Politecnico di Torino

Politecnico di Torino

Computer Engineering

A.a. 2024/2025

Graduation Session October 2025

Validation of Automatically Synthesized Smart Contract Invariants

A validation framework for the FLAMES tool

Supervisor:

Prof. Maurizio Morisio

Dr. Mojtaba Eshghie

Candidate:

Matteo Lauretano

"Attraverso la finestra sbarrata, vedo un quadrato di grano in un recinto, un cielo molto grande e un albero di gelso... E ogni giorno, il sole colora quel campo di sfumature diverse. Il mattino è dorato, il mezzogiorno è bianco, la sera diventa rame. Non posso uscire, ma questo piccolo angolo contiene già tutto un mondo."
– Vincent Van Gogh, Lettera a Theo

"Quando ti viene data la possibilità di scegliere, scegli sempre quello che ti fa tremare."
– Haruki Murakami, Kafka sulla spiaggia

Acknowledgements

Prima di tutto, mi sento in dovere di dire a te, caro lettore che stai per leggere questa tesi, che quest'ultima è stata uno dei lavori più impegnativi e allo stesso tempo soddisfacenti della mia vita. È la prima volta che produco qualcosa di cui mi sento veramente orgoglioso e su cui sento di aver messo tutto me stesso e spero che questo si percepisca man mano che si va avanti nella lettura. Inoltre, questi due anni passati prima a Torino e poi a Stoccolma sono stati i più caotici, intensi e pieni di esperienze della mia vita. Ho conosciuto molte persone, molte culture, molti usi e costumi che mi hanno arricchito profondamente, lasciando un'impronta indelebile che porterò per sempre con me.

Voglio ricordare, prima di iniziare la vera e propria sezione di ringraziamenti, la mia città, Avellino, che ho dovuto abbandonare per scoprire il mondo, ma che sarà sempre la mia amata terra, a cui *non vedo l'ora di ritornare* dopo lunghi periodi lontana da lei. Chi l'avrebbe detto che, un ragazzino timido e impacciato come me, sarebbe arrivato a questo punto, dopo aver girato prima l'Italia e poi l'Europa, completamente cambiato e pieno di amici sempre presenti, anche nei momenti più difficili? Mi sento molto fortunato ad avere tutto ciò, a priori non l'avrei mai immaginato e per questo mi sento profondamente grato ogni giorno della mia vita.

Detto questo, inizia la vera e propria sezione di ringraziamenti, quella specifica in cui ognuna delle persone che ha fatto parte della mia vita troverà un paragrafo apposta, dedicato a lei. Spero che questo sia un modo per ripagare almeno un minimo tutto ciò che hanno fatto per me, oltre ad un caffè al bar in Italia ovviamente. Piccola nota, a diffenza della tesi triennale questi ringraziamenti saranno molto personali, se vuoi caro lettore leggili pure ma tienili per te, non parlarne con me, mi imbarazzo facilmente.

Ringrazio il Prof. Maurizio Morisio per aver accettato di essere il mio relatore anche trovandomi dall'altra parte di Europa.

Ringrazio i miei genitori, insieme perchè non mi è possibile immaginarli separati. Grazie per avermi sempre dato retta, non aver mai mollato, per avermi sempre dato un abbraccio o una carezza quando ne avevo bisogno. Nei momenti di sconforto siete sempre stati il mio punto di riferimento, grazie per darmi sempre un esempio di cosa vuol dire amare e essere amati, nonostante tutto.

Ringrazio i miei fratelli, per esserci sempre quando ho bisogno di qualcosa, di qualsiasi cosa si tratti. Non sono il tipo da dirvelo in faccia, ma sono sempre stato grato di avermi come mio fratello e mia sorella, nonostante tutte le litogate e mazzate che ci siamo dati. Dopotutto è il nostro modo do esprimere il nostro affetto reciproco.

Ringrazio anche tutti i miei parenti, i miei zii e zie Eugenio, Antonietta, Anna, Gerardo, Ester, Antonio e i miei cugini e cugine Amato, Gerardina, Carmela e Antonio. Ci tengo a nominarli tutti perchè tutti nel momento del bisogno ci sono sempre stati, se ho potuto fare tutto quello che ho fatto in questi 24 anni di vita è anche grazie al loro supporto.

Voglio ringraziare anche mia zia Anna Maria, che purtroppo è venuta a mancare a causa del Covid, per avermi supportato sempre e avermi regalato il mio primo strumento musicale, una tastiera elettronica, dandomi la spinta per suonare il pianoforte e immergermi nel mondo della musica.

Ringrazio Donatella e Salvatore per avermi accolto nella loro casa questa estate, facendomi sentire parte della famiglia nella fase finale di revisione della tesi. Ringrazio la signora Caterina per avermi preparato tutto il cibo tipico possibile nel mio periodo di soggiorno in Sicilia, per avermi fatto sentire cosa vuol dire ricevere l'affetto di una nonna dopo tutto questo tempo.

Passiamo alla parte meno formale, i ringraziamenti agli *amici*. Il termine *amici* lo trovo riduttivo, perchè non sono solo questo, sono parte della mia vita, le persone con cui ho passato più tempo in assoluto, direi anche più della mia famiglia in questi cinque e più anni.

Ringrazio Luca, per esserci sempre stato come un fratello, per avermi sempre dato *fastidio*, dalla prima volta che ci siamo incontrati, passando dal trasferimento a Torino e l'Erasmus a Stoccolma, fino ad ora. I vocali di 15 minuti, le chiamate su Discord e le continue discussioni e litigi sono ormai una parte inamovibile della mia vita, chissà quando capirai che è più facile fare una chiamata piuttosto che recitare l'intero rosario in un vocale? Comunque sia, grazie veramente, mi hai sempre sostenuto, anche nei momenti in cui io stesso non ero capace di sostenere me stesso e mi sembrava non ci fosse via di uscita allo sconforto che provavo. Grazie per darmi sempre consiglio, sopratutto per avermi rassicurato sulla mia scelta di andare a Stoccolma quando io stesso ero incerto. Grazie per starmi sempre ad ascoltare, dalle *babbiate* ai miei sfoghi più pesanti. E niente, aspetto ancora un pranzo offerto, *cacc i sord pappooo*.

Ringrazio Sara, la persona più buona e cara che conosco. Una amica, una confidente e l'unica che mi è venuta a trovare nel freddo e nella neve di Stoccolma. Una delle poche persone capace di aiutarmi a sbrogliare i miei pensieri, a farmi sempre ridere e sorridere. Non so come avrei fatto senza di te nel periodo più brutto di questi ultimi anni, sei stata sempre pronta a darmi una parola di conforto e a farmi sentire meglio, nonostante la distanza. Grazie per essere sempre pronta

ad *inciuciare*, una delle mie attività preferite. Grazie per essere sempre capace di arricchire ogni nostra discussione, che sia seria o leggera. Grazie per essere semplicemente così come sei, solare, come il quadro *Albero di pesco in fiore* di Van Gogh. Per concludere, *ma che ci vuoi fare, se puzzi di pesce.*

Ringrazio *mbare* Gabriele per essere stato come un fratello durante la nostra convivenza in Collegio. Lo ringrazio per avermi aperto la porta della 417 ogni volta, a qualsiasi orario e per qualsiasi motivo io bussassi. Grazie per avermi aiutato ad uscire dalla mio essere introverso e ad aver iniziato a condividere ogni momento della quotidianità con me, dalla colazione alla cena, dallo studio allo svago, dalle cose più importanti a quelle non necessarie, ma si sa, non ci sono segreti tra fratelli. Nonostante i problemi che ci sono stati, nonostante le incomprensioni, nonostante tutto, non posso non essere grato per tutto quello che abbiamo fatto insieme. Ma una cosa ti devo ricordare per le prossime volte, *quando parlo io, muto ti devi stare e devi ascoltare.*

Ringrazio tutta la compagnia di Torino, Tonino, il Dottore, Michele, Manuel, Raffaele e Alice, ormai campana acquisita, per avermi fatto sentire a mio agio anche a Torino, così distante sia geograficamente che culturalmente dalla nostra Campania. Grazie per le serate passate insieme, in giro alla scoperta di Torino oppure a Via Vernazza o Via Osasco a vedere le partite dell'Italia. Ringrazio in particolare Chiara per essere stata la prima con me a imbaccarsi in questo viaggio verso l'ignoto, senza conoscere nulla del Politecnico o di Torino. Non ho altro da dire se non *lavali, lavali, lavali col fuoco.*

Ringrazio tutto il quarto piano del Collegio Einaudi e i miei *coinquilini*, Kekko, Andrea, Giulia e Giorgia, che mi hanno accolto in questo ambiente multiculturale e così nuovo per me, facendomi sempre sentire a mio agio. E' stato incredibile scoprire come basti passare da una regione all'altra per incontrare culture completamente diverse dalla mia. Grazie a tutti i momenti di quotidianità, dalla pausa studio alla *tisanina* prima di andare a letto. Grazie per tutte le serate passate insieme a giocare a giochi da tavolo, a guardare film o a girare senza meta a Torino, tra un gelato o una birra. E ricordatevi, *sono di Avellino, non di Napoli.*

Ringrazio Raffaele, mio amico dalle superiori, che mi conosce da più tempo di tutti. 10 anni non sono pochi, la metà di quelli serviti a finire la *galleria sotto il Corso*. Grazie per farmi sentire sempre, ogni volta che torno ad Avellino, come se non fossi mai andato via, come se non ci fossero mesi e chilometri a dividerci. Alla fine, non è cambiato nulla dalle superiori, la nostra amicizia rimane sempre la stessa e spero rimanga sempre così. Ma una domanda importante, *sto stadio quando lo fanno?*

Ringrazio anche i miei amici rimasti purtroppo *ingiustamente carcerati* a Salerno, Pietro e Manuel. Nonostante la distanza, sono sempre raggiungibili e sempre presenti, come se non fosse cambiato niente dalla triennale. Rimane un interrogativo, *ma il numero 17 da dove è uscito?*

Ho deciso di aggiungere una sezione un pò particolare in questi ringraziamenti, dedicata a quelle cose un pò più *banali* che però voglio comunque menzionare, perchè anche le cose più semplici possono dare la spinta per non fermarsi e continuare ad andare avanti.

Ringrazio la musica per avermi sempre accompagnato in tutti i miei viaggi, dal bus per arrivare all'università ai viaggi *della speranza* per tornare da Stoccolma a casa. In particolare voglio ringraziare i miei artisti preferiti, Nitro e Madman, che mi hanno accompagnato dal primo anno delle superiori fino ad ora.

Ringrazio il mio artista preferito, Vincent Van Gogh, per avermi ispirato ed emozionato con tutte le sue opere. La visita al Vincent Van Gogh Museum di Amsterdam rimarrà sempre nella mia memoria.

Ringrazio gli innumerevoli libri che ho letto, da *Uno, Nessuno e Centomila* a *It* fino a *Norwegian Wood*, che mi hanno fatto riflettere su me stesso e viaggiare con la testa. In particolare, ringrazio il giorno in cui ho deciso di chiedere a mia madre la collana intera di *Harry Potter*, che ha dato il via alla mia passione per la letteratura.

Ringrazio il mio scrittore italiano preferito, Luigi Pirandello, per aver cambiato il mio modo di vedere il mondo con le sue opere.

Ringrazio il mio scrittore preferito, Stephen King, per avermi fatto innamorare della scrittura, la quale è capace di esprimere svariati concetti con precisione assoluta.

Ringrazio lo scrittore Haruki Murakami, per avermi accompagnato con le sue opere in questo anno in Svezia, dandomi una nuova spinta nella lettura.

Ringrazio il mio mangaka preferito, Shin'ichi Sakamoto, per avermi sempre affascinato con il suo stile di disegno e intrattenuto con i suoi complessi e crudi racconti.

Ringrazio il mangaka Eiichiro Oda, che con la sua opera maestra One Piece mi ha introdotto all'universo dei manga e degli anime, rendendo questo mondo uno dei miei interessi principali tuttora.

Ringrazio Torino, che mi ha accolto nella mia prima esperienza da fuorisede e mi ha fatto sentire a casa.

Ringrazio Stoccolma, che nonostante tutte le difficoltà mi ha accolto e si è dimostrata una città straordinaria, piena di natura e cultura.

Ringrazio, anche se con dispiacere, Genshin Impact e Rocket League, per avermi accompagnato in tutta la mia carriera accademica, concedendomi sempre uno stacco dallo studio.

Ringrazio Expedition 33, che mi ha accompagnato nella stesura della tesi e nelle sue mille revisioni.

Ringrazio anche tutti i content creators che sono stati il background dei miei studi, Yotobi, Sabaku, Cydonia e molti altri, che mi hanno tenuto compagnia in questi momenti solo con me stesso.

Ringrazio Rami Kebab, che a Torino mi ha sfamato innumerevoli volte, in sessione o meno.

Ringrazio la catena di fast food Max, che in Svezia mi ha sfamato più di una volta.

Un'altra sezione che tengo ad aggiungere contiene i ringraziamenti presenti nella mia tesi del KTH, dedicati alle persone che ho conosciuto lì e quindi scritta in inglese.

I would like to thank my supervisor, Mojtaba Eshghie, and my examiner, Cyrille Artho, for their invaluable guidance and advice throughout the writing of this thesis. A special thanks to Mojtaba for his patience in answering all my questions throughout the process and for the continuous support he provided.

I also want to thank the Apt. 51 for making me feel like I had a family here, even though I was far from home. In particular, thank you to Ronya for being my confidante during this time, like a sister to me.

Last but not least, I want to thank my friend Sara for making me feel like I was in Italy, even while in Sweden.

Stockholm, August 2025

Matteo Lauretano

E per finire, prima di iniziare con la vera e propria tesi, aggiungo un QR code che se desideri, caro lettore, puoi scansionare e lasciare in esecuzione mentre leggi la tesi. In questo modo potrai ascoltare ciò che ho ascoltato io durante la scrittura di questa tesi, per vivere in minima parte le stesse sensazioni che io ho provato nella sua stesura.

E grazie anche a te, caro lettore, per aver letto questi ringraziamenti così prolissi e buona lettura, spero ti piaccia questa mia opera.

Abstract

Smart contracts are immutable programs deployed on blockchain platforms to enable decentralized applications and financial systems without intermediaries. However, their immutability and public execution make them especially vulnerable to security flaws, which can lead to irreversible significant financial losses.

A promising approach to improve security is the use of invariants, properties that must always hold during contract execution. FLAMES (Fine-tuned Large Language Model for Invariant Synthesis) is a tool that leverages fine-tuned large language models to automatically generate security-relevant invariants for Solidity smart contracts. Automatically generating and validating such invariants remains a key challenge. This thesis evaluates FLAMES ability to synthesize and validate security-relevant invariants for Solidity smart contracts. The goal is to determine whether these invariants are both syntactically correct and semantically effective. To this end, two automated evaluation pipelines were implemented. The first tests whether smart contracts with injected invariants still compile successfully. The second examines whether these invariants prevent real exploits while preserving benign functionality.

Experimental results show that FLAMES20K and FLAMES100K achieved high compilability rates and successfully patched known vulnerabilities. This work bridges the gap between theoretical invariant generation and practical smart contract security. By automating the validation of synthesized invariants, the proposed framework supports the development of more secure and reliable blockchain applications.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Smart contracts are self-executing programs that enforce the terms of digital agreements once predefined conditions are met. Originally conceptualized by Nick Szabo in the 1990s [1], they were envisioned as tools for minimizing trust and automating contract enforcement through precise code. In this way, they reduce reliance on intermediaries and minimize the risk of human mistakes in processes [2].

With the emergence of blockchain platforms like Ethereum [3], Szabo's vision became technically feasible. Today, smart contracts underpin decentralized applications (dApps) such as decentralized finance (DeFi), token standards, and on-chain governance. Their deployment on the blockchain ensures immutability and transparency; once published, the code cannot be altered, and its logic executes publicly and deterministically. While this fosters trustlessness, it also introduces significant engineering and security challenges. Smart contracts are often written in Solidity, where small bugs can lead to severe exploits which has been observed before [4].

The high-value nature of smart contracts and their open execution environment make them prime targets for attacks. Incidents such as the DAO hack in 2016 [5] and multiple DeFi breaches have resulted in irreversible losses totaling millions of dollars [6, 7, 8].

As decentralized applications continue to evolve and scale, ensuring the correctness and robustness of smart contracts becomes increasingly critical. These programs are no longer isolated scripts but foundational components of a global financial and computational infrastructure.

A common example is a crowdfunding platform. Instead of relying on an intermediary to manage the funds, a smart contract can be programmed to automatically collect contributions from backers. The rules are encoded in advance: if the campaign reaches its funding goal within the specified deadline, the smart contract releases the funds directly to the project creator; otherwise, it refunds the contributions to the backers.

1.1 Problem Statement

Smart contracts, once deployed to a blockchain network, are immutable and operate autonomously. This immutability, while beneficial for transparency and trust, also means that any vulnerabilities in the contract code become permanent and exploitable. Even minor logic errors can have catastrophic financial consequences [9, 10, 6], and the adversarial nature of public blockchains exacerbates the risk.

To address these risks, one promising direction involves the use of invariants [11]. They are properties of the contract state that must always hold true [12, 13]. For example, an invariant might specify that the total balance in a vault must never decrease outside of an authorized withdrawal function. If such invariants are enforced at runtime by the injection of `require/assert` statements directly into the contract's Solidity code, malicious transactions that violate them can be reverted automatically, neutralizing potential exploits.

Recent research has explored automated invariant synthesis, particularly through dynamic analysis techniques that extract candidate invariants from historical transaction data [14, 15]. While such methods are effective in capturing behavioral patterns, they often lack semantic rigor and generalizability, limiting their applicability to unseen or adversarial scenarios [11, 16].

Moreover, empirical studies [17] highlight recurring classes of vulnerabilities in deployed contracts and suggest that ad hoc fixes or manual auditing are insufficient. These findings underscore the need for automated, scalable mechanisms that can generate, verify, and integrate security-critical invariants directly into smart contract code.

Despite ongoing progress, the literature lacks a comprehensive and empirically validated framework for the end-to-end use of invariants in real-world smart contracts. This thesis addresses this gap by proposing a framework for the validation of automatically-synthesized invariants in Solidity smart contracts.

1.2 Purpose and Goals

The purpose of this thesis is to validate automatically synthesized invariants and develop a comprehensive automated evaluation framework for assessing the FLAMES tool. FLAMES is a tool that utilizes fine-tuned large language models to automatically produce security invariants for Solidity smart contracts. By analyzing contract code and predicting necessary safety checks, FLAMES assists in preventing common vulnerabilities such as overflows, unauthorized transfers, or state inconsistencies. This evaluation framework is specifically designed to test whether the generated invariants are both syntactically correct and semantically effective when applied to real-world vulnerable contracts.

This research aims to bridge the gap between theoretical vulnerability detection and practical application in deployed smart contracts. The work provides empirical evidence of FLAMES's ability to generate invariants that compile successfully and mitigate known security issues, offering a systematic approach to improving smart contract reliability.

1.3 Research Questions

RQ1–Compilability To what extent do invariants synthesised by FLAMES preserve the ability of existing smart contracts to compile?

