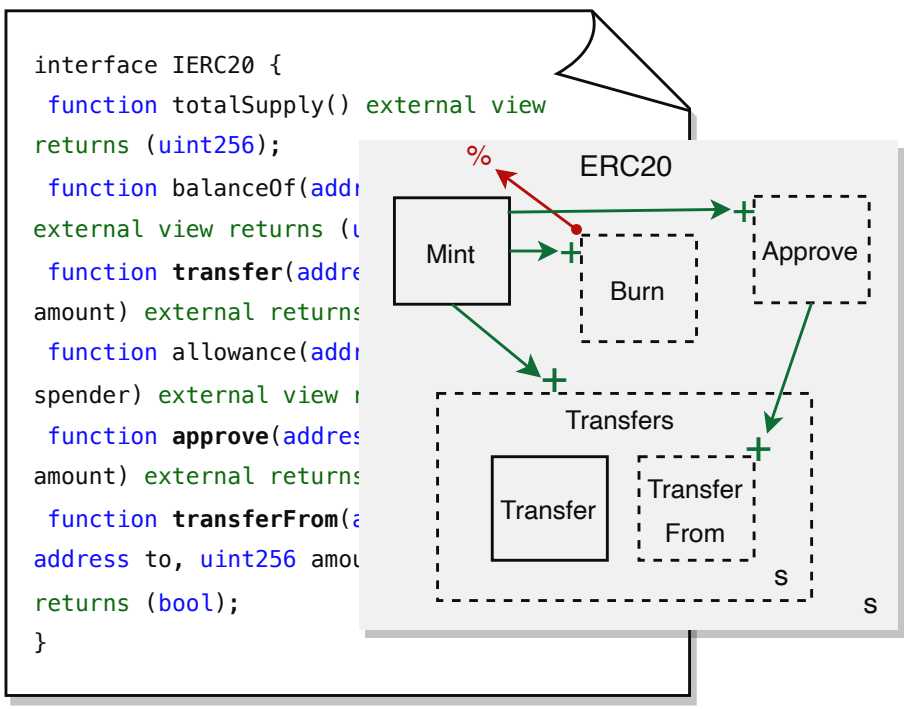
Doctoral Thesis in Computer Science

Securing Smart Contracts Against Business Logic Flaws

MOJTABA ESHGHIE

KTH ROYAL INSTITUTE OF TECHNOLOGY



Securing Smart Contracts Against Business Logic Flaws

MOJTABA ESHGHIE

Academic Dissertation which, with due permission of the KTH Royal Institute of Technology, is submitted for public defence for the Degree of Doctor of Philosophy on Wednesday the 11th of June 2025, at 9:00 a.m. in Kollegiesalen, Brinellvägen 6, Stockholm.

Doctoral Thesis in Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2025

TRITA-EECS-AVL-2025:56
ISBN 978-91-8106-288-5

Printed by: Universitetsservice US-AB, Sweden 2025

Abstract

Blockchain smart contracts empower decentralized applications (DApps) by enabling trustless, transparent, and immutable transactions. However, this immutability means that flaws in design or implementation lead to irreversible and costly security breaches. Therefore, rigorous verification and analysis of smart contracts before deployment is crucial. Furthermore, since not all vulnerabilities are detectable pre-deployment, monitoring after deployment is equally critical.

One particularly challenging yet understudied class of vulnerabilities in smart contracts is business logic flaws (BLFs), which occur when the implemented logic deviates subtly but critically from the intended behavior. BLFs are challenging to detect because they often require a deep understanding of the intended business rules rather than merely code-level analysis. This class of vulnerabilities has largely been overlooked in existing literature. This thesis introduces a comprehensive framework to address vulnerabilities across the entire DApp security lifecycle from specification and pre-deployment analysis to post-deployment monitoring with an emphasis on detecting, preventing, and mitigating BLFs.

