystem, this work reveals how mining code, when executed without authorization within third-party applications, introduces stealthy resource consumption that impacts system reliability. The investigation demonstrates the extent to which such applications evade detection, operate with limited visible impact, and spread across diverse application categories. This contribution illustrates how a single dimension can transition from a trust-reinforcing mechanism in its native context to a serious trustworthiness threat when embedded elsewhere, establishing the necessity for dimension-aware analysis that accounts for contextual appropriateness. This contribution informs security-oriented analysis and detection practices, highlighting blind spots in existing malware detection pipelines and motivating improvements to analysis tools used during app vetting and runtime monitoring.

1.4.2 Immutability Dimension: Preserving Data, Fragmenting Evolution [41]

The second contribution investigates the immutability dimension of smart contracts, revealing a fundamental tension between data preservation and software evolution. Immutability is frequently presented as a trust-enhancing property that prevents unauthorized tampering and ensures data integrity. However, this same dimension creates significant challenges for software maintainability when contracts contain faults or require updates. Because evolution necessitates redeployment at new addresses, version histories become fragmented, limiting traceability and hindering systematic program repair. This contribution introduces ContractTrace, an automated infrastructure that reconstructs coherent version lineages by leveraging the proxy pattern, enabling fault prevention and removal without sacrificing the immutability guarantee. The work demonstrates how careful dimension-level understanding enables targeted solutions that address trustworthiness challenges while preserving beneficial properties, illustrating the importance of dimension-specific rather than technology-wide interventions. It directly addresses software evolution and maintenance challenges in blockchain-based systems, providing engineers with tooling support to analyze version histories, study vulnerability propagation, and reason about repair strategies despite immutability constraints.

1.4.3 Transparency Dimension: Visible Yet Incomprehensible [2]

The third contribution addresses a critical gap between transparency as a blockchain property and transparency as experienced by stakeholders. While blockchain systems are commonly lauded for transparency that enables auditability, smart contracts frequently remain opaque to non-expert stakeholders, particularly when documentation is poor or absent. This comprehension gap undermines the trustworthiness benefits that transparency should provide, as auditability requires not merely visibility but understanding. This contribution introduces evalSmart, a tool that applies Large Language Models as automated evaluators of smart contract documentation quality. By leveraging AI's semantic analysis capabilities to assess and improve blockchain artifact comprehensibility, this work demonstrates how dimensions from different technologies can interact synergistically: AI capabilities enhance the practical trustworthiness impact of blockchain's transparency dimension. This convergence scenario illustrates how cross-technology dimension interactions can address limitations that exist within a single technology's dimensional space. By targeting documentation as a first-class software artifact, this contribution strengthens program comprehension, auditing, and maintainability, positioning documentation quality as a measurable and engineerable dimension of trustworthiness.

1.4.4 Reliability Dimension: Who Validates the Validators?

The fourth contribution confronts a critical meta-trustworthiness challenge that emerges as AI-based evaluators become increasingly prevalent in software engineering. When Large Language Models or other AI systems are employed to assess software artifact quality, a fundamental question arises: if the evaluators themselves lack demonstrated reliability, how can we trust the evaluations they produce? This contribution conducts a systematic analysis of LLM-based evaluator reliability in the context of smart contract documentation assessment, comparing multiple model configurations against expert human judgments. The investigation identifies factors that influence evaluator reliability and establishes guidelines for trustworthy AI-assisted evaluation. This work illustrates a dimension-level concern, the reliability dimension of AI capabilities that must be validated before AI can credibly serve as a trust-reinforcing mechanism for other technologies. The contribution demonstrates that dimension-aware analysis must extend to examining the trustworthiness properties of dimensions themselves, not merely their effects on target systems. This contribution is particularly relevant to software validation and quality assurance, as it provides evidence-based guidelines for safely integrating AI-based evaluators into software engineering pipelines.

1.4.5 Verifiability Dimension: From Claims to Proofs[42]

The final contribution addresses trustworthiness challenges in AI pipelines, which depend heavily on large datasets whose quality and compliance are often uncertain and inadequately documented. Existing documentation practices rely primarily on narrative descriptions that provide limited guarantees and lack verifiable evidence, leaving stakeholders unable to justify trust in dataset quality. This contribution proposes SIEVE, a framework that transforms dataset quality assessments into machine-verifiable Confidence Cards anchored on blockchain infrastructure. By leveraging blockchain's verifiability dimension to provide immutable, auditable records of

dataset quality checks, SIEVE enables continuous and reproducible verification of data reliability, robustness, and regulatory compliance. This work exemplifies bidirectional convergence: blockchain's verifiability dimension strengthens trustworthiness in AI systems, just as AI's analytical capabilities enhanced blockchain transparency in the third contribution. Together, these convergence contributions demonstrate how dimensions from different technologies can be strategically combined to address trustworthiness challenges that individual technologies cannot resolve in isolation. SIEVE supports data engineering and compliance-oriented software processes by enabling reproducible and verifiable dataset quality assessments, turning informal documentation into auditable software artifacts.

1.4.6 Overall Perspective

Taken together, the five contributions presented in this thesis operationalize the Trust in Depth approach across a diverse set of analytical and engineering scenarios. They cover isolated dimension analyses (first and second contributions), unidirectional cross-technology enhancement (third contribution), reflexive dimension validation (fourth contribution), and bidirectional technological convergence (fifth contribution). At the same time, these contributions span a broad range of software engineering contexts, including security analysis, system evolution, documentation, validation, and data engineering.

Overall, this thesis demonstrates that justifying trust in software systems integrating or interacting with emerging technologies requires fine-grained, dimension-aware analysis that is embedded within software engineering workflows, rather than coarse, technology-level abstractions. By systematically examining how emerging technology dimensions influence trustworthiness both individually and through their interactions, this work advances the theoretical foundations of trustworthiness analysis while providing practical guidance for engineering trustworthy software systems in an era of rapid technological change.

Our approach and contributions

This thesis contributes the Trust in Depth approach a dimension-aware methodology for trustworthiness analysis operationalized through five empirical studies that examine how blockchain and AI dimensions individually threaten or reinforce software trust, and how their convergence creates novel trustworthiness dynamics requiring systematic investigation.

1.5 Roadmap

The remainder of this thesis is organized into three main parts.

Part I, *Foundations and Approach*, establishes the conceptual and methodological basis of the dissertation. Chapter 2 introduces the background of the study by clarifying core terminology and foundational concepts related to trustworthiness, software engineering, and emerging technologies. It discusses how trustworthiness is conceptualized in software systems involving blockchain and artificial intelligence, examines their confluence with software engineering, and identifies key gaps that motivate the thesis orientation. Chapter 3 presents the *Trust in Depth* approach. It details its conceptualization and component modeling, illustrates dimension-aware analysis through concrete examples, explains how the approach is operationalized across the empirical contributions, and outlines its methodological foundations.

Part II, *Contributions*, comprises the empirical studies that apply the Trust in Depth approach to concrete software engineering contexts. Chapter 4 investigates the mining dimension of blockchain through an empirical study of cryptojacking malware on Android, analyzing how illicit mining practices undermine user trust and platform security. Chapter 5 addresses the immutability dimension of blockchain and its implications for software maintenance and security analysis, introducing *ContractTrace*, an infrastructure for reconstructing smart contract lineages despite immutability constraints. Chapter 6 focuses on blockchain transparency and explores how large language models can be leveraged to improve and evaluate the quality of smart contract documentation, thereby enhancing comprehensibility and trust. Chapter 7 examines the convergence of blockchain and AI in the context of dataset quality, presenting the *SIEVE* framework for blockchain-anchored, verifiable certification of AI code datasets.

Part III, *Synthesis and Outlook*, consolidates and reflects on the contributions of the thesis. Chapter 8 synthesizes the findings across all empirical studies, discusses how dimension-aware analysis advances the engineering of trustworthiness, and reflects on the broader implications of integrating AI and blockchain in software systems. It also outlines promising directions for future research.

Through this structure, the thesis provides both theoretical contributions to the engineering of trustworthiness in software systems involving emerging technologies and concrete, evidence-based guidance for software engineers and organizations seeking to design, analyze, and govern trustworthy systems that integrate or interact with blockchain and artificial intelligence.

Part I

Foundations and Approach

This part establishes the conceptual and methodological foundations of the thesis. Chapter 5 introduces essential terminology, examines trustworthiness challenges at the intersection of software engineering, AI, and blockchain, and identifies critical gaps motivating our research. Chapter 3 presents Trust in Depth, our dimension-aware framework for reasoning about trust in systems integrating emerging technologies. We introduce a component-based model, illustrate its application through blockchain transparency, and outline its operationalization through our empirical contributions the next part II.

2

Background

This chapter presents the foundational concepts and terminology necessary for understanding trustworthiness in software systems that integrate or interact with emerging technologies, specifically artificial intelligence and blockchain. It introduces the core properties of these technologies and examines how trustworthiness is defined, assessed, and addressed in software engineering, AI, and blockchain contexts. The chapter further explores bidirectional relationships between software engineering and these technologies, illustrating how each domain can support and influence the other.

Contents

2.1 Terminology and Foundational Concepts

2.1.1 Software Engineering and Software Systems

Software engineering focuses on the systematic study and management of software systems, encompassing their specification, development, operation, and maintenance [43, 44]. It provides principles, methods, and practices for constructing software that satisfies functional requirements while ensuring non-functional quality attributes such as reliability, security, maintainability, and performance. Software systems are the main artifacts produced by these activities, composed of interacting components and their execution environments, ranging from simple applications to complex distributed systems. Their analysis extends beyond functional correctness, emphasizing the quality and dependability essential for stakeholder confidence.

2.1.2 Trust and Trustworthiness

Trust represents a subjective judgment by a *trustor* regarding the expected behavior of a *trustee* under uncertainty and potential risk [45], whereas trustworthiness denotes the objective capability of a system to perform as expected [46]. In software engineering, trust is often conflated with system properties; however, it emerges from stakeholder assessments based on observable system behavior, process evidence, and contextual factors [47]. Trustworthiness, in contrast, encompasses multiple quality dimensions and is evaluated through verification, testing, and empirical evidence [19]. Distinguishing perceived trust from actual trustworthiness is critical, especially for intelligent systems where inappropriate reliance may result in significant harm.

2.1.3 Emerging Technologies

Emerging technologies exhibit radical novelty, rapid growth, internal coherence, significant potential impact, and high uncertainty [7]. Blockchain and artificial intelligence exemplify current emerging technologies, transforming traditional practices and presenting novel engineering challenges. Software engineering practices evolve to accommodate these domains, requiring adaptive methods for code generation, testing, verification, and data management [10, 32]. AI supports multiple software engineering stages, from requirements analysis and architecture design to coding, testing, and maintenance, introducing both productivity gains and trustworthiness considerations. Similarly, blockchain enables new software engineering practices by providing transparency, immutability, automated verification through smart contracts, and auditable development workflows.

2.1.4 Blockchain Technology

Blockchain is a distributed ledger technology enabling secure, tamper-resistant, and transparent record-keeping without a central authority [20]. Its core components include cryptographic primitives, nodes maintaining replicated ledgers, transactions, and consensus mechanisms [48]. Key properties such as decentralization, immutability, and transparency enhance integrity, auditability, and accountability [24], shifting trust from intermediaries to the technology itself. Smart contracts, autonomous and tamper-proof programs executing predefined rules on a blockchain, automate transaction enforcement and agreements, forming the basis for novel software paradigms while introducing specialized challenges for secure development and verification.

2.1.5 Artificial Intelligence

Artificial Intelligence enables machines to perform tasks requiring human-like reasoning and decision-making [26]. Unlike traditional software, AI exhibits data-driven and probabilistic behavior, with functionality emerging from patterns learned from training data rather than explicit programming. This characteristic necessitates specialized validation and monitoring to ensure reliability, fairness, and robustness. AI encompasses a range of techniques and paradigms, including symbolic reasoning, machine learning, and deep learning, each contributing distinct capabilities as well as unique trustworthiness challenges. Opacity, particularly in complex models such as deep neural networks, complicates verification, accountability, and stakeholder confidence, motivating research into explainable AI methods. Understanding and managing trust in AI systems requires integrating evidence from system behavior, development artifacts, and context-specific assessments, forming the basis for reliable and trustworthy deployment in software systems [27].

2.2 Trustworthiness in Software Systems with Emerging Technologies

Trustworthy software engineering seeks to systematically ensure that systems meet specified quality and dependability requirements [49]. Trustworthiness emerges from holistic system behavior and interactions with stakeholders. Dimensions including security, reliability, availability, safety, maintainability, privacy, and fairness are evaluated through evidence and verification [50, 16]. Assessment guides stakeholder decisions regarding system reliance, particularly under uncertainty [12].

Blockchain systems reconfigure trust, providing transparency, immutability, and decentralized consensus while still relying on user trust for effective operation [21, 22]. Security, integrity, and reliability remain critical, with development practices needing adaptation to address vulnerabilities in smart contracts and distributed execution [24]. AI systems introduce additional trustworthiness dimensions such as explainability, fairness, robustness, and accountability [51, 27]. Their probabilistic and data-driven nature [52] necessitates specialized testing, monitoring, and human-centered oversight.

2.3 Confluence Between Software Engineering, AI, and Blockchain

The interaction between software engineering, AI, and blockchain exhibits bidirectional characteristics, formalized using the X4Y notation. In this framework, X represents the technology providing capabilities, and Y denotes the domain being enhanced. AI supports software engineering tasks (AI4SE), while software engineering methods facilitate AI development (SE4AI) [17]. Similarly, blockchain enables novel software engineering practices (Blockchain4SE), while SE adapts to blockchain-specific requirements (SE4Blockchain). Engagement occurs through integration, where emerging technologies are embedded into system core functionality, or interaction, where technologies are accessed as external services. Each mode entails distinct trustworthiness implications that must be addressed systematically.

Blockchain-oriented software engineering (SE4Blockchain) confronts challenges inherent to distributed ledgers and smart contracts [53]. Immutability, decentralized consensus, and cryptographic security necessitate rigorous verification, formal

methods, and specialized development tools, including domain-specific languages and auditing procedures. Conversely, blockchain enhances traditional software engineering (Blockchain4SE) by providing transparent, tamper-proof records, smart contract automation, and auditable development processes [54, 55]. These capabilities are particularly valuable in distributed or outsourced development, supporting secure transactions, verifiable reward mechanisms, and license compliance for third-party components [56].

Software engineering for AI (SE4AI) addresses the unique characteristics of AI systems, emphasizing data quality, model validation, continuous monitoring, and lifecycle management [57]. Practices include version control for datasets and models, behavioral testing beyond code correctness, and documentation capturing assumptions, data lineage, and operational contexts. AI methods supporting software engineering (AI4SE) include code generation, bug detection, automated testing, requirements analysis, and maintenance assistance [58, 59]. While these approaches can improve productivity and quality, they require careful integration, continuous performance validation, and consideration of trust and over-reliance risks [60].

Blockchain also supports AI development (Blockchain4AI) by providing immutable records of datasets, training processes, and model deployments, enhancing reproducibility, accountability, and federated learning while safeguarding data integrity and privacy [9, 30, 61]. In turn, AI techniques enhance blockchain engineering (AI4Blockchain) by generating comprehensible smart contract documentation, detecting vulnerabilities through machine learning, analyzing network behavior, optimizing performance, and complementing human auditing practices [29, 62].

2.4 Gaps and Thesis Orientation

Engineering trustworthiness in software systems is inherently complex. There is no universally accepted definition of trust or trustworthiness, and the set of attributes considered essential varies across studies. Assessment outputs differ in format and granularity, and methods for evaluating trustworthiness remain highly inconsistent [3, 14]. This heterogeneity introduces uncertainty for software engineers, who must select appropriate metrics, validate system behavior, and justify trust to stakeholders, often resulting in fragmented assessments and inconsistent practices.

Emerging technologies such as blockchain and AI introduce additional trust-related considerations. Their unique properties can simultaneously enhance and undermine different trust attributes. Blockchain offers transparency, immutability, and auditability, which can reinforce reliability and accountability, but also introduces challenges in security, privacy, and governance. AI systems provide automation, predictive capabilities, and intelligent decision support, but raise concerns related to opacity, fairness, robustness, and explainability. Consequently, software engineering practices must adapt not only to mitigate new risks but also to leverage the opportunities these technologies offer to improve system trustworthiness.

Addressing these challenges and opportunities requires a systematic, dimension-aware approach that links technological properties, software engineering artifacts, and processes to specific trustworthiness outcomes. Such an approach enables engineers to manage emerging trust concerns, integrate technology-specific advantages, and ensure consistent, evidence-based reasoning about trust across the software lifecycle. This thesis proposes the *Trust in Depth* (TID) methodology to meet this need, providing structured guidance for analyzing, evaluating, and enhancing trustworthiness in systems that integrate or interact with AI and blockchain.

2.5 Summary

This chapter has introduced the foundational concepts and terminology necessary to understand trustworthiness in software systems that integrate or interact with emerging technologies. It highlighted the multidimensional and context-dependent nature of trust, and how software engineering practices are challenged by the evolving properties of artificial intelligence and blockchain. These technologies both introduce new trust concerns and provide opportunities to enhance software engineering processes.

By examining bidirectional interactions between software engineering, AI, and blockchain, and illustrating practical examples of systems and tools, this chapter established the conceptual and technical context for addressing trustworthiness. The identified challenges and opportunities motivate the need for a systematic, dimension-aware engineering approach, which forms the basis for the methodological contributions presented in subsequent chapters.

3

Approach: Trust In Depth

This chapter presents the trust in Depth appraoch/

Contents

Trustworthiness in software systems integrating or interacting with emerging technologies has traditionally been addressed at the technology level, often treating technologies such as Artificial Intelligence or Blockchain as monolithic entities. However, trustworthiness concerns do not arise uniformly across a technology; instead, they emerge from specific technological dimensions and their interaction with software engineering processes and artifacts within a given system. Technology-level assessments fail to capture these nuanced and sometimes contradictory effects, motivating a dimension-centric perspective.

The *Trust in Depth* (TID) framework adopts this perspective, analyzing trustworthiness through explicit relationships between emerging technology dimensions and software engineering processes or artifacts.

3.1 Conceptualization and Component Modeling of Trust in Depth

To formalize and operationalize this framework, we propose a **Component Model**, which represents the core elements and their interactions as modular entities suitable for engineering reasoning.

3.1.1 Core Components

The TID Component Model includes the following primary elements:

- **System Context (S):** Represents the specific software system under study that integrates or interacts with an emerging technology. This component captures operational conditions, platform constraints, stakeholder expectations, and regulatory requirements.
- **Emerging Technology (E):** Refers to the evolving technology under analysis, such as AI or Blockchain, characterized by rapid evolution, limited maturity, and variable usage practices.
- **Technology Dimension (C):** Represents one or more specific properties or capabilities of an emerging technology (e.g., mining, immutability, transparency). Formally, $C = \{c_1, c_2, \ldots, c_n\}$, where $n \geq 1$. Dimensions serve as the primary analytical units influencing trustworthiness.
- **Software Engineering Process or Artifact (A_j):** Captures existing engineering processes or artifacts affected by technology dimensions, such as validation, documentation, or malware analysis infrastructure.
- **Engineering Advancement (EA_k):** Represents new artifacts, tools, or process improvements developed to mitigate threats or reinforce trustworthiness in response to observed dimension effects.
- **Trustworthiness Effect (f):** Models the influence of a technology dimension on a process or artifact. Effects can be positive (trust-reinforcing), negative (trust-threatening), or neutral, depending on context and interaction with other components.

3.1.2 Component Interactions

Within this model, the interactions are formalized as:

$$f : (S, E, C, A_j) \rightarrow \{-1, 0, +1\},$$

where f denotes the observed trustworthiness effect of dimension c_i on artifact A_j in system S under technology E. Engineering advancements arise conditionally as:

$$EA_k = \begin{cases} g(S, E, C, A_j), & \text{if } f(S, E, C, A_j) < 0 \\ \varnothing, & \text{otherwise} \end{cases}$$

Finally, the dimension-specific trustworthiness impact is aggregated through:

$$TW_C(S) = h\Big(f(S, E, C, A_j), EA_k\Big),$$

where h accounts for contextual factors and interactions without implying a system-wide effect.

3.1.3 Benefits of the Component Model

Modeling Trust in Depth (TID) as a Component Model provides a modular and systematic perspective on trustworthiness in software systems. Each element: system context (S), emerging technology (E), technology dimensions (C), software engineering artifacts (A_j), and engineering advancements (EA_k), is represented explicitly, along with their interdependencies. This clarity facilitates reasoning about how emerging technology dimensions impact specific software engineering artifacts and where engineering interventions may be needed.

The component-based approach naturally supports extensions to multiple technologies, dimensions, or artifacts, enabling cross-technology and reflexive analyses. Moreover, it integrates seamlessly with formal engineering methods, design patterns, or simulation frameworks, bridging the gap between conceptual trustworthiness and practical software engineering applications.

3.1.4 Analytical Configurations

Building on the explicit component representation, the Trust in Depth framework supports several analytical configurations that structure the examination of trustworthiness impacts. In its simplest form, the framework enables isolated dimension analysis, where a single technology dimension is examined in relation to a specific software engineering artifact, such as the influence of blockchain immutability on maintenance processes.

Beyond isolated cases, the framework accommodates context-dependent analysis, in which the same technology dimension may produce differing or even opposing effects depending on the system context. For example, transparency may enhance verification and accountability in one system while undermining confidentiality in another.

The framework also supports cross-technology enhancement scenarios, where a dimension originating from one emerging technology amplifies or mitigates the effects of a dimension from another. An illustrative case is the use of AI-based semantic analysis to enhance the practical exploitation of blockchain transparency.

Finally, the framework enables reflexive analysis, addressing situations in which the software engineering artifacts under consideration are themselves evaluative mechanisms. In such cases, the trustworthiness of these mechanisms, such as AI-based code evaluation or analysis tools, becomes an object of assessment in its own right.

By supporting these analytical configurations, the framework operationalizes the component model and enables systematic reasoning about how technology dimensions

and their interactions shape trustworthiness across diverse software engineering contexts.

3.2 Illustrative Example: Blockchain Transparency in Dual Contexts

To demonstrate the framework's core mechanics, we formalize a scenario where the same blockchain dimension produces opposite effects on software engineering artifacts across two system contexts. Blockchain transparency, the property that all on-chain data and transaction history remain publicly visible to network participantsserves as our illustration. While frequently cited as a trust enabling feature, transparency actual impact on software engineering artifacts depends critically on system context and stakeholder requirements.

3.2.1 Context A: Verification of Probabilistic System Behavior

Software systems implementing probabilistic algorithms face a fundamental verification challenge: stakeholders must confirm that documented probability distributions match actual implementation. Verification and validation constitute fundamental software engineering processes that assess whether a system is correctly built and meets requirements [**?** **?**]. However, when users cannot access source code or observe internal state, traditional V&V processes remain opaque and rely entirely on trust in the operating organization.

Consider a representative case: a gaming platform implementing randomized reward distributions, commonly called loot boxes, where players purchase probabilistic access to virtual items. The software engineering challenge is *verifiable fairness*, enabling stakeholders to independently confirm that displayed probabilities correspond to deployed implementation without relying solely on operator attestations. This represents a broader class of probabilistic systems requiring stakeholder-accessible verification mechanisms, including lottery systems and fair allocation algorithms.

By deploying probability distribution logic within a blockchain smart contract, we can formalize this scenario. The system context S_1 represents a probabilistic system requiring stakeholder-verifiable fairness guarantees. The emerging technology E is blockchain, specifically leveraging two dimensions: programmability c_1 (smart contract execution capability) and transparency c_2 (public code visibility). The affected artifact A_1 is the verification and validation process itself. The trustworthiness effect is positive: $f(A_1 \mid S_1, E, \{c_1, c_2\}) > 0$.

The implied dimensions are programmability that makes Blockchain able to provide a flexible script code system to create smart contracts, currency, or other decentralization applications[63]; and blockhain transparency. These two dimension interact synergistically to transform V&V practices in ways neither dimension could achieve alone. Programmability enables probability logic to be encoded as executable smart contract code with deterministic execution across all network nodes. However, programmability alone would not enable stakeholder verification if the code remained proprietary or access-restricted. Transparency makes deployed code and execution history publicly visible to all network participants. However, transparency alone such as public transaction logs without executable logic would reveal outcomes but not the underlying algorithmic implementation.

Together, these dimensions enable any stakeholder to independently inspect code, execute verification procedures on their own infrastructure, and confirm correct implementation. This architectural transformation shifts V&V from centralized, trust-based, temporally-bounded processes conducted by developers or contracted auditors to distributed, trustless, continuously-accessible verification performed by stakeholders themselves. The transformation directly enhances auditability and verifiability as software engineering quality attributes.

Because programmability and transparency naturally align with verification requirements in probabilistic systems, no engineering intervention is needed. Thus $EA_1 = \varnothing$, and the trustworthiness outcome $TW_{\{c_1,c_2\}}(S_1)$ reflects enhanced trust through transformed V&V practices enabled by dimension interaction.

3.2.2 Context B: Multi-Party Systems with Confidentiality Requirements

Multi-organizational systems face conflicting engineering requirements: transaction validity must be verifiable across parties to prevent fraud and ensure coordination correctness, yet business-sensitive data must remain confidential to protect competitive advantage and comply with regulatory constraints. Data management and access control implementation in such systems must reconcile these fundamentally opposing demands.

Consider a representative case: a consortium blockchain coordinating supply chain operations among competing companies, tracking pricing negotiations, supplier identities, inventory levels, strategic partnerships, and competitive intelligence. This exemplifies a broader class of multi-party coordination systems including inter-organizational workflows, federated databases, and collaborative platforms where participants require mutual verification capabilities without exposing proprietary information. Traditional software engineering approaches to confidentiality rely on application-layer access control mechanisms, role-based permissions, and encryption, typically assuming a trusted central authority mediating data access.

We formalize this scenario with system context S_2 representing a multi-party coordination system with confidentiality requirements. The emerging technology E remains blockchain, but we now examine only the transparency dimension c_2, the same property that proved beneficial in Context A. The affected artifact A_2 is the data management and access control implementation. Crucially, the trustworthiness effect is now negative: $f(A_2 \mid S_2, E, c_2) < 0$.

The same transparency that enhanced V&V in S_1 now fundamentally complicates data management engineering in S_2. Blockchain's architectural requirement for replicated storage and execution across network nodes to achieve distributed consensus conflicts directly with standard software engineering practices for confidentiality. All transaction details stored on-chain become visible to every participating node, rendering conventional access control mechanisms ineffective. The transparency property forces engineers into unsatisfactory trade-offs: either store sensitive data off-chain, sacrificing the automation, immutability, and trust guarantees that motivated blockchain adoption, or accept excessive information exposure that violates confidentiality requirements and regulatory compliance obligations such as GDPR or trade secret protection.

This negative effect on fundamental software engineering practices necessitates intervention. The engineering advancement $EA_2 = g(S_2, E, c_2, A_2)$ corresponds in

practice to the development and integration of novel cryptographic engineering artifacts specifically designed to reconcile transparency with confidentiality. Zero-Knowledge Proofs (ZKPs) represent such an advancement: cryptographic protocols that enable one party to prove to another that a statement is true without revealing any information beyond the statement's validity [36]. In the multi-party coordination context, ZKPs allow organizations to prove transaction validity for instance, that a payment amount matches an invoice total without exposing the underlying sensitive values.

This represents a fundamental shift in software engineering practice for data management. Rather than restricting access to sensitive data through traditional access controls, the architecture restricts access to plaintext while enabling verification of encrypted or committed values. The engineering artifact is not merely a feature addition but a complete redesign of how verification and confidentiality are architected in distributed systems. The trustworthiness outcome $TW_{c_2}(S_2) = h(f(A_2 \mid S_2, E, c_2), EA_2)$ reflects this reconciliation: verification capabilities remain intact, preserving the auditability benefits that motivated blockchain adoption, while confidential information is cryptographically protected, addressing privacy and competitive concerns. This outcome demonstrates how engineering intervention can mitigate dimension-level threats while preserving beneficial properties.

3.2.3 Key Insights

This dual-context example demonstrates three core Trust in Depth principles that distinguish dimension-centric from technology-level analysis.

First, trustworthiness effects originate at the dimension level, not from blockchain as an undifferentiated whole. The impact on software engineering artifacts stems specifically from the transparency dimension, while other blockchain dimensions such as immutability affecting maintenance and evolution processes, or consensus mechanisms affecting performance and scalability engineering would affect different artifacts through entirely different mechanisms. Technology-level analysis treating blockchain monolithically cannot capture these nuanced, dimension-specific effects that vary independently.

Second, dimension effects exhibit context-dependent valence. The identical transparency dimension positively transforms V&V processes in probabilistic systems requiring open verification but negatively impacts data management practices in multi-party systems requiring confidentiality. This valence shift is determined not by any inherent property of transparency itself, but by the alignment between the dimension's architectural characteristics and the specific software engineering requirements within each system context. The same technical feature public data visibility serves as an enabler when verification requirements dominate and as an impediment when confidentiality requirements dominate.

Third, engineering interventions are introduced conditionally based on dimension effects. The framework prescribes new artifacts or process improvements only when natural dimension effects prove counterproductive. In S_1, transparency directly enhances existing verification objectives, requiring no compensating mechanism and yielding $EA_1 = \varnothing$. In S_2, transparency conflicts with established confidentiality practices, necessitating cryptographic innovations that reconcile verification with privacy and producing EA_2 as a new engineering artifact. This conditionality ensures that engineering effort is directed precisely where dimension-level challenges arise,

rather than uniformly across all contexts.

These principles establish why dimension-centric analysis surpasses technology-level assessment for trustworthiness evaluation. Different dimensions of the same technology affect different software engineering artifacts through different mechanisms, producing effects whose valence and magnitude vary across system contexts. Only by analyzing at the dimension level can we capture these variations and develop targeted, context-appropriate engineering responses.

3.3 Operationalization Through Empirical Contributions

These illustrative examples provide concrete instantiation of how emerging technology dimensions affect software engineering artifacts through context-dependent mechanisms. The framework's utility extends beyond these pedagogical scenarios to systematic investigation of real-world systems. Chapters 4 through 8 present five empirical contributions that operationalize Trust in Depth across diverse configurations and technologies.

Chapter 4 examines blockchain's mining dimension through isolated analysis, showing how this mechanism, while essential for network security in its native context, becomes a trustworthiness threat when repurposed in unauthorized contexts such as cryptojacking in mobile applications. The dimension affects resource management and malware detection artifacts in ways that differ fundamentally from its security-enhancing role in blockchain networks.

Chapter 5 investigates the immutability dimension through context-dependent analysis, demonstrating how smart contract immutability simultaneously preserves data integrity and complicates software maintenance and program repair processes. The chapter introduces dimension-aware solutions that enable contract evolution without sacrificing immutability's trust benefits, illustrating conditional engineering intervention responding to negative dimension effects.

Chapter 6 explores cross-technology enhancement, showing how AI's semantic analysis capabilities amplify blockchain transparency's practical impact on documentation comprehension. By leveraging large language models to evaluate smart contract documentation quality, this work demonstrates how dimensions from different technologies can interact synergistically to address limitations that exist within a single technology's dimensional space.

Chapter 7 addresses reflexive dimension validation, investigating the reliability dimension of AI-based evaluators themselves. When AI systems are employed to assess software artifact quality, a fundamental question arises: if the evaluators lack demonstrated reliability, how can we trust their evaluations? This contribution establishes that trustworthiness assessment tools must themselves be evaluated for trustworthiness before they can credibly serve as trust-reinforcing mechanisms.

Chapter 8 demonstrates bidirectional technological convergence, leveraging blockchain's verifiability dimension to strengthen dataset quality assurance processes in AI pipelines. By anchoring dataset quality assessments in immutable, auditable blockchain records, this work shows how blockchain verifiability enhances AI trustworthiness just as AI capabilities enhanced blockchain transparency in Chapter 6.