RQ2–Security Do FLAMES invariants prevent known exploits *without* altering benign behaviour?

1.4 Research Methodology

This thesis adopted a positivist philosophical stance, emphasizing objective measurement and empirical observation as the foundation for knowledge generation [18]. The research employed an applied, iterative methodology [18], where automated evaluation pipelines were systematically developed, tested, and refined through multiple cycles. Two distinct automated pipelines were developed. Each pipeline was designed to address a specific research question:

RQ1 Pipeline: focuses on evaluating the **syntactic correctness** of generated invariants. It verifies that the generated invariants can be successfully compiled and integrated into existing smart contracts without introducing compilation errors.

RQ2 Pipeline: assesses the **semantic effectiveness** of the generated invariants. It measures their practical utility in patching known vulnerabilities and evaluates whether the invariants change the contract's behavior.

The overall reasoning process was inductive. Theoretical insights and generalizations emerged from patterns identified in the empirical data generated by the automated pipelines [18].

1.5 Structure of the thesis

Chapter 2 provides the theoretical foundations relevant to the topics addressed in this thesis. Chapter 3 offers a comprehensive review of the existing literature. The experimental methodology and rationale behind key decisions are detailed

in Chapter 4. Chapter 5 shows the experimental findings accompanied by a thorough analysis. Chapter 6 presents a discussion of the limitations and the multi-dimensional trade-off between robustness, accuracy, and the human effort required to review the output. Finally, Chapter 7 summarizes the work, suggests directions for future research, and offers reflections on the impact of this study.

Chapter 2

Background

This chapter provides the knowledge necessary to understand the research presented in this thesis. We begin by examining two key resources: FLAMES [19], a fine-tuned large language model approach for invariant synthesis described in Section 2.1, and DISL [20], a dataset for smart contract analysis presented in Section 2.2. Subsequently, we explore in Section 2.3 the most common smart contract vulnerabilities that threaten blockchain applications, demonstrating how automated security measures such as require statement injection can mitigate these risks.

2.1 FLAMES: An AI Model for Defensive Code Synthesis

FLAMES [19] presents an approach to enhancing smart contract security through automated invariant generation. The framework leverages CodeLlama [21] models fine-tuned on smart contract code to automatically produce invariants for Solidity smart contracts. It employs three model variants: a smaller fine-tuned model trained on 20,000 smart contracts, a larger fine-tuned model trained on 100,000 smart contracts, and a baseline model using the original CodeLlama.

FLAMES incorporates a Fill-in-the-Middle (FIM) token infilling technique [22]. This technique enables the model to predict missing security checks with contextual relevance. The framework uses Abstract Syntax Tree (AST) analysis [23] to identify strategic points for inserting security checks, placing `<FILL_ME>` tokens accordingly. This injection ensures that the model learns to generate contextually appropriate security invariants at syntactically and semantically correct locations in the smart contract code.

An example is a Solidity function that transfers tokens between user accounts. A potential vulnerability arises if the sender's balance is insufficient for the transfer, which could result in underflow or failed transactions. Using FLAMES, an

invariant such as `require(balance[sender] >= amount, "Insufficient balance");` can be automatically generated and inserted at the relevant location. This ensures that the function enforces the necessary safety condition while preserving its intended functionality.

2.2 The DISL Dataset for Training AI Models

The DISL [20] (Dataset for Intelligent Smart Contract anaLysis) dataset serves as a resource for smart contract research and machine learning applications. The DISL dataset presents a collection of 514,506 unique Solidity files that have been deployed to Ethereum main net. This makes it one of the largest of real-world smart contracts available for research purposes.

DISL includes only the verified source code of smart contracts deployed on Ethereum, ensuring that the dataset comprises real, in-use contracts. Moreover, its focus on uniqueness of the included smart contracts, ensures a diverse dataset.

2.3 Smart Contract Vulnerabilities

Smart contracts face several security challenges due to their immutable nature and the high-stakes financial applications they often support.

2.3.1 Reentrancy Attacks

Reentrancy vulnerabilities occur when external calls to untrusted contracts can recursively invoke the calling contract before the first invocation completes. This can lead to unexpected state changes and allow attackers to drain funds. The most notorious example is the DAO attack of 2016 [5].

A simple `require` statement ensuring state updates are performed before external calls can prevent many reentrancy attacks. For instance, consider the safe withdrawal pattern in Figure 2.1.

2.3.2 Access Control Vulnerabilities

Access control issues arise when functions lack proper permission checks, allowing unauthorized users to execute privileged operations. These vulnerabilities can be prevented by implementing role-based access control with appropriate `require` statements, as illustrated in Figure 2.2.

```solidity
contract SecureWithdraw {
    mapping(address => uint256) public balances;

    function withdraw(uint256 amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");

        // Update state before external call
        balances[msg.sender] -= amount;

        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "External call failed");
    }

    receive() external payable {
        balances[msg.sender] += msg.value;
    }
}
```

Figure 2.1: SecureWithdraw contract using checks-effects-interactions to prevent reentrancy.

```solidity
contract AdminControl {
    address public owner;
    mapping(address => bool) public admins;

    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner can perform this action");
        _;
    }

    modifier onlyAdmin() {
        require(admins[msg.sender], "Insufficient privileges");
        _;
    }

    function addAdmin(address _admin) public onlyOwner {
        admins[_admin] = true;
    }

    function sensitiveFunction() public onlyAdmin {
        // privileged logic
    }
}
```

Figure 2.2: Example of role-based access control with owner and admin checks.

2.3.3 Arithmetic Vulnerabilities

Integer overflow and underflow vulnerabilities can cause unexpected behavior in smart contracts. While Solidity 0.8+ includes built-in overflow protection[24], older contracts and unchecked blocks remain vulnerable. Manual checks are shown in

Figure 2.3.

```
contract ArithmeticSafe {
    function safeAdd(uint256 a, uint256 b) public pure returns (uint256) {
    require(a + b >= a, "Addition overflow");
    return a + b;
    }

    function safeSub(uint256 a, uint256 b) public pure returns (uint256) {
        require(a >= b, "Subtraction underflow");
        return a - b;
    }
}
```

Figure 2.3: Manual overflow and underflow checks in arithmetic operations.

2.3.4 Front-Running Attacks

Front-running occurs when attackers observe pending transactions and submit their own transactions with higher gas prices to be executed first. This is particularly problematic in DeFi applications. Commit-reveal schemes help mitigate such risks, as shown in Figure 2.4.

```
contract CommitReveal {
    mapping(address => bytes32) public commits;
    uint256 public revealPhase;

    function commit(bytes32 hash) public {
        commits[msg.sender] = hash;
    }

    function reveal(uint256 value, bytes32 salt) public {
        require(block.timestamp >= revealPhase, "Still in commit phase");
        require(commits[msg.sender] != 0, "No commit found");
        require(keccak256(abi.encodePacked(value, salt)) == commits[msg.sender], "
    Invalid reveal");
        // process value
    }
}
```

Figure 2.4: Commit-reveal scheme to mitigate front-running.

2.3.5 Unchecked Low-Level Calls

Low-level calls such as `call`, `delegatecall`, and `staticcall` can fail silently if their return values are not checked. Always ensure their success explicitly, as shown

in Figure 2.5.

```
contract SafeCall {
    function callExternal(address target, bytes memory data) public {
        (bool success, ) = target.call(data);
        require(success, "Low-level call failed");
    }
}
```

Figure 2.5: Explicit return value check on low-level `call`.

2.3.6 Bad Randomness

Smart contracts cannot generate truly random numbers due to the deterministic nature of the blockchain. Predictable sources like block hashes can be exploited. A safer approach involves external oracles, as in Figure 2.6.

```
interface RandomnessOracle {
    function isValid() external view returns (bool);
    function getRandom() external view returns (uint256);
}

contract RandomSafe {
    RandomnessOracle public oracle;
    uint256 public lastRandomUpdate;
    uint256 constant MIN_DELAY = 1 hours;

    function updateRandom() public {
        require(block.timestamp > lastRandomUpdate + MIN_DELAY, "Randomness too
    recent");
        require(oracle.isValid(), "Invalid randomness source");

        uint256 random = oracle.getRandom();
        lastRandomUpdate = block.timestamp;

        // use random
    }
}
```

Figure 2.6: Randomness via external oracle with freshness and validity checks.

2.3.7 Denial of Service

Denial of Service (DoS) attacks in smart contracts aim to make a function or the entire contract unusable. These issues commonly arise from patterns such

as unbounded loops, reliance on external contract calls, or blocking operations. To prevent DoS vulnerabilities, developers should avoid writing logic that can be indefinitely blocked or fail due to untrusted interactions. Figure 2.7 demonstrates a safer approach by using pull-over-push patterns to mitigate such risks.

```
contract SecureWithdrawal {
    mapping(address => uint256) public balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() public {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "Nothing to withdraw");

        balances[msg.sender] = 0;

        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Failed to send Ether");
    }
}
```

Figure 2.7: Example of secure withdrawal pattern to avoid DoS via external calls.

Chapter 3

Related Works

Blockchains enable decentralized and trustless applications by removing the need for central authorities and enabling verifiable, tamper-resistant execution of transactions [2]. Smart contracts bring this vision to life by providing programmable logic that runs on the blockchain. They enforce rules and automate transactions in a secure and transparent manner. They have become the core enabler of decentralized finance (DeFi), non-fungible tokens (NFTs), and decentralized autonomous organizations (DAOs). As their complexity and adoption continue to grow, traditional manual development approaches face increasing scalability and security challenges. This has led to a growing interest in automated synthesis techniques capable of generating correct and secure smart contract code at scale.

The rise of large language models (LLMs) and advanced program synthesis techniques has opened new possibilities for automated smart contract generation. However, the immutable nature of deployed contracts and their direct control over valuable digital assets make the validation of synthesized code not merely important, but absolutely critical. Bugs and vulnerabilities are both harder to patch due to the immutability of the code section of smart contracts and also have a significant financial impact. While patching is possible using upgradability/proxy patterns [25, 26], this approach has limitations: once an attack transaction is confirmed on the blockchain, it cannot be reverted. As a result, vulnerabilities can lead to the irreversible loss of assets and funds, as demonstrated in past attacks [6]. This makes rigorous validation an indispensable part of any automated code synthesis workflows. A recent systematization of real-world incidents [27] reinforces this need by showing that many high-impact exploits are not caused solely by low-level bugs, but by combinations of vulnerabilities, many of which could be prevented using proper checks in smart contract functions.

This literature review provides an examination of current approaches, tools, and methodologies for validating automatically synthesized smart contract code.

3.1 Validation Techniques for Generated Code

The validation of automatically generated smart contracts presents unique challenges. Unlike manually written code, synthesized contracts may show unexpected patterns, novel vulnerability combinations, or subtle logical inconsistencies that emerge from the generation process itself. The following subsections examine the formal, dynamic, adversarial, and empirical methodologies that form the foundation of robust validation for generated smart contract code.

3.1.1 Formal Verification Approaches

Formal verification provides the strongest guarantees for smart contract correctness, making it particularly valuable for validating synthesized code where the development process may introduce subtle errors.

The theoretical foundations of smart contract verification trace back to classical symbolic execution techniques. King's seminal work [28] established the fundamental principles of symbolic execution, where program inputs are represented as symbolic values rather than concrete data, enabling systematic exploration of execution paths.

Building upon these foundations, Baldoni et al. [29] provide a survey of symbolic execution techniques, highlighting key challenges such as path explosion, constraint solving complexity, and the trade-offs between precision and scalability. These challenges are particularly relevant in the smart contract domain, where contracts may have complex state spaces and intricate interaction patterns.

Formal verification tools can be broadly categorized into symbolic execution-based analyzers, such as SolSEE [30] and SymGPT [31], and model checking approaches, such as ESBMC-Solidity [32].

SolSEE [30] performs symbolic execution on Solidity source code rather than bytecode, preserving high-level semantic information for debugging and interactive analysis.

SymGPT [31] integrates large language models with symbolic execution for smart contract auditing. This tool shows how AI can improve formal verification by generating property specifications and invariants, which are then verified using traditional symbolic methods..

ESBMC-Solidity [32] extends satisfiability modulo theories (SMT) based model checking to Solidity contracts, providing bounded model checking capabilities that can verify safety properties and detect common vulnerabilities such as integer overflows and reentrancy attacks.

3.1.2 Dynamic Analysis and Testing Methodologies

Testing approaches for synthesized smart contracts must address both functional correctness and security properties. Traditional testing methodologies require adaptation to handle the unique characteristics of generated code, including potentially complex interaction patterns and new vulnerability vectors.

Dynamic testing approaches for synthesized contracts span a spectrum from machine learning-guided fuzzing, exemplified by SmarTest [33], to temporal logic-based validation, as implemented in tools like SmartPulse [34] and T2 [35].

SmarTest [33] introduces a new approach by combining symbolic execution with language model guidance to find vulnerable transaction sequences. The tool addresses the path explosion problem by using machine learning to prioritize exploration of potentially vulnerable execution paths, demonstrating superior effectiveness in discovering complex multi-transaction vulnerabilities. This aligns with the empirical findings of [27], which show that multi-step exploit chains are responsible for a large portion of real-world smart contract losses. It highlightes the importance of testing methodologies that go beyond single-transaction execution traces.

SmartPulse [34] provides automated checking of temporal properties in smart contracts, utilizing temporal logic frameworks to reason about contract lifecycles and state transitions.

T2 [35] is specialized in temporal property verification, offering automatic generation of proofs for liveness and safety properties. Its integration with the LLVM framework enables analysis of contracts compiled from various high-level languages, broadening its applicability to different synthesis approaches.

A mention to formal temporal reasoning is provided by the work of Sergey et al. [36]. This work describes a treatment of temporal properties in smart contracts, demonstrating how traditional temporal logic can be adapted to reason about blockchain-specific concepts such as block ordering, transaction atomicity, and cross-contract interactions. Their approach using Coq for mechanized verification establishes important foundations for formal reasoning about contract temporality.

3.1.3 Threat Modeling and Adversarial Analysis

Security analysis for synthesized smart contracts requires specialized approaches that can handle both traditional vulnerability patterns and novel security issues that may arise from the synthesis process. The automated nature of code generation may introduce unexpected interaction patterns or subtle logical errors that traditional security analysis tools might miss.

Threat modeling and adversarial analysis tools for synthesized contracts can be categorized based on their core capability: from formal temporal safety verification,

as implemented in VerX [37], to benchmarking and multi-tool evaluation frameworks, such as SmartBugs 2.0 [38], and to adversarial exploit generation guided by formal specifications and language models, as proposed in XploGen [39].

VerX [37] provides safety verification capabilities with particular strength in temporal specification checking, enabling detection of complex vulnerability patterns that span multiple transactions.

SmartBugs 2.0 [38] offers a standardized execution framework for weakness detection, providing benchmarking capabilities that are crucial for evaluating the effectiveness of different analysis approaches on synthesized code. The framework's ability to compare multiple analysis tools systematically shows the strengths and limitations of different validation approaches when applied to automatically generated contracts.

The work by Eshghie and Artho [39] presents XploGen which is a methodology to generate benchmarks for smart contract analysis tools. Their approach uses formal specifications combined with LLMs to synthesize valid exploits. Their methodology achieved a 57% success rate in exploiting targeted contract aspects with an average of 3.5 transactions per exploit, demonstrating the efficiency of guided exploit generation. While such tools focus on exploit generation and validation, recent work has shown that many real-world attacks do not stem from isolated code-level bugs alone. Rezaei et al. [27] systematically analyzed 50 major incidents and revealed that exploit chains often arise from intertwined flaws across protocol logic, governance, external dependencies, and implementation. Their four-tier root-cause framework highlights the importance of going beyond static bug detection and considering systemic and operational factors when validating synthesized contracts.

3.1.4 Performance Evaluation and Quality Metrics

Empirical evaluation frameworks provide crucial infrastructure for comparing different synthesis and validation approaches. The development of complete benchmarks specifically designed for synthesized smart contracts is essential for advancing the field and ensuring reproducible research results.

Recent work by Bobadilla et al. using SB-Heist [40] examines whether automated fixes truly mitigate smart contract exploits, providing insights into the effectiveness of current vulnerability remediation approaches for synthesized code. This research highlights the gap between detecting vulnerabilities and effectively addressing them in automatically generated contracts, revealing that many automated fixes may not provide the security guarantees they claim to offer. The establishment of standardized evaluation metrics and benchmark datasets enables systematic comparison of different validation approaches.

3.2 Cutting-Edge Synthesis-Validation Integration Systems

The integration of synthesis and validation processes represents a paradigm shift from traditional sequential development models where verification occurs after code completion. Modern approaches increasingly recognize that effective validation of automatically generated smart contracts requires tight coupling between the generation and verification processes, creating feedback loops that improve both synthesis quality and validation coverage. This integration enables real-time constraint checking, specification-guided generation, and iterative refinement that would be impossible with purely post-hoc validation approaches.

3.2.1 Solidity-Centric Development Trends

Given Ethereum's dominant position in the smart contract ecosystem, Solidity has become the primary target for automated synthesis research. The language's complexity, with features such as inheritance, modifiers, events, and low-level assembly integration, presents significant challenges for automated generation and subsequent validation.

As a benchmarking framework, SolEval [41] provides infrastructure for comparing different synthesis approaches and their associated validation techniques. The framework gives a systematic evaluation of LLM-generated smart contracts by providing metrics and evaluation protocols. These metrics include standardized measures such as *pass@k*, the proportion of tasks for which at least one of the top-k generated candidates passes all test cases, and *compile@k*, the proportion of generations that result in syntactically correct, compilable code, along with gas usage estimation and vulnerability detection via tools like Slither.

3.2.2 Specification Mining and Property Synthesis

Specification mining and property synthesis approaches for smart contracts can be categorized based on their core methodology: from retrieval-augmented LLM-based property generation, as in PropertyGPT [42], to automated invariant inference for Solidity, as demonstrated by Liu et al. [14], to multimodal learning for invariant mining, as in SmartInv [43], and domain-adapted LLM-based synthesis, as implemented in FLAMES [19].

PropertyGPT [42] introduces techniques for retrieval-augmented property generation, using LLMs to generate formal specifications from natural language descriptions and code analysis. The retrieval-augmented approach ensures that generated properties are grounded in existing knowledge and best practices.

Invariant generation has emerged as a critical component of smart contract verification, as invariants capture essential correctness properties that must hold throughout contract execution. Liu et al. [14] present automated techniques for generating invariants specific to Solidity contracts, addressing challenges such as handling dynamic arrays, mappings, and complex state relationships.

SmartInv [43] employs multimodal learning to infer smart contract invariants, combining code analysis with natural language processing to understand contract documentation and comments.

FLAMES [19] uses fine-tuned large language models for invariant synthesis, showing how domain-specific training can improve the quality and relevance of generated invariants for smart contract verification. The fine-tuning approach generates meaningful invariants for synthesized contracts by incorporating domain-specific knowledge.

3.2.3 Code Generation with Integrated Verification

FSM-SCG [44] shows how finite state machines can guide LLM-based smart contract generation with validation. The approach uses formal state models to structure the synthesis process, making the resulting contracts more amenable to formal verification by ensuring that the generated code follows predictable state transition patterns. The iterative validation component of FSM-SCG allows continuous refinement based on feedback.