At its core, our work formalizes high-level business logic of contracts using Dynamic Condition Response (DCR) graphs. We categorize design patterns into low-level platform/implementation specific and high-level business logic design patterns. We formalize the high-level patterns such as time-based constraints and action dependencies into precise graphical formal models. These models provide developers with a clear visual blueprint of intended contract workflows. Formalized contracts not only facilitate automated verification and analysis but also lay the groundwork for our "model to mitigate" approach. In this methodology, smart contract functions are mapped to DCR activities and its logic is modeled using DCR relations to expose design flaws by making implicit assumptions about the code explicit.

Complementing our design-phase contributions, we propose an off-chain monitoring tool that observes on-chain transactions. By mapping each transaction against the DCR-specified logic, this tool detects deviations from intended behavior without instrumenting the deployed contract. Furthermore, our method effectively detects complex cross-chain attacks and violations of the specified behavior to secure the increasingly popular cross-chain DApps.

We address the lack of concrete exploit scenarios for business logic vulnerabilities using Large Language Models (LLMs) to inject realistic BLFs into contracts and synthesize corresponding exploits. Our approach uses DCR-based formal specifications as guidance, ensuring that synthesized vulnerabilities are both representative and realistic.

Recognizing the need for rigorous invariants to enforce security, we develop a semantic invariant differencing tool that given two smart contract `require/assert` statements it produces a verdict of which one is stronger. Our experiments prove its effectiveness in filtering out redundant or weak invariants to enhance the usefulness of dynamically mined invariants. In parallel, acknowledging the pivotal role of blockchain oracle security, we propose

an oracle data lifecycle model spanning from the creation of data to its deprecation. We systematically identify vulnerabilities at each stage of this model and survey mitigation strategies.

Collectively, these contributions provide an end-to-end approach to smart contract security by bridging formal design and analysis, dynamic monitoring, and invariant refinement to build more resilient decentralized applications.

Keywords: Smart Contract, Vulnerability, Formal Specification, DCR Graphs, Business Logic Flaws

Sammanfattning

Blockchainbaserade smarta kontrakt ger stöd för decentraliserade applikationer (DApps) genom att möjliggöra tillitslösa, transparenta och oföränderliga transaktioner. Emellertid innebär denna oföränderlighet att brister i design eller implementering leder till oåterkalleliga och kostsamma säkerhetsincidenter. Därför är rigorös verifiering och analys av smarta kontrakt innan driftsättning avgörande. Vidare kan inte alla sårbarheter upptäckas på förhand, varför övervakning efter driftsättning också är kritiskt.

En särskilt utmanande men tämligen outforskad klass av sårbarheter i smarta kontrakt är affärslogikbrister (BLF), som uppstår när den implementerade logiken avviker subtilt men kritiskt från det avsedda beteendet. BLF:er är svåra att upptäcka eftersom de ofta kräver en djup förståelse av de avsedda affärsreglerna snarare än enbart kodnivåanalys. Denna klass av sårbarheter har överlag förbisetts i befintlig litteratur. Den här avhandlingen introducerar ett omfattande ramverk för att hantera sårbarheter under hela säkerhetslivscykeln för DApps – från specifikation och analys före - till övervakning efter - driftsättning – med fokus på att upptäcka, förebygga och hindra BLF:er.

I grunden formaliserar vårt arbete affärslogik på hög nivå för kontrakt med hjälp av Dynamic Condition Response (DCR)-grafer. Vi kategoriserar designmönster som antingen lågnivå (plattforms- och implementeringsspecifika) eller högnivå (affärslogiska) designmönster. Vi formaliserar högnivåmönster, såsom tidsbaserade begränsningar och handlingsberoenden, med grafiska formella modeller. Dessa modeller ger utvecklare en tydlig visuell översikt över de avsedda kontraktsarbetsflödena. Formaliserade kontrakt underlättar inte bara automatiserad verifiering och analys utan lägger också grunden för vår "modellera för att hindra"-metod. I denna metod kopplas smarta kontraktsfunktioner till DCR-aktiviteter, och logiken modelleras med hjälp av DCR-relationer för att avslöja designfel genom att göra tidigare implicita antaganden om koden explicita.