Collectively, these contributions instantiate the framework across isolated dimension analysis, context-dependent scenarios, unidirectional cross-technology enhance-

ment, reflexive validation, and bidirectional convergence. This diversity demonstrates both the generality and practical applicability of the Trust in Depth approach to engineering trustworthy software systems that integrate or interact with emerging technologies.

3.4 Methodological Foundation

The Trust in Depth methodology operationalizes the conceptual framework through explicit, evidence-based linkages between technology dimensions and trustworthiness attributes. Only dimensions and trustworthiness impacts supported by peer-reviewed literature or empirical experimentation conducted as part of this research are considered valid. This constraint ensures analytical rigor while enabling interoperability with existing trustworthiness frameworks, as the evidence-based foundation provides a stable basis for comparison and synthesis across studies.

Each contribution presented in Chapters 4 through 8 instantiates the conceptual model under a different analytical configuration, collectively demonstrating the framework's capacity to address diverse trustworthiness scenarios. By conceptualizing trustworthiness as a dimension-driven, context-sensitive phenomenon rather than a technology-level property, Trust in Depth provides a unifying analytical lens for understanding how emerging technologies influence software engineering practice. The framework bridges theoretical analysis and practical engineering by explicitly incorporating both the effects of technology dimensions on existing artifacts and the engineering advancements developed to mitigate threats and reinforce trustworthiness where necessary.

3.5 Related work

Trustworthiness assessment in software engineering has been extensively studied from both perceptual and evidence-based perspectives. Prior research distinguishes between experience-based sources, derived from direct or indirect user interactions with software systems, and trust evidence collected from organizational artifacts, community platforms, and technical repositories [14]. Empirical investigations further operationalize these categories by identifying concrete information sources commonly consulted by practitioners, including version control systems, issue tracking platforms, question-and-answer portals, community forums, and security databases [64]. While these studies provide valuable insights into the informational foundations of trust assessments, existing models largely treat such sources as unstructured inputs to user perception, without explicitly modeling their technical origin or their interaction with software engineering processes. In particular, trustworthiness analyses at the technology level tend to consider emerging technologies as monolithic entities, overlooking the heterogeneous and sometimes contradictory effects induced by specific technological properties. To address these limitations, we introduce the *Trust in Depth* approach, which adopts a dimension-centric perspective and explicitly relates emerging technology dimensions to software engineering artifacts and processes, enabling a structured and traceable analysis of trustworthiness in complex software systems.

[50]propose a mathematical programming approach to allocate the trustworthy degree to each sub-attribute of some software attribute appropriately and then to make the trustworthy degree of this attribute maximize under some constraint conditions

Part II

Contributions

This part operationalizes Trust in Depth through five empirical contributions addressing trust challenges at the intersection of software engineering, blockchain, and AI. Chapter 4 examines cryptojacking malware on Android, revealing security vulnerabilities introduced by an illicit integration of code for blockchain mining. Chapter 5 presents ContractTrace for reconstructing smart contract lineages despite immutability; Chapter 6 introduces evalSmarT for LLM-based evaluation of generated smart contract comments ; Chapter 7 examines the reliability of LLM-based evaluation. Chapter 8 reverses the perspective with SIEVE, leveraging blockchain transparency to certify AI data card and for commun ity driven data cleaning for AI-based system. These contributions demonstrate how emerging technologies challenge and advance software engineering practices for trustworthy systems.

4

Illicit Mining : Unwrapping the Cryptojacking Malware Ecosystem on Android

This chapter constitutes an instantiation of the Trust in Depth framework through the study of cryptojacking malware in Android applications. Our empirical analysis of 346 potential mining apps, collected between April 2019 and May 2022, reveals that several applications persist on the Google Play Store despite violating developer policies and exploiting users devices for cryptocurrency mining. These apps leverage the mining dimension of blockchain technology, which directly affects security-oriented software analysis processes by enabling evasion, obfuscation, and covert dissemination practices that undermine existing detection artifacts. While some mining apps consume minimal resources, others remain operational without alerting typical users, demonstrating how this technological dimension can have both subtle and significant effects on malware detection. These findings motivate the development of targeted engineering advancements and provide actionable insights for improving cryptojacking detection in Android applications and more broadly in emerging-technology contexts.

This chapter is based on the work published in the following research paper:

- Adjibi, Boladji Vinny; MBODJI, Fatou Ndiaye; Allix, Kevin et al. 2022 In 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), p. 153-163.

Contents

4.1 Overview

The increased demand for cryptocurrencies following 2017's boom drove a burgeoning interest in mining [65]. The induced competition strengthened the need for miners to join mining pools to expect to make a profit. The introduction of services such as Coinhive, which made it possible to spread mining code across the web with minimal effort, further increased the attractiveness of the activity. It soon led to malicious entities covertly participating in mining pools using abused devices. This activity, known as cryptojacking, affects every type of device, from personal computers to smaller devices such as Android-powered devices. Despite their relatively low computing power, Android devices are an attractive target to hackers who can target more than four billion devices at once [66, 67].

Previous research has, for instance, identified that developers produced more than seven hundred Android mining apps between 2017 and 2019 [68]. The findings from this investigation were confirmed by another study highlighting the active use of the Google Play Store to distribute those apps [**?**]. The results from those studies attest that understanding the cryptojacking phenomenon on Android devices is paramount. Unfortunately, there exist no other studies of mining apps on Android besides those mentioned above. Moreover, the datasets used in those studies do not contain any mining app beyond 2019. Considering that two majors events happened around that time, namely (1) the Google Play Store's policy to ban apps that mine using the device's resources starting from July 2018 [69, 70] and (2) the discontinuation of Coinhive, which alone contributed to more than half of the mining apps in the studied period [71], the lack of research on the subject appears to us as a counter-intuitive outcome. In comparison, many recent studies of web-based [72, 73, 74], and general-purpose miners [75, 76, 77] have been conducted. The prevalence of the cryptojacking problem on Android devices and its expected damaging effects make it essential to understand the various mechanisms currently used to mine cryptocurrencies on Android devices [78]. More specifically, we are interested in answering the following research questions:

- **RQ1:** What are the methods and techniques used by developers to insert crypto mining code into Android apps, and how effective are existing detection techniques against those threats?
- **RQ2:** What processes are put in place by developers to ensure that their mining apps are downloaded and executed by as many users as they want?
- **RQ3**: To what extent does mining activity affect the behavior of an Android device, and how likely is an average user to suspect such action on their device?

Novelty

We investigate those questions with the intent of inspiring the research community into building a more effective defense mechanism against mining apps. Our attempt at addressing those questions consisted of an empirical study of 10 430 681 Android apps collected from various marketplaces between April 2019 and May 2022. To the best of our knowledge, this is the first study that focuses on Android mining apps in this timeframe. It allowed us to unveil an evasion technique based on assigning an audio filename to a gzipped archive containing a mining binary, which the program extracts at runtime. Those results, along with the decreased proportion of Javascript-based miners and the appearance of new crypto coins (Veruscoin, RPCoin, uPlexa), supplement prior findings [68] to provide a complete mapping

of the mining phenomenon on Android. To the best of our knowledge, this is the first study that aims to explain the marketing structure that sustains the mining ecosystem on Android. Our investigation resembles the study by Kotzias et al. [79] which discussed the topic of the sources from which users typically download malware. Unlike this study, our focus is on understanding what could get users off their guard to let apps stealthily mine coins on their devices. Earlier research works suggest the prevalence of gaming, streaming, application download, and adult content platforms with illicit mining [80, 81] on the web, but there is little scholarly work documenting illegal mining on the Android platform. We intend to fill this gap through the present study. Finally, our evaluation of the impact of mining activity, whether covert or overt, provides an up-to-date complement to the work conducted by Clay et al. [78] which estimated a five-fold increase in power consumption due to the execution of web-based miners. Besides the novelty factor, our study extends to binary-based miners and further considers the potential concealment of the mining activity. Our evaluation captures more diversity in the mining apps on Android.

Contributions

Overall, through our study of the suspected mining apps from a technical and non-technical standpoint, we make the following contributions:

1. We identified and duly notified the Google Play Store of the presence of at least five mining apps on their marketplace, which is a blatant violation of the applicable developer policies.
2. We discuss the case of some so-called wallet apps that hide malicious code into gzip archives renamed to have misleading extensions. We postulate that this is a technique likely to be used by evil entities.
3. We demonstrate the prevalence of hacks and cracks of premium Android apps as placeholders for mining code in cryptojacking apps.
4. We establish through various tests that, despite mining apps consuming more resources than regular apps, a social media app such as Facebook is twice as greedy as an evasive miner.

The remaining of this chapter is structured as follows. In Section 4.2, we present our methodology used in the study. Section 4.3 then discusses the findings from the manual analysis of the source code and static artifacts of the suspected mining apps. Next, our results from analyzing the non-technical aspects of the mining apps are described in Section 4.4. We then present our findings from profiling the resource consumption of mining apps in Section 4.5 before a brief discussion on the possible shortcomings of our study in Section 4.6.

4.2 Study Design

4.2.1 Data collection

The ban imposed by Google on mining apps on the Google Play Store in July 2018, along with the suspension of Coinhive late in March 2019 are both critical events that previous research has noted [68]. This study that leveraged apps collected from various companies' security advisories is arguably the only one to be aiming at understanding the crypto mining phenomenon on Android devices; a data collection process used in recent Android malware research [82]. Besides this data collection strategy, researchers have also relied on Android marketplaces such as Google Play

Store [83, 84, 85] or a combination of both [68]. The dissemination of the compiled apps has contributed to the availability of a wide range of Android malware data sets usable for numerous studies. Unfortunately, those datasets are limited due to their age or specificity to a research problem. The lack of diversity in the datasets renders them inappropriate for our work which we wanted to represent the current state of the ecosystem of crypto mining on Android. To this effect, we turned our focus on AndroZoo, arguably the most extensive dataset of Android apps for research [86]. The versatility and up-to-date nature of the dataset, which contains more than 19 million (in May 2022) Android apps collected from various sources, make it a good source of information to feed our research effort. Moreover, the apps available in the dataset are already scanned on VT (Virus Total), providing researchers with an initial set of information to spearhead their works.

Consequently, we rely our study on information collected from AndroZoo. More specifically, we combined the list of hashes with VT API's results to identify all the apps that 1) had not been submitted to VT before April 2019 and 2) have been assigned a label related to mining by at least two antivirus solutions. Our approach prevented an intersection of our dataset with that of Dashevskyi et al. [68] and enabled us to discard most of the non-relevant apps. The resultant initial dataset that we built with this strategy contains 346 SHA-256 corresponding to one or more versions of 92 different apps. In this work, an app refers to a unique pair of a package name and the marketplace of origin. For each app, we only consider the latest version in our data set for the study.

4.2.2 Manual analysis of apps

Through our manual, systematic and rigorous manual analysis of the apps in our dataset, we want to answer the following questions to establish their connection with the mining activity.

1. **Does the app mine cryptocurrencies?** To answer the question, we look for pieces of evidence that validate that the app embeds and uses some mining procedures. Based on the findings reported in previous research, we focus our search on the `app/src/main/assets` and `app/src/main/res/raw` folders where the developers usually put their mining artifacts [68]. Upon localizing the said file, we analyze the source code to confirm that the program exploits the identified mining procedure. This step allows us to identify some false positives, some miners, and a few apps for which we could not confirm beyond the point of doubt that the app is mining.
2. **How is the mining implemented into the app?** At this stage, we try to explain the specific technique the developers leveraged to insert the mining logic into the app. Besides distinguishing between web and binary-based miners, we look at the programming languages and other relevant elements that made the implementation possible.
3. **Is the device's user aware of the mining activity?** At this stage, we look at various information, such as the listing of the marketplace app when available, the source code, and layouts, to verify whether the user is aware that mining activity is occurring on their device.
4. **Who benefits from the mining activity?** This is to check who is the final beneficiary of the mining. The beneficiary is either the user or the developer. It could also be both when a donation wallet is specified. This question is answered

by looking for wallet addresses, configuration files, and similar elements that allow making an informed decision.

5. **What triggers the mining activity?** This question is to explain under which conditions the mining activity is started. The trigger could vary from a system event to an action from the user.
6. **How is the mining activity concealed?** This step is to identify strategies that developers use to evade static and dynamic detection of the mining activity.

Considering the high reliance of our approach on having access to the apps' source code, we downloaded the APKs using the AndroZoo API and further decompiled them using the JADX tool[1], one of the many available tool to extract the Java source code and the static resources bundled into apps. We also leveraged the detailed VT reports to clear up doubts when faced with uncertainty.

4.2.3 Understanding how developers distribute the mining apps

Our objective in this regard is to understand the various processes that the developers put in place to attract users and get them to install the mining apps. More specifically, we are interested in answering the following questions:

1. **In what type of apps do developer often insert their mining code?** Previous research has highlighted the use of piggybacked and repackaged apps to distribute Android malware in the wild [87, 88]. Based on those insights, we try to evaluate the prevalence of a particular type of app in our corpus of mining apps. We leveraged the Androguard tool[2] to retrieve each app's package name and label. We searched for those names on Google to find similar words to identify links with apps from other marketplaces.
2. **Are ratings and comments of the apps manipulated to increase adoption by the users?** This question stems from the findings of Harris et al. suggesting a strong link between an app's popularity and its attractiveness to users [89]. To assess that, we analyzed the ratings and comments of the mining apps in our dataset that present specific characteristics and suggest some manipulation. We leveraged the Google Play Scraper [3] tool's review API to collect the said information for the apps still available in the store while doing our work. We used the default settings of the package (i.e., country set to the USA and the language to English)

We present the results of those investigations in Section 4.4.

4.2.4 Evaluation of the impact on users

Thus far, the literature only assumed that mining applications are greedy in terms of resources. Even though there have been some estimates that clearly showed that those apps consume an enormous amount of energy[4], those reports are mainly related to the period when Coinhive still existed and was used with the intent of abusing resources. Still, the research community lacks a systematic measure of those values in the context of Android apps. In this work, we benchmark the resource consumption of 3 mining apps against some commonly used legitimate apps. The

[1] `https://github.com/skylot/jadx`
[2] https://github.com/androguard/androguard
[3] https://pypi.org/project/google-play-scraper/
[4] `https://www.kaspersky.com/blog/loapi-trojan/20510/`

results presented in Section 4.5 suggest that mining applications may not be as greedy as is often assumed.

In the subsequent sections, we describe the findings of our investigations for answering the outlined research questions.

4.3 RQ1: Understanding the implementation of the mining logic into Android apps

Our manual analysis of the potential mining apps consisted in looking into the external dependencies or Java packages used in the project, the `assets` directory that often contains HTML resources used by web-based miners on Android, the `res/raw` folder that is often used for binary files, and the `lib` directory where shared libraries are put to be loaded with the `System.loadLibrary()` method. Our analysis of those artifacts and how they are used allowed us to identify many apps that could be wrongly believed to perform mining and some other mining apps about which we provided more explanation on how they perform.

4.3.1 False positives and possibly unconfirmed miners

From our investigations, there were a number of apps for which our manual analysis did not allow us to conclude that they were mining. We report on those apps in the following sections.

4.3.1.1 The apps wrongly identified as miners

Upon analyzing the suspected mining apps from our data set, we identified that most were mining apps. Those false positives amount to 76% (70/92) of the entire data set. In this entire corpus, three main types of apps can be observed:

4.3.1.1.1 The *com.kaching.kingforaday* based packages Those represent half of the entire false positives in our data set (35/70) and were all downloaded from VirusShare into AndroZoo. The sparkling similarity in those apps starts with their naming convention which is done by appending the string .hack to the name of some apps with paid subscriptions (e.g., com.imangi.templerun.hack). Furthermore, the analysis of the source code shows the presence of a package named `com.kaching.kingofaday` which declares a service as seen in Listing 4.1 which is in charge of starting the mining activity. For this, it depends on a package named `com.coinhiveminer.CoinHive` which uses an HTML file that instantiates the Coinhive miner.

```
1 public class StartAdsReceiver extends BroadcastReceiver {
2     @Override // android.content.BroadcastReceiver
3     public void onReceive(Context context, Intent intent) {
4         context.startService(new KingForADay(context).MinerIntent());
5     }
6 }
```

Listing 4.1: Mining receiver code embedded into the com.kaching.kingforaday package.

However, though this logic is well implemented, it appears to not be instantiated anywhere in the code. We further ran a full string search in the folder to verify whether it was called anywhere but we could not get evidence for this at all. Moreover, the fact that the service was not declared in the Manifest means that it could not be

called anyway. Thus, we hypothesize that the app had been an active miner during the rise of Coinhive but then, since the service was later shutdown, the developers just removed the initialization code from it. Our attempt to validate that hypothesis by finding earlier versions of any of the concerned app was not successful unfortunately.

4.3.1.1.2 The apps with the unused mining files Very similar to the previous apps, the apps in this category embed either directly or through a third-party service, the mining code as a Javascript dependency or through the use of a binary. Among the 10 such apps, half of them have the mining code in them because of the inclusion of the Hextrix game[5] which is a game that clearly states its intention of mining. Despite that, the mining code was commented out in each of the relevant files. This is similar to the other half of the apps which used various mining services including Coinhive and CryptoLoot. In Listing 4.2, we show how the mining code was commented out in those apps. Here again, a full string search was performed in the directory to find and analyze all the code loading the webview, searching for instances where the comments might have been removed prior to loading the files but no activity of such kind was noticed.

```
1 <!-- <script src="<LINK TO SCRIPT>"></script> -->
2 <!-- <script> -->
3 <!-- var miner = new CoinHive.Anonymous(<SITE_KEY>, { -->
4         <!--         throttle:0.7 -->
5         <!--     }); -->
6         <!--     miner.start(); -->
7 <!-- </script> -->
```

Listing 4.2: Initialization code of Coinhive mining that is commented out in application

4.3.1.1.3 The wallets Our data set is comprised of 17 crypto wallets that were considered as miners by at least two antivirus solutions. Our analysis of the binaries embedded in those apps shows that they indeed contain the mining logic, as evidenced by the parameters to set the type of algorithm, donation address, etc. Moreover, some of those apps download entire copies of the blockchain in order to track the transactions related to the users' wallet. One notable case among the wallets however is that of the app `io.scalaproject.vault` still available on the Play Store, which presents a mining interface through which the user is redirected to download the actual mining app from a Github repository[6]. By analyzing the source code and various parameters used to call the suspicious binary, we confirmed that the apps are not exploiting the mining capabilities of the software.

4.3.1.1.4 The outliers 8 apps fall into this category which from our investigation, have no clear link to mining activity other than the appearance of strings such as *miner* or *mining* in the assets. One special case in this category has a list of links that it uses to serve as a firewall, preventing the user from accessing those links. And among those links, some look like mining pools address (e.g. 0.0.0.0rpd.cryptopool.eu) in a subfolder of the corresponding assets folder. Some of the other apps have no suspicious binary or JavaScript file that leads to thinking about a mining activity.

The high proportion of false positives and that count up to 25 detection by antivirus solutions can be explained by the fact that most of those tools are signature

[5] `https://hextris.github.io`

[6] `https://github.com/scala-network/MobileMiner/releases/`

based and do not actually check whether the code is being run or not. This in itself is not a good practice but we suggest a better attempt at finding whether the code is being run or not.

4.3.1.2 The ambiguous cases

Besides the apps that we were able to confirm were not doing any mining activity, for 7 of the remaining apps, we could not ascertain the presence of the mining code. The information collected about those apps is summarized below:

1. 2 apps presented as movie downloader and viewer based on the use of torrents. In one of those apps, there is the presence of the xmrig binary that can be used for mining. Surprisingly, the binary is not called anywhere in the code as confirmed by our search in all the relevant files. However, we noticed that those two apps at a point during the bootstrap try to download and install an app from the link[7]. We presume that the actual miner is installed from that link and further exploits the binaries already embedded in the current package. Unfortunately, at the time of running our analysis, the link was no longer available making it impossible for us to validate our hypothesis. A third app with a different purpose redirects the user to dynamically queried links which we could not verify because the link were no longer accessible.
2. 3 other apps, all retrieved from the *anzhi* marketplace include the `libjiagu` binary which serves as a packer meant to dissimulate source code and prevent their investigation by manual analysis. Our attempt at unpacking the binaries were unsuccessful so we could not justify whether those apps are mining or not.
3. A further duo of apps from the same developer which are presented as wallets and expose a great similarity in the source code. The most interesting bit of those apps is the fact that they hide a compressed archive under a file name `private.mp3` and placed under the `assets` folder. The file which is uncompressed at runtime contains Python files that allow to run a node of the Electrum coin. Adding to this deceptive and misleading approach to installing the binaries on the device, we identified from the source code the use of a `config.json` file that determines the mode in which the node is ran. Unfortunately, we could not find any information about the said file which hampered our ability to identify whether the mining activity was occurring or not.

As a summary, our investigation have pointed out a number of apps (74/87) for which we could not find any evidence of the execution of a mining activity. Thus, for the remaining of this chapter, only the remaining 13 unique apps amounting to 27 different APKs will be put in focus for us to try to understand how the miners work and explain the processes that are engaged into their creation and mass distribution to users.

4.3.2 Inspection of the mining apps

As a logical conclusion to our investigation, the 13 remaining apps are considered and verified as mining one or multiple crypto currencies. Among those apps, there is a preponderance of apps originating from Google Play Store with such apps amounting to 11 out of the 13 apps, the remainder coming from VirusShare. This impressive ratio of mining apps downloaded from Google Play Store in our time-frame betrays our initial intuition to have more miners from alternative markets. Our surprise

[7] http://b3.ge.tt/gett/9JtX8Kp2/com.opera.mini.native.pdf?index=0&pdf

is further reinforced by the fact that 6 of such apps were still present on the store at the time of writing. We postulate that some developers manage to bypass the controls made to prevent those apps from being flagged during the verification process. Understanding the process through which they successfully get those apps on the store are out of scope of this work. Since the store's regulations ban mining apps, all the 13 apps in our data are considered *illicit*. Nonetheless, we will distinguish between the apps according to whether the mining activity is known to the user or not.

Above any consideration to the awareness of the user on the mining activity, a general view to the mining apps shows a consistency with the results from previous works that identified the web and binary-based techniques as the means of inserting mining code in Android apps [68]. In our case only a small proportion of the apps are web-based (4/13). This rapid change when compared to the period before April 2019 where web-based miners amounted to almost 75% of all apps can be explained by the shutdown of the Coinhive service which was the main driver of the mobile mining ecosystem on Android. As far as the alternatives to Coinhive are concerned, we have identified the use of CryptoLoot and other proprietary scripts[8].

Another point of variation in the implementations of the mining apps lie in the use of various programming languages to support the code. As such, we have identified apps using JavaScript-based hybrid mobile developments toolkit such as Ionic, but also those developed using Kotlin and obviously the Java language which powers most of those. This evaluation was performed by looking at the source code of the assets for instance (presence of the <ion-app> tags in Ionic) and the included packages which can inform on the tool used to build the app. This diversity in techniques and procedures suggests that the development of mining apps for Android devices is possible for virtually every programming language supported on Android.

Our analysis of the scan reports collected from VT has shown that the number of tools that have detected a certain miner varies from 2 to 25 suggesting that the threshold of ten used in related works is not appropriate [90]. Our findings suggest that for categorical malware such as mining apps, a threshold of 2 seems sufficient.

In order to discriminate the between legitimate and illicit miners, we used information such as the package name, description of the app where available (collected from the store), the layouts in the decompiled file and also the source code. This has allowed to identify 6 legitimate miners among which one specifically sets all the rewards to the developers while the others get only a portion of the gains, with the other part being left to the user's address. The former's screen through which the user's approval is granted is visible in Fig. 4.1 and declines its intention to monetize their website visits by leveraging the users' CPU for mining crypto currencies.

The next section describes the underlying functioning and evasion techniques pertaining to the illicit miners in our data set.

4.3.3 Functioning and evasion techniques of the crypto jacking apps

The balance in the type of implementation used in the list of crypto jacking apps (3 web vs 4 binary based), the mined coins vary in nature with Monero, uPlexa and RPCoin all being represented in our data set. This describes a diversity in the available tools to embed illicit mining activity into apps. We

[8] `https://www.craftyourserv.net/mineur`

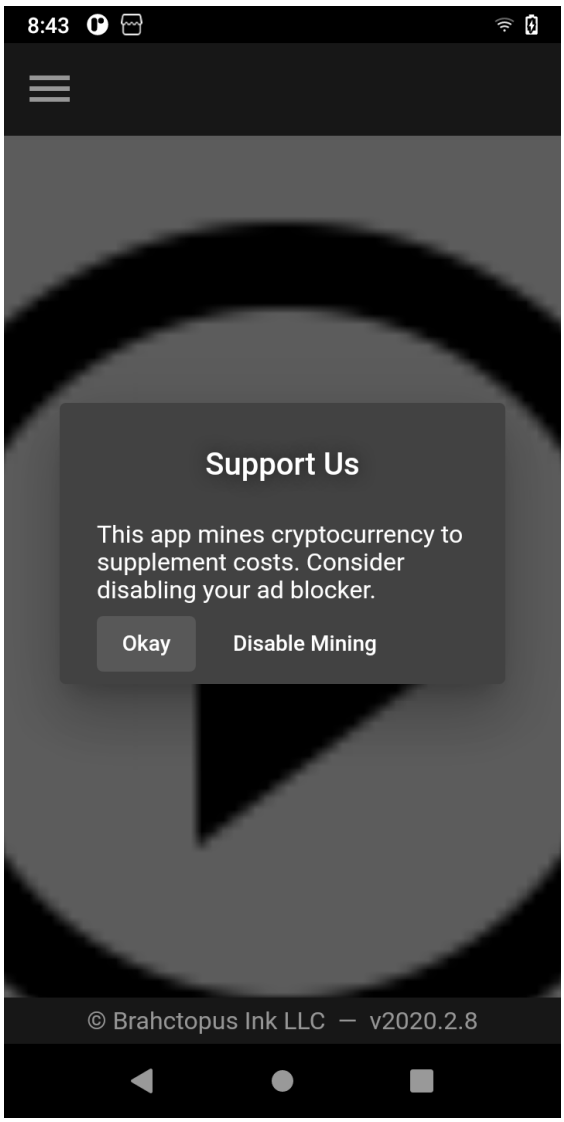


Figure 4.1: Screenshot of the *com.sunderapps.brahtopus* app's interface through which the user grants permission to mine cryptocurrency.

Table 4.1: Top 10 permissions requested by the cryptojacking apps.

Permission	**# Apps (%)**
android.permission.INTERNET	7 (100.0%)
android.permission.ACCESS_NETWORK_STATE	5 (71.43%)
android.permission.WRITE_EXTERNAL_STORAGE	5 (71.43%)
android.permission.READ_EXTERNAL_STORAGE	4 (57.14%)
android.permission.READ_PHONE_STATE	4 (57.14%)
android.permission.VIBRATE	4 (57.14%)
android.permission.WAKE_LOCK	4 (57.14%)
android.permission.ACCESS_COARSE_LOCATION	3 (42.86%)
android.permission.ACCESS_FINE_LOCATION	3 (42.86%)
android.permission.ACCESS_WIFI_STATE	3 (42.86%)

further analyzed the permissions requested by the mining apps which, as presented in Table 4.1, are quite similar to those identified by Dashevskyi et al [68]. When compared with legitimate apps, three permissions appear to be peculiar to the mining apps namely those related to the device's location (ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION), and the one related to the phone state (READ_PHONE_STATE) [91]. Those three permissions have been identified as being used in up to 68% Android apps according to a study of dangerous Android permissions [92]. As far as the system events are concerned, the mining apps in our dataset subscribe to a few events that do not appear in the list of events identified in Cao et al.'s study of Android malware [82]. Those events are related to the charging status of the device and to the presence or not of the user by the telephone. They are used in mining apps to find the best moment to mine without raising any suspicions from the user.

In the later sections of the chapter, we will leverage those information as a means to talk about the triggers for the mining activity, the various sophistication put in place and also the evasion techniques that we identified in the various apps. The apps can therefore be declined into the following categories:

4.3.3.1 The watchers

Those apps amounting to 3 exploit the possibility of running JavaScript on the web view interfaces of the apps. In a general sense, the script is executed immediately, as soon as the user navigates into the page which is loaded either through a web view or using a language such as Ionic. Then, the mining keeps running as long as the activity is alive. We were able to confirm the malicious intent of the apps through the presence of developer (or site keys) that are used internally by the mining service providers to distribute the gains back to the developers. In order to avoid detection, we have observed two main techniques: 1) the obfuscation of the JavaScript file which is also loaded from random links, and 2) the inclusion of the mining initialization code into larger files combined with the use of variables to dissimulate the mining activity. An example of initialization code can be seen in Listing 4.3 two different site keys are used in the same app based on the domain name from which the user is visiting.

```
function D(){
    var e="<SITE KEY 1>"
```

```
    if(-1!=location.href.indexOf("<LINK>")||-1!=location.href.indexOf
        ("<LINK>"))
         return void(e="<SITE KEY 2>");
    var n=new CoinHive.Anonymous(e);
    n.setThrottle(.5),n.start()
}
```

Listing 4.3: Initialization code of a web based miner

The use of those techniques ensures that a static detection tool which is for instance based on the length of the site key would be fooled and not able to detect the app as doing mining.

4.3.3.2 The zombie miners

Those ones are a group of 2 apps seemingly from the same developer that use the Monero miner file in order to launch the mining activity. After loading the system libraries, the apps connect to a mining pool and instantiate a service that keeps restarting each time there is a failure. This allows the mining process to continue running even if the user is not interacting with the app thus making up for a zombie like functioning. In order to not letting the user being aware of such activity, the apps listen to events related to the battery and stop the mining activity as soon as a threshold is reached. The apps download the configuration files for the mining from a link[9] which was no longer available during our analysis. We were also unable to run the app which kept crashing on the test device.

4.3.3.3 The fetchers

We denote by this, a set of apps that only download the mining binary at run time, or receive the task through another file. For the two apps in this category, the mining activity is started automatically when the device is powered on. Besides this first layer of obfuscation, we have observed in one of those apps that the SharedPreferences were used to store the commands and parameters to be executed to run the mining. This is then later used to launch the mining. And since the two apps in those categories offer services that are not supposed to consume too much memory, they make sure that the mining activity is running only when the user is away by subscribing to the *USER_PRESENT* system's event. This complex scheme has further been complicated by one of the apps that inserted the mining code under a Google's package name (com.google.ads) in an attempt to fool detection tools.

4.4 RQ2: How are users attracted to downloading and installing those applications?

In the Android ecosystem of mining apps, the development of the apps is just one part of the equation. Upon producing the software artifacts, the developer needs to find the appropriate ways in which they can distribute those apps to as many users as possible. In the current section, we look at two factors that we presume to positively impact the adoption of mining apps by Android users. We refer specifically to (1) the kind of app used and (2) its popularity. We identify the kind of app by grouping the apps based on the features that they claim to propose. As far as popularity is concerned, we investigate the manipulation of reviews to increase an app's attractiveness.

[9] http://flowcount.eidon.top:10085

Table 4.2: Top 10 system events subscribed to by the cryptojacking apps.

Event	# Apps (%)
BOOT_COMPLETED	3 (42.86%)
MEDIA_MOUNTED	3 (42.86%)
USER_PRESENT	3 (42.86%)
net.conn.CONNECTIVITY_CHANGE	3 (42.86%)
WEB_SEARCH	2 (28.57%)
ACTION_POWER_CONNECTED	2 (28.57%)
ACTION_POWER_DISCONNECTED	2 (28.57%)
com.android.vending.INSTALL_REFERRER	2 (28.57%)
TIME_SET	1 (14.29%)
service.notification.NotificationListenerService	1 (14.29%)

Table 4.3: Categories of the mining apps and their source.