Complementing this, VeriSolid [45] presents a correct-by-design framework for smart contract development based on formal state machine models. Developers define contracts using a high-level FSM specification with clear operational semantics. This specification is then subjected to formal verification to ensure properties such as deadlock-freedom and state reachability. Once verified, the specification is automatically compiled into Solidity code, preserving the correctness guarantees.

3.2.4 End-to-End Smart Contract Security

Security in smart contracts can be addressed at different stages of their lifecycle.

CodeBC [46] presents a security-aware approach to LLM-based smart contract generation, incorporating security considerations directly into the synthesis process through a three-stage fine-tuning method. Instead of relying on annotated vulnerability datasets or formal specifications, CodeBC leverages lightweight vulnerability and security tags to differentiate between secure and insecure code during training.

In addition to generation-time security, Solythesis [47] presents how contracts can be instrumented to enforce invariants during execution, showing that runtime validation overhead is negligible in the blockchain context. The system operates as a source-to-source Solidity compiler that takes a smart contract and user-specified

invariants as input, producing an instrumented contract that rejects transactions violating the invariants. These invariants are specified in a formal specification language that supports quantifiers, summations, and stateful expressions. It allows users to define complex properties over arrays, mappings, and global contract state.

3.2.5 Verification Infrastructure

Verification tools can be categorized by verification level: SOLC-VERIFY [48] operates at the source-code level and KEVM [49] at the bytecode level.

SOLC-VERIFY [48] provides a modular verification framework that enables developers to annotate contracts with specifications and automatically verify compliance. The tool's integration with the Solidity compiler workflow makes formal verification more accessible to practitioners working with synthesized contracts.

KEVM [49] provides complete formal semantics of the Ethereum Virtual Machine, establishing the formal foundations necessary for rigorous verification of smart contract execution.

3.2.6 Business Logic and Behavioral Verification

The work by Shishkin [50] addresses debugging smart contracts' business logic using symbolic model-checking, providing techniques for verifying that contracts correctly implement intended business rules. The proposed approach involves translating smart contracts written in Solidity into an intermediate formal model and expressing business logic properties as state and trace specifications.

HIGHGUARD [51] introduces cross-chain business logic monitoring, using dynamic condition response (DCR) graph models to specify and verify contract behaviors at runtime. The system employs DCR graphs [26, 25] as formal specifications to verify contract execution against these models. Another interesting aspect about this work is the capability of operating in cross-chain environments for detecting business logic flaws.

Chapter 4

Experimental Setup

This chapter describes the empirical protocol adopted to evaluate automatically synthesized invariants for smart contract security. This chapter is organised following conventional research-paper structure: we elaborate upon the experimental setup, datasets used, and evaluation metrics. All artefacts (code, data, and raw results) needed to reproduce the experiments are publicly available at:

- RQ1 Artefacts
- RQ2 Artefacts

4.1 Evaluation Dataset

The following datasets are available at Flames Results.

FLAMES-20K: 20,000 smart contracts dataset extracted from DISL [20] used for fine tuning the model. Each element in the dataset is composed of multiple attributes, including:

- `predicate` – the original (ground-truth) invariant associated with the contract,
- `results` – the invariant synthesized by the model,
- `original_idx` – index linking to the original entry in the DISL dataset.

FLAMES-100K: 100,000 smart contracts dataset derived from DISL, used for fine tuning the model. Each entry contains:

- `predicate` – the original invariant,
- `results` – the invariant synthesized by the model,

- `original_idx` – index pointing to the corresponding DISL record.

DISL: [1] the main source dataset of verified smart contracts from which FLAMES-20K and FLAMES-100K are derived. Each element includes multiple metadata attributes, such as:

- `contract_address` – on-chain address of the contract, to retrieve the source code from Etherscan.io,
- `compiler_version` – version of the Solidity compiler used to deploy it.

CodeLlama-7B: same contracts as FLAMES20K dataset, but the invariants are generated by default CodeLlama. This acts as a baseline to evaluate generation quality. Each entry contains:

- `predicate` – the original invariant,
- `codeLlama_results` – the invariant synthesized by the model.

SB-Heist-4.x: a subset ($n = 110$) of SB-Heist, consisting of contracts written in Solidity 0.4.x. Each contract contains ground-truth exploit annotations, used for evaluating security-relevant inference in RQ2.

4.2 Evaluation Method

The study is divided into two phases, each mapped to a research question:

RQ1–Compilability To what extent do invariants synthesised by FLAMES preserve the ability of existing smart contracts to compile?

RQ2–Security Do FLAMES invariants prevent known exploits *without* altering benign behaviour?

4.2.1 RQ1 Protocol

For every contract C in a dataset we perform: (i) native compilation to establish a baseline; (ii) invariant injection using the candidate model; and (iii) re-compilation with the original compiler version. All steps are fully automated and the automated pipeline is shown in Figure 4.1.

[1] `https://huggingface.co/datasets/ASSERT-KTH/DISL`

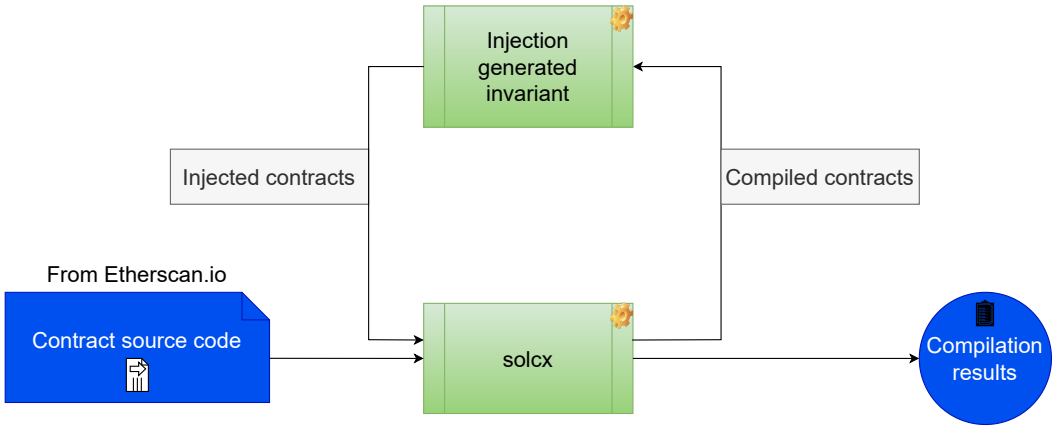

Figure 4.1: Automated pipeline RQ1 Experimental Setup

Metrics

- **Compilation Success**: percentage of contracts that compile after injection.
- **Failure categories**: syntax error, etc..

4.2.2 RQ2 Protocol

For each vulnerable contract V in SB-Heist, and for every combination of *inference strategy* $inf \in \{Aggregated, Isolated\}$ as described in Section 4.2.2, and *invariant-injection strategy* $inj \in \{VL, Pre, Post, \ldots\}$ as showed in Figure 4.2, we:

1) Patch V with the inf model-generated `require` statements at inj locations;

2) Execute the benign transaction suite (*regression test*);

3) Replay the ground-truth exploit;

4) Record the outcome (pass/fail).

```
1  pragma solidity ^0.4.10;
2
3  contract IntegerOverflowAdd {
4      mapping (address => uint256) public balanceOf;
5
6      // INSECURE
7      function transfer(address _to, uint256 _value)
       public {
8          // PRECONDITION - PRE
9
10         balanceOf[msg.sender] -= _value;
11
12         // Vulnerability: potential integer overflow
       if _value is very large
13         // VULNERABLE LINE - VL (line 18 in original):
14         balanceOf[_to] += _value;
15
16         // POSTCONDITION - POST
17     }
18 }
```

Figure 4.2: Annotated Solidity snippet showing precondition, postcondition, and vulnerability line.

Inference Strategies

Considering the inference process described in Section 2.1, we evaluate two different strategies for inferring invariants:

- **Aggregated** — at each inference step, we inject the `FILL-ME` token while retaining the previously inferred invariants within the contract (i.e., from Vulnerable Line to Pre-condition to Post-condition). This allows each new inference to build upon the results of the previous ones.
- **Isolated** — each inference step is performed independently, without incorporating any information or invariants generated in previous iterations.

Injection Strategies

We evaluate seven placement variants inspired by prior work on smart contracts discussed in Section 3.2.2:

- **VL (Vulnerability Line)** — invariant inserted exactly on the line that contains the vulnerability.
- **Pre (Pre-condition)** — immediately after the function signature that encloses the VL.
- **Post (Post-condition)** — immediately before the closing brace of the same function.
- **Pre + VL** — both pre-condition and vulnerability line.
- **VL + Post** — vulnerability line and post-condition.
- **Pre + Post** — pre-condition and post-condition.
- **Pre + VL + Post** — invariant at all three positions.

The automated pipeline is shown in Figure 4.3.

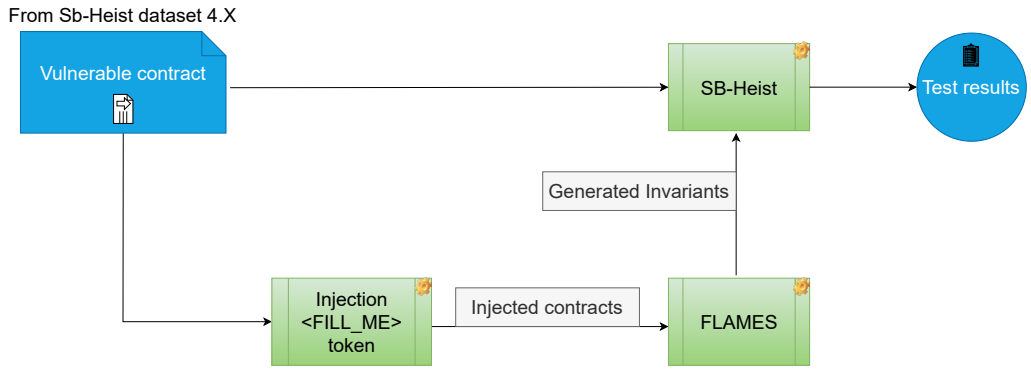

Figure 4.3: Automated pipeline RQ2 Experimental Setup

Metrics

RPR Regression Tests Pass Rate: the contract behaves identically for benign inputs.

RPV Ratio of patched vulnerabilities: the exploit is stopped by the injected invariant.

MP Matching Patches: contracts where both *RPR* and *RPV* hold.

4.3 Experimental Infrastructure

Experiments ran on a dedicated Ubuntu 22.04 server equipped with a 64-core AMD EPYC 7742 CPU, 256 GB RAM, and one NVIDIA A100 (80 GB) GPU. The GPU is only required for model inference; compilation and SB-Heist replay are CPU-bound.

Furthermore, we used the following software packages to build the experimental pipeline:

- **Python 3.12.1** (for scripts)
- **Solidity compiler**: `solc` via `py-solc-x` (versions 0.4.11–0.8.24) based on contracts under test individually.
- **SB-Heist** [40] for replay-based security testing.
- Auxiliary libraries: `pandas`, `requests`, `difflib`, `fuzzywuzzy`, and `re`

Container specifications and an exact `conda` environment file are included in the replication package.

Chapter 5

Results

This chapter presents and discusses the results obtained from the experimental setup described in Chapter 4. In particular, it addresses both research questions defined in Section 1.3. Specifically, RQ1 is addressed in Section 5.1, while RQ2 is discussed in Section 5.2. Finally, Section 6.2 analyzes possible limitations and threats to the validity of our work.

5.1 RQ1 Results: Compilability Evaluation

The objective of this phase was to assess whether smart contracts remain compilable after the injection of automatically synthesized invariants. For each contract, compilation was tested before and after the injection of synthesized invariants. A sample of 5,000 was randomly extracted from the FLAMES produced dataset hosted by HuggingFace [1] to establish a valuable dataset for the evaluation.

The following metrics were tracked:

- **Successfully compiled contracts** before invariant injection
- **Contracts that failed to compile** due to Etherscan-related issues described in Section 5.1.1
- **Contracts that compiled successfully** after injecting the synthesized invariants