Som komplement till våra bidrag i designfasen föreslår vi ett off-chain-övervakningsverktyg som observerar on-chain-transaktioner. Genom att koppla varje transaktion mot den DCR-specificerade logiken upptäcker detta verktyg avvikelser från det avsedda beteendet utan att instrumentera det driftsatta kontraktet. Dessutom kan vår metod effektivt upptäcka komplexa cross-chain-attacker och överträdelser av specificerade beteenden, vilket bidrar till att öka säkerheten för de allt mer populära cross-chain DApps. Vi tar itu med bristen på konkreta exploateringsscenarier för BLF:er genom att använda stora språkmodeller (LLM) för att injicera realistiska BLF:er i kontrakt och syntetisera motsvarande exploateringar. Vårt angreppssätt använder DCR-baserade formella specifikationer som vägledning, vilket säkerställer att de syntetiserade sårbarheterna är både representativa och realistiska.

Med insikten om behovet av rigorösa invarianter för att upprätthålla säkerhet utvecklar vi ett semantiskt verktyg för invariantdifferentiering som, givet två smarta kontraktsinvarianter, avgör vilken som är starkare. Våra experiment visar på verktygets effektivitet att filtrera bort redundanta eller

svaga invarianter och därigenom ge mer användbara dynamiskt utvunna invarianter. Därutöver, beaktandes vikten av blockchain-orakelsäkerhet, föreslår vi en modell för datalivscykeln för orakel som löper från dataskapande till dess utfasning. Vi identifierar systematiskt sårbarheter i varje steg av denna modell och kartlägger hindrande strategier.

Sammanfattningsvis erbjuder dessa bidrag ett helhetsangreppssätt för smart kontraktssäkerhet genom att koppla ihop formell design och analys, dynamisk övervakning samt förfining av invarianter – allt för att bygga mer motståndskraftiga decentraliserade applikationer.

List of Papers

The list of papers that are included in the thesis in chronological order:

I ***Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning***
Mojtaba Eshghie, Cyrille Artho, Dilian Gurov
In: Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering (EASE '21). Association for Computing Machinery, New York, NY, USA, pages 305–312, June 2021. DOI: 10.1145/3463274.3463348.

II ***Capturing Smart Contract Design with DCR Graphs***
Mojtaba Eshghie, Wolfgang Ahrendt, Cyrille Artho, Thomas Troels Hildebrandt, Gerardo Schneider
In: Software Engineering and Formal Methods. Springer Nature Switzerland, Cham, pages 106–125, 2023. DOI: 10.1007/978-3-031-47115-5_ 7.

III ***From Creation to Exploitation: The Oracle Lifecycle***
Mojtaba Eshghie, Mikael Jafari, Cyrille Artho
In: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C). IEEE, March 2024, pages 23–34. DOI: 10.1109/SANER-C62648.2024.00009.

IV ***HighGuard: Cross-Chain Business Logic Monitoring of Smart Contracts***
Mojtaba Eshghie, Cyrille Artho, Hans Stammler, Wolfgang Ahrendt, Thomas Troels Hildebrandt, Gerardo Schneider
In: 2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, October 2024, pages 2378–2381. DOI: 10.1145/3691620.3695356.

V ***Oracle-Guided Vulnerability Diversity and Exploit Synthesis of Smart Contracts Using LLMs***
Mojtaba Eshghie, Cyrille Artho
In: 2024 39th IEEE/ACM International Conference on Automated Software

Engineering (ASE). IEEE, October 2024, pages 2240–2244. ISSN: 2643-1572. DOI: 10.1145/3691620.369529

VI ***Formalizing Smart Contract Design Patterns with DCR Graphs***
Mojtaba Eshghie, Wolfgang Ahrendt, Cyrille Artho, Thomas Troels Hildebrandt, Gerardo Schneider
Under review in the International Journal on Software and Systems Modeling (SoSyM).