Package name	Category	Source
com.pangzlab.verus_box	Blockchain related apps	play.google.com
de.ludddetis.monerominer		play.google.com
dev.waterhole		play.google.com
free.bitcoin.mining.crypto		play.google.com
io.waterhole		play.google.com
info.bitcoinunlimited.voting		play.google.com
com.karameesh.app	Game and entertainment apps	play.google.com
com.sunderapps.brahctopus		play.google.com
com.games.tecdroid.freddynightmario		play.google.com
com.sunderapps.recaptureorganics		play.google.com
net.craftyourserv.www		play.google.com
mikado.bizcalpro	The cracks	VirusShare
com.dust.clear.ola	Utility apps	VirusShare

4.4.1 Type of apps containing mining logic

Our analysis allowed us to group the mining apps in our dataset into the following categories that are summarized in Table 4.3.

4.4.1.1 Blockchain-related apps

The most represented category in our dataset with six unique apps, this category contains three apps that state their intent of mining, two wallet apps, and one other app that leverages the blockchain to facilitate the organization of polls (info.bitcoinunlimited.voting). In their official listing on the marketplace, all those apps appear to be associated with the *Finance* category. This denotes the attractiveness of such apps to users who can fall prey to malicious developers who abuse their devices for mining as for two apps in this corpus (i.e., *dev.waterhole* and *io.waterhole*), published on Google Play Store by the same developer.

4.4.1.2 Game and entertainment apps

Five apps in our dataset fall under this category, including games, music, and advice content. Among all those apps, only two (*com.sunderapps.brahctopus*, *net.craftyourserv.www*) inform the user of the mining intent on the first usage of the app, as seen in Fig. 4.1 and Fig. 4.2a respectively. The relatively high proportion of those apps (5/13) denotes the interest of developers in maintaining users in the app for a long time to generate the most profit, as previously demonstrated by Tekiner et al. [93]. Furthermore, all the miners in this category use the web-based strategy that works better when the user is present.

4.4.1.3 The cracks

In an attempt to generate revenues from their production, developers often require users to pay before downloading or enabling the premium features of an app. Often unable to use the apps because of those limitations, users tend to resort to cracked apps help them bypass the verification mechanisms put in place by the developers. Despite posing as free software, the cracked apps have been proven to often contain malware [88]. Our sample of mining apps contains one cracked app carrying illicit mining codes. One of those apps posed as a free version to a paid calendar app available in the Google Play Store[10]. As soon as the user installs the app, it proceeds with its stealthy mining activity.

We can further extend our list of cracks with the false positives that we discussed in Section 4.3.1.1.1. The naming convention of those apps suggests that they all pretend to help users bypass the subscription controls of dozens of games and entertainment apps. We can therefore presume that they were indeed cracks shared through alternative Android marketplaces with the intent of distributing illicit mining codes.

4.4.1.4 Utility apps

Only one app (i.e. *com.dust.clear.ola*) falls into this category. From the search results available in Google and translated from Chinese to English[11], the app promises the user to clean their device's memory but also remove water droplets. The user willingly grants it many abusive permissions used by the app to conceal its mining activity. We expect that many similar apps carry mining code or related malware.

4.4.1.5 Interpretation

Despite the presence of two dominant types of apps that mining apps can be associated with the high diversity in such a small sample shows that mining apps can be found in any category of app. The small size of our dataset does not allow us to determine any specific category that developers prefer to distribute mining code.

4.4.2 Manipulation of comments and ratings to attract users

Our review of the ratings and comments associated with the apps still available on the Google Play Store during our study was based on the Google's policy for user reviews[12] that defines a helpful review as one that provides clear information to the reader, and present both the strong and weak points of their experience with the

[10] `https://play.google.com/store/apps/details?id=mikado.bizcalpro`
[11] https://app.mi.com/details?id=com.dust.clear.ola
[12] https://play.google.com/intl/en_us/about/comment-posting-policy/

Table 4.4: Summary of review information collected on Google Play Store about the mining apps.

Package Name	Average rating	Min. Num. of Installs	# of reviews
net.craftyourserv.www	3.7	1000	32
com.pangzlab.verus_box	4.3	5000	14
io.waterhole	-	-	13
de.ludddetis.monerominer	-	-	3
dev.waterhole	4.2	100	2
info.bitcoinunlimited.voting	5.0	10	1
com.games.tecdroid.freddynightmario	-	-	1
com.karameesh.app	-	-	1

app. As such, we focus our analysis on the various reviews of the apps looking for elements that pinpoint to a manipulation.

In total, we collected 67 reviews in English and French for eight different apps presented in Table 4.4. We performed a manual analysis of the metadata with two authors, both fluent in English and French actively involved in the process. The results were presented to other researchers in our group who agreed with our conclusions.

From our analysis, we made the following observations.

4.4.2.1 Useless and uninformative comments

Around five apps in our dataset have at most three reviews from users. In those specific cases, we hardly noticed any informative comments, with some reviews being as simple as "I like this style and feel safe". Moreover, the comments in this style are all submitted by users who gave five stars to the app. This goes against the fact that a user satisfied with an app that provides clear benefits would find many things to say to motivate others to download the app.

4.4.2.2 Untrustworthy comments

This specific case applies to the app *io.waterhole* for which a five-star rating was attributed after being available for less than five days. Even though we could not prove this practically, we believe that five days remains a tiny amount of time to provide objective feedback on an app without any prior bias. Moreover, the same comment only talks about the user feeling safe with the app, which does not have any link with the app's features.

4.4.2.3 Interpretation

Even though we could not prove our hypothesis beyond the point of doubt, we have observed in our case study of eight mining apps available on Google Play Store between April 2019 and May 2022 that the comments tend to be uninformative and intentionally submitted with high ratings to increase the popularity of the apps. We believe that this has contributed to the popularity of some of those apps.

4.5 RQ3: How noticeable is the execution of a cryptojacking app to the device's user

Due to the extensive computations needed to solve the proof-of-work algorithms, mining apps are expected to deplete devices' battery and resources. In this section, we quantify the values for various mining apps to estimate their impact on users. More specifically, we are interested in those apps' battery, CPU, and RAM consumption. .

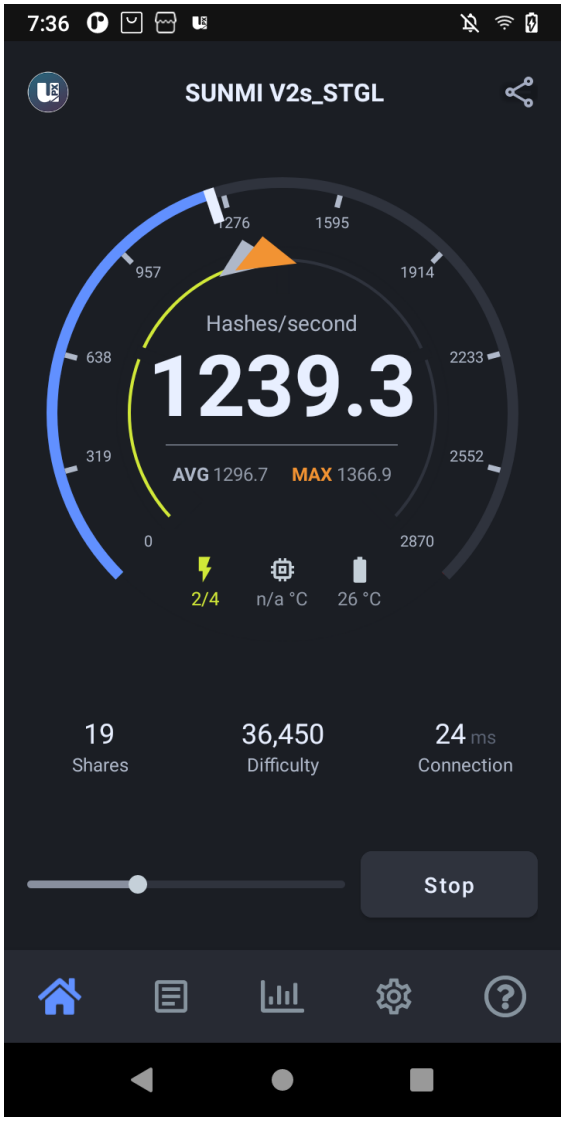
To provide such information, we decided to run two sets of experiments on one Android device powered with Android 11 with a 1.8 GB RAM powered with a Mediatek MT8766B[13] CPU running with four cores and sustained with a Lithium Polymer Battery 7.7V/3500mAh. On those devices, we installed and gathered metrics about various Android apps' battery, CPU, and RAM usage. We more specifically attempt to quantify those values for three main categories of apps: 1) benign apps that perform background tasks, 2) legit mining apps that users install in complete awareness, and 3) the apps that mine on the back of the users. Despite our initial intention of profiling licit and illicit miners from the web and binary-based families in our study, all the illegal web-based miners in our dataset were no longer functioning. Similarly, we could not successfully install any binary-based miners on our devices. Fortunately for that one, we were able to find an open-source mining app for Android that we used for those tests. Below is a list of the apps that we used for our analysis.

1. *com.facebook.katana* (139e0831...42ad3ba5), the official app for the Facebook social media users on Android was used as the benign app. We made this choice because of 1) the popularity of the app which was downloaded at least five billion users across the world [94]; and 2) the fact that it is pre-installed on many Android devices with sometimes limited possibility of users to uninstall them [95, 96]. Those facts suggest that this is one of the most common apps that Android devices owners ran on a daily basis. The ability of the app to execute background tasks in order to keep users atop of their community news renders day-to-day usage possible outside of the main screens of the app. Consequently, we measured the values for this app while it was executing in the background.
2. *net.craftyourserv.www* (5464fa82...b531c76e) as the licit web-based miner. This app provides an interface (Fig. 4.2a) where users can simultaneously visualize ads and mine the RUBY cryptocurrency. Therefore, we measured the average impact of the advertisement feature, which we subtracted from the values collected while the mining process was on. The resultant values are considered the impact of mining.
3. *mikado.bizcalpro* (b6001aa8...ece1ed41) as the binary-based cryptojacking app for which we collected the values while the app was executing in the background. This app being the only illicit miner that we studied, we consider its performance representative of illicit mining apps.
4. *com.uplexa.androidminer* (2b4cee1d8...1f86f43c)[14], the fully-fledged binary-based miner. After launching the mining process, the interface presents the user with stats about the mining activity, as seen in Fig. 4.2b. It then allows the user to leave the app while the mining continues in the background. We collected the various metrics during a period when the app was running in the background.

[13] `https://www.mediatek.com/products/tablets/mt8766b`

[14] `https://github.com/uPlexa/upx-android-miner/releases/download/v0.4.1/upx-android-miner-v4.1.apk`

(a) Screenshot of the *net.craftyourserv.www* app with the red area serving the ads and the green being used for mining.



(b) Screenshot of the *com.uplexa.androidminer* app showing the ongoing mining process.

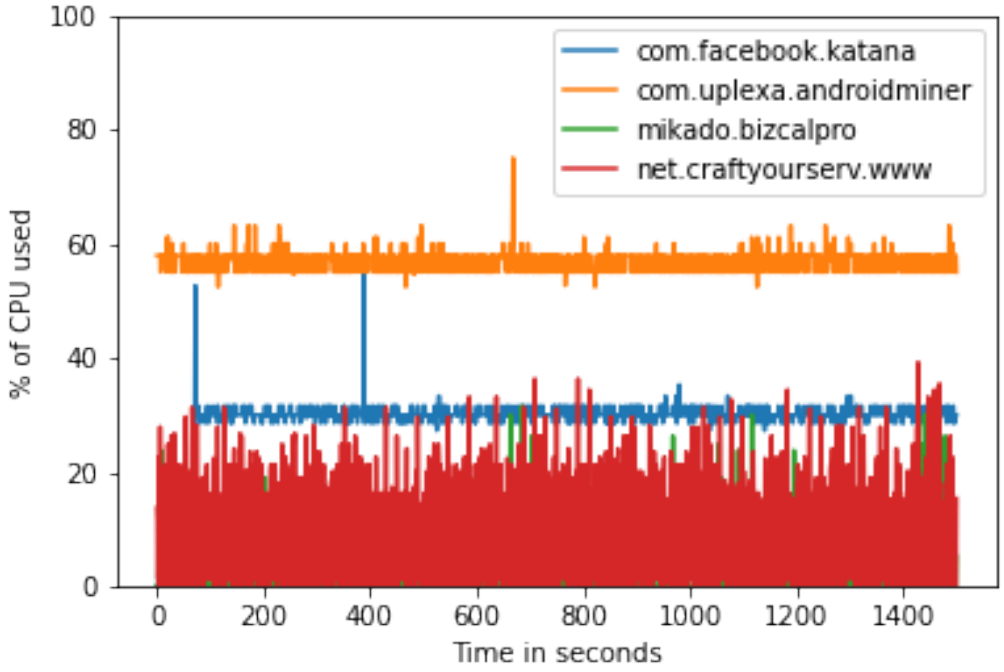


Figure 4.3: Comparison of the CPU usage of three mining apps compared to the Facebook app.

For each of the four applications, we further ran two sets of experiments described in the following sections.

4.5.1 Estimation of the CPU and RAM usage

For an uninterrupted duration of 1500 seconds (25 minutes), we ran each of the apps and collected data about the CPU and RAM usage from the Snapdragon Profiler tool[15] .The tool leverages the Android Debug Bridge (ADB) interface to provide real-time measurement of various metrics for an Android app. After collecting and processing the collected values for all the apps, it appears that only the *com.uplexa.androidminer* app uses more than twice the amount of CPU used by any other app. The Facebook app comes right behind and consumes slightly more than any other miner in our test set. Moreover, those results summarized in Fig. 4.3 show that the illicit miner does consume the least amount of CPU.

We observed the same pattern in the case of RAM usage. Here, the *com.uplexa.androidminer* app alone uses up to 75% of the available memory, which is almost 300x the closest value observed on Facebook (0.26%). Due to that significant disparity, we decided not to report this app's consumption in Fig. 4.4. Nevertheless, the graph shows that Facebook consumes at least 2.5x more RAM than all the other mining apps, with the web-based mining app being the less greedy. On the other side, the illicit miner does seem to use quite a constant amount of RAM throughout the experiment. Because it uses a proof-of-work algorithm that does not require too much information, one can assume that the CPU is constantly needed leading to the observed irregular pattern.

[15]https://developer.qualcomm.com/software/snapdragon-profiler

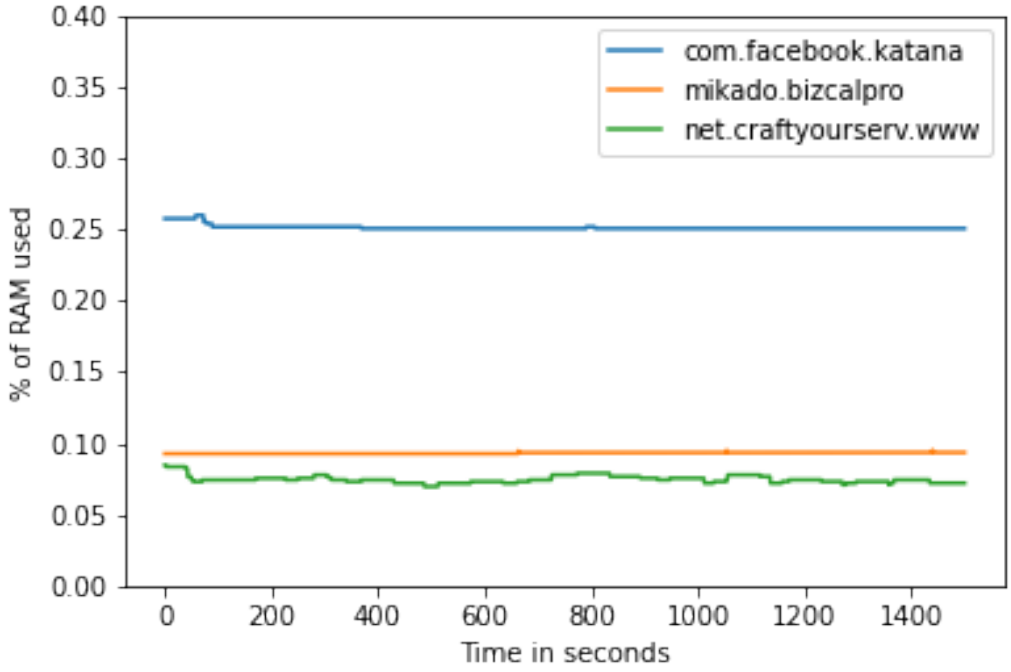


Figure 4.4: Comparison of the RAM usage of two mining apps and the Facebook app.

4.5.2 Estimation of the battery usage

For the battery usage of the apps, we leverage the features of ADB to generate and download bug reports and battery statistics summaries. Then, the files were analyzed using the Battery Historian tool[16]. The values presented in Table 4.5 are collected based on five series of five minutes execution of each app. We have reset the battery stats between each run to prevent data overlapping.

Table 4.5: Battery consumption of the various apps in a five-minute time frame (in percentage).

Package	Mean	Est. discharge time
com.uplexa.androidminer	0.196 ± 0.028	1 day, 10 hours
net.craftyourserv.www	0.096 ± 0.009	3 days, 14 hours
com.facebook.katana	0.060 ± 0.007	5 days, 18 hours
mikado.bizcalpro	0.032 ± 0.004	10 days, 20 hours

As evidenced in the results, the cryptojacking app surprisingly consumes less battery than all the other apps, including Facebook. From our estimate, it would take roughly ten days of execution for that app to deplete the battery. On the other hand, we also notice that legit mining apps use more battery than any other app. We explain this significant difference in resource consumption because illicit miners attempt to go stealthy by purposefully limiting their usage of the available resources.

[16] `https://github.com/google/battery-historian`

Table 4.6: Summary of the detection performed by some free antivirus solutions after scanning the device with the mining apps installed.

Antivirus	**# of malwares**	**# of miners**
com.sophos.smsec	5	0
com.bitdefender.antivirus	3	1
com.drweb	1	1

4.5.3 Interpretation

The results obtained from the profiling of the sample apps lead to the conclusion that while mining apps can deplete users' batteries daily, illicit miners seem to be diligent with their consumption. Moreover, compared with some popular apps such as Facebook, the impact of illegal miners seems negligible. Such a fact suggests that the average user is likely not to observe severe deviance from normal behavior when an app covertly mines on their device. Therefore, we can assume that many users might be running mining apps in total ignorance.

Even though the user could not notice the malware execution on their own, it is customary to assume that they could potentially install an antivirus that would let them know that a mining app is running on their device. Therefore, we decided to execute a set of free antivirus solutions alongside the mining apps to ascertain this. We justify our decision to only test free antiviruses by the fact that there are more accessible to users than the paid alternatives. We executed our tests using three antiviruses identified on Google Play Store after we searched for "antivirus." We summarized the results of those tests in Table 4.6.

From the corresponding results, it appears that only one app was simultaneously detected by all the antivirus at once (mikado.bizcalpro), with one not associating the actual label to it to let the user know that the app is mining. At most, five mining apps were detecting, accounting for less than half of the ground truth. The probability that an antivirus reports a mining app to the user remains critically low.

4.6 Limitations and Discussion

Given our reliance on AndroZoo as our only data collection point, it is evident that we could not provide an accurate measurement of the mining ecosystem. The first reason is that AndroZoo does not collect any apps besides those running on smartphones leaving out many other less capable devices such as wearables, TVs, and smart appliances. The incredible three-fold growth in the population of those devices in 2022 [67] makes us believe that it is an attractive target to malicious developers as much as the other IoT devices [97]. The exclusion of those devices means we are missing out on some possibly intriguing trends related to them. Similarly, the mere possibility of some apps not being included in AndroZoo for one reason or another could have impacted the scope of our data. However, the lack of any mining family in Cao et al.'s dataset collected using security advisories from respected security companies [82] suggest that a search on security blogs was unlikely to extend our dataset any further. Consequently, we argue that our results provide the best possible understanding of the mining phenomenon on Android.

The same applies to the use of VirusTotal on which we relied to identify the set

of apps to manually analyze. Despite the presence of several tools in VirusTotal, it is obvious that the tool is prone to errors which would lead to false and missed detection. Even though our manual analysis nullifies the likelihood of false detection in our final dataset, we are however likely to miss an important number of apps due to our strategy. Unfortunately, we are not able to provide an estimate to the number of miners that we potentially missed through this exercise. Nonetheless, VirusTotal remains the best existing tool for crowd-sourced malware analysis and has been used in research that is very similar to ours [82, 68].

Another potential threat to the validity of our work lies in the manual analysis performed on the malware samples. Though there might be some errors, we adopted the same technique as Dashevskyi et al. [68] to identify the mining logic. Moreover, we searched every project file in case of doubt regarding the mining activity. Our systematic approach to the problem has allowed us to identify some mining apps and present those for which we could not ascertain the presence of the mining code due to a lack of evidence. The absence of a programmable logic that can be used to reproduce our research constitutes on its own, a valid point on which scientist can doubt our results. However, proposing such a technique would have led to building a new detection mechanism which is out of scope of this study. Furthermore, we used a public dataset which can be exploited to verify the veracity of our conclusions. We believe that our efforts could help researchers in building effective detection mechanisms against mining apps on Android.

4.7 Summary

Our study of 346 Android apps suspected of mining cryptocurrencies on the devices provides a first-of-its-kind review of the advances in the cryptojacking phenomenon on Android since April 2019. Though our study confirms the results from studies covering an earlier period, it sheds light on a few problems: (1) the high-rate of false positives detection from many antivirus solutions, (2) the popularity of hacks and cracks as means of carrying the illicit mining logic, (3) the relatively low-resource consumption of cryptojacking apps on Android, and (4) the availability of mining apps in the Google Play Store. Those findings reveal a misfit prevention and fight system that leads to hundreds of thousands of users downloading mining apps that the developers successfully uploaded to the Google Play Store a breach of the developers' policy in place. The upfront notice from some developers of their intent to mine on the device leads us to question the verification process put in place by Google to prevent such apps from being present. The mere fact that more than 70% of users put great trust in the store [98] explains the high number of installs for each of those apps. We believe that Google should put in place better controls to increase the safety of its platform from malicious apps, including mining ones.

We resort to the research community to investigate the problem of how developers get through the verification process and further lure users into downloading malicious and unwanted apps on their devices. It would also be interesting to understand what happens to users who had previously installed those apps after Google removes them from the store. The underlying results would help make the process more reliable and would significantly increase the confidence and security of Android users against various sorts of abuse.

5

Despite Immutability Dimension: Retracing Smart Contract Versions for Security Analyses

Due to the inherent immutability of blockchain technology, smart contract updates require their deployment at new addresses rather than modifying existing ones, thus fragmenting version histories and creating critical blind spots for analyses. Indeed, for example, this fragmentation severely hinders security researchers' ability to track vulnerability lifecycles across contract versions. While platforms like Etherscan provide detailed information about Ethereum smart contracts, they lack crucial functionality to trace predecessor-successor relationships within smart contract lineages, preventing systematic analysis of how vulnerabilities emerge, propagate, and potentially remain unresolved across versions. To address the challenge of tracing smart contract lineages, we adopt a Design Science Research (DSR) approach and introduce , an automated infrastructure that accurately identifies and links versions of smart contracts into coherent lineages. This tool enables the construction of , an up-to-date, open-source dataset specifically designed to support security research on vulnerability, defect or any other property's evolution patterns in smart contracts. Through a security-focused case study we demonstrate how reveals previously obscured vulnerability lifecycles within smart contract lineages, tracking whether critical security flaws persist or get resolved across versions. This capability is essential for understanding vulnerability propagation patterns and evaluating the effectiveness of security patches in blockchain environments. In the evaluation phase of our DSR approach, we validated our lineage detection methodology against an alternative approach using Locality-Sensitive Hashing (LSH) to cluster contract versions, confirming the security relevance and accuracy of our technique.

This chapter is based on the work published in the following research paper:

- MBODJI, F. N., ADJIBI, V., DIOUF, M. A., Mendy, G., LIU, K., KLEIN, J., BISSYANDE, T. (2025). ContractTrace: Retracing Smart Contract Versions for Security Analyses. In Cybersecurity4D 2025. C4D.

Contents

5.1 Overview

Smart contracts, self-executing programs deployed on blockchains, have gained traction across industries like finance, healthcare, and real estate, where trust, security, and reliability are paramount. However, the inherent immutability of blockchain technology presents considerable challenges when tracking the evolution of these contracts. Therefore, updates to smart contracts require deployment to new addresses, effectively severing the links between different versions. This complicates the tracing of contract lineages as a sequence of versions of the same smart contract. In the literature, authors leveraging lineages for smart contract analyses claim to build on similarity metrics for building such lineages [99, 100]. We postulate that, due to the heavy reuse of code across smart contracts [101], illustrated by the high rates of code duplication within the Ethereum ecosystem, as well as the overall similarity of smart contract behaviors, similarity-based approaches will lead to unreliable lineages. Unfortunately, datasets described in the literature are not shared with the community for assessment or even reuse, hindering broad research on smart contract evolution.

To address the literature gap on smart contract lineages, we propose to build a large-scale, extensible, and open dataset of smart contracts where lineages are tracked. To that end, we rely on a conservative approach based on the concept of *proxy* in smart contract deployment. Proxy contracts, which act as intermediaries and redirect users to the latest contract version, offer a solution to the immutability problem in terms of interaction with the smart contract new versions. Unfortunately, even with proxies, establishing lineage across different contract addresses remains difficult. Existing platforms of smart contracts corpora, like Etherscan and smart-corpus[102] provide detailed information about deployed contracts but do not explicitly trace predecessor-successor relationships.

In this study, we introduce `ContractTrace`, an infrastructure designed to identify and collect smart contract lineages systematically. `ContractTrace` leverages proxy contracts to trace contract updates accurately and produces *lineageSet*, a comprehensive and open-source dataset of smart contract lineages. This dataset facilitates large-scale research on contract evolution and provides new insights into how smart contracts are maintained and updated over time. Currently, *lineageSet* contains 1 055 smart contracts distributed across 347 lineages. It is openly available on Github:

`https://anonymous.4open.science/r/sclineages-A9A2`

Community Benefits. Our infrastructure serves as a valuable resource for software engineering and security research, offering a comprehensive dataset of linked smart contract versions that can be leveraged for various analytical purposes. This enables researchers to explore contract evolution, vulnerability management, and the reliability of existing lineage construction techniques.

- **Case study #:** *Identifying vulnerability life-cycle for smart contracts.* Vulnerability fixes in software are often silent. Since *lineageSet* includes lineages of production smart contracts, we apply vulnerability detection tools on the different versions of the lineages to retrieve the code changes that have led to vulnerability warning apparitions and disappearances.
- **Evaluation:** *Revisiting the reliability of similarity-based construction of smart contracts.* Given the conservative way *lineageSet* was built, we can consider it as ground truth for validating approaches for lineage construction. we assess the reliability of similarity computation as proposed in prior literature. We

consider the Locality-Sensitive Hashing (LSH) method implemented in the Etherscan search engine as a similarity computation model. This evaluation is for validating the relevance of our methodology.

Overall, the main contributions of our work are as follows:

1. `ContractTrace`: we propose a straightforward but novel approach for building lineages, leveraging proxy contracts to ensure accurate lineage tracking, and overcoming the challenges associated with blockchain immutability.

2. lineageSet: we present *lineageSet*, a comprehensive and open-source dataset of smart contract lineages. This rich dataset stands out in the literature due to its:

   (a) *Extensive Scope*: Encompassing 1,055 smart contracts *lineageSet* provides a vast landscape for research exploration.

   (b) *Open Accessibility*: In contrast with prior work that has built lineages, *lineageSet* is freely available on GitHub and will foster reproduction studies and collaboration to advance research on smart contract analysis.

   (c) *Ground-Truth Foundation*: Due to its conservative construction method, *lineageSet* serves as a reliable benchmark for validating future lineage construction approaches.

   (d) *Security analysis*: *lineageSet* serves to examine the vulnerability life-cycle in smart contracts, allowing for a detailed analysis of how vulnerabilities emerge and are resolved across contract versions.

In the following section, we define and present key concepts used throughout the chapter. We also describe our research method, Design Science Research (DSR), which is applied in the subsequent sections, followed by the discussion, related works, and finally, the conclusion.

5.2 Background and Foundation

5.2.1 Ethereum Smart Contract

In 1997, Szabo envisioned smart contracts as self-executing programs that automate agreements between parties, eliminating the need for a trusted third party [103].

The emergence of blockchain technology in 2008, as documented in the white paper of Nakamoto [104], provided a robust platform for the implementation of smart contracts. Blockchain technology facilitates disintermediation by eliminating the need for third-party intermediaries in transactions, fostering trust among participants in a decentralized network. At its core, a blockchain is a chronologically ordered sequence of data blocks, referred to as a distributed ledger. This ledger is managed collaboratively by a peer-to-peer (P2P) network, ensuring decentralization and eliminating the need for participants to trust each other. The integrity of the data within the blockchain is cryptographically secured, with each block linked to the preceding one using a cryptographic hash, thus ensuring immutability and tamper-proof data storage.

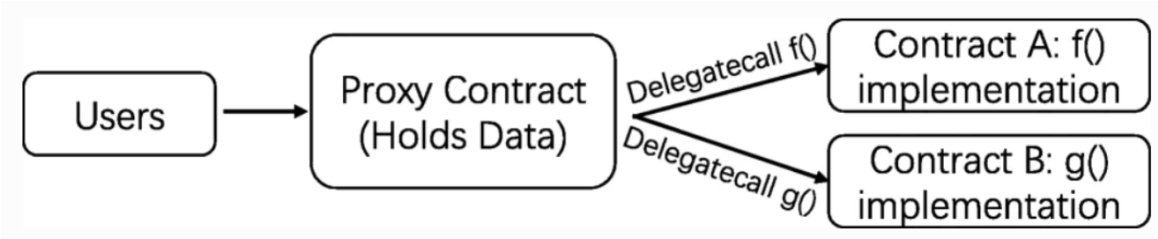


Figure 5.1: An Example of the Upgradable Contract from Chen et al. [1].

With the advent of blockchain technology, a smart contract, as formally described by Nick Szabo [103], has been effectively and fully implemented. Smart contracts are defined as software programs that, once deployed on a blockchain platform, automate the execution of agreement terms between parties, thereby eliminating the need for a third party.

Blockchain technology is currently implemented in various platforms, our study specifically focuses on Ethereum [105], the most prominent platform known for its extensive use of publicly-operated smart contracts. Ethereum offers a virtual machine capable of executing smart contracts utilizing blockchain technology.

5.2.2 Proxy pattern: A Nuance Between Immutability and Upgradable Smart Contracts

Smart contracts inherit the key characteristics of blockchains, such as decentralization and immutability. Once a smart contract is built and deployed at a specific address, it cannot be modified. Any new version of a contract must be redeployed at a new address.

To provide an interface for accessing updated code without the need for the new address, a proxy pattern can be implemented. When this pattern is applied, users interact with the proxy contract, which cannot be modified. However, the proxy is able to point to a configurable address, allowing it to call another contract known as the callee contract. Thus, the parts of the code that are subject to change are located in the callee contract. When an upgrade is needed, the developer redeploys a new version of the callee contract with a new address, and the proxy contract reconfigures its callee contract address accordingly. Typically, an admin user has the privilege to perform this address configuration. Therefore, users interact with the same proxy contract even when upgrades occur.

Fig. 5.1 illustrates this method, which is facilitated by constructs proposed by smart contract programming languages. For instance, in Solidity, the `delegatecall` allows a contract caller (or proxy) to use the code of a callee contract while remaining in the global context of the caller.

Our work builds upon this proxy concept to track smart contract updates.

5.2.3 Smart contracts versions

Predecessor/Successor Relationships in Smart Contracts: We adopt the notion

of predecessor/successor from prior work [99], where a predecessor is defined as the most recent version preceding the successor.

Smart Contract Lineage: In this study, lineage refers to a set of smart contracts where each contract can be paired with another based on a predecessor/successor relationship.