[1] `https://huggingface.co/datasets/ASSERT-KTH/FLAMES_results`

5.1.1 Etherscan compilation failures

A notable portion of compilation failures on Etherscan derive from issues with flattened source code retrieved via their API. Flattened source code refers to a single Solidity file created by combining all imported and dependent files of a smart contract into one file. These include inconsistent compiler settings (e.g., incorrect `pragma` versions) and duplicated or reordered imports.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.5.0;
3  pragma solidity ^0.5.0; // Duplicate pragma - can
       cause compiler warnings/errors
4  library SafeMath {
5      //code...
6  }
7  // Original contract
8  contract MyToken {
9      using SafeMath for uint256;
10     mapping(address => uint256) public balances;
11     constructor() public {
12         balances[msg.sender] = 1000;
13     }
14     function transfer(address to, uint256 amount)
       public {
15         balances[msg.sender] -= amount;
16         balances[to] += amount;
17     }
18 }
19 // Duplicate contract - compilation error: Identifier
       already declared
20 contract MyToken {
21     mapping(address => uint256) public balances;
22     constructor() public {
23         balances[msg.sender] = 1000;
24     }
25 }
```

Figure 5.1: Example of a Solidity contract flattening error: duplicate pragma and contract declarations.

Errors show in Figure 5.1 have been observed in multiple community cases. They include an ERC-20 contract combined with a Crowdsale and deployed on Ropsten

using OpenZeppelin v2.5.0[2] and the `ShahToken_flat.sol` contract compiled with Solidity `^0.8.5` [52]. Both exhibited issues with duplicated `pragma` directives or improperly flattened Solidity files.

5.1.2 Comparative Compilability Analysis

Figure 5.2 illustrates a comparative summary of compilation success rates across the three datasets.

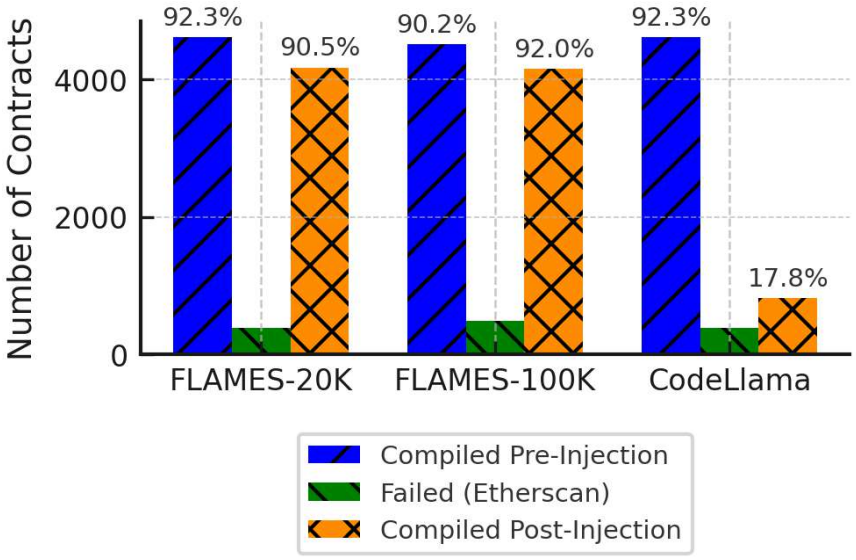

Figure 5.2: Compilation success rate after invariant injection across datasets.

The results obtained using the FLAMES20K model indicate a high level of syntactic and semantic compatibility between the generated invariants and the original contract code, with over 90% of contracts compiling successfully after invariant injection. When using the larger FLAMES100K model, the compilation success rate was slightly higher, suggesting that increasing the amount of training data may improve the quality or consistency of the synthesized invariants. Although the precise training/test split for these datasets is not specified, the observed trend highlights the potential benefits of scaling the training corpus for enhanced model performance.

[2]https://forum.openzeppelin.com/t/verification-failed-after-flattening-contract/2366

In clear contrast to the FLAMES datasets, the CodeLlama-generated invariants resulted in a significantly lower compilation success rate. Only 17.8% of contracts remained compilable after invariant injection. This suggests that the invariants produced by the CodeLlama model were syntactically or semantically incompatible in the majority of cases. This is due to less targeted training, bringing to incomplete invariants. These results indicate that, for the FLAMES20K and FLAMES100K models, the synthesis pipeline is effective at generating deployable security invariants in a substantial portion of cases. While the compilation success rate is not 100%, verifying compilation requires no human effort. As a result, a non-negligible failure rate does not diminish the utility of the correctly compiled invariants.

5.2 RQ2 Results: Validating Security Invariants

The second experimental phase aimed to assess the practical security effectiveness of the synthesized invariants injected into vulnerable smart contracts. The evaluation was conducted using the SB-Heist framework [40], which provides reproducible exploit scenarios derived from the SmartBugs-Curated dataset. The contracts were selected from the Sb-Heist dataset [40], in particular the 4.X Solitidy version dataset. This dataset does not overlap with the training dataset of FLAMES models.

For this experiment, contracts vulnerable to known exploits were patched using require statements generated by the 3 different FLAMES models described in 2.1. Each patched contract was then validated against the associated exploit using SB-Heist, which performs both regression and exploit-based testing. The goal was to determine whether the injected invariants successfully neutralized the documented vulnerabilities without introducing new unintended behavior.

Validation was carried out for each model across multiple injection strategies and inference strategies as described in Section 4.2.2. A patch was considered matching if it both passed the regression test and patched the vulnerability.

5.2.1 Evaluation Metrics

We tracked the following metrics:

- **Vulnerability Patched**: if the injected invariant covered the related exploit.
- **Regression Test success**: if the injected invariant did not change the contract behaviour.
- **Matching Patch**: the exploit was covered and the behaviour did not change after the injection.

To improve clarity, summary tables for the spider graphs are provided as follows: Tables 5.2 and 5.1 present the results for FLAMES20K, while Tables 5.4 and 5.3 show the results for FLAMES100K. Tables 5.6 and 5.5 report the results for CodeLlama-7B. Each cell in the tables reports results in the format: Regression test success / Vulnerability patched / Matching patch.

Table 5.1: Aggregated results for FLAMES20K by vulnerability category and inference strategy.

Vulnerability	VL	pre_post	pre	post	pre_VL_post	pre_VL	VL_post
denial_of_service (4)	3 / 1 / 1	2 / 1 / 0	4 / 1 / 1	2 / 1 / 0	2 / 1 / 0	3 / 1 / 1	2 / 1 / 0
other (2)	1 / 1 / 0	2 / 0 / 0	2 / 0 / 0	2 / 0 / 0	2 / 0 / 0	2 / 0 / 0	1 / 0 / 0
unchecked_low_level_calls (22)	15 / 16 / 9	9 / 9 / 0	17 / 1 / 0	13 / 7 / 0	8 / 15 / 3	17 / 15 / 10	10 / 13 / 3
access_control (16)	11 / 11 / 6	9 / 12 / 6	10 / 10 / 5	12 / 9 / 5	7 / 14 / 6	7 / 14 / 6	8 / 12 / 4
reentrancy (26)	18 / 5 / 0	14 / 18 / 7	21 / 5 / 1	13 / 16 / 4	15 / 16 / 6	19 / 6 / 2	14 / 17 / 6
front_running (6)	5 / 0 / 0	3 / 2 / 0	6 / 0 / 0	3 / 2 / 0	3 / 2 / 0	5 / 0 / 0	3 / 2 / 0
bad_randomness (8)	7 / 1 / 1	7 / 4 / 4	7 / 2 / 2	6 / 4 / 3	6 / 5 / 4	7 / 3 / 3	5 / 3 / 2
arithmetic (20)	14 / 13 / 7	10 / 12 / 5	15 / 8 / 4	12 / 12 / 7	13 / 11 / 6	15 / 11 / 6	11 / 12 / 5
time_manipulation (4)	2 / 1 / 1	1 / 3 / 0	4 / 0 / 0	3 / 2 / 1	1 / 2 / 0	2 / 2 / 1	2 / 2 / 1

Table 5.2: Isolated results for FLAMES20K by vulnerability category and inference strategy.

Vulnerability	VL	pre_post	pre	post	pre_VL_post	pre_VL	VL_post
denial_of_service (4)	3 / 1 / 1	2 / 1 / 0	3 / 2 / 1	2 / 1 / 0	1 / 2 / 0	2 / 2 / 1	2 / 1 / 0
other (2)	1 / 1 / 0	1 / 0 / 0	1 / 1 / 0	1 / 0 / 0	1 / 0 / 0	2 / 0 / 0	1 / 0 / 0
unchecked_low_level_calls (22)	14 / 16 / 8	7 / 10 / 3	13 / 11 / 5	7 / 10 / 2	9 / 10 / 4	15 / 14 / 7	7 / 10 / 2
access_control (16)	12 / 9 / 6	13 / 7 / 6	10 / 7 / 3	14 / 5 / 4	12 / 8 / 6	12 / 7 / 5	13 / 8 / 6
reentrancy (26)	21 / 4 / 1	8 / 6 / 0	24 / 2 / 0	7 / 7 / 0	7 / 7 / 0	20 / 6 / 2	7 / 7 / 0
front_running (6)	3 / 2 / 0	2 / 3 / 0	2 / 3 / 0	2 / 4 / 0	2 / 3 / 0	2 / 3 / 0	2 / 3 / 0
bad_randomness (8)	7 / 0 / 0	5 / 1 / 0	7 / 0 / 0	5 / 1 / 0	5 / 1 / 0	6 / 1 / 0	5 / 0 / 0
arithmetic (20)	16 / 11 / 7	13 / 12 / 8	16 / 11 / 7	17 / 10 / 10	13 / 12 / 8	16 / 11 / 7	13 / 13 / 8
time_manipulation (4)	1 / 1 / 0	0 / 2 / 0	2 / 1 / 0	1 / 2 / 0	0 / 2 / 0	0 / 2 / 0	0 / 2 / 0

Table 5.3: Aggregated results for FLAMES100K by vulnerability category and inference strategy.

Vulnerability	VL	pre_post	pre	post	pre_VL_post	pre_VL	VL_post
denial_of_service (4)	3 / 1 / 1	2 / 3 / 1	4 / 2 / 2	2 / 2 / 0	2 / 3 / 1	3 / 2 / 2	2 / 3 / 1
other (2)	2 / 0 / 0	2 / 0 / 0	2 / 0 / 0	2 / 0 / 0	2 / 0 / 0	2 / 0 / 0	2 / 0 / 0
unchecked_low_level_calls (22)	17 / 11 / 7	14 / 1 / 0	16 / 2 / 0	16 / 0 / 0	14 / 6 / 4	16 / 11 / 6	15 / 5 / 4
access_control (16)	10 / 12 / 6	11 / 12 / 7	11 / 12 / 7	13 / 10 / 7	9 / 14 / 7	9 / 14 / 7	10 / 13 / 7
reentrancy (26)	23 / 3 / 0	8 / 21 / 3	23 / 10 / 7	10 / 18 / 2	7 / 22 / 3	21 / 12 / 7	9 / 19 / 2
front_running (6)	4 / 1 / 0	3 / 3 / 0	4 / 2 / 0	4 / 2 / 0	3 / 3 / 0	3 / 2 / 0	4 / 2 / 0
bad_randomness (8)	7 / 0 / 0	5 / 3 / 1	7 / 1 / 1	5 / 3 / 1	5 / 3 / 1	6 / 2 / 1	5 / 2 / 1
arithmetic (20)	13 / 13 / 6	14 / 11 / 6	16 / 6 / 3	16 / 9 / 7	12 / 12 / 4	12 / 14 / 6	11 / 14 / 5
time_manipulation (4)	4 / 0 / 0	3 / 3 / 2	4 / 1 / 1	3 / 3 / 2	2 / 3 / 1	4 / 1 / 1	2 / 3 / 1

Table 5.4: Isolated results for FLAMES100K by vulnerability category and inference strategy.

Vulnerability	VL	pre_post	pre	post	pre_VL_post	pre_VL	VL_post
denial_of_service (4)	3 / 1 / 1	1 / 2 / 0	4 / 1 / 1	1 / 2 / 0	1 / 2 / 0	3 / 1 / 1	1 / 2 / 0
other (2)	2 / 0 / 0	1 / 0 / 0	2 / 0 / 0	1 / 0 / 0	1 / 0 / 0	2 / 0 / 0	1 / 0 / 0
unchecked_low_level_calls (22)	15 / 14 / 8	8 / 9 / 3	14 / 5 / 1	8 / 9 / 3	8 / 11 / 5	14 / 14 / 7	8 / 11 / 5
access_control (16)	10 / 12 / 6	9 / 13 / 7	9 / 12 / 6	12 / 10 / 6	9 / 13 / 7	9 / 12 / 6	10 / 13 / 7
reentrancy (26)	23 / 3 / 0	7 / 18 / 1	22 / 10 / 6	9 / 15 / 0	6 / 19 / 1	21 / 11 / 6	8 / 16 / 0
front_running (6)	4 / 1 / 0	2 / 1 / 0	3 / 1 / 0	4 / 1 / 0	2 / 1 / 0	3 / 1 / 0	3 / 1 / 0
bad_randomness (8)	7 / 0 / 0	5 / 5 / 4	7 / 0 / 0	5 / 5 / 4	5 / 5 / 4	7 / 0 / 0	5 / 4 / 4
arithmetic (20)	13 / 13 / 6	12 / 13 / 6	13 / 12 / 5	17 / 11 / 9	11 / 14 / 6	12 / 13 / 5	12 / 14 / 7
time_manipulation (4)	4 / 0 / 0	0 / 0 / 0	4 / 0 / 0	0 / 0 / 0	0 / 0 / 0	4 / 0 / 0	0 / 0 / 0

Table 5.5: Aggregated results for CodeLlama7B by vulnerability category and inference strategy.

Vulnerability	VL	pre_post	pre	post	pre_VL_post	pre_VL	VL_post
denial_of_service (4)	2 / 2 / 1	1 / 2 / 1	3 / 1 / 1	1 / 1 / 0	1 / 2 / 1	1 / 2 / 1	1 / 2 / 1
other (2)	2 / 0 / 0	1 / 0 / 0	1 / 0 / 0	1 / 0 / 0	1 / 0 / 0	1 / 0 / 0	1 / 0 / 0
unchecked_low_level_calls (22)	16 / 11 / 6	7 / 8 / 4	14 / 6 / 4	10 / 8 / 4	7 / 13 / 7	14 / 14 / 10	7 / 13 / 7
access_control (16)	12 / 8 / 7	3 / 4 / 3	8 / 6 / 4	4 / 3 / 3	3 / 4 / 3	8 / 6 / 4	4 / 3 / 3
reentrancy (26)	24 / 7 / 5	8 / 11 / 0	25 / 2 / 1	8 / 11 / 0	7 / 16 / 4	24 / 7 / 5	7 / 16 / 4
front_running (6)	2 / 3 / 0	0 / 2 / 0	2 / 2 / 0	0 / 2 / 0	0 / 2 / 0	1 / 3 / 0	0 / 2 / 0
bad_randomness (8)	2 / 3 / 0	3 / 2 / 1	5 / 1 / 0	4 / 2 / 1	1 / 3 / 1	1 / 3 / 0	2 / 3 / 1
arithmetic (20)	13 / 6 / 6	5 / 5 / 4	11 / 6 / 6	6 / 1 / 1	5 / 5 / 4	11 / 8 / 8	5 / 3 / 2
time_manipulation (4)	3 / 0 / 0	0 / 0 / 0	2 / 0 / 0	1 / 1 / 1	0 / 0 / 0	1 / 0 / 0	0 / 1 / 0

Table 5.6: Isolated results for CodeLlama-7B by vulnerability category and inference strategy.

Vulnerability	VL	pre_post	pre	post	pre_VL_post	pre_VL	VL_post
denial_of_service (4)	2 / 2 / 1	0 / 2 / 0	3 / 2 / 2	1 / 2 / 0	0 / 2 / 0	1 / 2 / 1	1 / 2 / 0
other (2)	2 / 0 / 0	0 / 0 / 0	1 / 0 / 0	0 / 0 / 0	0 / 0 / 0	1 / 0 / 0	0 / 0 / 0
unchecked_low_level_calls (22)	16 / 11 / 6	6 / 2 / 1	14 / 5 / 4	10 / 5 / 1	6 / 8 / 4	13 / 13 / 8	6 / 11 / 4
access_control (16)	11 / 8 / 6	6 / 9 / 6	9 / 9 / 6	7 / 8 / 4	6 / 9 / 6	9 / 9 / 6	6 / 8 / 5
reentrancy (26)	24 / 7 / 5	1 / 3 / 0	24 / 3 / 2	2 / 3 / 0	1 / 3 / 0	19 / 8 / 2	2 / 4 / 1
front_running (6)	2 / 3 / 0	1 / 2 / 0	3 / 2 / 0	1 / 2 / 0	0 / 3 / 0	2 / 3 / 0	0 / 3 / 0
bad_randomness (8)	2 / 3 / 0	0 / 1 / 0	7 / 1 / 0	0 / 1 / 0	0 / 1 / 0	2 / 3 / 0	0 / 1 / 0
arithmetic (20)	13 / 6 / 6	6 / 5 / 5	11 / 5 / 5	12 / 5 / 5	6 / 7 / 5	11 / 7 / 7	7 / 7 / 4