VII ***Model to Mitigate: Using DCR Graphs to Prevent Vulnerabilities in Smart Contracts***
Mojtaba Eshghie, Wolfgang Ahrendt, Cyrille Artho, Thomas Troels Hildebrandt, Gerardo Schneider
Under review in the Journal of Logical and Algebraic Methods in Programming (JLAMP).

VIII ***SInDi: Semantic Invariant Differencing For Solidity Smart Contracts***
Mojtaba Eshghie, Gustav Andersson Kasche, Cyrille Artho, Martin Monperrus

Other contributions by the author not included in the thesis in chronological order:

I ***CircleChain: Tokenizing Products with a Role-Based Scheme for a Circular Economy***
Mojtaba Eshghie, Li Quan, Gustav Andersson Kasche, Filip Jacobson, Cosimo Bassi, Cyrille Artho
arXiv preprint, DOI: 10.48550/arXiv.2205.11212.

II ***DISL: Fueling Research with A Large Dataset of Solidity Smart Contracts***
Gabriele Morello, Mojtaba Eshghie, Sofia Bobadilla, Martin Monperrus
arXiv preprint, DOI: 10.48550/arXiv.2403.16861.

III ***SoliDiffy: AST Differencing for Solidity Smart Contracts***
Mojtaba Eshghie, Viktor Åryd, Cyrille Artho, Martin Monperrus
arXiv preprint, DOI: 10.48550/arXiv.2411.07718.

Acknowledgement

I recommend that the Statue of Liberty on the East Coast be supplemented by a Statue of Responsibility on the West Coast.

– Viktor E. Frankl

First and foremost, my gratitude goes to my supervisor, Cyrille, whose strong guidance was matched only by his genuine kindness. His mentorship helped me grow both professionally and personally. My appreciation extends also to my second supervisor, Dilian, with whom I had the pleasure of collaborating on my first publication (DYNAMIT [1]).

The research stay at Chalmers University was pivotal for my academic growth. Collaboration with Wolfgang Ahrendt and Gerardo Schneider introduced me to formal specification of smart contracts and runtime verification that shaped the trajectory of my research. Their insights inspired some of the works presented in this thesis.

Meeting Thomas Hildebrandt at the University of Copenhagen marked another important point. My gratitude also extends to Morten Marquard for his technical assistance, and to DCR Solutions for providing the academic version of their software I used in my experiments.

Collaborating with Martin Monperrus and participating in the weekly meetings—*Minutes on ML on Code* and *Minutes on Smart Contracts*—helped me dive deeper into my main research interest: AI for software engineering. Martin's generosity with his time, resources, and enthusiasm for research has been motivating.

During the second half of my PhD, I supervised eleven master's and two bachelor's students whose projects challenged me to grow as a teacher and researcher. I appreciate the trust they placed in me as their supervisor, and I look forward to seeing them succeed in their careers.

I am deeply grateful to my loved ones–my partner, whose support made a difference in this journey; my family, especially my mother, whose perseverance I carry; and my brother, Mahan, whose humor brought joy to our calls.

Finally, I thank KTH Royal Institute of Technology, my academic home for the past four and a half years for providing the resources to pursue my passion.