5.2.4 Computing Smart Contract Similarity:

Several reasons contribute to code cloning in deployed smart contracts, such as deploying new versions, the open source characteristic of many smart contracts, the simplicity of copying code fragments instead of writing them from scratch, etc. [101]. The literature contains numerous studies on Ethereum smart contract code clone detection, including clone detection techniques such as SmartEmbed[106], Deckard [107], Nicad[108], and LSH-based approaches [109]. The LSH-based method is implemented in Etherscan, which fingerprints smart contracts and computes their similarity levels, categorizing them as low, medium, or high.

5.2.5 Dynamic and Up-to-Date Repositories for Smart Contract Analysis in Empirical Software Engineering

Etherscan[1] is a widely used Ethereum block explorer that allows developers to submit the source code of their smart contracts, making this code available for verification and transparency. This platform provides a comprehensive database essential for developers to verify and analyze smart contracts deployed on the Ethereum blockchain.

Smart Corpus [102] aims to be an organized and up-to-date repository where developers can systematically access Solidity source code and other metadata about Ethereum smart contracts. This repository facilitates the retrieval of detailed information on smart contracts and their software metrics, streamlining the process for developers and researchers.

Tools like Etherscan and Smart Corpus, which provide accessible and current information on smart contracts, are crucial for advancing empirical software engineering research. They enable a thorough understanding of real contracts deployed on Ethereum, supporting the development of more robust and transparent smart contracts.

These repositories offer a variety of information about smart contracts but do not provide the lineage of contracts. Our study is situated in the same context and aims to contribute information about smart contract lineages.

Hence, our objective is to create a repository where access to versions of smart contracts deployed on Ethereum is streamlined.

5.3 Design Science Research (DSR) as Method

We follow the Design Science Research (DSR) approach. The DSR process includes six steps: problem identification and motivation, objectives for a solu-

[1] `https://etherscan.io/`

tion, design and development, demonstration, evaluation, and communication. These are the essential activities that make up the DSR process:

(a) **Problem Identification and Motivation**: This phase involves defining the research problem and justify the importance of solving it. We recognized the challenge of tracking smart contract lineages. As a specific example, this gap hampers the ability to study how vulnerabilities emerge and evolve over time. This step is developed in detail in (5.4.1) and (5.4.2).

(b) **Define the Objectives for a Solution**: The goal of this step is to formulate solution objectives based on the problem and feasible solutions to define the desired outcome. In this study, we aimed to build an infrastructure to track and link contract versions, enabling research on their evolution and security. Further details of this step are found in (5.4.3)

(c) **Design and Development**: This step is to create the artifact by defining its desired functionality and architecture. We built `ContractTrace` to track smart contract lineages using proxy contracts and created *lineageSet*, a dataset for research purposes. This is covered in (5.5)

(d) **Demonstration**: This step is for utilizing the artifact to address real-world instances of the problem. We applied `ContractTrace` in a case study on the lifecycle of vulnerabilities in smart contracts. This is demonstrated in (5.6).

(e) **Evaluation**: The evaluation tests the effectiveness of the artifact. We validated our approach by applying Locality-Sensitive Hashing (LSH) for clustering contract versions, confirming the approach robustness. Details of this evaluation are in (5.7)

(f) **Communication**: The final step involves sharing the findings and the artifacts design to relevant stakeholders. We consider the communication step as the dissemination of results through publication of this study.

5.4 Problem, Motivation, and Objectives

This section presents the application of the first two phases of the Design Science Research approach.

5.4.1 Studies Highlighting the Need for Contract Lineage Information

Our literature review identified several studies that propose algorithms leveraging code similarity metrics for the classification of smart contract versions.

Chen et al. *[99].* The predecessor/successor concept was used in 2021 by Chen *et al.* to group smart contracts into pairs of predecessor/successor. To obtain predecessor/successor pairs for their research question, which was "Why do smart contracts self-destruct?", they consider that the predecessor had to have executed the self-destruct function and that the successor must be deployed by the same creator address as its predecessor, and it must also have similar

functionality to that of the predecessor. Hence, to determine the similarity rate between two contracts, they employed the SmartEmbed tool [106] and chose 0.6 as the minimum value for the similarity rate between the predecessor and successor. To filter out false pairs, they performed a manual check. Their work presents the following limitations: (1) Reproducing their work is difficult due to manual verification. (2) Only contract versions that executed a self-destruct function are included in their shared dataset. (3) Another limitation of their approach is the risk of false predecessor/successor pairs due to the manual verification process.

Huang et al. *[100].* To test their technique, which involves guiding smart contract updates by detecting code smells, they needed lineages of smart contracts. To construct a dataset of lineages, they used two criteria: the submission of contracts by the same creator and a similarity degree of 0.7 or higher between the contracts. Compared to Chen *et al.* , their approach requires a higher similarity rate. However, given the strong tendencies for code copying, using similarity to determine contract lineages may include contracts that do not actually belong to the lineage.

5.4.2 Motivation for Leveraging the Proxy Pattern for Lineage Construction

Our study is motivated by the need to establish up-to-date repositories that facilitate easy and rapid access to versions of smart contracts. These repositories can serve as valuable resources for analyses in studies such as those conducted by Chen *et al.* [99] and Huang *et al.* [100]. This dynamic approach aims to streamline the process of accessing and analyzing smart contracts deployed on Ethereum, thereby supporting empirical software engineering research.

To the best of our knowledge, there is no existing infrastructure that provides a comprehensive, openly accessible, and systematically organized dataset of smart contract lineages. Our main objective is to establish an infrastructure for building a publicly available dataset of smart contracts meticulously grouped and ordered within their respective lineages. To ensure the integrity of the research process and facilitate further exploration within the smart contract analysis community, the dataset will be both reproducible and reliable.

To address this gap, we propose an approach using the Proxy Pattern. Given that contract calls through proxies are trackable, we plan to leverage this feature to develop a method for constructing lineages. This approach will enable the creation of a comprehensive and systematically organized repository, enhancing the accessibility and utility of smart contract data for researchers and developers alike.

5.4.3 Objectives

Our study aims to create a comprehensive and systematically organized repository of smart contracts that will be publicly available and up-to-date. By leveraging the Proxy Pattern, we anticipate several key outcomes:

- Enhanced accessibility, allowing developers and researchers to benefit

from easy and rapid access to a well-organized dataset of smart contracts, which will facilitate empirical software engineering research.
- Detailed lineage information, providing meticulously grouped and ordered smart contracts within their respective lineages, offering valuable insights into the evolution and relationships of smart contracts.
- Reproducibility and reliability, ensuring the dataset supports the integrity of the research process and enables further exploration within the smart contract analysis community.
- Supporting empirical study: By providing comprehensive data on smart contracts deployed on Ethereum, the repository will serve as a valuable resource for conducting empirical studies.

Additionally, we will test the applicability of this repository in analysis scenario to demonstrate its practical utility and effectiveness in real-world research contexts.

These results will contribute significantly to the field of smart contract analysis, supporting the development of more robust and transparent smart contracts.

5.5 ContractTrace: Design and Development

This section presents the protocol for collecting smart contract lineages, its implementation, and the resulting infrastructure.

5.5.1 Experimental setup

Here, we present the collection of smart contracts and the rules defined to sort them into groups according to their belonging to the same lineage.

Step 1: Data Collection

The infrastructure relies on Etherscan to collect smart contracts. We target the contracts that are called via *proxy* contracts.

Targeted contracts: In our collection, we focus only on contracts updated by using the proxy technique. This is because, unlike other upgrade methods, proxies inherently track interactions with the contracts they govern. Hence, this design choice is to reduce the likelihood of false positives when classifying contracts within lineages. Unfortunately, for contracts updated without being associated with a proxy, we have not found a common denominator to retrieve a link between the various versions of the same contract. Prior works retrieved versions of contracts regardless of the technique used to update them. Still, their approaches require manual investigations that can be error-prone (e.g., in [99]) or rely on design choices leading to small datasets (e.g., in [100]). In our study, we aim to propose an approach that yields a large, evolving, and reliable set of contract versions. Consequently, automation is key. Hence, we only target contracts called in proxy contracts.

Data sources: Etherscan and BigQuery While Etherscan offers access to details of smart contracts, including source code and transaction information, it only provides direct access to a limited set of the most recently verified

contracts (500 latest verified contracts [2].). Verified contracts are contracts for which Etherscan has checked that their provided source code matches with their bytecode deployed on the blockchain at the given address. Since our goal is to find lots of proxy contracts, we do not rely solely on Etherscan. To overcome this limitation, we combine Etherscan with the publicly available Ethereum smart contract dataset on Google BigQuery [3], which offers a broader range of contract addresses.

Collecting proxies and their callee contracts: The proxy paradigm allows us to see the executing contract that receives the calls throughout the lifetime of the contract. Our approach leverages the fact that, the proxy calls the 'upgradeProxy' method of the interfacing contract, with the address of the new contract. We use Google BigQuery to identify all the contracts that called that function. The function has a specific keccak-256 value that could be computed to represent the signature of the function. Once the methods are identified, we automatically collect the addresses. These addresses are then used to query Etherscan and collect the details (in particular, the source code) of the callee smart contracts.

Upon getting that list of contracts, we proceed to classify them into their respective lineages based on a defined set of criteria, which will be explained in the next step.

Step 2: Design choice for lineages formation and Contract versioning

After collecting smart contracts that were called by proxy contracts, we classified them into lineages. In the following, we describe the classification process.

Notation Key: In terms of notations, we have:

- C is the set of contracts accessible in the data source (i.e., available in Etherscan);
- Σ is a contract lineage, defined as $\Sigma = S_1 S_2 .....S_n$, where the contract $S_i \in C$ is the ith versions of the contract code called by a unique contract proxy P;
- P is indeed unique, and $proxy(\Sigma) = P$; i.e., each lineage lies with a unique proxy contract.
- P is a proxy means P is a contract which executes a delegatecall instruction so, $instructionsE(P) = I_1 I_2 .....I_n$ where instructions I_i are the executed instructions in P's source code. Hence,$\exists I_x \in instructionsE(P)$ such as I_x contains a delegatecall instructions.
- We also have $address(S)$ which gives the address of a smart contract S;
- We note by $len(\Sigma)$ the number of contract versions in the lineage Σ.

Classification Rules:

A contract S is classified within a lineage Σ based on the following set of rules:

- **Rule 1: Lineage Members are Callees of the Lineage's Proxy**

[2] https://etherscan.io/contractsVerified/

[3] https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics?hl=en

$$(\Sigma = (S_1 S_2 \dots S_n) \text{ and } proxy(\Sigma) = P) \implies \forall S_i \in \Sigma, \exists I_x \in instructionsE(P)$$

such as I_x contains a delegatecall instructions and the callee address in an execution of I_x is $address(S_i)$. So, each lineage relies on one proxy and in a lineage, we cannot find a contract that was not called by the proxy.

- **Rule 2: A lineage has at least 2 versions of contract**

$$len(\Sigma) \geq 2$$

- **Rule 3: The versions are in chronological order and there is no overlap in the activity period of the lineage versions**
  We have:

$$firstDelegateCall(P, S)$$

(respectively

$$lastDelegateCall(P, S)$$

) gives the date of the first (respectively last) execution of a delegatecall instruction in the proxy contract P where the callee address in the delegatecall is $address(S)$;

$$(\Sigma = S_1 S_2 \dots S_n \text{ and } proxy(\Sigma) = P) \implies \forall S_i \in \Sigma$$

$$lastDelegateCall(P, S_i) < firstDelegateCall(P, S_{i+1}) \quad \text{with} \quad i < len(\Sigma)$$

The updated version should replace the previous version then, it starts its activity after the previous stop its activity.
Our goal in doing this work was to end up with a dataset that is of high confidence and can be trusted for many various tasks. The linear model that we decided to follow and the rules that we set make sure that we can confidently affirm that the said contracts are related. A tree-like structure would imply a lot of dependencies that could be hard to verify that they are related.

- **Predecessor/successor pairs of contracts:**
  These rules are applied to classify contracts in their lineage. Then, in each lineage, we classify contracts in couples of predecessor/successor.
  A predecessor (respectively a successor) of a contract is the contract corresponding to the most (respectively the least) recent version which precedes (respectively succeeds) the contract.

$$\Sigma = S_1 S_2 \dots S_n \implies \forall S_i \in \Sigma, predecessor(S_i) = S_{i-1} \quad \text{with} \quad i > 1$$

and

$$successor(S_i) = S_{i+1} \quad \text{with} \quad i < len(\Sigma)$$

- **Predecessor/successor pairs of file.sol:**
  Each contract version in a lineage is identified by one address; some contracts have more than one file with a ".sol" extension.
  The ".sol" extension refers to Solidity code files. Indeed, a contract

$$S = f_1 f_2 \dots f_n$$

  where f_i is a file with extension ".sol".

We also classify files from pairs of predecessor and successor contracts to facilitate analyses based on code changes. Each file is identified by its filename, which may change from one version to another. However, when examining predecessor-successor contract pairs, we observed that filename changes within the same subdirectories tend to involve only a small number of characters. For example, we found files named "LandRegistryV2.sol" and "LandRegistryV3.sol" accessible via the same subdirectories in two consecutive versions of smart contracts. We account for the possibility that these minor changes can appear in the filename, considering a similarity threshold with a two-character difference. We implement this proposal to build our infrastructure and collect smart contracts lineages.

5.5.2 Resulting dataset: *lineageSet*

Following the implementation and execution of the two-step process outlined above, we obtain *lineageSet*. This dataset provides details on smart contract lineages built based on proxy contracts and their called contracts. *lineageSet* aims to contribute to the field of smart contract software engineering research. Not only will *lineageSet* provide much-needed data to the community, but also it ensures that competing approaches are benchmarked transparently on the same diverse and large-scale data. *lineageSet* will keep growing because the collection is conducted regularly using Etherscan and BigQuery to collect real-world contracts deployed on the Ethereum platform. The implementation and results are open access[4]. We encourage users of our dataset to share their analysis results with the community by adding their links to the dedicated page in the repository. This page aims to foster collaboration between researchers on smart contract software engineering, promote open data and up-to-date datasets, and enable comprehensive analyses.

Figures: After our first execution, we have figures reported in 5.1

Our dataset *lineageSet* had 1055 smart contract addresses which were called in 347 distinct proxy addresses. Then, we identified 347 lineages having 706 predecessors/successors pair. Only 48.48% of smart contracts are open source. The total number of Solidity files in these contracts is 6049.

In figure 5.2, we classify lineages according to their size in terms of their number of contract versions. The lineage with the most number of contract versions encompasses 14 versions of that contract. one hundred seventy (170) smart contract lineages have only two (2) versions collected. It takes an average of 23 days to deploy a new version of a smart contract. We have restructured the lineages and the predecessor/successor relationships using open-source contracts. Additionally, we have paired the files and functions within these lineages. Approximately 88.73% of the Solidity files had at least one other version in the lineages. Additionally, we calculated the average

[4] `https://anonymous.4open.science/r/sclineages-A9A2`

Table 5.1: Summary of Dataset Figures

Metric	**Value**
Lineages identified	347
Pairs of predecessor/successor contracts	706
All smart contracts	1055
Percentage of open source smart contracts	48.48%
Solidity files in open source smart contracts	6049
Percentage of updated files	17.79%
Pairs of of predecessor/successor files	3450
Average days to deploy new version	23
Files in predecessor/successor files pairs	88.68%
Average similarity rate between files paired	98%
Files pairs with similarity rate $\geq$ 90%	97%
Number of functions classified in pair	43964

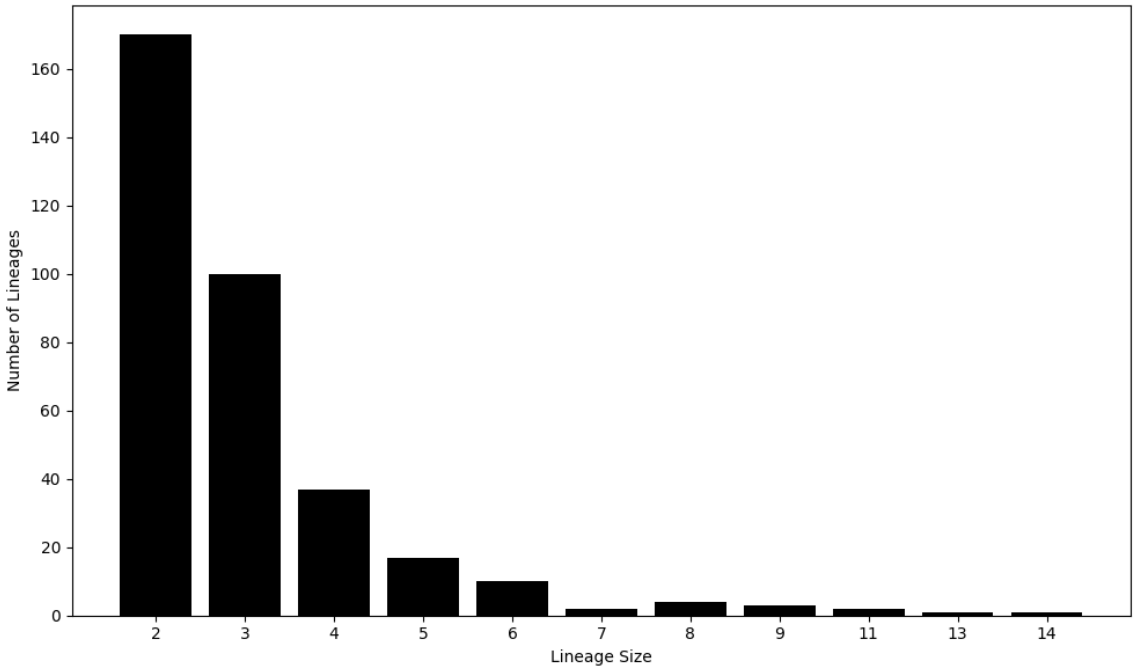


Figure 5.2: Sizes of lineages in *lineageSet*

similarity rate between these files, classified as predecessor-successor in 3450 pairs, and found an average rate of approximately 98%, whether comparing lines of code or entire files. In this file, we have 43964 pairs of contracts in which the data for the pie chart in Fig. 5.3 was derived from a comprehensive analysis of Solidity files, where each file was compared to its immediate predecessor to calculate the similarity rate. Specifically, Fig. 5.3 illustrates the distribution of similarity rates among pairs of Solidity files. Consequently, the pie chart categorizes the pairs into two distinct groups based on their similarity rates: those with a similarity rate of less than 90% and those with a similarity rate of 90% or higher.

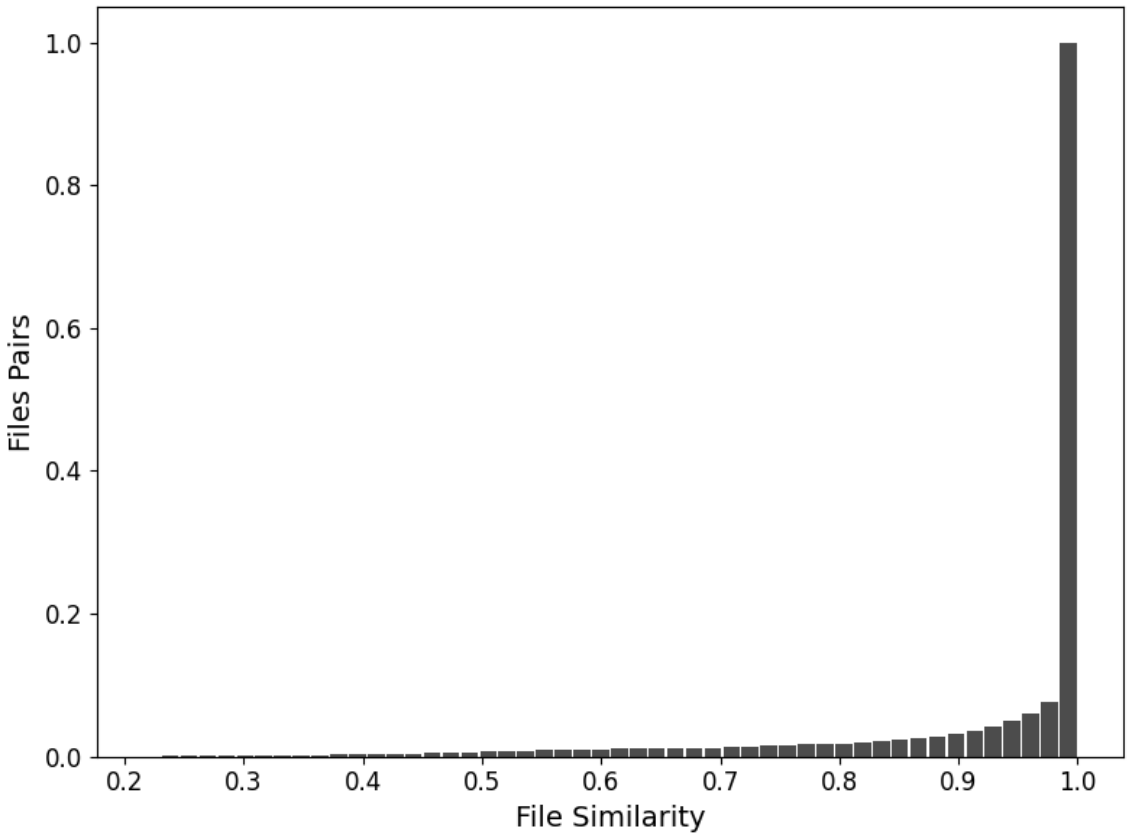


Figure 5.3: Distribution of Similarity Rates in Predecessor/Successor Pairs of Solidity Files

The analysis reveals that a significant majority, 97%, of the pairs exhibit a similarity rate of 90% or higher. On the other hand, only 3% of the pairs have a similarity rate of less than 90%, suggesting a relatively small proportion of files with lower similarity. This rate quantifies the degree of similarity between two consecutive file versions, providing insight into the extent of changes made between versions. Additionally, we could further analyze the evolution of smart contract functionalities over time.

5.6 Demonstration: Building a dataset on Smart contract vulnerabilities and code changes

Goal: With this case study, we aim to conduct an empirical analysis of vulnerability management in smart contracts, leveraging smart contract lineages of

lineageSet. To that end, we reuly on vulnerability detection tools that we apply on smart contract versions and track the appearance and disappearance of vulnerability warnings.

Prior work has created a dataset by tracking GitHub smart contract projects that have vulnerability fix commits [110]. In contrast, we aim to construct a vulnerability lifecycle dataset based on deployed smart contracts available on Etherscan. Indeed, Etherscan ensures the authenticity of smart contracts by allowing users to verify the actual deployed code, while GitHub only provides access to the code without guaranteeing that it was deployed on the blockchain. Moreover, Etherscan offers more comprehensive access to smart contracts directly deployed on the Ethereum blockchain, ensuring complete data on contract interactions, including all transactions and event logs, which may not be fully captured on GitHub. Additionally, in terms of size, the GitHub-based approach is limited (46 projects). This motivates our study to focus on deployed contracts. We aim to adapt specific questions from the previous study regarding vulnerability distribution and first and last occurrences to the context of deployed smart contracts.

- *Q1.1* How many vulnerabilities are reported by the vulnerability analysis tools, and in how many Solidity files do they occur?
- *Q1.2* How many vulnerabilities have disappeared, and how many new vulnerabilities have been introduced?
- *Q1.3* How many vulnerabilities have been patched?

Method: Life cycle analysis requires smart contract versions, which are not readily available on Etherscan. We therefore rely on *lineageSet*. We employ analysis tools such as Slither [111], Mythril [112], and Conkas [113] to detect vulnerabilities in these versions. Slither and Mythril were chosen for their effective balance between performance and execution cost [114], and Conkas was integrated subsequently.

We analyzed this dataset to answer the questions.

Results:

Vulnerabilities Distribution. The dataset includes 79,677 data points detailing vulnerabilities across 384,676 vulnerable lines of code contained in 4,449 different files from 470 unique smart contracts (91.41%) distributed across 165 distinct lineages in *lineageSet*. This high rate of vulnerable smart contracts is obtained by combining the tools through a union operation and significantly decreases when the tool results are combined through the intersection. This finding is similar to the results of previous works that analyzed smart contracts with nine vulnerability detection tools [114], highlighting performance issues of vulnerability detection tools.

Figure 5.4 illustrates the distribution of distinct vulnerabilities, vulnerable files, and total detected vulnerabilities across various security analysis tools. A logarithmic scale is used to account for the significant variation in results. Each element in the diagram is represented by a specific color: blue for distinct vulnerabilities, gray for vulnerable files, and red for total vulnerabilities. The figure provides a visual breakdown of how vulnerabilities and vulnerable files are identified across the different tools.

The diagram shows the differences in the types and distribution of vulnerabilities detected, offering valuable insights into the effectiveness and coverage of each tool.

Vulnerabilities life cycle. 49.19% of the vulnerabilities represent newly introduced vulnerabilities, while 21.53% of them have disappeared in a successor version. 16.73% of updated files in open source contracts of *lineageSet*, we have at least a vulnerability

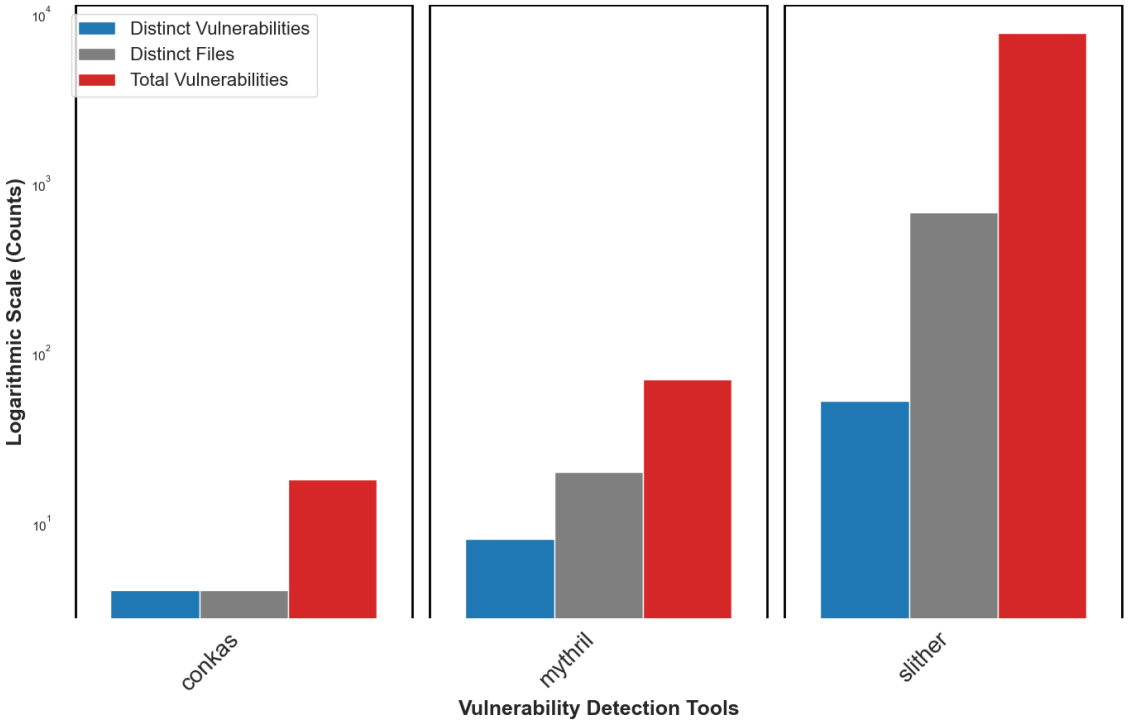


Figure 5.4: Q1.1: Vulnerabilities and Vulnerable Files by Tool

disappearing without a newly introduced vulnerability. These files were distributed across 164 contract versions in 99 contract lineages. *Vulnerabilities persistence* We note an average of 283 days for vulnerabilities to disappear without a new introduction in lineage, while our previous finding underscores the need for Automated Program Repair (APR) tools.

5.7 Evaluation: Revisiting the Reliability of Similarity-Based Construction of Smart Contract Lineages

Goal: Our objective is to evaluate the effectiveness of similarity-based lineage construction methods used in prior research. To that end, we leverage *lineageSet*, built conservatively, as a ground truth dataset. We consider the locality-sensitive hashing (LSH) to be a measure of similarity. It is used in the Etherscan search engine.

Methodology: LSH is integrated within Etherscan, underlying its search engine, which allows for greater automation and flexibility in processing future Ethereum features. We compute smart contract similarity using Etherscan-based LSHimplementation. The similarity scores are expressed in three categories: Low, Medium, and High. Because our lineages in *lineageSet* are Ethereum smart contracts, they can serve as ground truth to discover false positives and false negatives based on the applied similarity thresholds of Etherscan. The experiment answers the following research question:

RQ2: To what extent does the LSH-based approach accurately identify lineage relationships between Ethereum smart contracts?

The LSH-based approach of the Etherscan search engine evaluates smart contract similarity by comparing their fingerprints. In practice, it is possible to make a request on Etherscan to obtain smart contracts that are similar to a given contract, with their degree of similarity categorized as low, medium, or high.

Given a smart contract S, the engine will output smart contracts that are similar. We consider them as candidates for being in the same lineage as S. To summarize:

(1) We collect contracts that are similar to S

(2) We define a similarity threshold T

(3) We form a lineage with S and the collected contracts that have a similarity level to S of at least T

Results: Since the Etherscan LSH-based similarity computation relies on fingerprints, we differentiate two types of smart contracts which may have different level of reliable fingerprints: open-source smart contracts tend to have more artifacts compared to non-open-source contracts, thus their fingerprints may be more accurate.

The LSH-based approach defines various scenarios to form lineages. The differences between scenarios lie in the defined minimum similarity threshold T that contracts in the same lineage must meet and whether the model targets all contracts or just open-source contracts.

The evaluation methodology consists of three steps:

1. **Ground Truth Lineages Data**: *lineageSet* is used as the benchmark for evaluating the performance of the LSH-based method.

2. **Lineage Construction with the LSH Model**: The model is used to predict other contracts that belong to the same lineage as those in the Ground Truth Lineages Data (*lineageSet*) based on the aforementioned scenarios.

3. **Evaluation**: We compared the lineages formed by the model with the ground-truth lineages in *lineageSet*. The evaluation measured overall precision and recall for each scenario.

Table 5.2: Evaluation of LSH-based Approach for Smart Contract Lineage Identification

Contract Type	Similarity threshold	Precision (%)	Recall (%)	Observations
Open-source	Low	48.33	15.80	Higher precision
	Medium	62.27	11.63	
	High	70.08	6.09	
All contracts	Low	44.58	8.08	Lower precision and recall
	Medium	56.12	6.05	
	High	63.40	3.12	

Table 5.2 indicates a trade-off between precision and recall across different similarity thresholds, with open-source contracts showing better overall performance compared to all contracts combined. Indeed, open-source contracts exhibited higher precision across all similarity thresholds, starting at 48.33% at the low threshold and reaching 70.08% at the high threshold, although recall decreased significantly from

15.80% to 6.09%. In contrast, all contracts, including non-open-source ones, showed lower precision and recall. Their precision began at 44.58% at the low threshold and peaked at 63.40% at the high threshold, while recall declined from 8.08% to 3.12%. These findings indicate that focusing on open-source contracts improves precision, but recall presents a challenge as similarity thresholds increase. This trade-off underscores the importance of balancing precision and recall based on specific use cases and the availability of open-source data. This observation emphasizes the critical need to balance precision and recall when utilizing similarity computed by the LSH method for lineage formation.

Key Insight

Low Recall Across All Scenarios: Similarity-based approach to building lineages for smart contracts leads to low recall. When considering all contracts in Ethereum, the conservative proxy-based approach used to build *lineageSet* has a significantly higher precision and recall. Even when the required similarity is low, recall remains poor with LSH, including when considering open-source contracts.

The results of this approach align with previous research on code smart contract code reuse [101] which postulate that they have many code reuses. The findings of this case study also reinforce our relevance to construct lineages based on proxies.