time_manipulation (4)	3 / 0 / 0	0 / 0 / 0	1 / 0 / 0	0 / 0 / 0	0 / 0 / 0	1 / 0 / 0	0 / 0 / 0

Spider Graphs

The spider graphs presented in the following Sections [5.2.2; 5.2.3; 5.2.4] display, along each axis, either a vulnerability type or an injection strategy. For each category, the graphs report the corresponding counts of Regression test successes, Vulnerability patched, and Matching Patch. In the "Vulnerability per best strategy" spider graph (e.g., Figure 5.3), each vulnerability label is accompanied by the injection strategy that resulted in the highest number of matching patches, showing the most effective approach for that specific vulnerability.

5.2.2 FLAMES20K Results

In this section, we present the results of both inference strategies applied to FLAMES20K. Figures 5.3 and 5.5 illustrate the results for each vulnerability under the aggregated and isolated inference strategies, respectively, while Figures 5.4 and 5.6 display the results for each injection strategy under the same configurations.

Aggregated Inference Results

Under the aggregated strategy, performance is balanced across most categories. *Arithmetic* records the highest matching patches (40), while *reentrancy* achieves the strongest test coverage (114 successes, 83 patched vulnerabilities). The *pre_VL* strategy leads overall with 29 matching patches.

Isolated Inference Results

Without context propagation, FLAMES20K retains strong performance. *Arithmetic* remains the top category (40 matching patches), and *VL* is the most effective strategy (23 matching patches).

5.2.3 FLAMES100K Results

In this section, we report the results of both inference strategies for FLAMES100K. Figures 5.7 and 5.9 present the outcomes per vulnerability for the aggregated and isolated settings, respectively, whereas Figures 5.8 and 5.10 show the results per injection strategy for the two settings.

Aggregated Inference Results

Compared to FLAMES20K, performance improves, especially in *access_control* (48 matching patches). The *pre_VL* strategy reaches the highest score in this setting (30 matching patches).

Isolated Inference Results

This model outperforms all others in the isolated configuration, with *access_control* (45) and *arithmetic* (44) leading in matching patches. The *pre_VL* strategy remains the most effective (25 matching patches).

5.2.4 CodeLlama Results

In this section, we present the results of both inference strategies for CodeLlama-7B. Figures 5.11 and 5.13 display the outcomes per vulnerability for the aggregated

and isolated settings, respectively, while Figures 5.12 and 5.14 illustrate the results per injection strategy for the two settings.

Aggregated Inference Results

CodeLlama-7B struggles with complex categories such as *access_control* but performs comparatively well in *unchecked_low_level_calls* (42 matching patches). *Pre_VL* yields the best aggregated results (28 matching patches).

Isolated Inference Results

Performance improves slightly when run in isolation, particularly in *access_control* (39) and *arithmetic* (37). The *pre_VL* and *VL* strategies tie for the lead (24 matching patches).

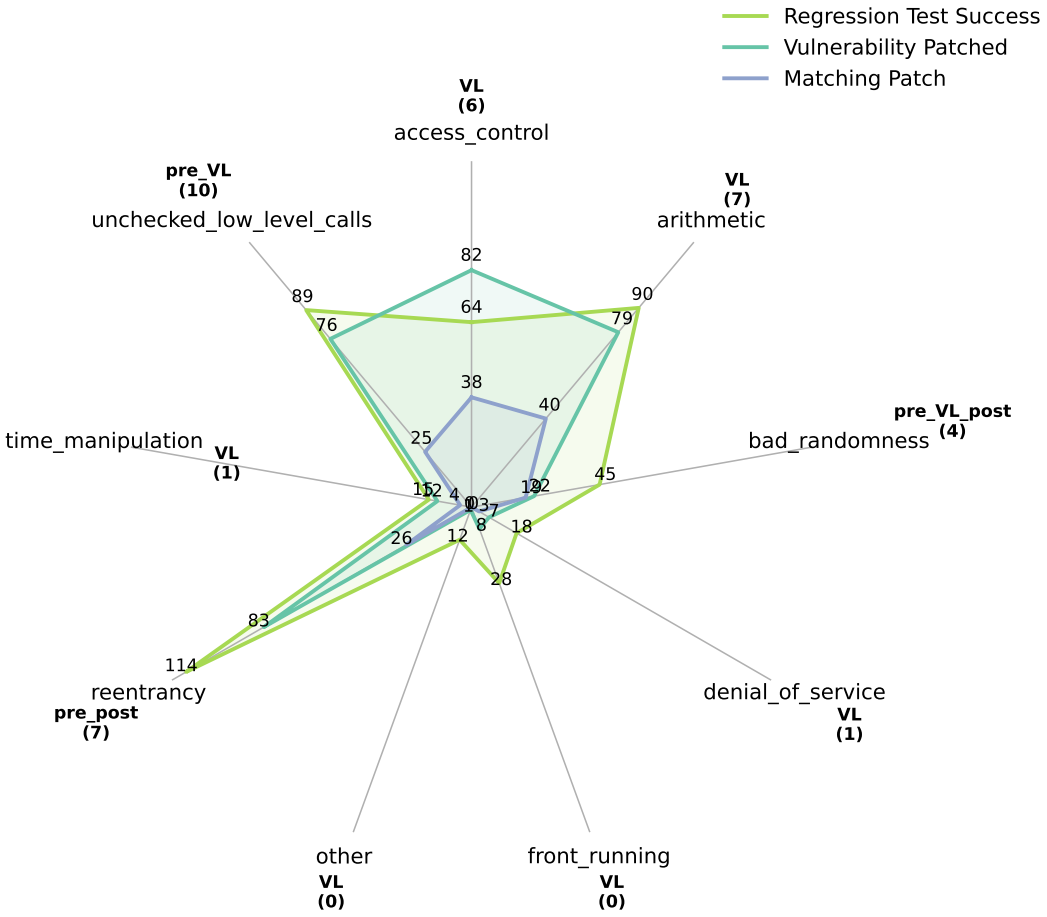

Figure 5.3: Vulnerability per best strategy spider graph for FLAMES20K, aggregated results

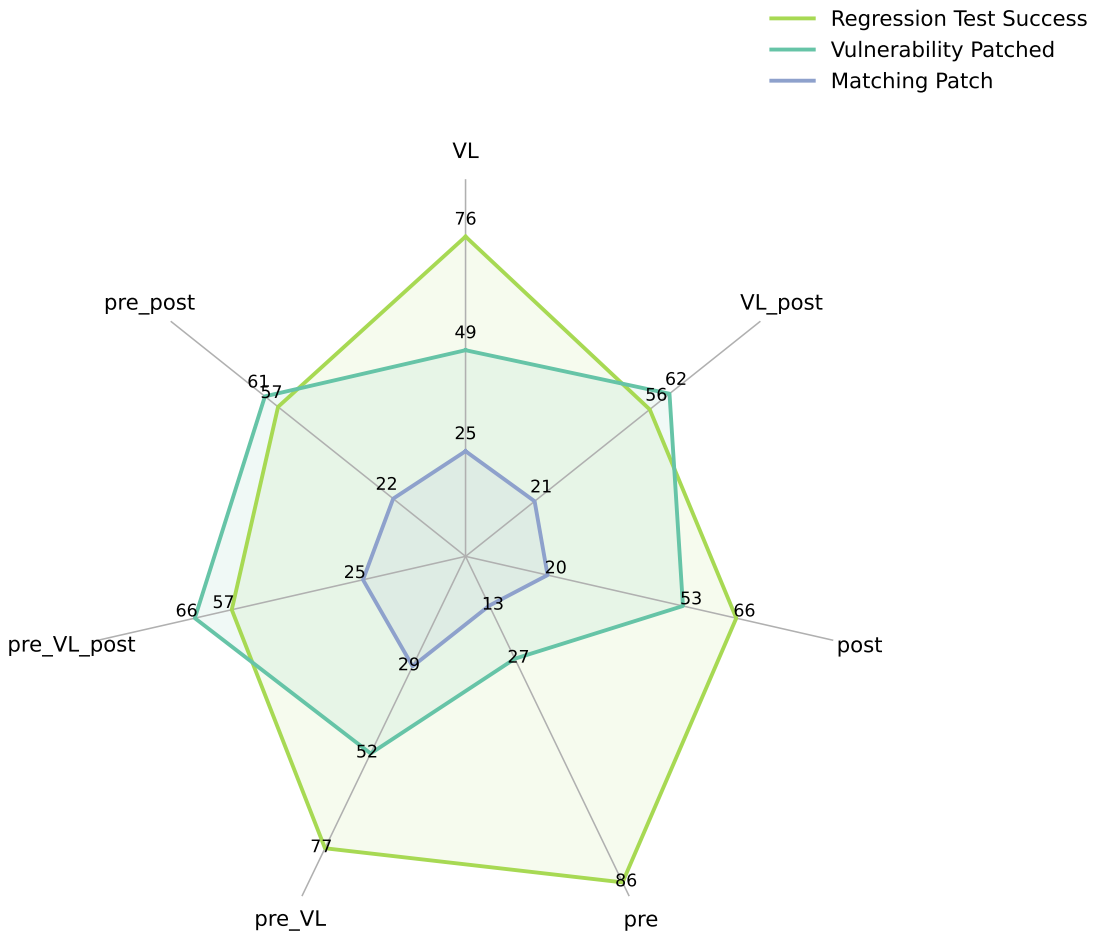

Figure 5.4: Performances per strategy spider graph for FLAMES20K, aggregated results

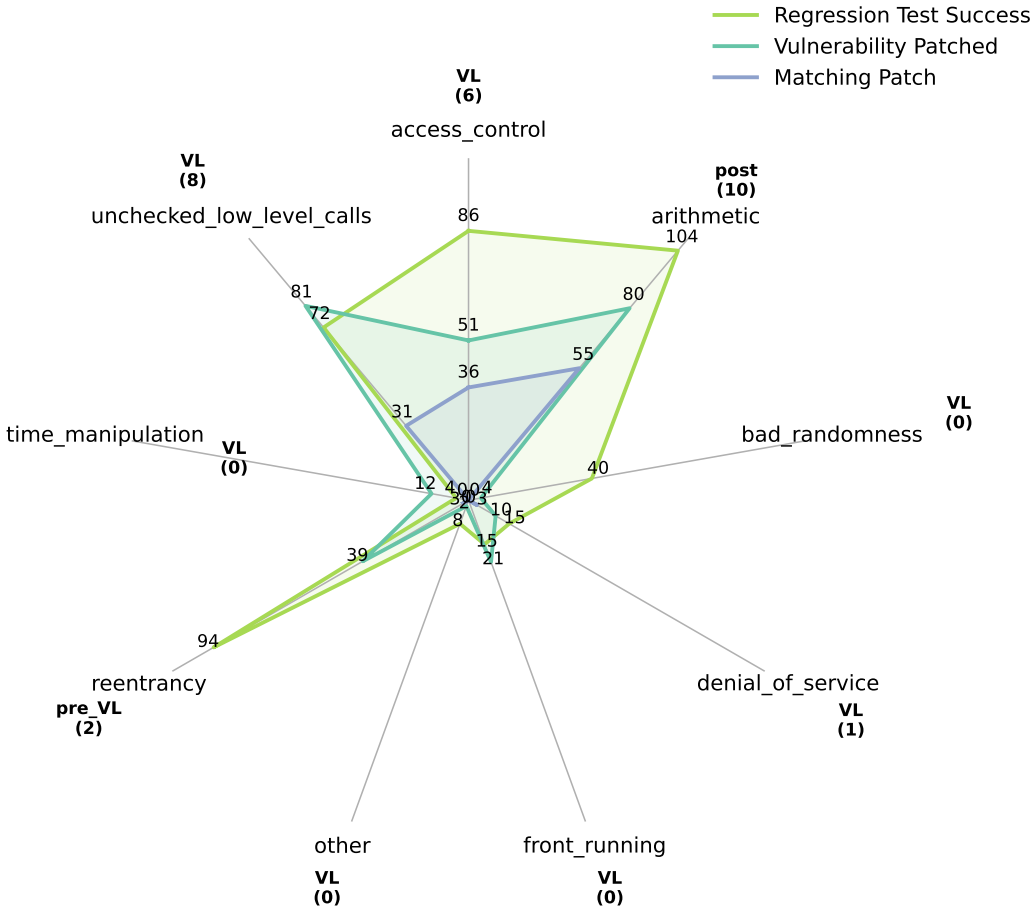

Figure 5.5: Vulnerability per best strategy spider graph for FLAMES20K, isolated results

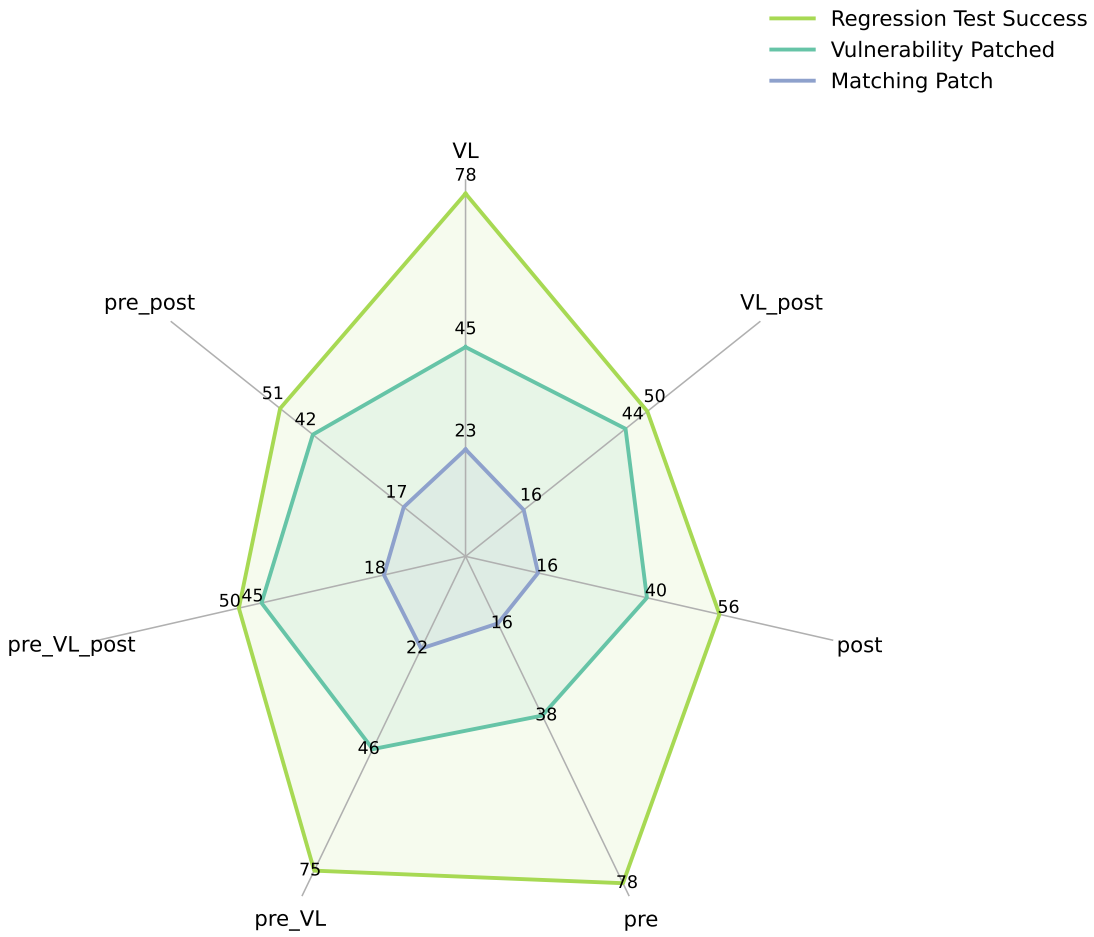

Figure 5.6: Performances per strategy spider graph for FLAMES20K, isolated results

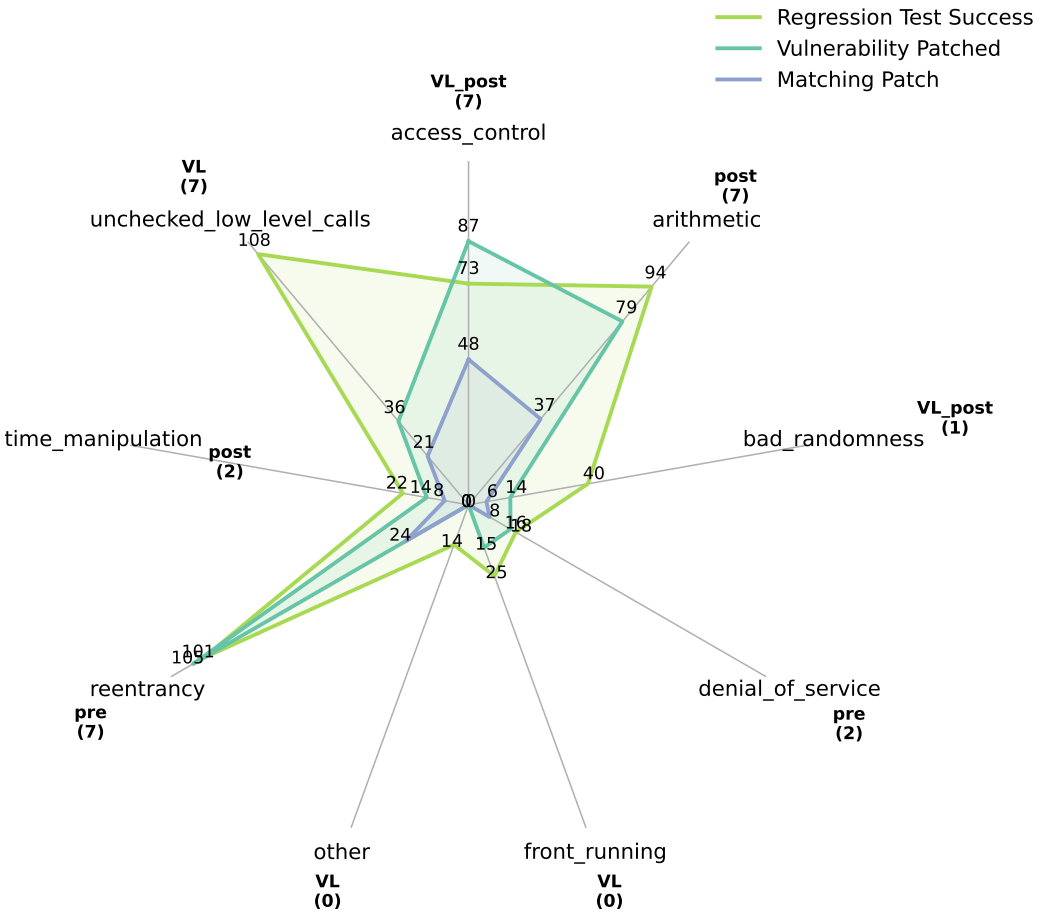

Figure 5.7: Vulnerability per best strategy spider graph for FLAMES100K, aggregated results

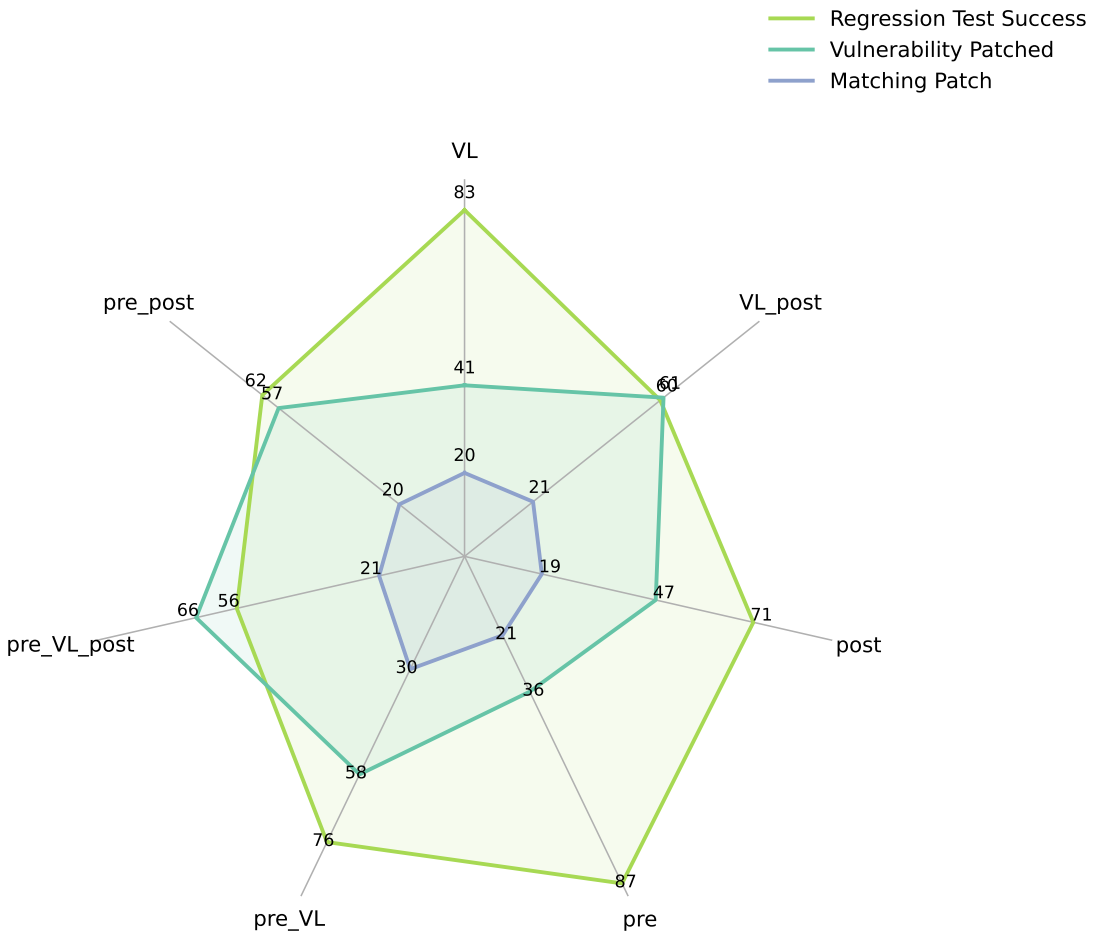

Figure 5.8: Performances per strategy spider graph for FLAMES100K, aggregated results

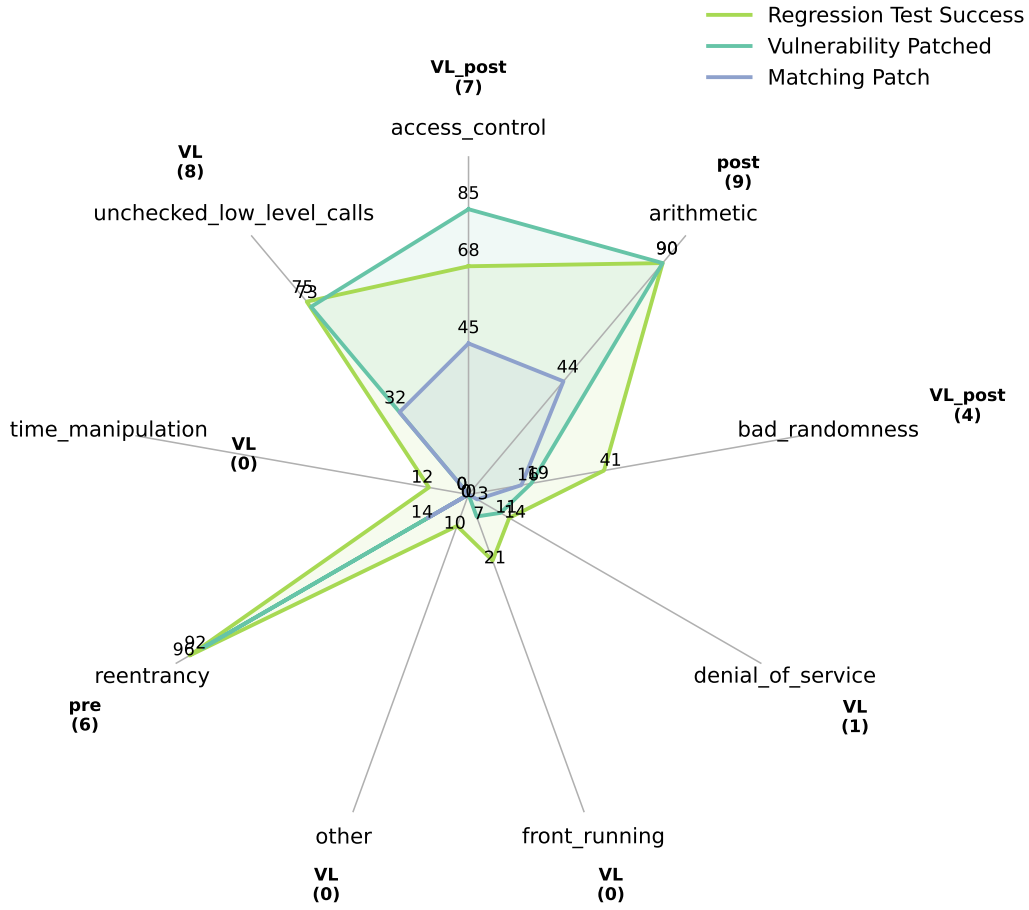

Figure 5.9: Vulnerability per best strategy spider graph for FLAMES100K, isolated results

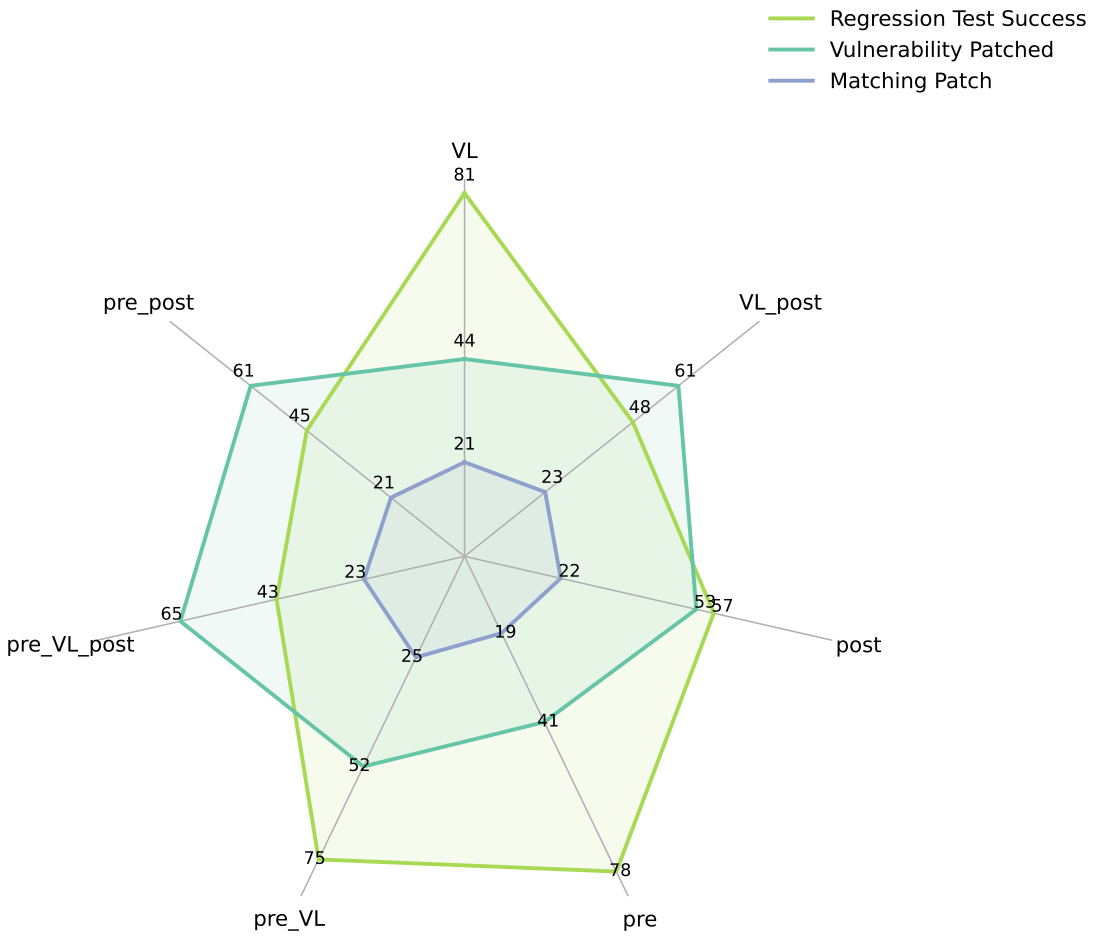

Figure 5.10: Performances per strategy spider graph for FLAMES100K, isolated results

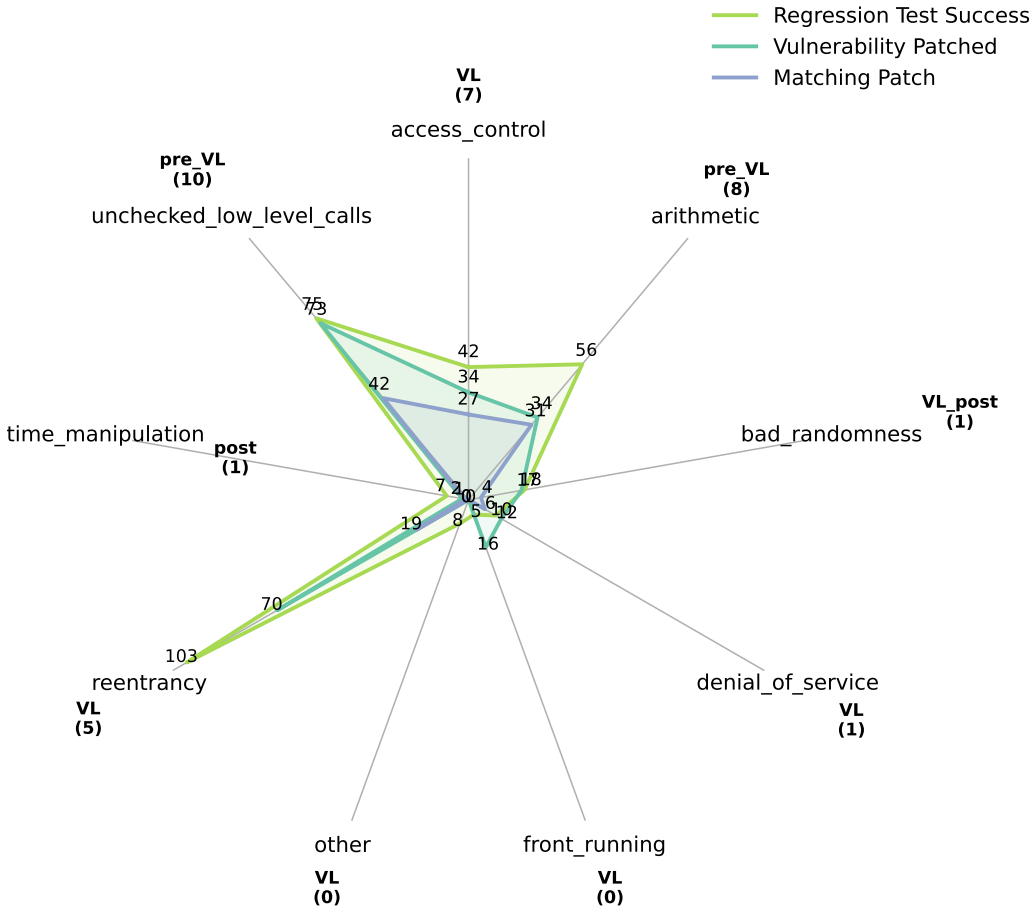

Figure 5.11: Vulnerability per best strategy spider graph CodeLlama, aggregated results

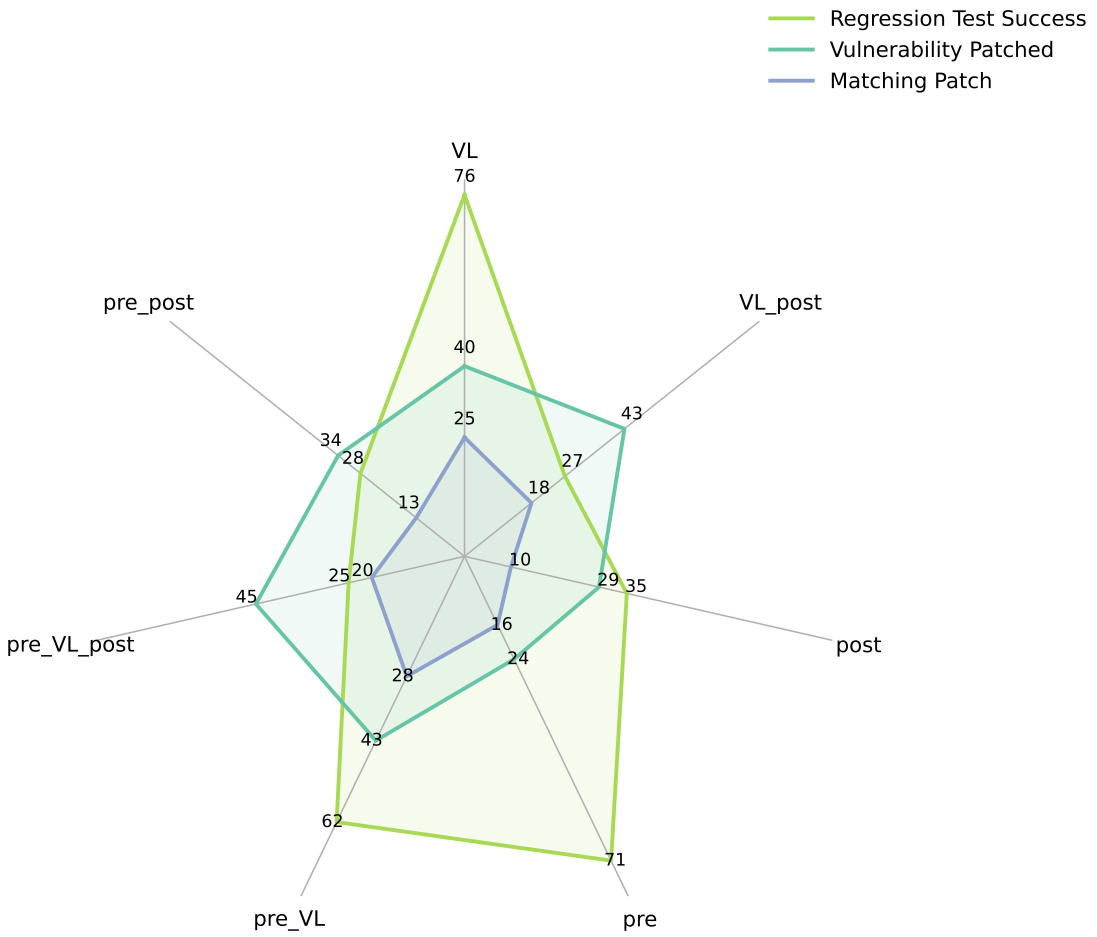

Figure 5.12: Performances per strategy spider graph for CodeLlama, aggregated results

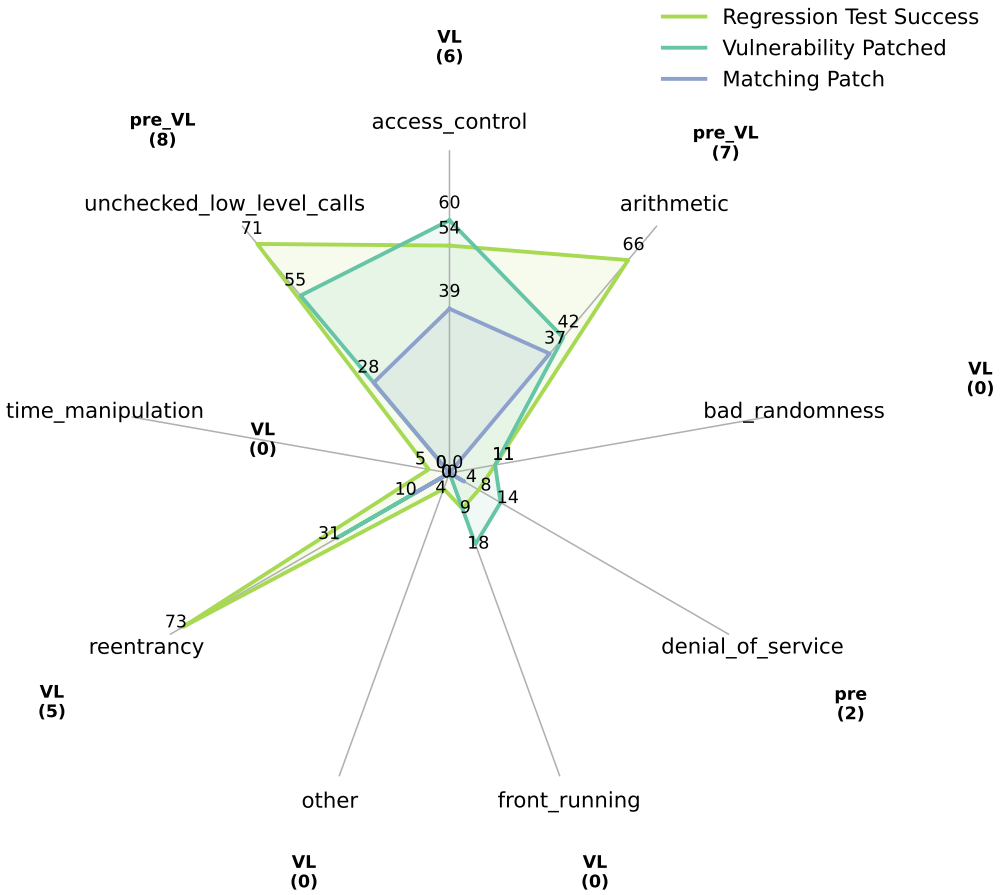

Figure 5.13: Vulnerability per best strategy spider graph for CodeLlama, isolated results

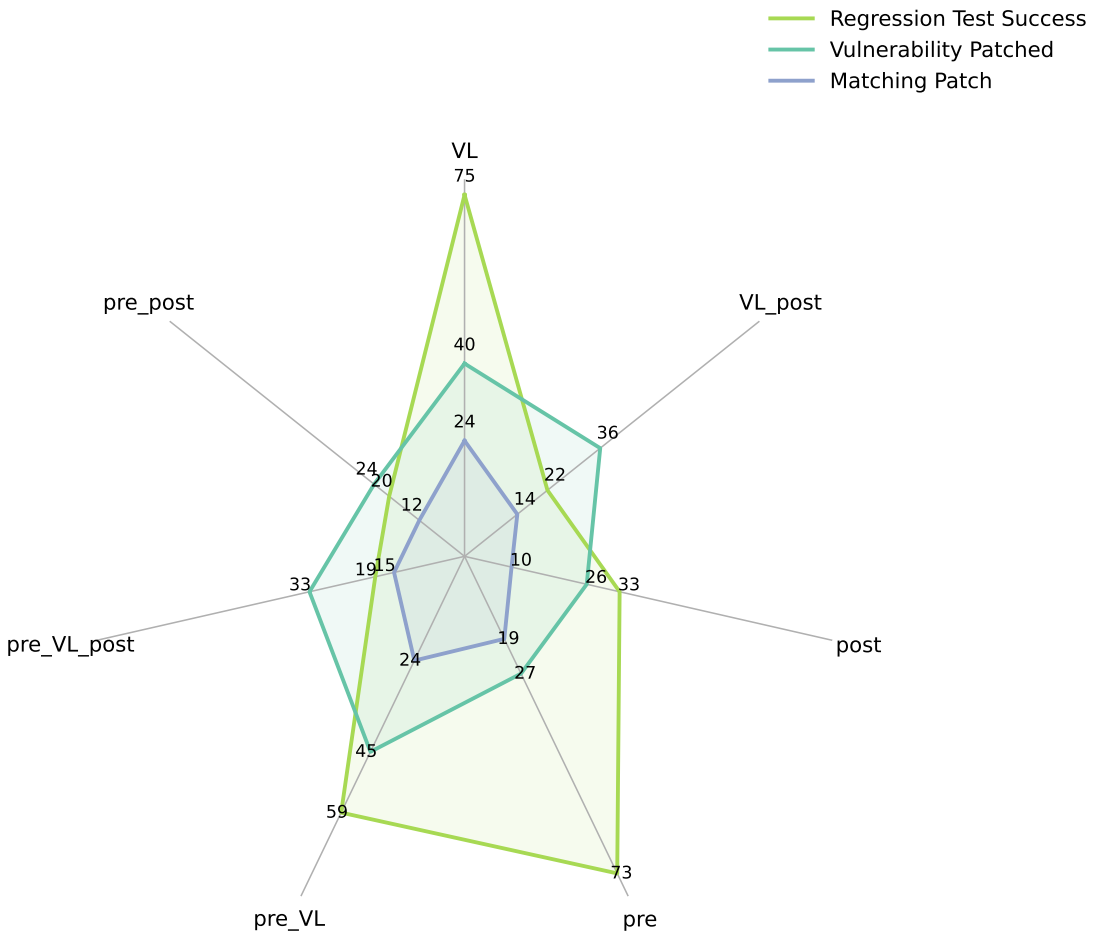

Figure 5.14: Performances per strategy spider graph for CodeLlama, isolated results

5.2.5 Comparative Validity Analysis

Table 5.7: Validation results for all models in both inference strategies

	Test Regression Successes	Vulnerabilities Patched	Matching Patches	Most Patched Vulnerability	Best Injection Strategy
FLAMES20K					
Aggregated	475	370	151	*arithmetic*	*pre_VL*
Isolated	438	300	128	*arithmetic*	*VL*
FLAMES100K					
Aggregated	486	355	152	*access_control*	*pre_VL*
Isolated	427	377	154	*access_control*	*pre_VL*
CodeLlama-7B					
Aggregated	324	258	130	*unchecked_low_level_call*	*pre_VL*
Isolated	301	231	118	*access_control*	*VL - pre_VL*

Table 5.7 summarizes the overall performance of different models and inference strategies in terms of test regression success, number of vulnerabilities patched, matching patches, the most frequently patched vulnerability type, and the best-performing injection strategy. The table allows for a comparison between aggregated and isolated settings for all the FLAMES models.

For the FLAMES20K, the aggregated configuration achieves higher test regression success (475) and more vulnerabilities patched (370) compared to the isolated configuration. However, both configurations share the same most frequently patched vulnerability type, *arithmetic*. This suggests that the nature of vulnerabilities does not change between configurations. Interestingly, while the aggregated setup favors the *pre_VL* strategy, the isolated setup benefits more from the *VL* strategy, indicating that isolation emphasizes verification-focused inference.

In the case of FLAMES100K, the pattern differs slightly. Aggregated results yield the highest test regression success (486), whereas isolated settings result in a slightly lower regression success (427) but achieve a higher number of vulnerabilities patched (377) and matching patches (154). Notably, FLAMES100K in isolated configuration achieves the most matching patches across all models and configurations. This result demonstrates that isolated inference can maximize patch quality and approval rates, possibly due to reduced interference from unrelated vulnerabilities. Both configurations identify *access_control* as the most patched vulnerability type and favor *pre_VL* as the optimal strategy, reinforcing its general effectiveness.

Comparing the two FLAMES models, FLAMES100K shows generally stronger performance in terms of test regression success and matching patches, especially in the isolated configuration. While FLAMES20K achieves slightly better regression success in the aggregated setting, FLAMES100K provides more effective and higher-quality patches overall. This improvement is likely attributable to the larger

dataset size, which facilitates more effective learning and better generalization of patching strategies. These results suggest that dataset scale plays a critical role in determining both the quantity and quality of generated patches.

For CodeLlama-7B, aggregated performance shows a test regression success of 324 and 258 vulnerabilities patched, with *unchecked_low_level_call* being the most frequently patched vulnerability. The isolated configuration exhibits a drop in performance (301 regression success and 231 vulnerabilities patched), and the most patched vulnerability shifts to *access_control*. Additionally, the best strategy in the isolated setting are *VL* and *pre_VL*, suggesting that multiple injection strategies may need to be blended to handle diverse vulnerability types effectively. Compared to the previous FLAMES models, CodeLlama-7B demonstrates the weakest overall performance, both in terms of regression success and number of vulnerabilities patched. This performance degradation highlights potential limitations in its capacity to generate effective patches under the tested configurations.

Overall, these results reveal several key trends. Aggregated configurations tend to maximize test regression success. Isolated configurations can sometimes achieve higher patch quality, as reflected in the number of matching patches. The choice of the most effective injection strategy is context-dependent: *pre_VL* performs consistently well across FLAMES models, whereas for CodeLlama-7B, a hybrid approach is preferable when vulnerabilities are isolated. Moreover, fine-tuning on a specialized dataset proves crucial for generating high-quality invariants. FLAMES models are significantly more accurate than the general-purpose CodeLlama, and increasing the fine-tuning data size yields further improvements. Finally, the shift in the most frequently patched vulnerability type between models and configurations highlights the interaction between model architecture, dataset size, and inference strategy in determining overall effectiveness.

5.2.6 Outliers and Study Cases

In this section, we describe and analyze various case studies involving the generated invariants and the outliers observed in the results presented in Sections 5.2.2, 5.2.3, and 5.2.4.

Synthesized `require(false)`

The three models occasionally generate the `require(false);` invariant. It represents a specific type of response indicating that the model could not identify a valid invariant to patch the vulnerability at that particular line in the contract. In such cases, the model determines that the most effective approach to prevent exploitation of the vulnerability is to force the contract to revert. Since the `require(false);` statement always evaluates to false, it will invariably cause the contract execution

to fail and revert.

When comparing models, a clear pattern emerges. FLAMES100K produces the fewest `require(false);` statements, FLAMES20K a moderate amount, and CodeLlama-7B the most. This distribution suggests that CodeLlama-7B struggles more often to synthesize correct and contextually appropriate invariants, falling back to unconditional reversion. In contrast, FLAMES100K more consistently infers the intended contract semantics and generates targeted invariants.

Although these cases are generally symptomatic of incomplete vulnerability understanding, they also reveal the role of inference strategy. Aggregated inference can occasionally enable non-trivial alternatives to `require(false);`, even in weaker models. For instance, CodeLlama-7B under aggregated inference was able to generate a `require(true);` invariant in the post-injection scenario for the 0x4b71 contract (Figure 5.15). While semantically vacuous, as described in Section 5.2.6, this output still provides insight into the synthesis process. It shows that additional context from earlier inference steps can influence the generated invariant. In some cases, this context can steer the model away from the most conservative fallback. More cases are shown in Appendix Section A.1.