Acronyms

List of commonly used acronyms:

BLF	Business Logic Flaw
CVL	Certora Verification Language
DAO	Decentralized Autonomous Organization
DApp	Decentralized Application
DCR	Dynamic Condition Response
DeFi	Decentralized Finance
DSL	Domain Specific Language
EVM	Ethereum Virtual Machine
FSM	Finite State Machine
LLM	Large Language Model
LTL	Linear Temporal Logic
ML	Machine Learning
SC	Smart Contract

Contents

Part I

Thesis

1

Introduction

Blockchain technology has transformed decentralized systems from theoretical constructs into practical infrastructures by enabling trustless and transparent execution of transactions without central authorities [2, 3, 4]. At the core of this transformation are *smart contracts*, which are self-executing programs that enforce agreements without intermediaries. These contracts power decentralized applications (DApps), facilitating financial transactions, governance mechanisms, and digital asset management.

However, smart contracts also introduce new security challenges. Unlike traditional software, deployed smart contracts are immutable, meaning that vulnerabilities cannot be easily patched once discovered. The high financial stakes and the irreversible nature of transactions on blockchain make security incidents challenging, often resulting in substantial financial losses [5]. The need for robust security mechanisms throughout the entire lifecycle of DApps has become evident as high-profile attacks expose weaknesses in smart contract design and implementation.

This thesis delves into the challenges of decentralized application security. In the upcoming sections, we explore the motivation behind current security practices, review notable incidents, and discuss the frameworks that inform our approach to DApp security.

1.1 Context and Motivation

Smart contracts, as reactive programs running on blockchain virtual machines, have been conceptually discussed since the 1990s [4], but only rose to prominence with platforms like Ethereum [6] (launched 2015) that allowed complex programmable contracts. In the early days, security received relatively little attention [7]. Developers and users were infatuated with the potential of trustless automation, often overlooking the risks. Unlike traditional software, a deployed smart contract is immutable–it cannot be easily patched if a flaw is discovered–

which means any vulnerability can have permanent and costly consequences. This immutability, combined with the fact that contracts often hold or transfer valuable assets, set the stage for severe security incidents when bugs inevitably arise [5].

The wake-up call came in June 2016 with the DAO hack [8]. The DAO (Decentralized Autonomous Organization) was a landmark Ethereum smart contract intended as a crowd-managed investment fund [9]. A subtle coding flaw (a reentrancy vulnerability) in the DAO's withdrawal function allowed an attacker to repeatedly drain funds before the balance could update, ultimately siphoning roughly 3.6 million ETH. This exploit vividly demonstrated that even small oversights in the contract code could be catastrophic, undermining the safety of "code is law". The incident had a global impact on the ecosystem: Ethereum's community controversially executed a hard fork to reverse the theft (creating Ethereum Classic as a split from those who opposed reversal) [8, 10]. The DAO hack marked a turning point: smart contract security was thrust to the forefront, revealing the fragility of on-chain code and the high stakes involved when bugs occur.

Despite increased awareness post-DAO, vulnerabilities plagued smart contracts in subsequent years. In July 2017, a bug in the popular Parity multi-signature wallet contract led to another major breach [11]. An attacker exploited an unprotected function in Parity's wallet code, allowing them to seize ownership of the contract and steal over 150 000 ETH from several multi-sig wallets [11]. Merely four months later, in November 2017, Parity wallet users suffered a second disaster: a user accidentally triggered a flaw in Parity's library contract code, freezing about 150 million USD worth of Ether [12]. In this incident, no thief profited; instead, the funds became permanently locked due to a bug in the contract's self-destruct logic. The Parity incidents highlighted not only deliberate attacks but also how inadvertent actions or overlooked edge cases can wreak havoc.

Examining the early incidents reveals a pattern of recurring vulnerability types that have evolved alongside DApps. One of the most infamous is reentrancy, exemplified by the DAO hack [10]. A reentrancy allows an attacker's contract to call back into a vulnerable contract before the previous operation finishes [13], often enabling multiple withdrawals of funds. Reentrancy remains a critical risk to this day–many subsequent attacks (e.g., on DeFi lending platforms) have been variants of this issue [14], prompting developers to adopt defensive coding patterns (such as the Checks-Effects-Interactions design pattern [15]) to mitigate it.