5.8 Discussion, Limitation and Related work

5.8.1 Lineage construction

Our approach shares commonalities with those of Chen et al. [99] and Huang et al. [100].

Table 5.3: Comparison of our study with previous works

	SCLineage	**Chen et al. [99]**	**Huang et al. [100]**
Targeted method of Contract upgrades	Proxy-based upgrades	with self-destructed version	All
Target deployed contracts	Yes	Yes	Yes
Automation	Fully	No	Fully
Granularity	Contracts, files, and functions	Contract level	Contract level
Similarity-based	No	Yes	Yes
Corpus availability	All contract addresses	Some (self-destructed)	Any
Continuously updated	Yes	No	No

A key advantage of our approach lies in its automation. This ensures scalability and efficiency and reduces the risk of human error. Additionally, the implementation details and results of our work are publicly available on GitHub. Despite the technical differences, we believe there is potential for synergy between our approach and those of Chen et al. and Huang et al. Their techniques could be complementary to ours, particularly for targeting updated contracts that do not utilize a proxy pattern. We plan to investigate the combination of these methods in future work.

5.8.2 Lineage construction Implications

lineageSet extends previous empirical software engineering works on up-to-date repositories for smart contracts, such as Etherscan and Smart Corpus[102]. In java program, the Megadiff[115] study which is on code diff for Java programs enabled studies on automatic program repair (e.j [116]). *lineageSet* is similar to Megadiff and can therefore enable research in field of smart contract similar to those leveraging megadiff. In sum, *lineageSet* emerges as a large-scale, open-source, and reproducible dataset specifically designed for smart contract lineage analysis. The automation and public availability of our approach further enhance its usability and value for the research community.

However, it is important to acknowledge the limitations of our approach, which will be discussed in the following subsection.

5.9 Threats to validity

Our approach has inherent limitations, which we discuss and justify in the table below, explaining our design choices.

Table 5.4: Threats to the validity of `ContractTrace`.

Source	**Risk**	**Rationale for Design Choice**
Proxy contracts only	Limited coverage	Ensures reliable upgrade tracking
Rule 3: Linear versioning assumption	Overlapping versions possible	Reduces complexity

Targeted contracts: Our approach deliberately focuses on contracts upgraded through proxy mechanisms. This inevitably excludes lineages of contracts that were updated without proxies. Indeed, according to a study conducted in 2021 targeting 178 developers [1]: 39.39% of the selected respondents admitted that they discarded the old contract directly and deployed a new one, while 35.76% of them reported developing upgradable contracts. Thus, to be conservative, our dataset excludes many contract lineages. We did not employ similarity-based approaches, which could have allowed us to address various types of contracts. This is motivated by the homogeneity of Ethereum smart contracts, which is facilitated by code plagiarism, among other factors [101].

Rule 3 Contract Version Ordering: As a design choice, we opted for a linear versioning assumption within lineages. This implies that contract versions are assumed to be deployed chronologically, without overlapping activity periods. This

decision was made with the objective of constructing a high-confidence dataset suitable for various applications. The linear model of this established rule allows us to confidently assert the relationships between the identified contracts. A more complex, tree-like structure would introduce intricate dependencies that could be challenging to verify. The weakness of this design choice is that we may exclude some versions in the lineages. In summary, our approach to collecting smart contract lineages introduces the potential to overlook certain contract versions within lineages. Additionally, we have excluded contracts that were not updated using the proxy method. However, these design choices are justified by our prioritization of a conservative approach that minimizes the inclusion of false positives and ensures the integrity of the lineage classifications.

5.10 Summary

Due to the inherent immutability of blockchain technology, smart contract updates require their deployment at new addresses rather than modifying existing ones, thus fragmenting version histories and creating critical blind spots for analyses. Indeed, for example, this fragmentation severely hinders security researchers' ability to track vulnerability lifecycles across contract versions. While platforms like Etherscan provide detailed information about Ethereum smart contracts, they lack crucial functionality to trace predecessor-successor relationships within smart contract lineages, preventing systematic analysis of how vulnerabilities emerge, propagate, and potentially remain unresolved across versions.

To address the challenge of tracing smart contract lineages, we adopt a Design Science Research (DSR) approach and introduce `ContractTrace`, an automated infrastructure that accurately identifies and links versions of smart contracts into coherent lineages. This tool enables the construction of *lineageSet*, an up-to-date, open-source dataset specifically designed to support security research on vulnerability, defect or any other property's evolution patterns in smart contracts.

Through a security-focused case study we demonstrate how `ContractTrace` reveals previously obscured vulnerability lifecycles within smart contract lineages, tracking whether critical security flaws persist or get resolved across versions. This capability is essential for understanding vulnerability propagation patterns and evaluating the effectiveness of security patches in blockchain environments. In the evaluation phase of our DSR approach, we validated our lineage detection methodology against an alternative approach using Locality-Sensitive Hashing (LSH) to cluster contract versions, confirming the security relevance and accuracy of our technique.

We addressed the challenge of identifying and classifying versions of smart contracts, which is crucial for various research areas in smart contract engineering. By applying a Design Science Research approach, we proposed `ContractTrace`, a novel infrastructure for systematically collecting and classifying smart contract lineages. It leverages the proxy pattern to minimize errors in lineage identification. Our up-to-date open-source dataset, *lineageSet* produced by `ContractTrace`, facilitates extensive research and analyses.

We demonstrated the utility of our approach in software engineering research, by conducting a case study focused on vulnerability tracking across smart contract versions, using *lineageSet*. Additionally, our methodology was evaluated through Locality-Sensitive Hashing (LSH) to test the effectiveness of clustering contract

versions, confirming its robustness and relevance.

Building on the findings from both the case study and evaluation, we propose several directions for future research:

- `ContractTrace` and *lineageSet* provide valuable resources for advancing research in software engineering and security, facilitating more effective studies on smart contract vulnerabilities and their resolution. These tools can also be leveraged for developing smart contract repair techniques.
- Our findings from this case study reinforce those of previous works [114, 110] and emphasize the need for further exploration of Automated Program Repair (APR) techniques for smart contracts, particularly those with enhanced precision capabilities. When integrated with `ContractTrace`, these techniques could facilitate the creation of an up-to-date dataset that tracks code changes made to address vulnerabilities, similar to Big-Vul, a dataset focused on C/C++ code vulnerabilities [117].
- Investigating enhancements to the performance of LSH-based similarity metrics could improve the construction of contract lineages, extending their applicability to all types of contracts, irrespective of their update mechanisms. Future models can be tested using *lineageSet* as a ground-truth resource for lineage construction.

6

Boosting Blockchain Transparency with LLMs : Automated Evaluation of Generated Comments for Smart Contracts

Smart contracts require high-quality documentation due to their financial and security-critical nature, yet generating accurate and relevant comments remains challenging. Existing evaluation approaches for automatically generated comments lack both scalability and sensitivity to blockchain-specific concerns such as access control, state transitions, and payment logic. We present a domain-informed evaluation framework that leverages large language models (LLMs) as scalable proxies for expert assessment of smart contract comments. Our approach employs contract-aware prompting strategies to evaluate comments across key semantic dimensions including functionality, state changes, access control, and financial operations. We further introduce a ranking and refinement methodology that enables selection and improvement of comments from multiple generation techniques, allowing adaptation to diverse stakeholder requirements. Experimental results demonstrate that our domain-aware evaluation approach significantly improves the relevance and clarity of generated comments compared to generic evaluation methods. Our findings show that LLMs can effectively serve as evaluators of smart contract comment quality of more targeted, domain-appropriate documentation. This work advances automated documentation practices for blockchain development, providing developers with more reliable tools for creating maintainable smart contract codebases.

This chapter is based on the work published in the following research paper:

- MBODJI, F. N., Mame Mariem Ciss SOUGOUFARA, OUEDRAOGO, W. A. M. C., DIALLO, A., LIU, K., KLEIN, J., BISSYANDE, T. (2025). evalSmarT: An LLM-Based Framework for Evaluating Smart Contract Generated Comments. In The 40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025. Seoul, South Korea: IEEE/ACM.

Contents

6.1 Overview

Smart contracts represent a foundational technology in decentralized systems, enabling the autonomous execution of agreements. Due to their immutable and financial nature, they are particularly sensitive to design flaws and documentation errors. Clear, accurate, and complete documentation is therefore essential not only for code maintainability, but also for security audits, regulatory compliance, and effective collaboration across teams.

Despite this need, smart contracts are often poorly documented [118]. Even when comments are present, they frequently lack relevance or alignment with the underlying code. This has motivated the development of automatic comment generation models tailored to smart contracts [119, 29]. However, evaluating the quality of these generated comments remains a significant challenge.

Traditional evaluation methods rely on surface-level metrics such as BLEU, METEOR, and ROUGE, which fail to capture the semantic and domain-specific aspects of smart contract behavior. Human evaluation, while more nuanced, is time-consuming, subjective, and difficult to scale. In contrast, the emerging paradigm of using large language models (LLMs) as evaluators referred to as *LLM-as-a-Judge* offers a promising alternative. Prior work has demonstrated the effectiveness of LLMs in evaluating requirements [120] and code summaries [121], particularly when guided by carefully designed prompts.

We introduce `evalSmarT`, a modular and extensible framework for evaluating smart contract comment generation using LLMs. Unlike general-purpose summarization tasks, smart contract documentation requires domain-specific knowledge of Solidity and the Ethereum ecosystem. `evalSmarT` addresses this challenge by combining multiple LLMs with ten prompting strategies that incorporate domain awareness, language-specific features, and evaluation framing. Our framework enables scalable, reproducible, and semantically rich evaluation of generated comments, bridging the gap between traditional metrics and expert judgment

6.2 Motivation and Problem Statement

Smart contracts are critical components of decentralized systems. Their immutable and financial nature makes them particularly sensitive to design flaws and documentation errors. While automatic comment generation models have emerged to support smart contract comprehension, their evaluation remains limited. Traditional metrics such as BLEU, METEOR, and ROUGE fail to capture domain-specific concerns like security, gas optimization, or Solidity-specific constructs. Human evaluation, though more nuanced, is costly and unscalable.

Below, we present evaluation methods used in existing studies.

Motivation: Lack of Evaluation Diversity

Most studies rely on traditional automatic metrics such as BLEU, METEOR, and ROUGE, with **0% employing LLM-based evaluation** and 55% incorporating human judgment. This reveals a significant gap in methodological diversity, particularly the absence of modern LLM-based evaluation approaches.

Table 6.1: Existing code comment methods and their evaluation metrics

Method names	Evaluation metrics
SMTranslator [122]	Human judgment
STAN [123]	Human judgment
MMTrans [124]	BLEU, METEOR, ROUGE
SMARTDOC [125]	BLEU, ROUGE, Human judgment
CCGIR [119]	BLEU, METEOR, ROUGE
SolcTrans [126]	BLEU, Human judgment
SCCLLM [29]	BLEU, ROUGE
SCLA [127]	BLEU, METEOR, ROUGE
FMCF [128]	BLEU, METEOR, ROUGE
SmartBT [129]	BLEU, ROUGE, Human judgment
CCGRA [130]	BLEU, METEOR, ROUGE-L, Human judgment

Inference of the Diversity Lack

Although some recent works explore LLMs for the **generation** of smart contract comments [129, 29], to the best of our knowledge, **none** use LLMs as **evaluators**. This contrasts with broader blockchain research, where LLMs are increasingly adopted [131], and highlights the need to investigate their potential in comment evaluation.

Problem and Objective

Research Problem

Problem Statement: Despite advances in smart contract comment generation, evaluation practices remain limited to surface-level metrics or manual human judgment. This lack of semantic depth and scalability highlights a critical gap: the potential of large language models (LLMs) as evaluators remains unexplored.
Objective: To design a tool that uses large language models (LLMs) as automated evaluators for smart contract comment generation, aiming to enhance evaluation depth and scalability, while addressing smart contract-specific concerns.

6.3 System Overview: `evalSmarT`

We present `evalSmarT`, a modular framework for evaluating automatically generated comments for smart contracts. It leverages the *LLM-as-a-Judge* paradigm to assess comment quality. The system is designed to be model-agnostic and can integrate any large language model accessible via local deployment (e.g., through Ollama) or remote APIs (e.g., via OpenRouter). It supports flexible prompt engineering and evaluation workflows, enabling researchers and practitioners to experiment with diverse model–prompt configurations tailored to smart contract documentation.

`evalSmarT` is implemented using both local (Ollama) and remote (OpenRouter) LLM access, and supports evaluation of comments generated by tools such as SCCLLM, MMTrans, and CCGIR.

6.3.1 Evaluation Metrics

The evaluators assess comment quality across four dimensions: accuracy, completeness, clarity, and helpfulness. The helpfulness metric incorporates audience-specific utility assessment, acknowledging the heterogeneous stakeholder landscape in smart contract ecosystems.

Evaluation Metrics

1. Accuracy (0–100)
2. Completeness (0–100)
3. Clarity (0–100)
4. Helpfulness: Identify which audiences would find the comment useful from:

- developer_maintaining_contract
- developer_reusing_code
- developer_integrating_contract
- non_technical_user
- business_analyst

6.3.2 Prompting Strategies and Evaluation Protocol

We define an LLM evaluator as a tuple $\langle M, P \rangle$, where M is the model and P is the prompt template. Prompts are designed to incorporate:

- **Domain knowledge** (e.g., blockchain-specific logic, permission enforcement)
- **Language features** (e.g., Solidity constructs, modifiers, events)
- **Evaluation framing** (e.g., QA-based reasoning)

Table 6.2: Prompting Strategy Design Matrix

Prompt	Domain	Language	QA
P1: Baseline	No	No	No
P2: Domain-aware	Yes	No	No
P3: Language-aware	No	Yes	No
P4: Baseline + QA	No	No	Yes
P5: Domain + QA	Yes	No	Yes
P6: Language + QA	No	Yes	Yes
P7: Unguided Domain	Min	No	No
P8: Unguided Language	No	Min	No
P9: Domain + Language + QA	Yes	Yes	Yes
P10: Domain + Language	Yes	Yes	No

Legend: Yes = Full integration; Min = Minimal guidance; No = Not applied

evalSmart

Type: a modular and extensible framework for evaluating automatically generated comments for smart contracts.
Components: Multiple LLM evaluators with diverse prompt strategies;
Coverage: Around 400 evaluators: 40 LLMs with 10 prompting strategies
Metrics accuracy, completeness, clarity, and helpfulness.

6.4 Demonstration and Use Cases

We demonstrate `evalSmarT` through the evaluation of a smart contract comment generation tool. The system loads a set of (code,comment) pairs produced by the

tool, applies multiple LLM-based evaluators, and generates structured assessments across four dimensions: accuracy, completeness, clarity, and helpfulness. This process illustrates how `evalSmarT` can be used to benchmark the performance of comment generation models in a reproducible and scalable manner.

Beyond this core demonstration, `evalSmarT` enables several practical and research-oriented use cases:

6.4.1 Benchmarking for Research

Researchers can use `evalSmarT` to compare outputs from multiple comment generation tools. The framework supports up to 400 evaluator configurations (based on approximately 40 LLMs and 10 prompt strategies), allowing for fine-grained analysis of model performance under varied evaluation conditions.

6.4.2 Best Output Selection

In practical settings, `evalSmarT` ranks multiple generated comments and selects the most appropriate one. This supports developers in choosing the most informative and accurate documentation, especially when integrating or maintaining smart contracts.

6.4.3 Prompt and Evaluator Extension

The system is designed to be extensible. Users can add new prompts or integrate additional models, enabling continuous refinement of evaluation strategies and adaptation to emerging documentation needs in blockchain development.

Demonstration Summary

The demonstration showcases how `evalSmarT` evaluates the output of a smart contract comment generation tool. A set of code–comment pairs is loaded, and multiple LLM-based evaluators are applied to assess the quality of the generated comments. The system outputs structured scores and justifications across four evaluation dimensions. This demonstration highlights the tool's ability to benchmark models, select the most informative comment, and support reproducible, scalable evaluation workflows.

6.5 Illustrative Evaluation and Insights

To support the demonstration of `evalSmarT`, we conducted a focused experiment showcasing its evaluation capabilities on real-world smart contract summaries. Rather than presenting an exhaustive benchmark, we aim here to illustrate how the tool operates in practice and to highlight the rationale behind its internal evaluator configuration.

6.5.1 Selecting the Default Evaluator

We experimented with 40 evaluator configurations (10 prompts $\times$ 4 LLMs). Among these, the combination of GPT-4 and prompt P6 (language-aware + QA framing) achieved the best alignment with human expert annotations across accuracy, completeness, clarity, and helpfulness dimensions. This configuration serves as the default evaluator in our demonstration.

6.5.2 Example: Comparing SCCLLM and CCGIR

To illustrate the tool in use, we applied `evalSmarT` to evaluate comments generated by two state-of-the-art smart contract summarization systems: SCCLLM [29] and CCGIR [119]. We collected real-world smart contract functions from Etherscan, processed them through both tools, and used our selected evaluator (GPT-4 with prompt P6: language-aware + QA) to assess the quality of the generated comments.

The example showcases how `evalSmarT` facilitates structured comparison between models across the four evaluation dimensions, while also providing audience-specific helpfulness annotations.

`evalSmarT` Configuration for Model Comparison

LLM Component: GPT-4
Prompt: P6–Language-aware-QA-framing
Smart Contract Source: Real-world contracts collected from Etherscan
Generated Comments: Outputs from SCCLLM and CCGIR
Expected Output: Structured evaluation across accuracy, completeness, clarity, and helpfulness, including identification of relevant audiences

6.5.3 Findings

The evaluation revealed clear differences in the performance of the two systems. SCCLLM produced more accurate and complete comments, with better alignment to contract semantics and audience needs. In contrast, CCGIR showed notable weaknesses, particularly when evaluated on smart contracts that differed significantly from its training distribution.

Table 6.3: Evaluation scores for SCCLLM using the GPT-4 + P6 (Language-aware + QA) evaluator.

Acc.	Comp.	Clar.	Overall	Mnt.	Reuse	Integr.	NonTech	Analyst
88.53	73.90	96.22	86.22	0.97	0.99	0.76	0.02	0.06

The results in Table 6.3 highlight the strong performance of SCCLLM when evaluated using the GPT-4 + P6 (Language-aware + QA) configuration. The generated comments exhibit high scores in clarity (96.22) and accuracy (88.53), with a slightly lower but still solid score in completeness (73.90). These scores contribute to a robust overall evaluation average of 86.22.

From an audience-specific perspective, the comments are especially helpful for developers maintaining (0.97) or reusing (0.99) the contract, and to a lesser extent for those integrating (0.76) it. However, the helpfulness drops significantly for non-technical users (0.02) and business analysts (0.06). This indicates that while SCCLLM produces highly accurate and readable comments, its utility remains concentrated among technically proficient users. Broader accessibility would likely require additional language simplification or audience-specific tailoring.

The evaluation scores for CCGIR (Table 6.4) reveal significantly lower performance compared to SCCLLM. Accuracy (10.00) and completeness (6.00) are particularly low, indicating that the generated comments often fail to correctly and fully describe the smart contract functions. While clarity (57.00) is somewhat better, it remains moderate, suggesting the comments are not sufficiently clear or informative.

Table 6.4: Evaluation scores for CCGIR using the GPT-4 + P6 (Language-aware + QA) evaluator.

Acc.	Comp.	Clar.	Overall	Mnt.	Reuse	Integr.	NonTech	Analyst
10.00	6.00	57.00	24.33	0.40	0.40	0.40	0.00	0.00

Regarding audience-specific helpfulness, CCGIR's comments show limited utility for developers maintaining, reusing, or integrating the contracts, with only 40

These findings highlight CCGIR's limitations in generating high-quality and audience-tailored comments for smart contracts, especially when compared to the stronger performance of SCCLLM under the same evaluation conditions.

Summary of Findings

SCCLLM [29]:
Generated comments were generally accurate, complete, and clear. Helpfulness tags showed relevance for technical audiences, especially developers reusing or maintaining contracts.
CCGIR [119]:
Comments lacked precision and completeness, often missing key logic. Performance degraded significantly when the input contracts diverged from the types seen during training.
Conclusion:
`evalSmarT` revealed that SCCLLM generalizes better to unseen contracts, while CCGIR struggles with out-of-distribution functions. This highlights the importance of evaluation tools that account for generalization, semantic accuracy, and audience relevance.

Tool Summary Snapshot

Users: Researchers and practitioners working on smart contract comprehension and comment generation.
Challenge: Lack of semantic, scalable, and domain-aware evaluation of smart contract comments.
Method: LLM-based evaluators using customizable prompt-model configurations for structured evaluation.
Validation: Successfully benchmarked SCCLLM and CCGIR; findings align with human judgment and reveal evaluator sensitivity to prompt design.

6.6 Summary

Smart contract comment generation has gained traction as a means to improve code comprehension and maintainability in blockchain systems. However, evaluating the quality of generated comments remains a challenge. Traditional metrics such as BLEU and ROUGE fail to capture domain-specific nuances, while human evaluation is costly and unscalable. In this chapter, we present `evalSmarT`, a modular and extensible framework that leverages large language models (LLMs) as evaluators. The system supports over 400 evaluator configurations by combining approximately 40

LLMs with 10 prompting strategies. We demonstrate its application in benchmarking comment generation tools and selecting the most informative outputs. Our results show that prompt design significantly impacts alignment with human judgment, and that LLM-based evaluation offers a scalable and semantically rich alternative to existing methods.

We presented `evalSmarT`, a modular framework leveraging the *LLM-as-a-Judge* paradigm for evaluating smart contract generated comment. The tool enables structured, scalable assessment across multiple quality dimensions by integrating a wide range of LLMs and prompt strategies. Through a use case with SCCLLM and CCGIR, we demonstrated `evalSmarT` capabilities for benchmarking, best-output selection, and flexible evaluator configuration. The tool is open-source and readily usable by both researchers and practitioners. Future work will include a more comprehensive evaluation campaign and integration of fine-tuned domain-specific evaluators.

Resources

Video Demo: `https://youtu.be/HXS_Yiszoz4`

Code and Data: `https://anonymous.4open.science/r/SC_code_summarization-4653`

7

Boosting Blockchain Transparency with LLMs : Judging the judges

The evaluation of automatically generated code documentation remains a fundamental challenge in software engineering, particularly for domain-specific contexts such as smart contract development. While Large Language Models (LLMs) have emerged as promising automated evaluators, their reliability in specialized domains requires systematic investigation. This chapter presents an empirical study examining the trustworthiness of LLM-based evaluation for smart contract code summarization. We analyze 16,000 evaluation outputs from 40 distinct LLM evaluator configurations, comparing their judgments against human expert annotations across multiple quality dimensions. Our investigation focuses on four critical aspects of evaluator reliability: alignment with human judgment, self-consistency, preference bias, and sensitivity to prompt design. The findings provide actionable insights for researchers and practitioners implementing LLM-based evaluation pipelines in domain-specific software engineering tasks.

- MBODJI, F. N., M. M. C, SOUGOUFARA, LIU, K., KLEIN, J., BISSYANDE, T. (2025). Assessing the Reliability of Large Language Models as Evaluators for Smart Contract Code Summarization.

Contents

7.1 Overview

Code summarization, the task of generating natural language descriptions for programming code segments, plays a crucial role in software documentation and maintenance [132]. The quality of these automatically generated summaries directly impacts developer productivity, code comprehension, and knowledge transfer within software projects. As summarization techniques become increasingly sophisticated, particularly with the advent of neural approaches, the challenge of reliably evaluating summary quality has become more pronounced.

Traditional evaluation approaches rely primarily on automated metrics such as BLEU [133], ROUGE [134], and METEOR [135], which measure surface-level similarity between generated and reference summaries. While these metrics provide rapid assessment capabilities, they often fail to capture semantic nuances and domain-specific correctness. Conversely, human evaluation offers deeper insights into summary quality but requires substantial time and expert resources, making it impractical for large-scale assessment [136].

Recent advances in Large Language Models (LLMs) have introduced a promising alternative: using LLMs themselves as evaluators, commonly referred to as the LLM-as-a-judge paradigm [137]. Studies across diverse domains have demonstrated that LLM-based evaluation can achieve strong correlation with human judgments while maintaining scalability [138]. However, these evaluations exhibit sensitivity to design choices, particularly prompt formulation and model selection [139, 140]. Furthermore, most existing work focuses on general-purpose tasks, leaving domain-specific evaluation contexts underexplored.

Smart contract development presents a particularly challenging domain for automated evaluation. The immutable and financially critical nature of blockchain systems demands exceptional precision in documentation. Our prior work, EvalSmart [2], introduced a framework for automatically evaluating and refining smart contract comments using LLMs. However, fundamental questions regarding the reliability of these LLM-based judgments remain unaddressed. Specifically, we lack empirical evidence regarding how well LLM evaluators align with human expertise, whether they consistently evaluate their own outputs, and how prompt design influences evaluation quality in this specialized domain.

This study addresses these gaps through a systematic empirical investigation of LLM evaluator reliability for smart contract code summarization. We examine 40 distinct LLM evaluator configurations, combining four different models with ten carefully designed prompt templates. These evaluators assess both initially generated comments and LLM-refined versions, producing 16,000 evaluation instances that we compare against expert human annotations. Our analysis focuses on four dimensions of evaluator trustworthiness: human alignment, self-consistency, self-preference bias, and prompt design impact.

The contributions of this work are threefold. First, we provide the first large-scale empirical assessment of LLM evaluator reliability specifically for smart contract documentation, analyzing correlation patterns across multiple quality dimensions. Second, we systematically investigate how prompt design factors including domain knowledge integration, language specific features, and evaluation framing influence evaluator performance. Third, we offer evidence-based recommendations for configuring LLM-based evaluation systems in domain-specific software engineering contexts.

7.2 Background and Related Work

7.2.1 Code Summarization and Evaluation

Code summarization generates high-level natural language descriptions for programming code segments, providing developers with clear explanations of functionality, logic, and purpose [141]. The effectiveness of summarization systems depends critically on evaluation methodology. Traditional automated metrics operate by comparing generated summaries against reference summaries using various similarity measures. BLEU calculates n-gram precision between candidate and reference texts, while ROUGE emphasizes recall-oriented matching. METEOR extends these approaches by incorporating synonymy and paraphrase detection [135].

Despite their widespread adoption, reference-based metrics exhibit significant limitations. They assume that good summaries must closely match reference texts, potentially penalizing semantically equivalent but lexically different descriptions. Human evaluation addresses these shortcomings by engaging experienced developers to assess accuracy, fluency, and effectiveness [136]. However, such evaluation requires months to complete and involves non-reusable human labor [133], limiting its practical applicability.

7.2.2 LLMs as Evaluators

The emergence of powerful Large Language Models has catalyzed research into automated evaluation systems that can approximate human judgment. LLM-as-a-judge approaches prompt an LLM to assess AI-generated or human-authored content, producing scores or qualitative feedback [137]. Studies across education, healthcare, software engineering, and text summarization have explored models including GPT-3.5, GPT-4, LLaMA variants, Mixtral, Gemini, and PaLM-2 [142, 139, 143, 144]. These investigations assess diverse criteria including coherence, logical consistency, fluency, relevance, comprehensiveness, and factual accuracy [145, 144, 120].

Research demonstrates that LLM evaluators can outperform traditional reference-based metrics in correlation with human quality judgments [138]. However, evaluation quality depends heavily on prompt design. Different approaches include zero-shot and few-shot prompting, rubric-guided evaluation, chain-of-thought reasoning, and structured versus unstructured assessment formats [143, 145, 139]. Output formats range from numeric scores and categorical ratings to detailed explanations and improvement suggestions [120, 139].

Despite promising results, LLM-as-a-judge systems face important limitations. They remain sensitive to design choices, with performance varying substantially based on prompt formulation and underlying model selection [140]. Most existing work concentrates on single domains or narrow task sets, requiring re-investigation and adaptation when applied to new contexts. Furthermore, questions persist regarding potential biases, including self-preference bias where LLMs favor their own generated content, and consistency issues when evaluating similar inputs.

7.2.3 Smart Contract Documentation

Smart contracts execute automatically on blockchain platforms, enforcing agreements through code rather than legal frameworks. Their immutable nature and direct control over financial assets demand exceptional documentation quality. Errors or ambiguities in smart contract documentation can lead to misunderstanding

of contract behavior, integration failures, or security vulnerabilities. Traditional code summarization approaches, when applied to smart contracts, must account for domain-specific constructs including permission modifiers, event emissions, state transitions, and security patterns.

Prior work on smart contract summarization has primarily focused on generation techniques rather than evaluation methodology. EvalSmart [2] introduced a modular framework enabling LLM-based evaluation and refinement of smart contract comments. The system incorporates domain knowledge about blockchain concepts and Solidity language features into evaluation prompts, producing both quality scores and improved comment versions. However, the reliability of these LLM-generated assessments, their alignment with expert developer expectations, and the influence of specific prompt design choices remain open research questions requiring empirical investigation.

7.3 Research Questions

This study investigates the reliability and trustworthiness of Large Language Models (LLMs) when used as evaluators of smart contract code comments. The analysis is structured around the following research questions.

RQ1: Human Alignment To what extent do LLM-based evaluators align with expert human judgments when assessing the quality of smart contract comments?

RQ2: LLM Refinement Impact - Can LLM-generated refinements improve the quality of smart contract comments? Specifically, we examine whether refined comments demonstrate measurable improvements in accuracy, completeness, clarity, and audience targeting compared to their original versions.

RQ3 Self-Consistency. How consistently do LLM evaluators assess comments that they themselves have refined? When an LLM evaluator produces an improved comment version and later re-evaluates it, consistency in scoring would indicate reliable self-assessment. Inconsistency might suggest evaluation instability or drift in judgment criteria

7.4 Experimental Design

7.4.1 Dataset Construction

The experimental dataset integrates smart contract code, automated summarization tools, LLM evaluators, and human expert annotations. Five smart contracts were selected. Two comment generation tools, SCCLLM and CCGIR, produced ten initial comments across all contracts.

The initial comments were evaluated using EvalSmart [2], a platform developed in a previous study for automated evaluation of code comments. Forty distinct LLM configurations, comprising four models and ten prompt templates, were configured within EvalSmart to conduct the evaluation. For each comment, LLM evaluators assigned numeric scores ranging from zero to one hundred for the three quantitative quality dimensions: accuracy, completeness, and clarity, each accompanied by a textual justification. For the helpfulness dimension, evaluators identified the relevant audiences for whom the comment would be useful and provided a justification, without assigning a numeric score. Each evaluator also produced a refined version of the comment intended to address identified deficiencies. This first evaluation phase thus generated four hundred evaluation instances, each containing dimension-specific

scores or audience labels, justifications, and one refined comment. Human evaluators assessed the refined comments following the same four dimensions and the same helpfulness instructions, indicating the relevant audiences and providing justifications. Numeric scores from zero to five were assigned for the three quantitative dimensions to serve as a ground truth reference for alignment analysis.

The four hundred refined comments were annotated by two human evaluators, an experienced blockchain researcher and a professional software engineer. This human annotation provides a ground truth for assessing LLM evaluator reliability.

Finally, all forty LLM evaluators reassessed the refined comments, resulting in sixteen thousand evaluation instances. This second phase enables analysis of self-consistency, self-preference bias, and the impact of comment refinement on alignment with human judgments. Table 7.1 summarizes the dataset composition.

Table 7.1: Dataset Composition Summary

Component	Count
LLM evaluator configurations	40
Human evaluators	2
Evaluated comments	400
LLM evaluation instances	16,000
Quality dimensions assessed	4

7.4.2 LLM Evaluator Configurations

An LLM evaluator is defined as the tuple $\langle M, P \rangle$, where M represents the language model and P denotes the prompt template. As shown in Table 7.3, four models are included, comprising two open-source models (LLaMA 3.2, Mistral) and two proprietary models (GPT-3.5 Turbo, Gemini). Ten prompt templates vary systematically along three dimensions: domain knowledge integration, language feature emphasis, and evaluation framing (question-answering versus direct instruction). Table 7.2 presents the complete prompt design matrix.