```
1  pragma solidity ^0.4.24;
2  contract airPort{
3      function transfer(address from,address caddress,
       address[] _tos,uint v)public returns (bool){
4          require(_tos.length > 0);
5          bytes4 id=bytes4(keccak256("transferFrom(
       address,address,uint256)"));
6          for(uint i=0;i<_tos.length;i++){
7              caddress.call(id,from,_tos[i],v);
8          }
9          return true;
10 require(true);       //POST INVARIANT
11     }
12 }
```

Figure 5.15: 0x4b71 contract with post injection strategy applied in aggregated inference strategy CodeLlama-7B.

Synthesized `require(true)`

In certain scenarios, the model generates a `require(true);` invariant. This typically occurs for one of two reasons. First, the model may correctly determine that there is no exploitable vulnerability at the specific line in the contract, and therefore no

patch is needed. Second, the model may mistakenly fall into this first case by failing to detect an existing vulnerability. As a result, it still inserts a `require(true);` statement as a neutral placeholder. Since this condition always evaluates to true, it does not affect the execution or behavior of the contract in any way. An example of this behavior are the 0xe894 and 0x4051 contracts. CodeLlama-7B with both inference strategies produces the `require(true);` invariant in the post injection scenario as shown in Figure 5.16 and Figure 5.17. Notably, in this case the VL invariant provides an Matching Patch, and in all strategy combinations where the VL appears, including those with the post invariant, the patch is matching as expected by the `require(true);`.