Another common class of bugs is arithmetic overflow/underflow [16]. Early Solidity versions did not automatically guard against numeric overflow/underflow, leading to mistakes where computations could wrap around unexpectedly. While not as publicized as the DAO hack, the consequences have been severe in several cases. For example, in April 2018, attackers exploited an integer overflow in the BeautyChain (BEC) token contract's batch transfer function to create an astronomically large number of tokens out of thin air [17]. Similar overflow flaws were found in multiple ERC-20 tokens around that time, forcing exchanges to halt trading and pushing developers to use safer math libraries for token arithmetic.

Access control and authorization weaknesses form another key category [16, 18]. Smart contracts must explicitly specify which addresses or entities can perform privileged actions; any gap can be fatal. The Parity multi-sig hack in 2017 is a prime example: a wallet contract failed to restrict access to an internal initialization function, which allowed users to re-initialize the contract and make themselves the owner. Once in control, the attacker extracted all funds from the wallet contracts. In this case, the vulnerability was essentially an authentication bypass–a function that should have been protected was left public. The ecosystem's response was swift: projects rushed to audit their contracts for similar flaws, and developers became far more careful with function visibility and ownership logic. Nonetheless, access control bugs have continued to surface from exchange wallet breaches to DeFi governance attacks, proving that rigorous verification is necessary for any function that moves assets [19].

The rise of Decentralized Finance (DeFi) applications enabled on-chain financial services by enabling permissionless lending, borrowing, and trading without intermediaries. The rapid growth of these protocols [20] also introduced novel attack vectors. Attackers combined flash loans, oracle manipulation, and reentrancy to exploit protocols and steal hundreds of millions of USD [5, 14]. For example, in August 2021, an attacker exploited an input validation flaw in the Poly Network cross-chain bridge, managing to extract over 600 million in various cryptocurrencies—one of the largest hacks in DeFi history.

Business logic flaw (defined in Section 2.2) represents another subtle yet critical vulnerability class in smart contracts, arising when the implemented code deviates from the intended business rules. Unlike overt coding errors mentioned before, these flaws are embedded within the contract's logic, making them challenging to detect. A notable example is the 2024 Level Finance hack, where attackers exploited a flaw in the business logic which enabled repeated claims of referral rewards to siphon funds illicitly [21].

Financially, the losses are staggering–by recent estimates, attacks and bugs in smart contracts have cumulatively cost over 2.2 billion in 2024 [20]. Each high-profile exploit tends to spark a temporary loss of user confidence in decentralized platforms, sometimes triggering market dips or an exodus of liquidity from DeFi projects [14]. Trust is difficult to earn in a "code-is-law" environment; once shaken by a major hack, both users and regulators grow more skeptical of the technology.

1.2 Security Lifecycle of Decentralized Applications

In the aftermath of the mentioned incidents, the blockchain community recognized that more rigorous approaches were needed to prevent future disasters. Since then, security audits relying on manual expert effort and automated tools for smart contracts have been the pillar of defense against vulnerabilities. These audits typically aim to catch vulnerabilities after the contract is developed and before on-chain deployment (Stage 2 in Figure 1.1). In an audit, experienced pro-

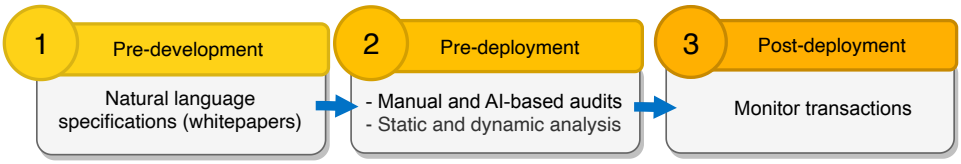


Figure 1.1: Security maintenance lifecycle of DApps

fessionals scrutinize the contract's source code, line by line, using both manual code review and automated analysis tools [7, 22, 23] to identify known vulnerability types. The auditors use a range of static [24, 25, 26] and dynamic [27, 28, 29] analysis and recently LLM-based analysis [30, 31, 32] methods. Audits often enforce best practices such as proper use of security design patterns [15, 33].