The first dimension, domain knowledge integration, determines how extensively blockchain and smart contract concepts appear in evaluation instructions. Full integration provides detailed explanations of blockchain terminology. Minimal integration mentions domain context briefly without detailed exposition. Absence of integration treats smart contracts as generic code requiring no specialized knowledge.

The second dimension, language feature emphasis, controls attention to Solidity-specific constructs and syntax. Full emphasis explicitly instructs evaluators to consider modifiers, events, inheritance patterns, visibility specifiers, and other Solidity-specific elements when assessing comment quality. Minimal emphasis briefly acknowledges language-specific features without detailed enumeration. Absence leaves language features unmentioned.

The third dimension, evaluation framing, determines whether assessment employs question-answering structure or direct instruction. Question-answering framing presents evaluation as answering specific questions about comment quality. Direct instruction framing simply states evaluation criteria without interrogative structure.

Table 7.2: Prompt Design Configuration Matrix [2]

Template	Domain Knowledge	Language Features	QA Framing
P1	None	None	None
P2	Full	None	None
P3	None	Full	None
P4	None	None	Yes
P5	Full	None	Yes
P6	None	Full	Yes
P7	Minimal	None	None
P8	None	Minimal	None
P9	Full	Full	Yes
P10	Full	Full	None

Type	**Model**	**Full Name**
Open source	LLaMA 3.2 3B Instruct	llama3.2:3b-instruct-fp16
	Mistral	mistral
Closed source	GPT-3.5 Turbo	gpt-3.5-turbo
	Gemini 2.5 Flash	gemini-2.5-flash-preview-05-20

Table 7.3: Overview of selected language models categorized by type and full identifier.

7.4.3 Human Annotation Protocol

Human annotation followed a structured procedure designed to ensure consistency and reproducibility. Annotators had access to the original code, the initial comments, and LLM-generated scores and reasoning. Each refined comment was scored from zero to five for accuracy, clarity, and completeness, while helpfulness was assessed across five stakeholder categories. This choice is motivated by a prior study that found that human evaluators tend to reconsider their decisions when they are aware of the model's evaluation and its accompanying explanation [120]. This protocol establishes a ground truth for evaluating the models.

7.4.4 Analysis Methodology

Our analysis employs multiple correlation measures to assess alignment between LLM evaluator scores and human annotations. To ensure comparability between scoring schemes, LLM scores originally provided on a zero-to-one-hundred scale were linearly rescaled to a zero-to-five scale to match the human evaluation scale.

Pearson correlation captures linear relationships between numeric scores, providing insight into whether LLM and human evaluations follow similar trends across the score range. Spearman correlation assesses monotonic relationships using rank-based comparison, offering robustness to non-linear scaling differences [146]. The Intraclass Correlation Coefficient (ICC) measures absolute agreement, accounting for systematic differences in scale usage between evaluators [147, 148, 149].

7.5 Results and Discussion

7.5.1 RQ1: Human Alignment

We evaluated how well LLM evaluators align with human judgments using numeric metrics (ICC, Pearson, Spearman) and audience-based F1 scores.

The analysis shows that the choice of LLM model is the main factor affecting alignment. OpenAI GPT-4 and Google Gemini consistently achieve higher numeric and audience alignment scores compared to LLaMA. For instance, numeric alignment (Pearson) reaches up to 0.62 for GPT-4, while LLaMA remains around 0.22. Audience F1 scores also reflect this trend, with GPT-4 and Gemini around 0.48–0.49, versus 0.24 for LLaMA 3.2. ANOVA results confirm these differences are statistically significant across all numeric and audience metrics, highlighting that model selection is crucial for reliable automated evaluations.

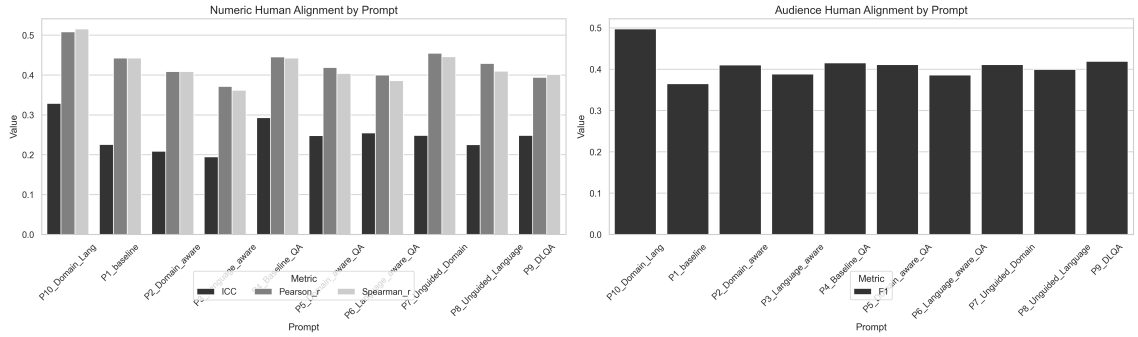


Figure 7.1: Human alignment metrics per prompt template (ordered by complexity). Left: Numeric alignment (ICC, Pearson, Spearman). Right: Audience alignment (F1 score).

Regarding prompts (7.1), numeric and audience alignment vary less between templates, suggesting that prompt choice has a more limited effect than model selection. Nevertheless, prompts with integrated domain knowledge or QA instructions, such as P10_Domain_Lang and P5_Domain_aware_QA, slightly outperform simpler prompts like P1_baseline or P3_Language_aware. This indicates that increasing prompt complexity or incorporating domain-specific guidance can provide incremental improvements in alignment, though the overall effect is small compared to the impact of the model itself.

Overall, the results emphasize that high-performing LLM models are key to achieving strong human alignment, while careful prompt design can provide additional but modest benefits.

7.5.2 RQ2: Impact of LLM Refinement on Comment Quality

To evaluate whether LLM-generated refinements improve comment quality, we compared human annotations of 10 original comments with 400 refined versions produced by 40 LLM evaluator configurations (4 models × 10 prompts).

Table 7.4 summarizes the overall impact of LLM refinement across quality dimensions. Refinements showed positive improvements on all dimensions, with completeness increasing the most (mean $\Delta = 0.95$), followed by accuracy ($\Delta = 0.82$) and clarity ($\Delta = 0.42$).

Table 7.4: Overall impact of LLM refinement on comment quality

Dimension	Mean Δ	% Improved	N
Accuracy	0.82	18.0%	400
Completeness	0.95	19.7%	400
Clarity	0.42	15.3%	400

7.5.2.1 Evaluator Performance

Figure 7.2 illustrates the wide variability across evaluators. The composite improvement score (average of accuracy, completeness, and clarity deltas) ranges from −2.40 to +2.20 on a 5-point scale.

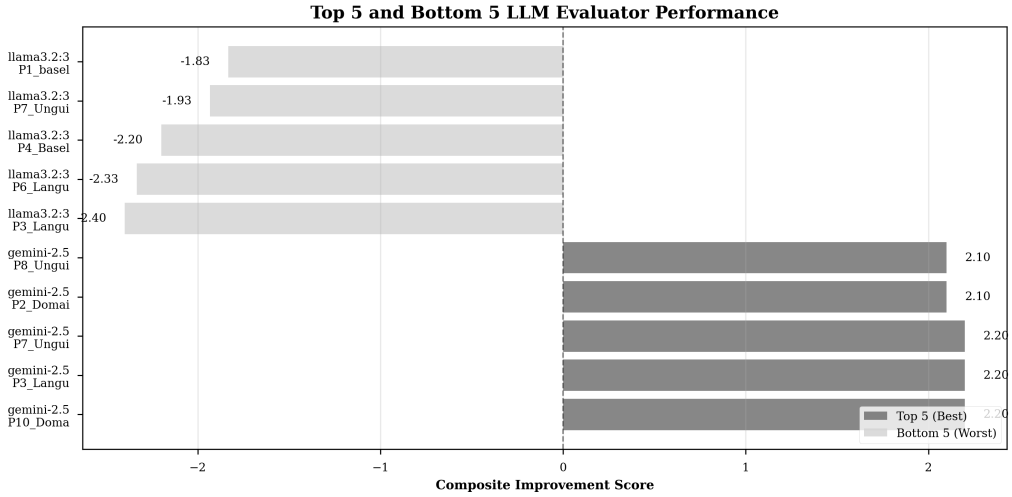


Figure 7.2: Composite improvement scores across 40 LLM evaluator configurations. Positive scores indicate overall improvement; negative scores indicate degradation.

Top performers were mostly Google Gemini configurations using domain-aware or language-aware prompts, with the best achieving +2.4 in accuracy, +2.5 in completeness, and +1.7 in clarity. Bottom performers were LLaMA configurations, with some refinements degrading all dimensions by up to -2.4.

Overall, LLM refinements demonstrate three key patterns. First, model capability drives performance, with Gemini and GPT-4 outperforming LLaMA. Second, technical audiences are favored, while non-technical audiences are frequently removed. Third, prompt variations matter for strong models but cannot compensate for weaker models' limitations. The high variability and low overall improvement rates indicate that LLM refinement is more suitable for generating candidate suggestions rather than fully autonomous comment improvement.

7.5.3 RQ3: Self-Consistency Analysis

We analyze cases in which the same evaluator both refined a comment and later re-evaluated it.

Self-consistency ranges from 67% to 77% across metrics. Accuracy shows the highest stability (76.8%), followed by completeness (72.9%), clarity (69.6%), and overall score (67.0%). Audience consistency reaches 68.0%. Approximately one third

of cases exhibit regression, where the refined comment receives a lower score than the original.

The score difference between evaluation rounds (Δ = re-evaluation−initial evaluation) is substantial in magnitude. Improvements average between +22 and +44 points, while regressions range from −23 to −41 points, indicating limited scoring stability across rounds. Model choice strongly affects consistency. GPT-4 achieves 88–90% consistency across numeric metrics. Mistral ranges between 75–85%. LLaMA shows the lowest stability, reaching 49% on overall score and displaying high variance across metrics. Prompt configuration affects self-consistency. P5 and P3 achieve the highest consistency rates (up to 86%), whereas the baseline prompt show the lowest stability, with P6 reaching 54% on clarity. GPT-4 with P5 and P8 achieves 100% consistency across numeric metrics. Evaluators add 0.49 audience categories and remove 0.38 on average between rounds.

Key Finding. Self-consistency varies substantially across models and prompt configurations, with model capability exerting the strongest effect.

7.6 Perspectives

The role of prompt configuration remains a key factor, as our results suggest domain-aware and QA-enhanced prompts improve reliability, while unguided or purely language-aware prompts yield weaker alignment. Investigating the impact of different prompt strategies more systematically could guide best practices for leveraging LLMs as evaluators.

Additionally, the refinement effect warrants attention: assessing whether improvements made by LLMs to comments lead to better alignment with human judgment can inform the potential of LLMs not only as evaluators but also as quality enhancers. Finally, a thorough model comparison between proprietary models like GPT-4 and open-source alternatives such as Llama 3.2 highlights performance differences in reliability and alignment, pointing to trade-offs between accessibility, cost, and evaluation quality.

7.7 Summary

In this chapter, we investigated the reliability of LLMs as evaluators of smart contract code comments. Our results show that LLMs can achieve strong alignment with human judgments for completeness, moderate alignment for accuracy, and weaker alignment for clarity. Prompt configuration and model choice significantly influence this reliability, with domain-aware and QA-enhanced prompts yielding the best performance.

These findings highlight both the potential and limitations of using LLMs for automated evaluation: while they can serve as partial proxies for human judgment, careful consideration of task framing, model selection, and possible biases is essential. Future work should explore LLM self-consistency, bias tendencies, and refinement impact to fully harness their evaluation capabilities.

8

Adding Blockchain Transparence in AI dataset : Towards Verifiable Certification for Code-datasets

Code agents and empirical software engineering rely on public code datasets, yet these datasets lack verifiable *quality guarantees. Static "dataset cards" inform, but they are neither auditable nor do they offer statistical guarantees, making it difficult to attest to dataset quality. Teams build isolated, ad-hoc cleaning pipelines. This fragments effort and raises cost. We present SIEVE, a community-driven framework. It turns per-property checks into* Confidence Cards*—machine-readable, verifiable certificates with* anytime-valid *statistical bounds. We outline a research plan to bring SIEVE to maturity, replacing narrative cards with* anytime-verifiable *certification. This shift is expected to lower quality-assurance costs and increase trust in code-datasets.*

This chapter is based on the work published in the following research paper:

- Mbodji, F. N., Diallo, E. H., Samhi, J., Liu, K., Klein, J., Bissyande, T. F. (2025). SIEVE: Towards Verifiable Certification for Code-datasets. arXiv preprint arXiv:2510.02166.

Contents

8.1 Overview

Data underpins modern science and machine learning. It powers recommendation systems, code-generation tools, and products used at global scale. Yet dataset trust remains fragile: once published, we often cannot tell if a dataset is complete, clean, or legally compliant. If a dataset contains biases or compliance failures, the flaws propagate, compromising research validity and seeding failures in deployed systems. Other domains (e.g., chip design, infrastructure) certify quality before use. However, for datasets, the foundation of empirical science, we still lack transparent, machine-verifiable certification.

Early documentation efforts set the norm for human-readable records: *Datasheets for Datasets* formalized a structured questionnaire covering motivation, collection, and limitations [150]; the *Data Nutrition Label* proposed modular summaries to surface issues at a glance [151]; and *Data Cards* emphasized user-centric, purpose-driven documentation to aid responsible deployment [152]. To bridge prose and pipelines, recent work standardizes machine-readable metadata: *Open Datasheets* contributes a JSON schema to export structured documentation that downstream systems can parse [153]; *Croissant-RAI* define a Web-native vocabulary for lifecycle, labeling, safety/fairness, and compliance, enabling direct load and validation of RAI (Responsible AI) metadata [154]. While these efforts standardize RAI integration, their effectiveness depends entirely on adoption by dataset providers.

In reality, dataset documents remain scarce. An audit of 7,433 Hugging Face dataset cards found that only 30.9% of repositories contain non-empty cards, although those datasets account for 95% of downloads [155]. Even among the most popular datasets, the critical section *"Considerations for Using the Data"* which should describe biases, limitations, and downstream impacts averages only about 2.1% of the content [155]. At the same time, the *EU AI Act* requires providers to publish training-data summaries and maintain technical documentation for regulatory oversight [156]. The gap between regulatory expectations and current practice illustrates how far the ecosystem is from evidence-backed dataset certification.

Beyond under-documentation, risks are already materializing: widely adopted datasets may carry biases or violations, yet they have been used to support scientific conclusions. [157] shows massive indirect leakage of benchmark data into closed-source LLMs during evaluation.

Code-datasets particularly differ from other corpora: they are executable artefacts whose auditing is both operationally and semantically demanding. In practice, audits require reconstructing toolchains, pinning compilers and package registries, resolving transitive dependencies, and running builds/tests whose outcomes can drift as ecosystems evolve. Meanwhile, repositories become inaccessible, APIs deprecate, new CVEs surface, and stale projects silently bias analyses, making "the same dataset" hard to reproduce across time and machines.

To better understand real needs, we conducted a survey (Cf. 8.2) from which we identified recurring properties required by code datasets. Figure 8.1 contrasts what popular code-ataset cards currently document with these needs.

[1]Properties definitions: `buildability` = repo builds in a smoke run; `test_smoke` = if tests exist, a short run passes; `link_valid` = entries resolve to repo+commit; `dependency_health` = vulnerable dependencies; `license_resolves` = license present & compatible.

Property[1]	**CodeNet** [158]	**CSNet** [159]	**HumanEval** [160]	**APPS** [161]	**The Stack v2** [162]
Buildability	**Yes**	**No**	**No**	**No**	**No**
Test smoke	**Partial**	**Partial**	**Partial**	**No**	**No**
Link valid	**No**	**No**	N/A	**No**	**No**
Dependency health	**No**	**No**	**No**	**No**	**Partial**
License resolves	**No**	**Yes**	**No**	**No**	**Partial**

Evidence pointers:
(Yes)
- CodeNet buildability in the "status" column;
- CSNet: licenses for the source code in the _licenses.pkl

(Partial)
- CodeNet: tests provided but Only for AIZU;
- CSNet: human relevance judgement are given;
- HumanEval: function to test generated in the "test" column;
- The Stack v2 : acknowledge that the training dataset could contain malicious code and the limitation of license attribution

(N/A)
- HumanEval: APPS card: data are handwritten.

(No)
- Informations not found in the dataset cards

Figure 8.1: Documentation *coverage* of practitioner–critical properties *as advertised in dataset cards/docs*. **Yes** = explicitly documented as addressed; **Partial** = partially/indirectly stated; **No** = not stated; N/A = not applicable.
Currently documented information in widely used datasets.

Gap in Datasets and Objectives

Gap. While the ecosystem is converging on standards for where information should reside (Croissant-RAI), and regulators are demanding more (EU AI Act), to the best of our knowledge, there is no measurable evidence on the quality of code datasets, and even less concerning the properties demanded by researchers and practitioners.

Objective. SIEVE: the pioneering solution toward a transparent, machine-verifiable, per-property certificate for code datasets, reporting quality with anytime-valid statistical bounds. These certificates provide verifiable proof of dataset quality.

8.2 Understanding Dataset Challenges

This section investigates practical challenges encountered when using code datasets.

Interview

We conducted semi-structured [163] interviews. The details are given in the table 8.1.

Key Findings:

As summarized in table 8.3, our interviews with SE researchers and practitioners surfaced three recurring patterns: (i) dataset issues are *rarely reported* and projects are often *quietly abandoned*, wasting effort; (ii) *compliance and policy risks* (e.g., licensing, sensitive content) are typically discovered *indirectly and late* in the workflow; and (iii) even within the same group, teams in different SE subareas *do not share signals*, so common risks remain invisible. In short, quality problems are discovered

Recruitment	Participants contacted with study overview
Format	online or face-to-face
Participants	18: (15 SE researchers and 3 AI engineers)
Focus	Dataset quality challenges

Table 8.1: Interview methodology summary

#	Interview Question
1	What common quality challenges have you encountered in code datasets?
2	How have you identified concerns or issues in datasets you worked with?
3	What suggestions do you have for improving dataset documentation and reporting of issues?
4	Can you provide examples of specific datasets where such issues were observed?

Table 8.2: Key questions asked during the interviews.

Table 8.3: Key interview insights on code–dataset issues

Aspect	**Observation**
Indirect discovery (compliance)	Compliance risks (licensing) are rarely detected directly; they surface via colleagues, talks, or reviews.
Missed or low-quality capture	Valuable data is often not captured; indiscriminate scraping and weak filters yield noisy or low-quality corpora.
Abandonment pattern	Teams frequently invest time, then abandon datasets due to quality issues; many cannot later recall the dataset names.
Recall shaped by feedback	Datasets criticized by reviewers or reused by peers are more salient than those abandoned quietly.

reactively rather than *proactively*. These observations motivate our approach: replace ad-hoc, one-off cleaning with a *proactive, certification layer*. Accordingly, we are designing a *systematic analysis* of widely used code datasets to identify concrete manifestations of these issues and to prioritize the property definitions and pinned oracles that SIEVE will certify.

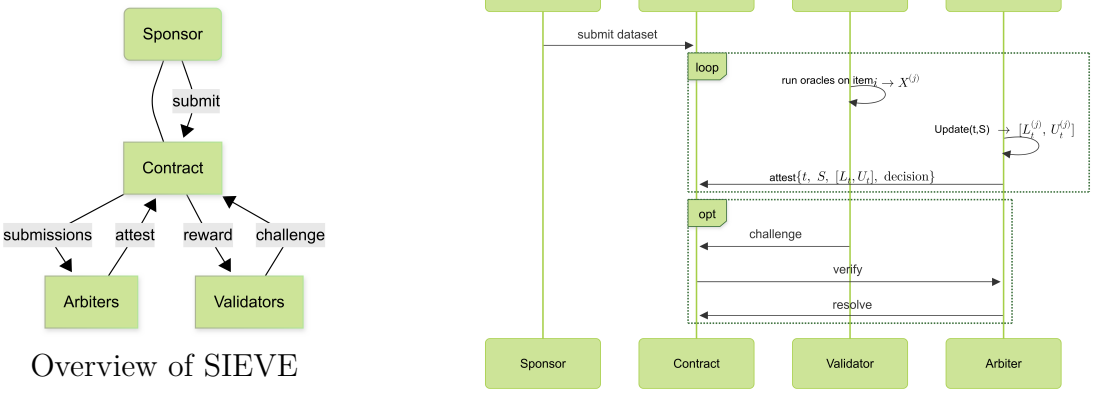
Informed by the insights from these interviews and targeting a potential solution, below, we present our proposal: SIEVE.

8.3 Proposed Framework: SIEVE

As datasets gain value, public and private stakeholders invest heavily in cleaning and maintaining ever-changing corpora. They need continuous, reproducible assurance of quality, yet current efforts are fragmented and often duplicate the same dataset pre-process work. SIEVE empowers the stakeholder consortium to co-sponsor

datasets and collaboratively refine their quality and properties on an ongoing basis. It also transforms checks into transparent, machine-verifiable certificates with quantitative guarantees, thereby reducing redundant effort and enhancing trust.

8.3.1 Global View



Overview of SIEVE

(a)

(b)

Figure 8.2: Overview of SIEVE: (a) Global view, (b) Workflow.

Actors and roles

As depicted in Fig.8.2a, **sponsors** submit datasets for audit. They also bear the cost of processing the entire audit and provide rewards as incentives for validators. The reward is assumed to be a recognition asset, similar to academic contributions such as reviewing papers. In scenarios involving private entities, continuous submission of their local test datasets for validation may be directly enforced by the sponsors. Sponsorship-related business models fall outside the scope of this work.

Because reviewing datasets containing hundreds of millions of records is both complex and expensive, sponsors may not require a full row-by-row assessment. Therefore, we introduce two tolerance measures per property: (i) an error bound ε, which specifies the accepted error on a given property, and (ii) a coverage parameter $(1 - \delta)$, which limits the cost derived from auditing.

Validators are dataset users (e.g., researchers, engineers) who derive the public samples and run lightweight property checks (`oracles`) on these samples. Through sponsors, validators may also define properties aligned with their needs.

Arbiters reproduce validator evidence, aggregate results, and *attest* the current confidence score. Their role can be configured differently depending on the deployment. In an academic context, arbiters may act as reviewers who simply aggregate and recheck validators' claims; in other settings, AI models could serve this role. In all cases, arbiters are auditable, and validators may challenge their outputs. If a conflict arises, a contradiction report is issued to highlight violations of the attestation.

Smart Contract

SIEVE leverages a contract [2] as a trust anchor that makes a dataset audit transparent and verifiable for stakeholders. It anchors the dataset and audit rules, fixes public randomness for unbiased sampling, escrows and settles funds under transparent rules, and keeps an append-only log of attestations and challenges. All checks run as off-chain evidences; the chain stores only commitments, ensuring independence from the sponsor and reproducible audits with an on-chain footprint.

8.3.2 Confidence Card

A *Confidence Card* is a machine-readable record stating, for a dataset version and a binary property P (violation/no-violation), the current evidence: sample count t, observed violations S_t, a live interval $[L_t, U_t]$ for the true violation rate p, and a decision state. It is updated as more items are checked and can be replayed by any third party. We use anytime-valid confidence sequences (CS): at every sample count t (number of distinct items evaluated), CS provide an interval for p that remains valid no matter when we look or stop (continuous monitoring).
Assumptions. Uniform seeded sampling. deterministic, version-pinned oracle; tolerance ε and coverage $1-\delta$ fixed.
Guarantee. We maintain $[L_t, U_t]$ such that

$$\Pr\Big(\forall t \geq 1 :\ p \in [L_t, U_t]\Big) \ \geq\ 1-\delta,$$

valid under arbitrary peeking/stopping.
Construction (Bernoulli, KL time-uniform). Let

$$d(a \,\|\, b) \ =\ a \log\!\left(\frac{a}{b}\right) + (1-a)\log\!\left(\frac{1-a}{1-b}\right),$$

denote the binary Kullback–Leibler divergence between Bernoulli parameters a and b, and define the anytime penalty

$$\psi_t(\delta) \ =\ \log\!\left(\frac{2\log_2(2t)}{\delta}\right),$$

as in [164, 165, 166].

At each time t, we invoke a standard routine that maps $(t, \widehat{p}_t = S_t/t, \delta)$ to a confidence interval $[L_t, U_t]$ using a time-uniform Bernoulli bound (we adopt the KL-based formulation of [164]). Specifically,

$$\begin{cases} U_t = \inf\Big\{\, u \in [\widehat{p}_t, 1] : t\, d(\widehat{p}_t \,\|\, u) \ \geq\ \psi_t(\delta) \Big\}, \\ L_t = \sup\Big\{\, \ell \in [0, \widehat{p}_t] : t\, d(\widehat{p}_t \,\|\, \ell) \ \geq\ \psi_t(\delta) \Big\}. \end{cases}$$

8.3.3 Workflow

This section presents the SIEVE workflow, which is structured into the following steps and illustrated in Fig. 8.2b:

1. The sponsor submits a dataset for audit including:

[2] https://ethereum.org/smart-contracts/; accessed on March 30, 2026

- `DatasetID` = (`rootHash`, `URLs`) : exact dataset version (e.g., commit SHA/CID) and eventual link to the dataset.
- `Property set` $\mathcal{P} = \{(P_j, \varepsilon_j, \delta_j)\}_{j=1}^{J}$
- `Oracles`: content digests (e.g., repo+commit) of the checker for each P_j.

2. The contract rejects duplicates for the same `rootHash` and locks a public randomness seed. All parties derive the same uniform schedule of indices via a pseudorandom function.

3. Repeated until a terminal decision:

   (a) Validators submit the next unclaimed seeded index; arbiters enforce membership and de-dup.

   (b) For each property P_j, compute $X^{(j)} \in \{0, 1\}$ on the sampled item.

   (c) Publish {`indices`, `bits`, `oracles`, `logs`} to a off-chain store (e.g., IPFS) and its digest/URI on-chain.

   (d) Arbiters reproduce the pack, update $(t, S_t^{(j)})$, and call to obtain $[L_t^{(j)}, U_t^{(j)}]$ for each P_j (Sec. 8.3.2).

   (e) Arbiters co-sign `attest`$(t, S, [L_t, U_t],$ `decision`$)$;

   (f) *Stopping rule.*

$$\text{State}(t) \;=\; \begin{cases} \textsc{Clean}, & \text{if } \forall j \;:\; U_t^{(j)} \le \varepsilon_j, \\ \textsc{Dirty}, & \text{if } \exists j \;:\; L_t^{(j)} \ge \varepsilon_j, \\ \textsc{Pending}, & \text{otherwise.} \end{cases}$$

4. When a terminal decision is reached, the per-property card is stored by content address and referenced on-chain next to `rootHash` and `seed`.

SIEVE Confidence Card

`dataset` : {`rootHash`, `seed`}
`property` : (P_j, ε_j, δ_j, `oracle_digest`)
`evidence` : (t, $S_t^{(j)}$, $\widehat{p}_t^{(j)}{=}S_t^{(j)}/t$, $[L_t^{(j)}, U_t^{(j)}]$)
`decision` : (State(t), T2ε_j if State(t) = CLEAN)

Reading rule. CLEAN iff $U_t^{(j)} \le \varepsilon_j$; DIRTY iff $L_t^{(j)} \ge \varepsilon_j$; otherwise PENDING.
Cleanliness lower bound: $1 - U_t^{(j)}$ at coverage $1 - \delta_j$.
Example. Let P_j = `buildability`, $\varepsilon_j = 0.5\%$, $1 - \delta_j = 95\%$. Suppose the card shows $t = 2{,}500$, $S_t^{(j)} = 7 \Rightarrow \widehat{p}_t^{(j)} = 0.28\%$, and $[L_t^{(j)}, U_t^{(j)}] = [0.13\%, 0.48\%]$. Since $U_t^{(j)} = 0.48\% \le 0.5\%$, the decision is CLEAN and T2$\varepsilon_j = 2{,}500$. The dataset's certified cleanliness for this property is at least $1 - U_t^{(j)} = 99.52\%$ (with 95% anytime coverage).

5. Validators may challenge arbiters `challenge(auditId, t, evidence_uri)`. The contract records the resolution (and any penalties in incentive-enabled deployments).

By aligning sponsors needs for clear guarantees with an efficient community participation, SIEVE turns ad-hoc, duplicated preprocessing into a transparent, replayable audit. Each dataset version receives a machine-readable *Confidence Card* that (i) states what was checked and with what tolerance, (ii) publishes live, anytime-valid bounds, and (iii) is tamper-resistant (pinned oracles, reproducible sampling, content-addressed records). Thus, we bring less duplicated cleaning (shared, reusable evidence), lower onboarding cost for downstream users (i.e., cards become portable to CI/catalogs), and higher trust for all stakeholders (decisions are auditable and hard to game), without full rescans of the whole dataset.

8.4 Future Plans

Our future plans focus on operationalizing SIEVE beyond the core statistics (Sec. 8.3.2) so that (RQ1) evidence is captured with near-zero friction inside developer tools, (RQ2) individual cleaning effort and duplication measurably decrease, and (RQ3) the framework demonstrably delivers value in real-world settings.

8.4.1 Editor/CI integration (RQ1):

Ship a lightweight **SIEVE-Client** (VS Code/JetBrains) that opportunistically captures build/test/dependency signals, packages an *EvidencePack* with one-click consent, and submits it.

8.4.2 Efficiency & cost (RQ2):

Add cache/skip rules for heavy checks, artifact/layer reuse, and a dashboard that tracks sample efficiency (T2ε), cleanliness growth ($1 - U_t$), and cost per certified point.

8.4.3 Deployment (RQ3):

Run multi-dataset pilots, publish public cards/artefacts (`rootHash`, `seed`, oracle, evidences), and wire cards to data catalogs.

Following this plan we expect, reproducible pipeline where editors/CI make evidence “nearly free”, cards certify properties with anytime-valid bounds, and pilots show measurable reductions in duplicated cleaning effort and increased trust thus validating the SIEVE for community-driven, per-property dataset certification.

8.5 Summary

Code agents and empirical software engineering rely on public code datasets, yet these datasets lack *verifiable* quality guarantees. Static “dataset cards“ inform, but they are neither auditable nor do they offer statistical guarantees, making it difficult to attest to dataset quality. Teams build isolated, ad-hoc cleaning pipelines. This fragments effort and raises cost. We present SIEVE, a community-driven framework. It turns per-property checks into *Confidence Cards*, machine-readable, verifiable certificates with *anytime-valid* statistical bounds. We outline a research plan to bring SIEVE to maturity, replacing narrative cards with *anytime-verifiable* certification. This shift is expected to lower quality-assurance costs and increase trust in code-datasets.

We introduced SIEVE, a community-driven framework that turns dataset-quality claims into anytime-valid statistical certificates. without scanning entire datasets.

Our goal is to make SIEVE a lightweight yet dependable layer: a card schema, a library, pinned oracles for common properties, and easy editor/CI clients. Dataset hubs and CI systems can consume cards to enforce gates or display cleanliness lower bounds. Practitioners stop rebuilding private filters; instead, they contribute evidence that improves a shared, *anytime-verifiable* certificate.

Part III

Synthesis and Outlook

This part synthesizes our contributions and articulates future directions. Chapter 9 demonstrates how our five empirical studies align with the dual objectives of adapting software engineering practices to trust challenges and leveraging emerging technologies as trust enablers. Through a dimension-aware taxonomy, we position our contributions within the Trust in Depth framework and discuss broader implications for trustworthiness engineering. Chapter 10 charts future research directions, identifying opportunities to expand the framework to additional technology dimensions, develop comprehensive toolchains for dimension-aware trust reasoning, and validate our approach across diverse application domains.

9

Discussion and Synthesis

This chapter synthesizes the thesis contributions and shows how they strengthen software engineering in the era of blockchain and AI. It identifies the key engineering challenges these technologies introduce and presents validated technical interventions grounded in real software engineering practice.