```
1  pragma solidity ^0.4.24;
2  contract airDrop{
3      function transfer(address from,address caddress,
       address[] _tos,uint v, uint _decimals)public
       returns (bool){
4  require(_tos.length <= 20);      //PRE INVARIANT
5          require(_tos.length > 0);
6          bytes4 id=bytes4(keccak256("transferFrom(
       address,address,uint256)"));
7          uint _value = v * 10 ** _decimals;
8          for(uint i=0;i<_tos.length;i++){
9  require(caddress.call(id,from,_tos[i],_value));//VL
       INVARIANT
10             caddress.call(id,from,_tos[i],_value);
11         }
12         return true;
13 require(true);       //POST INVARIANT
14     }
15 }
```

Figure 5.16: 0xe894 contract with pre_VL_post injection strategy applied in CodeLlama-7B.

```
1  pragma solidity ^0.4.24;
2  contract airdrop{
3      function transfer(address from,address caddress,
       address[] _tos,uint v)public returns (bool){
4  require(msg.sender==from);       //PRE INVARIANT
5          require(_tos.length > 0);
6          bytes4 id=bytes4(keccak256("transferFrom(
       address,address,uint256)"));
7          for(uint i=0;i<_tos.length;i++){
8  require(caddress.call(id,from,_tos[i],v)); //VL
       INVARIANT
9          caddress.call(id,from,_tos[i],v);
10         }
11         return true;
12 require(true);       //POST INVARIANT
13     }
14 }
```

Figure 5.17: 0x4051 contract with pre_VL_post injection strategy applied in CodeLlama-7B.

As previously noted, this type of invariant may arise due to two primary factors. First, the identified vulnerability is not directly patchable at the specific line of code. Second, the model may be producing a false negative. It incorrectly assumes the code is secure when, in fact, a vulnerability exists. FLAMES100K is the only model that produces 0 instances of `require(true);`. This demonstrates its capacity to generate meaningful invariants regardless of the injection strategy and inference strategy employed. Regarding FLAMES20K, it produces `require(true);` statements when using the isolated inference strategy to produce post invariants. This is likely due to the lack of context provided by previous invariants, which is available in the aggregated inference strategy. CodeLlama-7B generates these types of invariant with both inference strategies. This behavior highlights the superior capacity of FLAMES models to understand the contract's context and consequently produce sound invariants.

FLAMES20K Better Than FLAMES100K

Globally, the FLAMES100K model demonstrates superior performance compared to the other models. However, there are specific cases where the FLAMES20K model successfully generates a Matching Patch while the FLAMES100K model fails to do so. A notable example is the FibonacciBalance.sol case. Using

the aggregated inference strategy with the pre_post injection approach, the FLAMES20K model generates a Matching Patch. In contrast, the FLAMES100K model fails both the regression test and produces no valuable patch. In the case of FLAMES20K, the patch is considered matching because both the invariants, `require(msg.data.length > 0);` and `require(calculatedFibNumber > 0);`, work together to effectively mitigate the access_control vulnerability as shown in Figure 5.18. The first ensures that the fallback function is not called without data, preventing accidental or fuzzed calls. The second acts as a true guard by enforcing that only specific library functions that correctly update the `calculatedFibNumber` are allowed to execute. This combination blocks unauthorized access and misuse of `delegatecall`. In contrast, the FLAMES100K case includes the same pre-invariant `require(msg.data.length > 0);`, but replaces the post-condition with `require(msg.value == 0);` as shown in Figure 5.19. While this check is logically sound, ensuring that no Ether is sent during the fallback, it does not fully address the core vulnerability. It fails to restrict which library functions can be called via `delegatecall`. Therefore, this patch is incomplete and does not fully mitigate the vulnerability.

```
pragma solidity ^0.4.22;
contract FibonacciBalance {
    address public fibonacciLibrary;
    // the current fibonacci number to withdraw
    uint public calculatedFibNumber;
    // the starting fibonacci sequence number
    //...code
    // allow users to call fibonacci library functions
    function() public {
require(msg.data.length>0);     //PRE INVARIANT
    require(fibonacciLibrary.delegatecall(msg.data));
require(calculatedFibNumber>0);     //POST INVARIANT
    }
}
// library contract - calculates fibonacci-like
   numbers;
contract FibonacciLib {
    // initializing the standard fibonacci sequence;
    uint public start;
    uint public calculatedFibNumber;
    // modify the zeroth number in the sequence
    function setStart(uint _start) public {
        start = _start;
    }
    function setFibonacci(uint n) public {
        calculatedFibNumber = fibonacci(n);
    }
    function fibonacci(uint n) internal returns (uint)
    {
        //...code
    }
}
```

Figure 5.18: FibonacciBalance.sol contract with pre_post injection strategy applied in aggregated inference strategy FLAMES20K.

```
1  pragma solidity ^0.4.22;
2  contract FibonacciBalance {
3      address public fibonacciLibrary;
4      // the current fibonacci number to withdraw
5      uint public calculatedFibNumber;
6      // the starting fibonacci sequence number
7      //...code
8      // allow users to call fibonacci library functions
9      function() public {
10 require(msg.data.length>0);      //PRE INVARIANT
11     require(fibonacciLibrary.delegatecall(msg.data));
12 require(msg.value==0);      //POST INVARIANT
13     }
14 }
15 // library contract - calculates fibonacci-like
       numbers;
16 contract FibonacciLib {
17     // initializing the standard fibonacci sequence;
18     uint public start;
19     uint public calculatedFibNumber;
20     // modify the zeroth number in the sequence
21     function setStart(uint _start) public {
22         start = _start;
23     }
24     function setFibonacci(uint n) public {
25         calculatedFibNumber = fibonacci(n);
26     }
27     function fibonacci(uint n) internal returns (uint)
       {
28         //...code
29     }
30 }
```

Figure 5.19: FibonacciBalance.sol contract with pre_post injection strategy applied in aggregated inference strategy FLAMES100K.

Another notable case is the 0x52d2. In this scenario, FLAMES20K produces a Matching Patch using the isolated inference strategy in the VL injection scenario. Meanwhile, its stronger counterpart fails to fix the vulnerability. The vulnerability in question is an unchecked_low_level_call, as shown in Figure 5.20. The FLAMES20K model effectively mitigates this issue through a `require(Token(0xd2a4c91875a07c740680799768e67dfe7fd5d34e).transfer(addr,0wei))` statement. This

statement ensures that a preceding call to a known token contract succeeds before executing the dangerous `addr.call.value(0 wei)();`. The added constraint introduces an implicit limitation on arbitrary call behavior. This approach effectively reduces the attack surface and addresses the vulnerability. On the other hand, the FLAMES100K patch inserts a `require(msg.sender==owner);` check as shown in Figure 5.21, which, while reasonable in context, does not directly address the core issue. Since the low-level call remains unchecked, the vulnerability persists.

```
1  pragma solidity ^0.4.19;
2  contract Token {
3      function transfer(address _to, uint _value)
       returns (bool success);
4      function balanceOf(address _owner) constant
       returns (uint balance);
5  }
6  contract EtherGet {
7      address owner;
8      function EtherGet() {
9          owner = msg.sender;
10     }
11     function withdrawTokens(address tokenContract)
       public {
12         Token tc = Token(tokenContract);
13         tc.transfer(owner, tc.balanceOf(this));
14     }
15     function withdrawEther() public {
16         owner.transfer(this.balance);
17     }
18     function getTokens(uint num, address addr) public
       {
19         for(uint i = 0; i < num; i++){
20 require(Token(0
       xd2a4c91875a07c740680799768e67dfe7fd5d34e).transfer
       (addr,0wei));      //VL INVARIANT
21             addr.call.value(0 wei)();
22         }
23     }
24 }
```

Figure 5.20: 0x52d2 contract with VL injection strategy applied in isolated inference strategy FLAMES20K.

```
1  pragma solidity ^0.4.19;
2  contract Token {
3      function transfer(address _to, uint _value)
       returns (bool success);
4      function balanceOf(address _owner) constant
       returns (uint balance);
5  }
6  contract EtherGet {
7      address owner;
8      function EtherGet() {
9          owner = msg.sender;
10     }
11     function withdrawTokens(address tokenContract)
       public {
12         Token tc = Token(tokenContract);
13         tc.transfer(owner, tc.balanceOf(this));
14     }
15     function withdrawEther() public {
16         owner.transfer(this.balance);
17     }
18     function getTokens(uint num, address addr) public
       {
19         for(uint i = 0; i < num; i++){
20 require(msg.sender==owner);      //VL INVARIANT
21             addr.call.value(0 wei)();
22         }
23     }
24 }
```

Figure 5.21: 0x52d2 contract with VL injection strategy applied in isolated inference strategy FLAMES100K.

Therefore, despite its larger search space and stronger overall performance, FLAMES100K fails to generate a functionally correct patch in this case. While FLAMES100K demonstrates superior results globally, these outliers reveal that increased model capacity does not guarantee success in every scenario. This suggests that certain specific cases may benefit from a different approach. The more focused methodology of the smaller FLAMES20K model can be advantageous in these instances. These findings highlight the context-dependent nature of automated patch generation.

CodeLlama-7B Better Than FLAMES

```
pragma solidity ^0.4.25;
contract DosNumber {
    uint numElements = 0;
    uint[] array;
    function insertNnumbers(uint value,uint numbers)
   public {
require(numElements<1500);       //PRE INVARIANT
        for(uint i=0;i<numbers;i++) {
            if(numElements == array.length) {
                array.length += 1;
            }
            array[numElements++] = value;
        }
    }
    function clear() public {
        require(numElements>1500);
        numElements = 0;
    }
    function clearDOS() public {
        require(numElements>1500);
        array = new uint[](0);
        numElements = 0;
    }
    function getLengthArray() public view returns(uint
   ) {
        return numElements;
    }
    function getRealLengthArray() public view returns(
   uint) {
        return array.length;
    }
}
```

Figure 5.22: dos_number.sol contract with pre injection strategy applied in isolated inference strategy CodeLlama-7B.

As demonstrated in Section 5.2.4, CodeLlama-7B shows worse performance than the FLAMES models. However, there are some outliers where CodeLlama successfully produces matching patches while the FLAMES models fail to do so.

One such case is dos_number.sol. This occurs using the isolated inference

strategy with the pre injection strategy. CodeLlama produces a Matching Patch in this scenario while FLAMES100K is only capable of passing the regression test. In Figure 5.22, the denial_of_service vulnerability is effectively patched by using a `require` statement that limits the overall size of the array via the condition `numElements < 1500`. This constraint prevents the array from growing indefinitely. In this way it controls the gas consumption and avoids gas limit exhaustion during the insertion loop. In contrast, the patch produced by FLAMES100K, shown in Figure 5.23, fails to fix the vulnerability. It only limits the number of elements inserted per transaction (e.g., through `require(numbers <= 382);`) without restricting the total array size. This allows multiple consecutive calls to accumulate elements beyond safe gas limits, leaving the contract vulnerable to denial_of_service attacks via gas exhaustion.

```
pragma solidity ^0.4.25;
contract DosNumber {
    uint numElements = 0;
    uint[] array;
    function insertNnumbers(uint value,uint numbers)
   public {
require(numbers<=382);       //PRE INVARIANT
        for(uint i=0;i<numbers;i++) {
            if(numElements == array.length) {
                array.length += 1;
            }
            array[numElements++] = value;
        }
    }
    function clear() public {
        require(numElements>1500);
        numElements = 0;
    }
    function clearDOS() public {
        require(numElements>1500);
        array = new uint[](0);
        numElements = 0;
    }
    function getLengthArray() public view returns(uint
   ) {
        return numElements;
    }
    function getRealLengthArray() public view returns(
   uint) {
        return array.length;
    }
}
```

Figure 5.23: dos_number.sol contract with pre injection strategy applied in isolated inference strategy FLAMES100K.

Another interesting case is the token.sol contract analyzed under the aggregated inference strategy with the pre injection invariant. CodeLlama-7B generates a Matching Patch, while FLAMES20K fails to adequately address the arithmetic vulnerability.

As shown in Figure 5.24, CodeLlama-7B correctly patches the underflow by

adding `require(balances[msg.sender] >= _value);` before the subtraction. This ensures the sender has sufficient balance for the transfer. Since the contract uses Solidity versions prior to 0.8.0, this check is essential to prevent dangerous wraparound behavior. In contrast, FLAMES20K's patch in Figure 5.25 misses the core issue. It adds the check `require(_to != 0x0);` to prevent transfers to the zero address, a safeguard against accidental token burning. However, it does not address the original ineffective check `require(balances[msg.sender] - _value >= 0);`.

This check fails because in Solidity 0.4.18, the subtraction is performed before the comparison. When _value exceeds the balance, the subtraction underflows, resulting in a very large positive number that still passes the `>=0` check.

Therefore, the patch introduces address validation but leaves the arithmetic vulnerability unresolved.

```
1  pragma solidity ^0.4.18;
2  contract Token {
3      mapping(address => uint) balances;
4      uint public totalSupply;
5      function Token(uint _initialSupply) {
6          balances[msg.sender] = totalSupply =
       _initialSupply;
7      }
8      function transfer(address _to, uint _value) public
       returns (bool) {
9      require(balances[msg.sender] >= _value); //PRE
       INVARIANT
10         require(balances[msg.sender] - _value >= 0);
11         balances[msg.sender] -= _value;
12         balances[_to] += _value;
13         return true;
14     }
15     function balanceOf(address _owner) public constant
       returns (uint balance) {
16         return balances[_owner];
17     }
18 }
```

Figure 5.24: token.sol contract with pre injection strategy applied in aggregated inference strategy CodeLlama-7B.

CodeLlama-7B's occasional success despite its generally weaker performance reinforces the observation made in Section 5.2.6. These exceptional cases demonstrate

that automated patch generation is inherently unpredictable and context-sensitive. These findings highlight the context-dependent nature of automated patch generation and suggest to explore multi-model strategies that can adapt to different vulnerability characteristics.

```
1  pragma solidity ^0.4.18;
2  contract Token {
3      mapping(address => uint) balances;
4      uint public totalSupply;
5      function Token(uint _initialSupply) {
6          balances[msg.sender] = totalSupply =
       _initialSupply;
7      }
8      function transfer(address _to, uint _value) public
       returns (bool) {
9      require(_to!=0x0);       //PRE INVARIANT
10         require(balances[msg.sender] - _value >= 0);
11         balances[msg.sender] -= _value;
12         balances[_to] += _value;
13         return true;
14     }
15     function balanceOf(address _owner) public constant
       returns (uint balance) {
16         return balances[_owner];
17     }
18 }
```

Figure 5.25: token.sol contract with pre injection strategy applied in aggregated inference strategy FLAMES20K.

Chapter 6

Discussion

In this chapter, we reflect on the implications of our experimental results. In Section 6.1, we analyze the robustness, accuracy, and human effort involved in using the FLAMES models. In Section 6.2, we discuss potential threats to validity, considering limitations in our experimental setup.

6.1 Robustness, Accuracy and Human Effort

An important aspect to consider when evaluating the performance of the models is the multi-dimensional trade-off between robustness, accuracy, and the human effort required to review the output.

As shown in Section 5.1.2, FLAMES models do not achieve perfect compilation rates. However, since the compilation verification process is fully automated, the practical value of the tool remains high for invariants that compile successfully, even when some attempts fail. In this context, robustness should be assessed not solely by success rate, but by the model's ability to produce valid, deployable invariants when compilation succeeds. The tool's robustness lies in its capacity to fail gracefully. Unsuccessful attempts are automatically filtered out without requiring human intervention.

The generated invariants vary significantly across models and inference strategies, creating important implications for both accuracy and review effort. FLAMES100K demonstrates the highest overall correctness, producing fewer semantically vacuous fallbacks as described in Section 5.2.6 and Section 5.2.6, which translates directly into reduced human review burden. This superior performance is further evidenced by FLAMES100K generating the highest number of matching patches and achieving the best overall results across evaluation metrics as shown in Section 5.2. FLAMES20K shows moderate performance with a correspondingly intermediate number of matching patches, while CodeLlama-7B tends to rely more heavily on

conservative outputs and produces the fewest matching patches, indicating lower semantic understanding of contract logic. This progression clearly demonstrates how model sophistication directly correlates with both the quantity and quality of deployable invariants, establishing a clear performance hierarchy that practitioners must consider when balancing computational resources against output quality.

The human effort required to review and approve patches remains manageable for cases that compile successfully, as reviewers only need to inspect invariants that the model has synthesized rather than manually verifying compilation itself. However, the effort required scales inversely with the chosen model. The automated filtering of non-compilable outputs ensures that human effort is never wasted on syntactically invalid invariants, allowing practitioners to focus their limited review time on semantically meaningful invariants rather than basic syntactic validation.

This multi-dimensional trade-off reveals that the synthesis pipeline's practical value depends heavily on deployment context and organizational constraints. Even with partial compilation success, the tool provides significant advantages by reducing the overall time and expertise required to generate security-relevant contract invariants. In this way, it maintains flexibility in how organizations balance robustness, accuracy, and review effort according to their specific needs and resources.

6.2 Threats to Validity

This study provides empirical evidence for the effectiveness of FLAMES synthesized invariants in improving the security of smart contracts. The evaluation was conducted on a curated and widely adopted benchmark dataset derived from SmartBugs Curated [38]. It offers a realistic yet controlled setting for reproducible experiments. The use of fully automated pipelines allowed for consistent testing of both compilability and vulnerability prevention across hundreds of real contracts.

Nevertheless, a few threats to validity remain, especially with regard to the security validation pipeline.

First, the experiments are limited to smart contracts written in Solidity version 4.X. This excludes newer versions and alternative programming languages. These versions may introduce language-specific features or behaviors that are not captured in this evaluation.

Second, each contract in the evaluation dataset contains only a single vulnerability. While this simplification facilitates controlled testing and analysis, it does not reflect the complexity of real-world contracts. Real-world contracts often contain multiple vulnerabilities that may interact with each other.

Lastly, the validity of the comparative analysis is limited by the fact that CodeLlama-7B was used as the sole baseline model. This limitation may reduce

the robustness of the conclusions. Performance differences could vary if alternative or more diverse baseline models were considered.

Despite these considerations, the evaluation framework and results presented in this study offer valuable insights into the potential of invariant synthesis as a strategy for smart contract hardening. The experimental setup is reproducible and extensible. In this way, it provides a solid foundation for future research to explore more diverse settings and richer vulnerability profiles.

Chapter 7

Conclusion and Future Work

This chapter serves three primary purposes: to assess the extent to which our research objectives have been achieved, to identify promising avenues for future work, and to reflect on the broader implications. Section 7.1 addresses the fulfillment of our goals, Section 7.2 outlines potential research directions and Section 7.3 provides critical reflections on the research journey.

7.1 Conclusion

The purpose of this thesis was to validate automatically synthesized invariants and evaluate the ability of the FLAMES tool to produce compilable and effective invariants for patching vulnerabilities in real smart contracts. This evaluation was conducted through two comprehensive automated pipelines. Both FLAMES models were systematically tested and compared against CodeLlama-7B to assess their relative performance.

The experimental results demonstrate that the FLAMES tool successfully produces deployable invariants that surpass off-the-shelf models in vulnerability-preventiveness. FLAMES20K achieved a compilability rate of 90.5% after invariant injection, while FLAMES100K reached 92%. More importantly, smart contracts were effectively patched against known vulnerabilities, with both models producing a substantial number of matching patches. Both FLAMES models consistently showed superior performance compared to the CodeLlama-7B baseline model across all evaluation metrics. Generating security invariants that revert attack transactions prevents the deployment of exploitable contracts. Furthermore, if included in automated contract repair pipelines, it helps patching vulnerabilities early in the development cycle mitigating potential financial losses and security breaches. Such preventive measures are crucial in the blockchain ecosystem, where deployed contracts are immutable and vulnerabilities can lead to significant economic damage.

7.2 Future Work

Several promising research directions emerge from the validation of synthesized invariants discussed in this thesis.

Firstly, the current work could benefit from a more comprehensive comparative analysis by evaluating additional state-of-the-art models alongside FLAMES. Specifically, testing models such as ChatGPT-4o or Claude Sonnet 4 would significantly enhance the quality of the comparison presented in this work. Currently, the only baseline comparison has been conducted against CodeLlama-7B. It limits the scope of our evaluation and may not reflect the full landscape of available capabilities.

Secondly, a promising avenue for improvement involves utilizing a larger and more complex dataset for the validation phase described in Section 4.2.2. The current study employed the 4.X Solidity version dataset from SB-Heist, which is relatively small and contains contracts of limited complexity. Testing the capacity of FLAMES models to produce valid invariants on a more challenging and semantically richer dataset would provide valuable insights into the scalability and robustness of the approach.

Finally, there is significant potential in validating contracts containing multiple or different types of vulnerabilities. In the current work, the models generate patches for contracts with vulnerabilities described in Section 2.3. Each contract contains only a single type of vulnerability. Extending this to contracts with multiple vulnerabilities of different kinds would be both intellectually interesting and practically valuable. Real-world smart contracts often exhibit interconnected security issues that require comprehensive analysis and remediation strategies.

7.3 Reflections

While the FLAMES tool represents a significant advancement in automated smart contract vulnerability patching, this research journey has revealed both the immense potential and inherent challenges of applying artificial intelligence techniques to blockchain security.

The implications of this research extend far beyond the technical domain. In alignment with the UN SDGs, the development of more reliable smart contract security tools could substantially impact SDGs **8** (decent work and economic growth), **9** (industry, innovation, and infrastructure), and **10** (reduced inequalities). By enhancing the security and trustworthiness of blockchain technology, automated security tools like FLAMES have the potential to democratize access to decentralized financial services. However, realizing this democratization potential depends critically on the continued development of robust, accessible, and reliable security frameworks.

This research also illuminates a fundamental tension between automation and human oversight in security-critical systems. While FLAMES successfully demonstrates the feasibility of automated patch generation, the essential role of human validation and understanding of generated invariants cannot be overlooked. This delicate balance between automation and human expertise will likely shape the future trajectory of blockchain security tools, ensuring that technological advancement serves both efficiency and safety in equal measure.

Appendix A

Additional outliers

In this appendix, we present additional outlier cases that were not included in the main analysis to avoid redundancy.

A.1 Additional `require(false)` cases

Among the tested models, only FLAMES20K produces the `require(false);` invariant for the `dos_simple.sol` contract in the post-injection scenario, as demonstrated in Figure A.1. This behavior is consistent across both inference strategies employed.

Another case where the `require(false);` invariant appears is in the `incorrect_constructor_name1.sol` contract. All models generate the `require(false);` invariant in the pre injection scenario across both inference strategies, with the exception of FLAMES100K when using the aggregated inference strategy, as shown in Figure A.2.

This demonstrates the capability of the FLAMES100K model with aggregated inference strategy to produce valid invariants by leveraging information from previous steps in the inference process.

```
1  pragma solidity ^0.4.25;
2  contract DosOneFunc {
3      address[] listAddresses;
4      function ifillArray() public returns (bool){
5          if(listAddresses.length<1500) {
6              for(uint i=0;i<350;i++) {
7                  listAddresses.push(msg.sender);
8              }
9              return true;
10         } else {
11             listAddresses = new address[](0);
12             return false;
13         }
14 require(false);      //POST INVARIANT
15     }
16 }
```

Figure A.1: dos_simple.sol contract with post injection strategy applied in FLAMES20K.

```
1  pragma solidity ^0.4.24;
2  contract Missing{
3  require(msg.sender==tx.origin);      //PRE INVARIANT
4      address private owner;
5      modifier onlyowner {
6          require(msg.sender==owner);
7          _;
8      }
9      function IamMissing()
10         public
11     {
12         owner = msg.sender;
13     }
14     function () payable {}
15     function withdraw()
16         public
17         onlyowner
18     {
19        owner.transfer(this.balance);
20     }
21 }
```

Figure A.2: incorrect_constructor_name1.sol contract with pre injection strategy applied in aggregated inference strategy FLAMES100K.

Bibliography

[1] Nick Szabo. «Formalizing and securing relationships on public networks». In: *First Monday* 2.9 (1997) (cit. on p. 1).

[2] Nick Szabo. *Smart Contracts: Building Blocks for Digital Markets*. `http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html`. 1996 (cit. on pp. 1, 11).

[3] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger*. Tech. rep. Ethereum Foundation, 2014 (cit. on p. 1).

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. «A Survey of Attacks on Ethereum Smart Contracts (SoK)». In: *Principles of Security and Trust*. Ed. by Matteo Maffei and Mark Ryan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186. ISBN: 978-3-662-54455-6 (cit. on p. 1).

[5] Ethereum Foundation. *Hard Fork Completed*. Ethereum Foundation. 2016. URL: `https://blog.ethereum.org/2016/07/20/hard-fork-completed` (cit. on pp. 1, 6).

[6] Mojtaba Eshghie, Mikael Jafari, and Cyrille Artho. «From Creation to Exploitation: The Oracle Lifecycle». In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*. 2024, pp. 23–34. DOI: `10.1109/SANER-C62648.2024.00009` (cit. on pp. 1, 2, 11).

[7] Loi Luu, Duc-Hieu Chu, Hoi Olickel, Prateek Saxena, and Aquinas Hobor. «Making smart contracts smarter». In: *ACM Conference on Computer and Communications Security*. 2016, pp. 254–269 (cit. on p. 1).

[8] Christoph Torres, Marco Steichen, and Radu State. «The art of the scam: Demystifying honeypots in Ethereum smart contracts». In: *USENIX Security Symposium* (2021) (cit. on p. 1).

[9] Dan Grossman, Emina Torlak, and Xi Wang. «Online detection of effectively callback free objects with applications to smart contracts». In: *Proceedings of the ACM on Programming Languages*. Vol. 1. OOPSLA. 2017, pp. 1–28 (cit. on p. 2).

[10] Zhining Wu, Lei Song, Yi Zhang, and Jianjun Zhou. «An empirical analysis of real-world smart contract vulnerabilities: Lessons from 10 years of attacks». In: *IEEE Transactions on Software Engineering* (2023) (cit. on p. 2).

[11] Mojtaba Eshghie, Gustav Andersson Kasche, Cyrille Artho, and Martin Monperrus. *SInDi: Semantic Invariant Differencing for Solidity Smart Contracts*. Manuscript (preprint). URN: urn:nbn:se:kth:diva-363066, OAI: oai:DiVA.org:kth-363066, DivA id: diva2:1956137. KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, May 2025 (cit. on p. 2).

[12] C. A. R. Hoare. «An axiomatic basis for computer programming». In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259 (cit. on p. 2).

[13] Robert W. Floyd. «Assigning Meanings to Programs». In: *Program Verification: Fundamental Issues in Computer Science*. Ed. by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. Dordrecht: Springer Netherlands, 1993, pp. 65–81. ISBN: 978-94-011-1793-7. DOI: 10.1007/978-94-011-1793-7_4. URL: https://doi.org/10.1007/978-94-011-1793-7_4 (cit. on p. 2).

[14] Ye Liu, Chengxuan Zhang, and Yi Li. *Automated Invariant Generation for Solidity Smart Contracts*. 2024. arXiv: 2401.00650 [cs.SE]. URL: https://arxiv.org/abs/2401.00650 (cit. on pp. 2, 15, 16).

[15] Zhiyang Chen, Ye Liu, Sidi Mohamed Beillahi, Yi Li, and Fan Long. «Demystifying Invariant Effectiveness for Securing Smart Contracts». In: *Proceedings of ACM/IEEE FSE 2024* (2024) (cit. on p. 2).

[16] Gustav Andersson Kasche. *InvPurge: Reducing Invariant Noise with Logical Differencing*. 2024 (cit. on p. 2).

[17] Yue Wang, Lin Liu, Chao Zhou, and Ming Zhang. «An Empirical Study of Vulnerabilities in Real-world Ethereum Smart Contracts». In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2023, pp. 233–244 (cit. on p. 2).

[18] Anna Håkansson. «Portal of Research Methods and Methodologies for Research Projects and Degree Projects». In: *The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing*. CSREA Press, 2013, pp. 67–73 (cit. on p. 3).

[19] Gabriele Morello. *FLAMES: Fine-tuned Large Language Model for Invariant Synthesis*. 2024 (cit. on pp. 5, 15, 16).

[20] Gabriele Morello, Mojtaba Eshghie, Sofia Bobadilla, and Martin Monperrus. *DISL: Fueling Research with A Large Dataset of Solidity Smart Contracts*. 2024. arXiv: 2403.16861 [cs.SE]. URL: https://arxiv.org/abs/2403.16861 (cit. on pp. 5, 6, 19).

[21] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: 2308.12950 [cs.CL]. URL: https://arxiv.org/abs/2308.12950 (cit. on p. 5).

[22] Wanrong Zhu, Zhiting Hu, and Eric Xing. «Text Infilling». In: *arXiv preprint arXiv:1901.00158* (2019). URL: https://arxiv.org/abs/1901.00158 (cit. on p. 5).

[23] Mojtaba Eshghie, Viktor Åryd, Cyrille Artho, and Martin Monperrus. *SoliDiffy: AST Differencing for Solidity Smart Contracts*. 2025. arXiv: 2411.07718 [cs.SE]. URL: https://arxiv.org/abs/2411.07718 (cit. on p. 5).

[24] Solidity Team. *Solidity v0.8.0 Breaking Changes*. https://docs.soliditylang.org/en/latest/080-breaking-changes.html. 2020. URL: https://docs.soliditylang.org/en/latest/080-breaking-changes.html (cit. on p. 7).

[25] Mojtaba Eshghie, Wolfgang Ahrendt, Cyrille Artho, Thomas Troels Hildebrandt, and Gerardo Schneider. *Formalizing Smart Contract Design Patterns with DCR Graphs*. 2025. (Visited on 06/02/2025) (cit. on pp. 11, 17).

[26] Mojtaba Eshghie, Wolfgang Ahrendt, Cyrille Artho, Thomas Troels Hildebrandt, and Gerardo Schneider. «Capturing Smart Contract Design with DCR Graphs». In: *Software Engineering and Formal Methods*. Ed. by Carla Ferreira and Tim A. C. Willemse. Cham: Springer Nature Switzerland, 2023, pp. 106–125. ISBN: 978-3-031-47115-5 (cit. on pp. 11, 17).

[27] Hadis Rezaei, Mojtaba Eshghie, Karl Anderesson, and Francesco Palmieri. *SoK: Root Cause of $1 Billion Loss in Smart Contract Real-World Attacks via a Systematic Literature Review of Vulnerabilities*. 2025. arXiv: 2507.20175 [cs.CR]. URL: https://arxiv.org/abs/2507.20175 (cit. on pp. 11, 13, 14).

[28] James C. King. «Symbolic execution and program testing». In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: https://doi.org/10.1145/360248.360252 (cit. on p. 12).

[29] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. «A Survey of Symbolic Execution Techniques». In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300. DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657 (cit. on p. 12).

[30] Shang-Wei Lin, Palina Tolmach, Ye Liu, and Yi Li. «SolSEE: a source-level symbolic execution engine for solidity». In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 1687–1691. ISBN: 9781450394130. DOI: 10.1145/3540250.3558923. URL: https://doi.org/10.1145/3540250.3558923 (cit. on p. 12).

[31] Shihao Xia, Mengting He, Shuai Shao, Tingting Yu, Yiying Zhang, and Linhai Song. *SymGPT: Auditing Smart Contracts via Combining Symbolic Execution with Large Language Models*. 2025. arXiv: 2502.07644 [cs.AI]. URL: https://arxiv.org/abs/2502.07644 (cit. on p. 12).

[32] Kunjian Song, Nedas Matulevicius, Eddie B. de Lima Filho, and Lucas C. Cordeiro. «ESBMC-Solidity: An SMT-Based Model Checker for Solidity Smart Contracts». In: *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2022, pp. 65–69. DOI: 10.1145/3510454.3516855 (cit. on p. 12).

[33] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. «SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution». In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1361–1378. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/so (cit. on p. 13).

[34] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. «SmartPulse: Automated Checking of Temporal Properties in Smart Contracts». In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 555–571. DOI: 10.1109/SP40001.2021.00085 (cit. on p. 13).

[35] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. «T2: Temporal Property Verification». In: *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 387–393. ISBN: 9783662496732. DOI: 10.1007/978-3-662-49674-9_22. URL: https://doi.org/10.1007/978-3-662-49674-9_22 (cit. on p. 13).

[36] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. «Temporal Properties of Smart Contracts». In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*. Limassol, Cyprus: Springer-Verlag, 2018, pp. 323–338. ISBN: 978-3-030-03426-9. DOI: 10.1007/978-3-030-03427-6_25. URL: https://doi.org/10.1007/978-3-030-03427-6_25 (cit. on p. 13).

[37] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. «VerX: Safety Verification of Smart Contracts». In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1661–1677. DOI: 10.1109/SP40000.2020.00024 (cit. on p. 14).

[38] Monika di Angelo, Thomas Durieux, João F. Ferreira, and Gernot Salzer. *SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts*. 2023. arXiv: 2306.05057 [cs.CR]. URL: https://arxiv.org/abs/2306.05057 (cit. on pp. 14, 62).

[39] Mojtaba Eshghie and Cyrille Artho. «Oracle-Guided Vulnerability Diversity and Exploit Synthesis of Smart Contracts Using LLMs». In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 2240–2248. ISBN: 9798400712487. DOI: 10.1145/3691620.3695292. URL: https://doi.org/10.1145/3691620.3695292 (cit. on p. 14).

[40] Sofia Bobadilla, Monica Jin, and Martin Monperrus. *Do Automated Fixes Truly Mitigate Smart Contract Exploits?* 2025. arXiv: 2501.04600 [cs.SE]. URL: https://arxiv.org/abs/2501.04600 (cit. on pp. 14, 24, 28).

[41] Zhiyuan Peng, Xin Yin, Rui Qian, Peiqin Lin, Yongkang Liu, Chenhao Ying, and Yuan Luo. *SolEval: Benchmarking Large Language Models for Repository-level Solidity Code Generation*. 2025. arXiv: 2502.18793 [cs.SE]. URL: https://arxiv.org/abs/2502.18793 (cit. on p. 15).

[42] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. «PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation». In: *Proceedings 2025 Network and Distributed System Security Symposium*. NDSS 2025. Internet Society, 2025. DOI: 10.14722/ndss.2025.241357. URL: http://dx.doi.org/10.14722/ndss.2025.241357 (cit. on p. 15).

[43] Sally Junsong Wang, Kexin Pei, and Junfeng Yang. «SmartInv: Multimodal Learning for Smart Contract Invariant Inference». In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024, pp. 2217–2235. DOI: 10.1109/SP54263.2024.00126 (cit. on pp. 15, 16).

[44] Hao Luo, Yuhao Lin, Xiao Yan, Xintong Hu, Yuxiang Wang, Qiming Zeng, Hao Wang, and Jiawei Jiang. *Guiding LLM-based Smart Contract Generation with Finite State Machine*. 2025. arXiv: 2505.08542 [cs.AI]. URL: https://arxiv.org/abs/2505.08542 (cit. on p. 16).

[45] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. «VeriSolid: Correct-by-Design Smart Contracts for Ethereum». In: *Financial Cryptography and Data Security*. Ed. by Ian Goldberg and Tyler Moore. Cham: Springer International Publishing, 2019, pp. 446–465. ISBN: 978-3-030-32101-7 (cit. on p. 16).

[46] Lingxiang Wang, Hainan Zhang, Qinnan Zhang, Ziwei Wang, Hongwei Zheng, Jin Dong, and Zhiming Zheng. *CodeBC: A More Secure Large Language Model for Smart Contract Code Generation in Blockchain*. 2025. arXiv: 2504.21043 [cs.CR]. URL: https://arxiv.org/abs/2504.21043 (cit. on p. 16).

[47] Ao Li, Jemin Andrew Choi, and Fan Long. «Securing smart contract with runtime validation». In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 438–453. ISBN: 9781450376136. DOI: 10.1145/3385412.3385982. URL: https://doi.org/10.1145/3385412.3385982 (cit. on p. 16).

[48] Ákos Hajdu and Dejan Jovanovic. «solc-verify: A Modular Verifier for Solidity Smart Contracts». In: *Verified Software. Theories, Tools, and Experiments*. Springer International Publishing, 2020, pp. 161–179. ISBN: 9783030416003. DOI: 10.1007/978-3-030-41600-3_11. URL: http://dx.doi.org/10.1007/978-3-030-41600-3_11 (cit. on p. 17).

[49] Everett Hildenbrandt et al. «KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine». In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 2018, pp. 204–217. DOI: 10.1109/CSF.2018.00022 (cit. on p. 17).

[50] Evgeniy Shishkin. *Debugging Smart Contract's Business Logic Using Symbolic Model-Checking*. 2018. arXiv: 1812.00619 [cs.LO]. URL: https://arxiv.org/abs/1812.00619 (cit. on p. 17).

[51] Mojtaba Eshghie, Cyrille Artho, Hans Stammler, Wolfgang Ahrendt, Thomas Hildebrandt, and Gerardo Schneider. «HighGuard: Cross-Chain Business Logic Monitoring of Smart Contracts». In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 2378–2381. ISBN: 9798400712487. DOI: 10.1145/3691620.3695356. URL: https://doi.org/10.1145/3691620.3695356 (cit. on p. 17).

[52] Etherscan Team. *Etherscan Developer APIs*. `https://docs.etherscan.io/api-endpoints/contracts`. 2023 (cit. on p. 27).