Contents

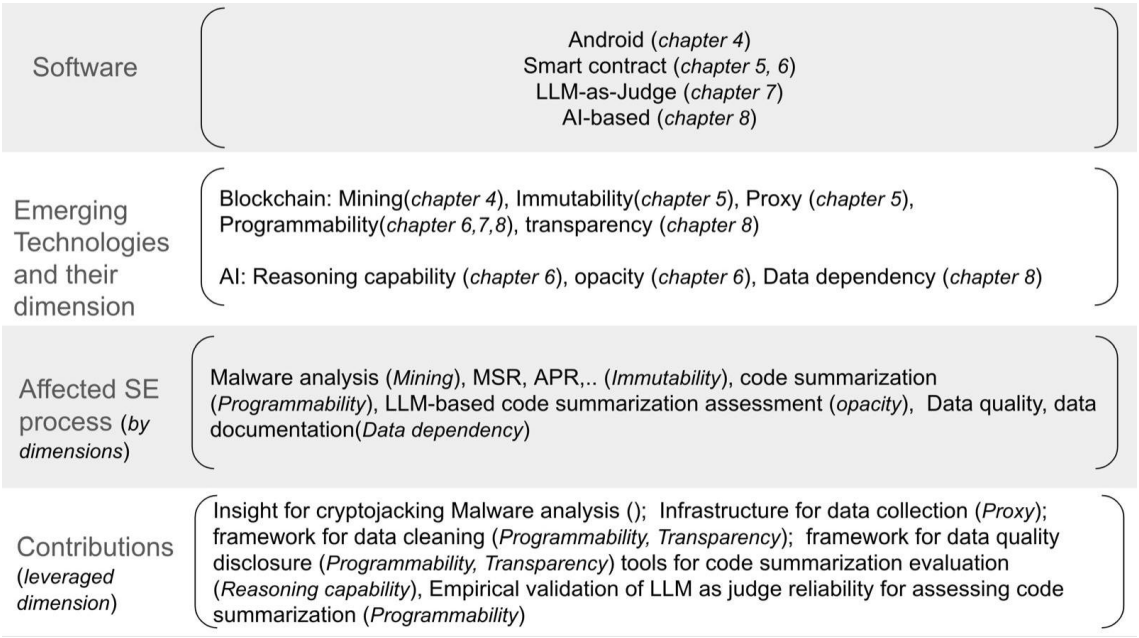


Figure 9.1: Contributions Summary

9.1 Overview

This chapter presents the thesis synthesis and discussion and shows how we apply Trust in Depth through chapters 5-8 and how these contributions align with our objectives. The thesis objectives are: (i) adapting software engineering practices to newly introduced trust challenges, and (ii) leveraging the distinctive capabilities of emerging technologies to reinforce trust in software systems. In Chapter 3, we presented Trust in Depth, our component-based model conceptualizing our contributions which investigate software integrating or interacting with an emerging technology such as blockchain or AI, identify dimension(s) of the emerging technology, and assess their impacts on software engineering processes for trustworthy systems. These impacts can challenge or advance software engineering. Based on these components, Fig. 9.1 presents a taxonomy of our five contributions in chapters 5-8.

9.2 Contribution Alignment with the Research Objectives

This dissertation addresses a central challenge in trustworthy software engineering: the difficulty of reasoning systematically about trustworthiness when software systems integrate or interact with emerging technologies such as blockchain and artificial intelligence. Existing approaches often rely on abstract trust models or technology-centric assumptions that are difficult to operationalize in concrete engineering contexts. In contrast, this thesis adopts a dimension-aware and software-engineering-centric perspective, grounding trust reasoning in observable artifacts, empirical evidence, and validated technical interventions. Indeed, in Fig. 9.1, each affected SE process requires a software engineering adaptation that we take into account in each item of the component contribution (or advances in our conceptualization model). All chapters deal with an affected SE process by a given dimension of an emerging technology and contribute with an adaptation as preconized by our objective (i). In

chapters 5, 6, and 8, the contributions leverage emerging technology dimensions as preconized in our objective (ii).

The contributions presented in Chapters 4 to 8 collectively demonstrate that trustworthiness emerges from the interaction between specific technological dimensions, the intentions and capabilities of involved actors, and the software engineering practices used to design, analyze, and govern software systems. In doing so, they directly address the two objectives stated in Chapter 1. First, they show how software engineering practices must adapt when emerging technologies introduce new trust-related challenges that invalidate existing assumptions. Second, they demonstrate how the unique attributes of these technologies can be harnessed to bolster trust via specific engineering interventions.

9.3 Adapting Software Engineering Practices to New Trust Challenges

Several contributions reveal how emerging technologies introduce trust challenges that are not adequately addressed by established software engineering practices. Importantly, these challenges do not stem from the technologies themselves, but from how their dimensions are exploited, misused, or insufficiently accounted for within existing engineering and governance infrastructures.

Chapter 4 illustrates this point through the study of cryptojacking in Android applications. The issue identified is not the mining capability per se, but its illicit integration into mobile applications by malicious developers, combined with weaknesses in platform-level verification processes. Cryptojacking apps exploit users trust in app marketplaces and evade detection by adopting resource-efficient strategies that fall below traditional thresholds used in malware analysis. This contribution exposes a misalignment between current mobile security practices and the threat models enabled by blockchain-related economic incentives. From a software engineering perspective, it highlights the need to revise detection, vetting, and monitoring practices to account for adversarial uses of otherwise legitimate computational capabilities.

Chapter 5 examines a different kind of challenge, arising from the interaction between blockchain immutability and software engineering research practices. While immutability is commonly associated with integrity and non-repudiation, it disrupts the ability to reconstruct software evolution histories for smart contracts. The absence of explicit version links undermines software repository mining and longitudinal security analyses, which are central to many software engineering methods. The proposed lineage reconstruction infrastructure restores these analytical capabilities, not by altering blockchain properties, but by introducing an external engineering mechanism that compensates for their structural constraints. This contribution demonstrates that trust-enhancing technological properties can inadvertently hinder established software engineering practices if they are not accompanied by suitable supporting infrastructure.

Chapters 6 and 7 further highlight adaptation challenges related to transparency and evaluation. Although smart contracts are publicly accessible, their transparency is limited in practice by poor documentation and the difficulty of assessing generated explanations. The evalSmarT framework addresses this gap by enabling scalable and semantically informed evaluation of smart contract comments using large language models. However, Chapter 7 shows that the adoption of such AI-based evaluation

tools introduces its own trust challenges. Without empirical validation, LLM-based judgments risk becoming opaque sources of authority. By systematically evaluating the reliability of LLM-as-a-Judge approaches, this contribution establishes conditions under which such tools can be responsibly integrated into software engineering workflows.

Taken together, these contributions show that adapting to new trust challenges requires more than extending existing checklists or metrics. It requires revisiting the assumptions underlying software engineering practices and redesigning analysis, validation, and governance mechanisms in response to the concrete effects of emerging technological dimensions.

9.4 Leveraging Emerging Technologies as Trust Enablers

In parallel with identifying new challenges, this dissertation demonstrates how emerging technologies can be actively leveraged to reinforce trustworthiness when their properties are intentionally integrated into software engineering processes.

Chapter 6 illustrates how blockchain transparency can be transformed from a passive property into actionable trust evidence. While smart contract code is publicly available by design, transparency alone does not guarantee comprehensibility or auditability. By enabling domain-aware and reproducible evaluation of generated documentation, evalSmarT operationalizes transparency in a way that supports practical trust assessment by developers and auditors. This contribution shows how combining AI-based reasoning with blockchain artifacts can enhance the usability of transparency for software engineering tasks.

Chapter 8 extends this perspective to AI-centric software engineering workflows. Dataset quality is a critical trust concern for AI systems, yet existing documentation mechanisms rely largely on narrative descriptions that cannot be independently verified. The SIEVE framework addresses this limitation by anchoring dataset-quality claims in blockchain-based, anytime-valid certificates. Rather than enforcing fixed quality definitions, SIEVE provides an infrastructure that supports incremental evidence accumulation and shared verification. This contribution illustrates how blockchain immutability and verifiability can be exploited as enabling mechanisms for governance, reproducibility, and trust in software engineering artifacts.

These results demonstrate that emerging technologies do not merely impose constraints on software engineering. When their dimensions are carefully aligned with engineering objectives, they can provide new forms of verifiable and reusable trust evidence that extend beyond what traditional approaches offer.

9.5 A Dimension-Aware Taxonomy of the Contributions

This dissertation adopts the Trust in Depth (TID) approach to analyze how trustworthiness concerns emerge, evolve, and can be addressed in software systems that integrate or interact with emerging technologies. Applying TID across the empirical investigations reported in Chapters 4 to 8 resulted in a structured and comparable characterization of the contributions. Following the X4Y lens wich mean using X for advancing Y, this chapter synthesizes the findings through a dimension-

aware taxonomy that makes explicit the relationships between emerging technologies, their technological dimensions, the software engineering areas they affect, and the corresponding engineering interventions.

The taxonomy is constructed along four explicitly separated layers. First, emerging technologies are treated as the entry points of analysis, distinguishing blockchain and artificial intelligence as independent but sometimes converging sources of trust-related effects. Second, each technology is decomposed into concrete technological dimensions (e.g., mining, immutability, programmability, opacity, or reliance on data), in line with the core principle of TID that trust effects do not arise from technologies as monolithic entities. Third, for each dimension, the software engineering area impacted prior to intervention is identified, together with the nature of the impact (e.g., threat, enablement, or ambiguity). Finally, each contribution is characterized by the software engineering intervention it proposes, including its type, content, applicability, and positioning with respect to the direction of influence between software engineering and the emerging technology.

Applying this taxonomy to the five empirical chapters highlights several recurring patterns. In Chapter 4, the blockchain mining dimension is shown to affect software security practices, specifically malware analysis in mobile ecosystems [34]. The mining capability itself is not treated as inherently harmful; rather, its illicit introduction into application code by malicious developers creates new threat models that existing security practices fail to capture. The intervention therefore takes the form of actionable insights that support security analysis, positioning the contribution as an instance of software engineering adapting to blockchain-induced challenges.

Chapter 5 illustrates a different trust dynamic associated with blockchain immutability [41]. While immutability is commonly associated with integrity and non-repudiation, its interaction with smart contract development disrupts software repository mining practices by breaking explicit version continuity. The impacted software engineering area is the analysis of software evolution in smart contracts. The proposed intervention is an infrastructure that reconstructs contract lineages, restoring the feasibility of longitudinal security analyses. In this case, the contribution again reflects a SE4BL perspective, where software engineering compensates for constraints introduced by a blockchain dimension.

The same chapter also reveals that specific blockchain design patterns, such as the proxy pattern enabled by smart contract programmability, can mitigate immutability-related limitations [41]. By enabling controlled evolution of deployed contracts, this dimension supports software engineering analyses rather than threatening them. The resulting intervention remains infrastructural, but its category reflects a more reciprocal relationship, where blockchain capabilities are deliberately exploited to support software engineering practices (SmartContract4SE).

Chapter 6 focuses on blockchain programmability in modern smart contract languages and its implications for system transparency. Although smart contract code is publicly accessible, insufficient documentation limits practical comprehensibility and thus undermines trust [2]. Here, the impacted software engineering area is documentation quality. The intervention is a tool-based approach that leverages AI reasoning capabilities [137]to automatically evaluate generated documentation. This contribution illustrates how software engineering can enforce blockchain transparency through AI-based tooling, and is positioned as SE4BL particularly LLM4SmartContract.

Artificial intelligence dimensions are further examined in Chapters 6 and 7 [2]. The

reasoning capability of large language models enables new forms of documentation evaluation, directly supporting software engineering activities. However, Chapter 7 demonstrates that AI opacity introduces ambiguity regarding the reliability of such tools. The affected area is system reliability, specifically the trustworthiness of AI-based software engineering tools. The intervention consists of empirical validation of LLM-as-a-Judge approaches, establishing conditions under which these tools align with human judgment. This contribution is categorized as SE4AI (SE4LLM), as it focuses on adapting software engineering validation practices to AI-induced uncertainty.

Finally, Chapter 8 addresses the reliance of AI-based software on data [42]. Both dataset quality and dataset documentation (e.g., data cards) are identified as impacted software engineering areas, with ambiguous trust effects when quality claims cannot be verified. The proposed interventions include a framework for community-driven data cleaning and a blockchain-anchored certification mechanism for dataset properties. These interventions position blockchain as an enabling infrastructure for improving trust in AI pipelines, resulting in a BL4AI contribution grounded in software engineering governance practices.

Taken together, this taxonomy demonstrates how applying Trust in Depth leads to a systematic decomposition of trustworthiness concerns across heterogeneous contexts. It shows that trust effects depend on how specific technological dimensions interact with concrete software engineering areas, and that effective trust engineering requires interventions tailored to these interactions. By making these relationships explicit, the taxonomy provides a structured synthesis of the thesis contributions and illustrates how TID supports both the adaptation of software engineering to new trust challenges and the exploitation of emerging technologies as trust enablers.

9.6 Implications for Trustworthiness Engineering

The synthesis of these contributions supports a central insight of this dissertation: trustworthiness in software systems integrating emerging technologies cannot be reduced to intrinsic properties of blockchain or AI, nor to isolated trust attributes. Instead, it emerges from the continuous interaction between technological dimensions, actor behavior, and software engineering practices.

The Trust in Depth approach provides a structured way to reason about these interactions. By focusing on dimension-specific effects and their manifestations in concrete artifacts and processes, it enables targeted and empirically grounded interventions. Rather than replacing existing trust models, Trust in Depth complements them by anchoring trust reasoning in observable engineering realities.

By demonstrating this approach across heterogeneous domains: mobile applications, smart contracts, AI-assisted evaluation tools, and datasets, this dissertation contributes a pragmatic and extensible foundation for advancing trustworthy software engineering in contexts shaped by rapidly evolving technologies.

10

Research Agenda: Extending Trust in Depth

This chapter outlines a comprehensive research agenda that extends and deepens the Trust in Depth (TID) *framework introduced in this dissertation. While the contributions presented in Chapters 4-8 demonstrate the relevance of dimension-aware reasoning for engineering trustworthy software systems, they represent only an initial exploration of a much broader research space. The agenda proposed in this chapter aims to systematize future investigations by deepening the analysis of trust dimensions, broadening the coverage of software engineering processes, and extending the scope of Trust in Depth across technologies and stakeholders.*

Contents

10.1 Overview of the Research Agenda

As illustrated in Fig. 9.1, the current application of Trust in Depth addresses a limited subset of trust-related dimensions for both blockchain and artificial intelligence technologies. For blockchain, we investigated mining, immutability, and programmability as challenge dimensions requiring software engineering (SE) adaptation, while leveraging proxy-based programmability patterns and transparency as SE advances. For AI, we analyzed data dependency and opacity as challenges, while exploiting reasoning capabilities to advance documentation practices.

Although these contributions validate the usefulness of Trust in Depth as an analytical framework, they leave several questions open. In particular, both blockchain and AI expose additional dimensions that remain unexplored, many software engineering knowledge areas are not yet covered, and trust is still treated primarily as a technical property rather than a socio-technical phenomenon.

The research agenda is therefore structured around four complementary directions: (1) deepening the analysis of trust dimensions by moving from high-level dimensions to internal dimensions, (2) broadening the investigation across architectural layers and software engineering knowledge areas, (3) extending SE adaptation for AI beyond data management and reasoning, and (4) evolving Trust in Depth from a techno-centric framework toward a socio-technical perspective.

10.2 Deepening Trust Dimensions Through Internal Dimensions

The current work has shown that analyzing trust at the level of high-level dimensions may hide important internal mechanisms. This was particularly evident in the case of blockchain programmability, where the investigation of proxy-based programmability patterns revealed that internal design choices can significantly influence trust properties.

10.2.1 From Dimensions to Internal Dimensions

Future work should systematically extend this approach by identifying and analyzing *internal dimensions* within each trust-related dimension. For instance, within blockchain programmability, additional internal dimensions such as gas reliance and gas optimization mechanisms warrant investigation, as they directly impact security, performance, and developer trust. Similarly, determinism may act as both a trust-enhancing property and a source of vulnerabilities when misused.

Beyond programmability, other blockchain dimensions such as throughput and scalability introduce trust challenges related to system performance and reliability [167]. Users and developers must trust that systems can handle anticipated workloads without degradation and that scalability mechanisms do not undermine security guarantees. Investigating the internal dimensions of scalability mechanisms would allow Trust in Depth to capture these subtleties more precisely.

10.2.2 Extending Beyond the Contract Layer

Most blockchain-related contributions in this dissertation focus on the smart contract layer. While Chapter 4 addressed the mining dimension at the consensus layer, this coverage remains limited. Blockchains are structured as multi-layered

systems, and each layer introduces distinct trust challenges that require SE adaptation [168, 54].

Future work should therefore expand the application of Trust in Depth to additional layers, including: the network layer, where trust depends on secure and reliable data transmission; the data layer, which raises challenges related to persistence, consistency, and tamper resistance; and the application layer, where user-facing components must preserve transaction integrity and prevent misuse.

Such a layer-aware analysis would enable the study of cross-layer trust dependencies, including how vulnerabilities at one layer propagate to others and how SE practices can be adapted to manage these interactions.

10.3 Broadening Software Engineering Coverage Using SWEBOK

Beyond technological dimensions and layers, an essential extension of the research agenda is to broaden the coverage of software engineering processes. To date, the application of Trust in Depth has primarily focused on selected activities such as data management, documentation, and analysis.

Future work should explicitly anchor Trust in Depth within established SE knowledge areas as defined by the *Software Engineering Body of Knowledge (SWEBOK)*. This includes, but is not limited to: requirements engineering, where trust assumptions and guarantees can be made explicit; software architecture and design, where trust-related dimensions influence modularity and layering decisions; software testing and verification, where trust properties must be validated under probabilistic, distributed, or immutable behaviors; and software maintenance and evolution, particularly in systems where immutability or continuous learning challenges traditional lifecycle assumptions.

By grounding Trust in Depth in recognized SE knowledge areas, this agenda aims to ensure that trust is treated as a cross-cutting concern throughout the software lifecycle rather than as an isolated technical attribute.

10.4 Extending SE Adaptation for AI Systems

In the AI domain, this dissertation primarily addressed SE adaptation through data management and documentation, while analyzing opacity and data dependency as key trust challenges. However, this focus captures only part of the AI software lifecycle.

Future work should extend SE adaptation for AI by incorporating additional dimensions identified in prior work on machine learning engineering challenges [169]. These include: model learning, where enhancements and improvements at the model level should be systematically captured and shared; model verification, particularly to prevent issues such as data leakage, which can lead to overoptimistic evaluation results [170]; and model deployment, which concerns the integration of trained models into operational software infrastructures.

Existing artifacts developed in this dissertation, such as SIEVE, can be extended to support these dimensions by enabling community-driven sharing, certification of model claims, and coherent links between data, models, and deployment contexts.

10.5 Extending Trust in Depth to Other Emerging Technologies

Although Trust in Depth was developed and validated in the context of blockchain and AI, its core principles are not inherently limited to these technologies. Future work should explore the applicability of the framework to other emerging technologies, such as quantum computing and the Internet of Things, which introduce novel trust challenges related to uncertainty, scale, and system heterogeneity.

Such extensions would help assess the generalizability of Trust in Depth and identify recurring trust-related dimensions across technological domains.

10.6 From Techno-Centric to Socio-Technical Trust

Finally, the current formulation of Trust in Depth is largely techno-centric. However, trust is fundamentally a human phenomenon shaped by perception, expectations, and context.

While technical properties such as transparency, immutability, and verifiability create conditions for trust, this focus ensures methodological rigor, it does not fully capture how trust signals are perceived, interpreted, and acted upon by human stakeholders. Some contributions already incorporate limited user-related perspectives, including the analysis of user reviews in the cryptojacking study and practitioner-informed design considerations in SIEVE. Future work could more systematically integrate user-centered evidence, involving developers, auditors, and end users, to study how technical trust signals influence adoption, misuse, and decision-making in practice.

In this context, affordance theory offers a promising extension to the Trust in Depth framework. By emphasizing that perceived opportunities and constraints emerge from the interaction between users and technology rather than from technology alone [171]. In complementary work outside the scope of this dissertation, an affordance–experimentation–actualization perspective has been explored to support the design of evolutive systems integrating AI and blockchain [172]. Future research could operationalize this lens to analyze trust dynamics in adaptive and continuously evolving software systems.

Incorporating socio-technical dimensions would enable Trust in Depth to capture how trust evidence is perceived, interpreted, and acted upon by developers, auditors, and end users, thereby aligning the framework more closely with real-world software ecosystems.

10.7 Research Roadmap

The research roadmap presented here translates the directions outlined in the previous sections into a structured plan of action for future work. It focuses on extending Trust in Depth (TID) by grounding future investigations in systematic literature evidence and by using TID as a model to structure and analyze research gaps.

Phase 1: Systematic Literature Reviews of Emerging Technologies. The first phase consists of conducting systematic literature reviews (SLRs) on emerging technologies. These SLRs aim to identify trust-related dimensions, internal dimensions, layers, and software engineering practices discussed in the literature. This phase provides an

evidence-based understanding of how trust is currently conceptualized and addressed across emerging technology domains.

Phase 2: Cross-Technology Mapping of Trust Dimensions. Building on the results of the SLRs, a third study synthesizes the findings through a cross-technology mapping. This mapping follows the same structuring principles as Fig. 9.1, while shifting from a thesis-centered perspective to a literature-oriented one. It consolidates dimensions, layers, and software engineering processes identified in the literature and highlights gaps and underexplored areas that remain insufficiently addressed.

Phase 3: Addressing Research Gaps Using Trust in Depth. This phase focuses on addressing the research gaps identified through the cross-technology mapping by applying the Trust in Depth model. It involves adapting and extending software engineering practices to specific dimensions, layers, and technologies, with the objective of producing concrete empirical studies, methodological contributions, and trust-aware software engineering approaches grounded in the TID framework.

Phase 4: Extending and Positioning SIEVE as a Supporting Artifact. The final phase concentrates on extending SIEVE to support additional trust-related dimensions and software engineering processes, particularly in the context of SE for AI. These extensions provide a foundation for integrating research results produced in earlier phases and for facilitating their adoption by practitioners. Through this evolution, SIEVE supports the operationalization of Trust in Depth and contributes to bridging research and practice in trustworthy software engineering.

10.8 Summary

In summary, this research agenda aims to evolve Trust in Depth by: deepening the analysis of trust dimensions through internal dimensions, broadening coverage across architectural layers and SE knowledge areas, extending SE adaptation for AI beyond data management, generalizing the framework to additional emerging technologies, and incorporating socio-technical perspectives on trust.

Together, these directions seek to transform Trust in Depth into a comprehensive, empirically grounded framework for reasoning about trustworthiness across emerging technologies and software engineering practices.

11

Conclusion

This concluding chapter crystallizes the dissertation's core contributions, highlighting their implications for the architectural integrity and engineering of trustworthy software within emerging technological paradigms.

This dissertation has systematically investigated how trustworthiness can be reasoned about, assessed, and engineered in software systems that integrate or interact with emerging technologies such as blockchain and artificial intelligence. By adopting a dimension-aware perspective, we demonstrated that trust is not an intrinsic property of these technologies, but rather an emergent outcome resulting from the interaction between technological dimensions, software artifacts, and engineering practices. Each empirical contribution illustrates this principle by linking a specific dimension of an emerging technology to concrete software engineering interventions that either mitigate trust-related challenges or leverage technological properties to enhance confidence in system behavior.

The empirical studies presented in Chapters 4 to 8 reveal two complementary facets of trustworthy software engineering. On one hand, emerging technologies introduce new challenges that can undermine trust if traditional software engineering practices are applied without adaptation. Examples include the introduction of illicit mining logic in Android applications, which circumvents standard verification processes, or blockchain immutability, which complicates longitudinal analysis of smart contract vulnerabilities. On the other hand, the same technological dimensions can be intentionally harnessed to strengthen trust when integrated thoughtfully into engineering workflows. Mechanisms such as transparency operationalization through domain-aware evaluation of smart contract documentation and blockchain-anchored certification of dataset quality demonstrate how technology can actively support reliable and verifiable software engineering outcomes.

A central insight of this dissertation is that trustworthy software systems require a systematic, dimension-aware approach that considers both the opportunities and constraints introduced by emerging technologies. The Trust in Depth framework provides such an approach by guiding engineers and researchers to identify dimension-specific trust effects, evaluate their interaction with software artifacts and processes, and design targeted interventions. This framework complements existing trust models by grounding abstract concepts in empirically validated engineering contexts, offering both methodological rigor and practical relevance.

While this dissertation emphasizes technical trust evidence, such as verifiable artifacts, automated analyses, and infrastructure-level guarantees, it also points toward the need for incorporating experience-based perspectives in future work. Integrating feedback from developers, auditors, and end users, and applying conceptual lenses such as affordance theory, can enrich our understanding of how trust signals are perceived, interpreted, and acted upon in real-world software ecosystems.

Finally, this work contributes to the broader goal of standardizing trust assessment practices within software engineering. By systematically mapping technological dimensions, trust attributes, and software artifacts to established knowledge areas, this dissertation provides a foundation for generalizable, repeatable, and practice-oriented interventions. Overall, the contributions demonstrate that trustworthiness is not a static property to be measured once, but a dynamic and emergent quality that can be shaped, maintained, and enhanced through thoughtful engineering practices in systems shaped by rapidly evolving technologies.

List of Papers

Papers included in this dissertation:

- published
  - Adjibi, B. V., MBODJI, F. N., Allix, K., KLEIN, J., BISSYANDE, T. F. D. A. (2022). The Devil is in the Details: Unwrapping the Cryptojacking Malware Ecosystem on Android. 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), 153-163. doi:10.1109/SCAM55253.2022.00023
  - MBODJI, F. N., ADJIBI, V., DIOUF, M. A., Mendy, G., LIU, K., KLEIN, J., BISSYANDE, T. (2025). ContractTrace: Retracing Smart Contract Versions for Security Analyses. In Cybersecurity4D 2025. C4D.
  - MBODJI, F. N., Mame Mariem Ciss SOUGOUFARA, OUEDRAOGO, W. A. M. C., DIALLO, A., LIU, K., KLEIN, J., BISSYANDE, T. (2025). evalSmarT: An LLM-Based Framework for Evaluating Smart Contract Generated Comments. In The 40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025. Seoul, South Korea: IEEE/ACM.
- Under review
  - MBODJI, F. N., Mame Mariem Ciss SOUGOUFARA, LIU, K., KLEIN, J., BISSYANDE, T. (2025). eval++: Assessing the Reliability of Large Language Models as Evaluators for Smart Contract Code Summarization
  - Mbodji, F. N., Diallo, E. H., Samhi, J., Liu, K., Klein, J., & Bissyande, T. F. (2025). SIEVE: Towards Verifiable Certification for Code datasets. arXiv preprint arXiv:2510.02166.

Papers published and not included in this dissertation:

- MBODJI, F. N., A. Lo NIANG, DIALLO, A., KLEIN, J., BISSYANDE, T. (2025). Affordance–Experimentation–Actualization- Evolution for AI-based Support of Innovation. In Reimagine the Future Through the Advancement of Information Technology. Polokwane, South Africa: IEEE AFRICON 2025 Conference.

Academic and Community Service

Research Community Service:

- Journal Reviewer: *Empirical Software Engineering* (EMSE)
- Conference Reviewer: IEEE AFRICON

Teaching and Supervision:

- Internship Supervisor, Master level, SnT/TruX, University of Luxembourg
- Guest Lecturer, Introduction to Blockchain , *Université Virtuelle du Burkina Faso* (UVBF), 2022

Outreach and Innovation:

- Co-organiser and Trainer, Annual Cybersecurity Summer School, *LuxWays Project*, LITA/ESP (Senegal) × University of Luxembourg, 2021–2025
- Co-founder and Product Owner, Spin-off Project, University of Luxembourg *(spin-off supported by SnT Technology Transfer Office)*
- Technical Expert, UEMOA Mobile Banking Project, *SnT4Dev Project*

Bibliography

[1] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges," *Empirical Software Engineering*, vol. 26, no. 6, p. 117, 2021.

[2] F. N. MBODJI, M. M. C. SOUGOUFARA, W. A. M. C. OUEDRAOGO, A. DIALLO, K. LIU, J. KLEIN, and T. BISSYANDE, "evalsmart: An llm-based framework for evaluating smart contract generated comments," in *The 40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025*. IEEE/ACM, 16 November 2025, 4 pages, 4 tables. [Online]. Available: https://youtu.be/HXS

[3] Y. Zhang, Y. Zhang, and M. Hai, "An evaluation model of software trustworthiness based on fuzzy comprehensive evaluation method," in *2012 International Conference on Industrial Control and Electronics Engineering*, 2012, pp. 616–619.

[4] A. Heiskanen, M. Newman, and M. Eklin, "Control, trust, power, and the dynamics of information system outsourcing relationships: A process study of contractual software development," *The Journal of Strategic Information Systems*, vol. 17, no. 4, pp. 268–286, 2008.

[5] J. Li, B. Mao, Z. Liang, Z. Zhang, Q. Lin, and X. Yao, "Trust and trustworthiness: What they are and how to achieve them," in *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. IEEE, 2021, pp. 711–717.

[6] Z. Yan, "A comprehensive trust model for component software," in *Proceedings of the 4th international workshop on Security, privacy and trust in pervasive and ubiquitous computing*, 2008, pp. 1–6.

[7] D. Rotolo, D. Hicks, and B. R. Martin, "What is an emerging technology?" *Research policy*, vol. 44, no. 10, pp. 1827–1843, 2015.

[8] C. Ebert, M. Kuhrmann, and R. Prikladnicki, "Global software engineering: Evolution and trends," in *2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)*. IEEE, 2016, pp. 144–153.

[9] M. N. Chaudhry, S. S. U. Din, Z. U. R. Zia, M. K. Abid, and N. Aslam, "Achieving scalable and secure systems: The confluence of ml, ai, iot, block-chain, and software engineering," *Journal of Computing & Biomedical Informatics*, 2024.

[10] V. Bhatnagar, T. Kaur, G. Singh, and R. Mishra, "A comprehensive study on software engineering and emerging technologies," in *2024 5th International Conference on Smart Electronics and Communication (ICOSEC)*. IEEE, 2024, pp. 1579–1582.

[11] P. Nath, J. R. Mushahary, U. Roy, M. Brahma, and P. K. Singh, "Ai and blockchain-based source code vulnerability detection and prevention system for multiparty software development," *Computers and Electrical Engineering*, vol. 106, p. 108607, 2023.

[12] S. Ding, S.-L. Yang, and C. Fu, "A novel evidential reasoning based method for software trustworthiness evaluation under the uncertain and unreliable environment," *Expert Systems with Applications*, vol. 39, no. 3, pp. 2700–2709, 2012.

[13] L. Shi and S. Yang, "The evaluation of software trustworthiness with fahp and ftopsis methods," in *2009 International Conference on Computational Intelligence and Software Engineering*, 2009, pp. 1–5.

[14] F. Hou and S. Jansen, "A survey of the state-of-the-art approaches for evaluating trust in software ecosystems," *Journal of Software: Evolution and Process*, vol. 36, no. 10, p. e2695, 2024.

[15] J. F. Prados-Castillo, J. M. M. Martín, I. Alexeeva-Alexeev, and J. M. Guaita-Martínez, "Understanding blockchain by digital natives: innovation, trust and psychological factors," *Journal of Innovation & Knowledge*, vol. 10, no. 6, p. 100806, 2025.

[16] S. Becker, W. Hasselbring, A. Paul, M. Boskovic, H. Koziolek, J. Ploski, A. Dhama, H. Lipskoch, M. Rohr, D. Winteler *et al.*, "Trustworthy software systems: a discussion of basic concepts and terminology," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 6, pp. 1–18, 2006.

[17] S. Martínez-Fernández, J. Bogner, X. Franch, M. Oriol, J. Siebert, A. Trendowicz, A. M. Vollmer, and S. Wagner, "Software engineering for ai-based systems: a survey," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–59, 2022.

[18] F. Khomh, B. Adams, J. Cheng, M. Fokaefs, and G. Antoniol, "Software engineering for machine-learning applications: The road ahead," *IEEE Software*, vol. 35, no. 5, pp. 81–84, 2018.

[19] B. Liu, G. Li, H. Zhang, Y. Jin, Z. Wang, and D. Shao, "The gap between trustworthy ai research and trustworthy software research: a tertiary study," *ACM Computing Surveys*, vol. 57, no. 3, pp. 1–40, 2024.

[20] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[21] T. Locher, S. Obermeier, and Y. A. Pignolet, "When can a distributed ledger replace a trusted third party?" in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications*

(GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). IEEE, 2018, pp. 1069–1077.

[22] E. Toufaily and T. Zalan, "In blockchain we trust? demystifying the "trust" mechanism in blockchain ecosystems," *Technological Forecasting and Social Change*, vol. 206, p. 123574, 2024.

[23] F. Völter, N. Urbach, and J. Padget, "Trusting the trust machine: Evaluating trust signals of blockchain applications," *International Journal of Information Management*, vol. 68, p. 102429, 2023.

[24] M. Fahmideh, J. Grundy, A. Ahmad, J. Shen, J. Yan, D. Mougouei, P. Wang, A. Ghose, A. Gunawardana, U. Aickelin *et al.*, "Engineering blockchain-based software systems: Foundations, survey, and future directions," *ACM Computing Surveys*, vol. 55, no. 6, pp. 1–44, 2022.

[25] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, "Blockchain-oriented software engineering: challenges and new directions," in *International Conference on Software Engineering Companion*, 2017, pp. 169–171.

[26] G. Anthes, "Artificial intelligence poised to ride a new wave," *Communications of the ACM*, vol. 60, no. 7, pp. 19–21, 2017.

[27] L. Alzubaidi, A. Al-Sabaawi, J. Bai, A. Dukhan, A. H. Alkenani, A. Al-Asadi, H. A. Alwzwazy, M. Manoufali, M. A. Fadhel, A. Albahri *et al.*, "Towards risk-free trustworthy artificial intelligence: Significance and requirements," *International Journal of Intelligent Systems*, vol. 2023, no. 1, p. 4459198, 2023.

[28] D. Partridge and Y. Wilks, "Does ai have a methodology which is different from software engineering?" *Artificial intelligence review*, vol. 1, no. 2, pp. 111–120, 1987.

[29] J. Zhao, X. Chen, G. Yang, and Y. Shen, "Automatic smart contract comment generation via large language models and in-context learning," *Information and Software Technology*, vol. 168, p. 107405, 2024.

[30] K. Salah, M. H. U. Rehman, N. Nizamuddin, and A. Al-Fuqaha, "Blockchain for ai: Review and open research challenges," *IEEE access*, vol. 7, pp. 10 127–10 149, 2019.

[31] S. Patel, I. Jolly, G. K. Sharma, and S. Kumar, "Blockchain with artificial intelligence," *IPEM JOURNAL OF COMPUTER APPLICATION & RESEARCH*, p. 53.

[32] K. Inkollu, S. K. Gorle, S. R. Kondabattula, P. B. Shankar, and M. B. Reddy, "A review on software engineering: Perspective of emerging technologies & challenges." *RICE*, pp. 23–27, 2023.

[33] A. Carvalho, "Bringing transparency and trustworthiness to loot boxes with blockchain and smart contracts," *Decision Support Systems*, vol. 144, p. 113508, 2021.

[34] B. V. Adjibi, F. N. Mbodji, T. F. Bissyandé, K. Allix, and J. Klein, "The devil is in the details: Unwrapping the cryptojacking malware ecosystem on android," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2022, pp. 153–163.

[35] J. Sedlmeir, J. Lautenschlager, G. Fridgen, and N. Urbach, " the transparency challenge of blockchain in organizations," *Electronic Markets*, vol. 32, no. 3, pp. 1779–1794, 2022.

[36] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali*, 2019, pp. 203–225.

[37] T. Gebru, J. Morgenstern, B. Vecchione, J. W. Vaughan, H. Wallach, H. D. Iii, and K. Crawford, "Datasheets for datasets," *Communications of the ACM*, vol. 64, no. 12, pp. 86–92, 2021.

[38] G. Li, C. Zhi, J. Chen, J. Han, and S. Deng, "Exploring parameter-efficient fine-tuning of large language model on automated program repair," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 719–731.

[39] J. A. H. López, B. Chen, M. Saad, T. Sharma, and D. Varró, "On inter-dataset code duplication and data leakage in large language models," *IEEE Transactions on Software Engineering*, 2024.

[40] N. Ostern, "Do you trust a trust-free transaction? toward a trust framework model for blockchain technology," 2018.

[41] F. N. Mbodji, V. ADJIBI, M. A. Diouf, G. Mendy, K. Liu, J. Klein, and T. BISSYANDE, "Contracttrace: Retracing smart contract versions for security analyses," *Cybersecurity4D*, 2025.

[42] F. N. Mbodji, E.-h. Diallo, J. Samhi, K. Liu, J. Klein, and T. F. Bissyande, "Sieve: Towards verifiable certification for code-datasets," *arXiv preprint arXiv:2510.02166*, 2025.

[43] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 3, pp. 1–51, 2018.

[44] R. Milewicz and M. Mundt, "Building bridges: Establishing a dialogue between software engineering research and computational science," *arXiv preprint arXiv:2201.04007*, 2022.

[45] R. C. Mayer, J. H. Davis, and F. D. Schoorman, "An integrative model of organizational trust," *Academy of management review*, vol. 20, no. 3, pp. 709–734, 1995.

[46] A. Jacovi, A. Marasović, T. Miller, and Y. Goldberg, "Formalizing trust in artificial intelligence: Prerequisites, causes and goals of human trust in ai," in *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, 2021, pp. 624–635.

[47] F. Hou and S. Jansen, "A systematic literature review on trust in the software ecosystem," *Empirical Software Engineering*, vol. 28, no. 1, p. 8, 2023.

[48] D. Puthal, N. Malik, S. P. Mohanty, E. Kougianos, and G. Das, "Everything you wanted to know about the blockchain: Its promise, components, processes, and problems," *IEEE Consumer Electronics Magazine*, vol. 7, no. 4, pp. 6–14, 2018.

[49] J. Mattioli, H. Sohier, A. Delaborde, K. Amokrane-Ferka, A. Awadid, Z. Chihani, S. Khalfaoui, and G. Pedroza, "An overview of key trustworthiness attributes and kpis for trusted ml-based systems engineering," *AI and Ethics*, vol. 4, no. 1, pp. 15–25, 2024.

[50] H. Tao, H. Wu, and Y. Chen, "An approach of trustworthy measurement allocation based on sub-attributes of software," *Mathematics*, vol. 7, no. 3, p. 237, 2019.

[51] E. Toreini, M. Aitken, K. Coopamootoo, K. Elliott, C. G. Zelaya, and A. Van Moorsel, "The relationship between trust in ai and trustworthy machine learning technologies," in *Proceedings of the 2020 conference on fairness, accountability, and transparency*, 2020, pp. 272–283.

[52] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," *Advances in neural information processing systems*, vol. 28, 2015.

[53] A. Vacca, A. Di Sorbo, C. A. Visaggio, and G. Canfora, "A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges," *Journal of Systems and Software*, vol. 174, p. 110891, 2021.

[54] S. Demi, R. Colomo-Palacios, and M. Sanchez-Gordon, "Software engineering applications enabled by blockchain technology: A systematic mapping study," *Applied sciences*, vol. 11, no. 7, p. 2960, 2021.

[55] D. Dzhalila, D. Siahaan, R. Fauzan, R. Asyrofi, and M. I. Karimi, "A systematic literature review on blockchain technology in software engineering," *Jurnal ELTIKOM: Jurnal Teknik Elektro, Teknologi Informasi Dan Komputer*, vol. 7, no. 1, pp. 38–49, 2023.

[56] F. Tariq and R. Colomo-Palacios, "Use of blockchain smart contracts in software engineering: A systematic mapping," in *International conference on computational science and its applications*. Springer, 2019, pp. 327–337.

[57] S. Masuda, K. Ono, T. Yasue, and N. Hosokawa, "A survey of software quality for machine learning applications," in *2018 IEEE International conference on software testing, verification and validation workshops (ICSTW)*. IEEE, 2018, pp. 279–284.

[58] M. Perkusich, L. C. e Silva, A. Costa, F. Ramos, R. Saraiva, A. Freire, E. Dilorenzo, E. Dantas, D. Santos, K. Gorgônio *et al.*, "Intelligent software engineering in the context of agile software development: A systematic literature review," *Information and Software Technology*, vol. 119, p. 106241, 2020.

[59] M. Barenkamp, J. Rebstadt, and O. Thomas, "Applications of ai in classical software engineering," *AI Perspectives*, vol. 2, no. 1, p. 1, 2020.

[60] G. Veletsianos, "Cognitive and affective benefits of an animated pedagogical agent: Considering contextual relevance and aesthetics," *Journal of Educational Computing Research*, vol. 36, no. 4, pp. 373–377, 2007.

[61] M. M. Karim, D. H. Van, S. Khan, Q. Qu, and Y. Kholodov, "Ai agents meet blockchain: A survey on secure and scalable collaboration for multi-agents," *Future Internet*, vol. 17, no. 2, p. 57, 2025.

[62] S. J. Alsunaidi, H. Aljamaan, and M. Hammoudeh, "Leveraging machine learning models to improve smart contract security: A survey of vulnerabilities and detection methods," *ACM Computing Surveys*, 2025.

[63] Y. Xinyi, Z. Yi, and Y. He, "Technical characteristics and model of blockchain," in *2018 10th International Conference on Communication Software and Networks (ICCSN)*, 2018, pp. 562–566.

[64] Z. Li, W. Liu, H. Chen, X. Wang, X. Liao, L. Xing, M. Zha, H. Jin, and D. Zou, "Robbery on DevOps: Understanding and mitigating illicit cryptomining on continuous integration service platforms," in *IEEE Symp. Secur. Privacy*, ser. SP '22. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 363–378.

[65] N. Tovanich, N. Soulié, N. Heulot, and P. Isenberg, "The evolution of mining pools and miners' behaviors in the Bitcoin blockchain," *IEEE Trans. Netw. Service Manage.*, vol. 19, no. 3, pp. 1–12, Mar. 2022, early access.

[66] Google, "Google keynote (Google I/O '21) - American sign language," May 2021, accessed June 2022. [Online]. Available: https://www.youtube.com/watch?v=Mlk888FiI8A

[67] ——, "Google keynote (Google I/O '22)," May 2022, accessed June 2022. [Online]. Available: https://www.youtube.com/watch?v=nP-nMZpLM1A

[68] S. Dashevskyi, Y. Zhauniarovich, O. Gadyatskaya, A. Pilgun, and H. Ouhssain, "Dissecting Android cryptocurrency miners," in *Proc. 10th ACM Conf. Data Appl. Secur. Privacy*, ser. CODASPY '20, SIGSAC. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 191–202.

[69] Google, "Financial services: Play console help," 2022, accessed June 2022. [Online]. Available: https://support.google.com/googleplay/android-developer/answer/9876821?hl=en

[70] BBC News, "Google bans crypto-mining apps from Play Store," Jul. 2018, accessed in June 2022. [Online]. Available: https://www.bbc.com/news/technology-44980936

[71] S. Varlioglu, B. Gonen, M. Ozer, and M. Bastug, "Is cryptojacking dead after coinhive shutdown?" in *3rd Int. Conf. Inf. Comput. Techn.*, ser. ICICT '20, San Jose, CA, USA, Mar. 2020, pp. 385–389.

[72] F. Tommasi, C. Catalano, U. Corvaglia, and I. Taurino, "MinerAlert: An hybrid approach for web mining detection," *J. Comput. Virol. Hack. Techn.*, pp. 1–14, Mar. 2022.

[73] F. Naseem, A. Aris, L. Babun, E. Tekiner, and A. S. Uluagac, "MINOS: A lightweight real-time cryptojacking detection system," in *Netw. Distrib. Syst. Secur. Symp.*, ser. NDSS '21, no. 28. Virtual: Internet Society, Feb. 2021, pp. 244–259.

[74] M. Caprolu, S. Raponi, G. Oligeri, and R. Di Pietro, "Cryptomining makes noise: Detecting cryptojacking via machine learning," *Comput. Commun.*, vol. 171, pp. 126–139, Feb. 2021.

[75] S. Varlioglu, N. Elsayed, Z. ElSayed, and M. Ozer, "The dangerous combo: Fileless malware and cryptojacking," in *SoutheastCon 2022*. IEEE, Mar. 2022, pp. 125–132.

[76] H. Badih and Y. Alagrash, "Crypto-jacking threat detection based on blockchain framework and deception techniques," *Amer. J. Sci. Eng.*, vol. 2, no. 1, pp. 1–10, Jul. 2021.

[77] N. Lachtar, A. A. Elkhail, A. Bacha, and H. Malik, "An application agnostic defense against the dark arts of cryptojacking," in *51st Annu. Int. Conf. Dependable Syst. Netw.*, ser. DSN '21, IEEE/IFIP. Taipei, Taiwan: IEEE, Jun. 2021, pp. 314–325.

[78] J. Clay, A. Hargrave, and R. Sridhar, "A power analysis of cryptocurrency mining: A mobile device perspective," in *16th Annu. Conf. Privacy, Secur. Trust*, ser. PST '18. Belfast, Ireland: IEEE, Aug. 2018, pp. 1–5.

[79] P. Kotzias, J. Caballero, and L. Bilge, "How did that get in my phone? unwanted app distribution on Android devices," in *IEEE Symp. Secur. Privacy*, ser. SP '21. San Francisco, CA, USA: IEEE, May 2021, pp. 53–69.

[80] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting in-browser covert cryptocurrency mining in the wild," in *World Wide Web Conf.*, ser. WWW '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 840–852.

[81] M. Russo, N. Šrndić, and P. Laskov, "Detection of illicit cryptomining using network metadata," *EURASIP J. Inf. Secur.*, vol. 1, no. 11, pp. 1–20, Dec. 2021.

[82] M. Cao, K. Ahmed, and J. Rubin, "Rotten apples spoil the bunch: An anatomy of Google Play malware," in *Proc. 44th Int. Conf. Softw. Eng.*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 1919–1931.

[83] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Symp. Secur. Privacy*, ser. SP '12. San Francisco, CA, USA: IEEE, May 2012, pp. 95–109.

[84] H. Wang, J. Si, H. Li, and Y. Guo, "RmvDroid: Towards a reliable Android malware dataset with app metadata," in *IEEE/ACM 16th Int. Conf. Mining Softw. Repositories*, ser. MSR '19. Montreal, QC, Canada: IEEE, May 2019, pp. 404–408.

[85] D. Arp, Michael, Spreitzenbarth, M. Huebner, and H. G. K. Rieck, "DREBIN: effective and explainable detection of Android malware in your pocket," in *21th Annu. Netw. Distrib. Syst. Secur. Symp.*, ser. NDSS '14, San Diego, Carlifornia, USA, Feb. 2014. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/11_3_1.pdf

[86] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 468–471.

[87] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding Android app piggybacking: A systematic study of malicious code grafting," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 6, pp. 1269–1284, Jan. 2017.

[88] K. Khanmohammadi, N. Ebrahimi, A. Hamou-Lhadj, and R. Khoury, "Empirical study of android repackaged applications," *Empirical Softw. Eng.*, vol. 24, no. 6, pp. 3587–3629, Dec. 2019.

[89] M. A. Harris, R. Brookshire, and A. G. Chin, "Identifying factors influencing consumers' intent to install mobile applications," *Int. J. Inf. Manage.*, vol. 36, no. 3, pp. 441–450, Jun. 2016.

[90] S. Pastrana and G. Suarez-Tangil, "A first look at the crypto-mining malware ecosystem: A decade of unrestricted wealth," in *Proc. Internet Meas. Conf.*, ser. IMC '19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 73–86.

[91] A. Saif, H. AL-KILANI, M. Qasaimeh, and A. Al-Refai, "Analysis of Android applications permissions," in *Int. Conf. Data Sci., E-Learn. Inf. Syst.*, ser. DATA '21. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 243–249.

[92] A. Alshehri, P. Marcinek, A. Alzahrani, H. Alshahrani, and H. Fu, "Puredroid: Permission usage and risk estimation for android applications," in *Proc. 3rd Int. Conf. Inf. Syst. Data Mining*, ser. ICISDM '19. Houston, TX, USA: Association for Computing Machinery, Apr. 2019, pp. 179–184.

[93] E. Tekiner, A. Acar, A. S. Uluagac, E. Kirda, and A. A. Selcuk, "SoK: Cryptojacking malware," in *Eur. Symp. Secur. Privacy*, ser. EuroS P '21. Vienna, Austria: IEEE, Sep. 2021, pp. 120–139.

[94] Meta Platforms, Inc., "Facebook - Apps on Google Play," Aug. 2022, accessed on 2nd August 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.facebook.katana

[95] H. Liu, P. Patras, and D. J. Leith, "Android mobile OS snooping by Samsung, Xiaomi, Huawei and Realme handsets," techreport, Oct. 2021. [Online]. Available: https://www.scss.tcd.ie/doug.leith/Android_privacy_report.pdf

[96] S. Frier, "Samsung phone users perturbed to find they can't delete Facebook," Jan. 2019, accessed on 2nd August 2022. [Online]. Available: https://www.bloomberg.com/news/articles/2019-01-08/samsung-phone-users-get-a-shock-they-can-t-delete-facebook#xj4y7vzkg

[97] E. Tekiner, A. Acar, and A. S. Uluagac, "A lightweight IoT cryptojacking detection mechanism in heterogeneous smart home networks," in *Netw. Distrib. Syst. Secur. Symp.*, ser. NDSS '22, no. 29. San Diego, CA, USA: Internet Society, Apr. 2022, pp. 208–223.

[98] A. Mylonas, A. Kastania, and D. Gritzalis, "Delegate the smartphone user? security awareness in smartphone platforms," *Comput, Secur.*, vol. 34, pp. 47–66, May 2013.

[99] J. Chen, X. Xia, D. Lo, and J. Grundy, "Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–37, 2021.

[100] Y. Huang, Q. Kong, N. Jia, X. Chen, and Z. Zheng, "Recommending differentiated code to support smart contract update," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 260–270.

[101] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing code clones in the ethereum smart contract ecosystem," in *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*. Springer, 2020, pp. 654–675.

[102] G. A. Pierro, R. Tonelli, and M. Marchesi, "An organized repository of ethereum smart contracts' source codes and metrics," *Future internet*, vol. 12, no. 11, p. 197, 2020.

[103] N. Szabo, "Formalizing and securing relationships on public networks," *First monday*, 1997.

[104] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[105] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, pp. 2–1, 2014.

[106] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 394–397.

[107] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 96–105.

[108] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *2008 16th iEEE international conference on program comprehension*. IEEE, 2008, pp. 172–181.

[109] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *Vldb*, vol. 99, no. 6, 1999, pp. 518–529.

[110] Y. Wang, X. Chen, Y. Huang, H.-N. Zhu, J. Bian, and Z. Zheng, "An empirical study on real bug fixes from solidity smart contract projects," *Journal of Systems and Software*, vol. 204, p. 111787, 2023.

[111] J. Feist, G. Grieco, and A. 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)*. IEEE, 2019, pp. 8–15.

[112] *https://github.com/ConsenSys/mythril.*

[113] *https://github.com/nveloso/conkas.*

[114] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[115] M. Monperrus, M. Martinez, H. Ye, F. Madeiral, T. Durieux, and Z. Yu, "Megadiff: A dataset of 600k java source code changes categorized by diff size," *arXiv preprint arXiv:2108.04631*, 2021.

[116] H. Ye, "Improving the precision of automatic program repair with machine learning," Ph.D. dissertation, KTH Royal Institute of Technology, 2023.

[117] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.

[118] A. Pinna, S. Ibba, G. Baralla, R. Tonelli, and M. Marchesi, "A massive analysis of ethereum smart contracts empirical study and code metrics," *Ieee Access*, vol. 7, pp. 78 194–78 213, 2019.

[119] G. Yang, K. Liu, X. Chen, Y. Zhou, C. Yu, and H. Lin, "Ccgir: Information retrieval-based code comment generation method for smart contracts," *Knowledge-Based Systems*, vol. 237, p. 107858, 2022.

[120] S. Lubos, A. Felfernig, T. N. T. Tran, D. Garber, M. El Mansi, S. P. Erdeniz, and V.-M. Le, "Leveraging llms for the quality assurance of software requirements," in *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, 2024, pp. 389–397.

[121] Y. Wu, Y. Wan, Z. Chu, W. Zhao, Y. Liu, H. Zhang, X. Shi, and P. S. Yu, "Can large language models serve as evaluators for code summarization?" *arXiv preprint arXiv:2412.01333*, 2024.

[122] M. Li, J. Weng, A. Yang, J. Weng, and Y. Zhang, "Towards interpreting smart contract against contract fraud: A practical and automatic realization," *Cryptology ePrint Archive*, 2020.

[123] X. Li, T. Chen, X. Luo, T. Zhang, L. Yu, and Z. Xu, "Stan: Towards describing bytecodes of smart contract," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2020, pp. 273–284.

[124] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang, "A multi-modal transformer-based code summarization approach for smart contracts," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 1–12.

[125] X. Hu, Z. Gao, X. Xia, D. Lo, and X. Yang, "Automating user notice generation for smart contract functions," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 5–17.

[126] C. Shi, Y. Xiang, J. Yu, K. Sood, and L. Gao, "Machine translation-based fine-grained comments generation for solidity smart contracts," *Information and Software Technology*, vol. 153, p. 107065, 2023.

[127] Y. Mao, X. Li, W. Li, X. Wang, and L. Xie, "Scla: Automated smart contract summarization via llms and semantic augmentation," *arXiv preprint arXiv:2402.04863*, 2024.

[128] G. Lei, D. Zhang, J. Xiao, G. Fan, Y. Cao, and Z. Feng, "Fmcf: A fusing multiple code features approach based on transformer for solidity smart contracts source code summarization," *Applied Soft Computing*, vol. 166, p. 112238, 2024.

[129] J. Xiang, Z. Gao, L. Bao, X. Hu, J. Chen, and X. Xia, "Automating comment generation for smart contract from bytecode," *ACM Transactions on Software Engineering and Methodology*, 2024.

[130] Z. Zhang, S. Chen, G. Fan, G. Yang, and Z. Feng, "Ccgra: Smart contract code comment generation with retrieval-enhanced approach." in *SEKE*, 2023, pp. 212–217.

[131] Z. He, Z. Li, S. Yang, H. Ye, A. Qiao, X. Zhang, X. Luo, and T. Chen, "Large language models for blockchain security: A systematic literature review," *arXiv preprint arXiv:2403.14280*, 2024.

[132] M. Rauf, S. Padó, and M. Pradel, "Meta learning for code summarization," *arXiv preprint arXiv:2201.08310*, 2022.

[133] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[134] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[135] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

[136] C. Zhang, J. Wang, Q. Zhou, T. Xu, K. Tang, H. Gui, and F. Liu, "A survey of automatic source code summarization," *Symmetry*, vol. 14, no. 3, p. 471, 2022.

[137] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 595–46 623, 2023.

[138] Y. Liu, D. Iter, Y. Xu, S. Wang, R. Xu, and C. Zhu, "G-eval: Nlg evaluation using gpt-4 with better human alignment (2023)," *arXiv preprint arXiv:2303.16634*, vol. 12, 2023.

[139] H. Alhawasi and A. Youssef, "Using llms for evaluating qa systems: Exploration and assessment," in *2024 2nd International Conference on Foundation and Large Language Models (FLLM)*, 2024, pp. 462–469.

[140] A. Kumar, S. Haiduc, P. P. Das, and P. P. Chakrabarti, "Llms as evaluators: A novel approach to evaluate bug report summarization," *arXiv preprint arXiv:2409.00630*, 2024.

[141] X. Zhang, X. Hou, X. Qiao, and W. Song, "A review of automatic source code summarization," *Empirical Software Engineering*, vol. 29, no. 6, p. 162, 2024.

[142] M. T. R. Laskar, C. Chen, S. B. Tn *et al.*, "Are large language models reliable judges? a study on the factuality evaluation capabilities of llms," in *Proceedings of the Third Workshop on Natural Language Generation, Evaluation, and Metrics (GEM)*, 2023, pp. 310–316.

[143] K. Seßler, M. Fürstenberg, B. Bühler, and E. Kasneci, "Can ai grade your essays? a comparative analysis of large language models and teacher ratings in multidimensional essay scoring," ser. LAK '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 462–472. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/3706468.3706527

[144] A. Subramanian, Z. Yang, I. Azimi, and A. M. Rahmani, "Graph-augmented llms for personalized health insights: A case study in sleep analysis," in *2024 IEEE 20th International Conference on Body Sensor Networks (BSN)*, 2024, pp. 1–4.

[145] S. Yang, Z. Yuan, S. Li, R. Peng, K. Liu, and P. Yang, "Advancing agricultural decision-making with a multi-dimensional evaluation of large language models for sustainable pest management," in *2024 IEEE 22nd International Conference on Industrial Informatics (INDIN)*, 2024, pp. 1–6.

[146] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[147] P. E. Shrout and J. L. Fleiss, "Intraclass correlations: uses in assessing rater reliability." *Psychological bulletin*, vol. 86, no. 2, p. 420, 1979.

[148] K. O. McGraw and S. P. Wong, "Forming inferences about some intraclass correlation coefficients." *Psychological methods*, vol. 1, no. 1, p. 30, 1996.

[149] T. K. Koo and M. Y. Li, "A guideline of selecting and reporting intraclass correlation coefficients for reliability research," *Journal of chiropractic medicine*, vol. 15, no. 2, pp. 155–163, 2016.

[150] T. Gebru, J. Morgenstern, B. Vecchione, J. W. Vaughan, H. Wallach, H. D. III, and K. Crawford, "Datasheets for datasets," 2021. [Online]. Available: https://arxiv.org/abs/1803.09010

[151] S. Holland, A. Hosny, S. Newman, J. Joseph, and K. Chmielinski, "The dataset nutrition label: A framework to drive higher data quality standards," 2018. [Online]. Available: https://arxiv.org/abs/1805.03677

[152] X. Yang, W. Liang, and J. Zou, "Navigating dataset documentations in ai: A large-scale analysis of dataset cards on hugging face," 2024. [Online]. Available: https://arxiv.org/abs/2401.13822

[153] A. C. Roman, J. W. Vaughan, V. See, S. Ballard, J. Torres, C. Robinson, and J. M. L. Ferres, "Open datasheets: Machine-readable documentation for open datasets and responsible ai assessments," *arXiv preprint arXiv:2312.06153*, 2023.

[154] N. Jain, M. Akhtar, J. Giner-Miguelez, R. Shinde, J. Vanschoren, S. Vogler, S. Goswami, Y. Rao, T. Santos, L. Oala *et al.*, "A standardized machine-readable dataset documentation format for responsible ai," *arXiv preprint arXiv:2407.16883*, 2024.

[155] C. Geren, A. Board, G. G. Dagher, T. Andersen, and J. Zhuang, "Blockchain for large language model security and safety: A holistic survey," *ACM SIGKDD explorations newsletter*, vol. 26, no. 2, pp. 1–20, 2025.

[156] "Regulation (eu) 2024/1689 of the european parliament and of the council of 13 june 2024 laying down harmonised rules on artificial intelligence (artificial intelligence act) and amending certain union legislative acts," https://eur-lex .europa.eu/eli/reg/2024/1689/oj, 2024.

[157] S. Balloccu, P. Schmidtová, M. Lango, and O. Dusek, "Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source LLMs," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, Y. Graham and M. Purver, Eds. St. Julian's, Malta: Association for Computational Linguistics, Mar. 2024, pp. 67–93. [Online]. Available: https://aclanthology.org/2024.eacl-long.5/

[158] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv preprint arXiv:2105.12655*, 2021.

[159] GitHub and M. Research, "Codesearchnet — datasets, tools, and benchmarks," https://github.com/github/CodeSearchNet, 2019, accessed: 2025-09-27.

[160] OpenAI, "Dataset card for openai humaneval," https://huggingface.co/datasets/openai/openai_humaneval, 2021, accessed: 2025-09-27.

[161] CodeParrot, "Apps: Automated programming progress standard — dataset card," https://huggingface.co/datasets/codeparrot/apps, 2021, accessed: 2025-09-27.

[162] BigCode, "The stack v2 — dataset card," https://huggingface.co/datasets/bigcode/the-stack-v2, 2024, accessed: 2025-09-27.

[163] S. E. Hove and B. Anda, "Experiences from conducting semi-structured interviews in empirical software engineering research," in *11th IEEE International Software Metrics Symposium (METRICS'05)*. IEEE, 2005, pp. 10–pp.

[164] S. R. Howard, A. Ramdas, J. McAuliffe, and J. Sekhon, "Time-uniform, nonparametric, nonasymptotic confidence sequences," *Annals of Statistics*, 2021.

[165] J. Ville, *Etude Critique de la Notion de Collectif*, 1939.

[166] P. Grünwald, R. de Heide, and W. Koolen, "Safe testing," *JRSS B*, 2019.

[167] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and P. Rimba, "A taxonomy of blockchain-based systems for architecture design," in *2017 IEEE international conference on software architecture (ICSA)*. IEEE, 2017, pp. 243–252.

[168] L. Olivieri and F. Spoto, "Software verification challenges in the blockchain ecosystem," *International Journal on Software Tools for Technology Transfer*, vol. 26, no. 4, pp. 431–444, 2024.

[169] A. Paleyes, R.-G. Urma, and N. D. Lawrence, "Challenges in deploying machine learning: a survey of case studies," *ACM computing surveys*, vol. 55, no. 6, pp. 1–29, 2022.

[170] S. Kapoor and A. Narayanan, "Leakage and the reproducibility crisis in machine-learning-based science," *Patterns*, vol. 4, no. 9, 2023.

[171] M. S. Sodhi, Z. Seyedghorban, H. Tahernejad, and D. Samson, "Why emerging supply chain technologies initially disappoint: Blockchain, iot, and ai," *Production and Operations Management*, vol. 31, no. 6, pp. 2517–2537, 2022.

[172] F. N. MBODJI, A. L. NIANG, A. DIALLO, J. KLEIN, and T. BISSYANDE, "Affordance–experimentation–actualization-evolution for ai-based support of innovation," in *IEEE AFRICON 2025 Conference*. IEEE, Polokwane, South Africa, 2025.

12

Appendix

Table 12.1: Summary of TID Instances: Dimension Effects on Software and Software Engineering

TID Instance / Dimension(s)	**Software Context**	**Effect on Software**	**Effect on SE**	**Engineering Contribution**
Mining (Blockchain)	Android apps	- Unauthorized crypto mining - Resource consumption	- Detection complexity - Stealth operations	- Empirical study - Detection insights
Immutability (Blockchain)	Smart contracts	- Data integrity - Tamper-resistance	- Hard to update - Fragmented version history	- `ContractTrace` - Version reconstruction
Transparency (Blockchain) + Semantic Analysis (AI)	Smart contracts	- Auditability - Documentation availability	- Poor comprehension - Evaluator variability	- `evalSmart` - Documentation improvement
Reliability (AI)	AI evaluators	- Assessment outputs	- Unreliable evaluations - Trust in evaluation process	- Systematic evaluation - Guidelines for trustworthy use
Verifiability (Blockchain) + Dataset Evaluation (AI)	AI pipelines	- Immutable audit logs - Verified dataset quality	- Difficulty ensuring reproducibility - Documentation gaps	- `SIEVE` - Confidence Cards