Ensuring security of decentralized applications involves all of the precautionary measures taken during designing (stage 1 in Figure 1.1), implementing (stage 2), and monitoring (stage 3) of these applications on blockchain. Figure 1.1 shows these three stages in summary. Current auditing techniques (manual and automated) mainly concern the second stage. Our contributions suggest a more holistic approach starting from stage 1.

Ethereum smart contract development involves a continuous security lifecycle encompassing the pre-development, pre-deployment, and post-deployment stages (Figure 1.1). Here, we review these stages to highlight how security measures are implemented over a DApp's lifetime.

1.2.1 Pre-Development Stage

In the early design phase, DApp designers rely on natural language (e.g., whitepapers or design documentations) to describe the intended protocol behavior and features of the protocol [34]. These documents serve as the *de facto* blueprint for development and audits. A clear, precise specification can prevent vulnerabilities by defining correct behavior. Writing a detailed specifications involves documenting trust assumptions, defining the contract's state machine, and enumerating potential misuse cases. Best practices include explicit security invariants, adversarial thinking, and formalization which are largely non-present in protocol whitepapers.

Ethereum Natural Language Specification Format (NatSpec) have been introduced as a more developer-friendly specification to put docuementation inside the source code comments [35]. However, NatSpec remains as a user-friendly commenting approach for developers, rather than a formal specification.

1.2.2 Pre-Deployment Stage

Manual auditing remains an effective approach for uncovering complex vulnerabilities such as business logic flaws. Automated tools excel at detecting generic issues

but struggle with protocol-specific vulnerabilities [10, 36]. Auditing is however constrained by time, expertise, and reliance on *assumptions* made by the auditors about the designer intentions.

Formal verification methods apply mathematical rigor to smart contracts to enhance security assurance at this stage, going beyond incomplete static or dynamic analysis. Formal verification relies on the contract's behavior specified in a formal logic used by theorem provers or model checkers to prove that the code satisfies the specifications. Formal methods, which originated in mission-critical domains such as aerospace, have been increasingly adopted in blockchain security [36]. For instance, frameworks such as Certora [37] and VeriSolid [38] allow auditors to define security properties and automatically verify them against contract logic. In practice, formal verification supplements traditional audits, as traditional automated/manual code review and testing methods—while useful—often fail to cover all edge cases. Formal verification techniques enable exhaustive exploration of contract states, reducing the likelihood of unexpected states slipping through this stage. These techniques depend greatly on the quality of specifications in the first stage.

1.2.3 Post-Deployment Stage

Post-deployment security relies on blockchain monitoring services (e.g., Forta, OpenZeppelin Defender) and emergency responses such as pause mechanisms [15]. To fix post-deployment issues, upgradeable contracts use proxy patterns [15, 39] which enables pointing to a new contract address for logic while using the original contract storage for data. However, upgrade mechanisms introduce new risks, such as administrative takeovers and improper storage layout.

1.3 The Rise of Business Logic Flaws

Unlike traditional security issues rooted in implementation errors, such as reentrancy, business logic flaws arise when the contract's encoded rules fail to enforce the intended workflow of its underlying business process. These flaws often stem from *incomplete* or *incorrect* assumptions about how different roles interact within the system, leading to unintended behaviors that adversaries can exploit (see Chapter 2). Attackers do not need to break cryptographic primitives or find low-level bugs; instead, they manipulate contract functionality to bypass safeguards, misappropriate funds, or violate protocol constraints. Given the prevalence of complex protocols such as decentralized finance (DeFi) applications and cross-chain protocols, where financial transactions depend on multi-step processes, business logic vulnerabilities have become an attractive target for attackers.

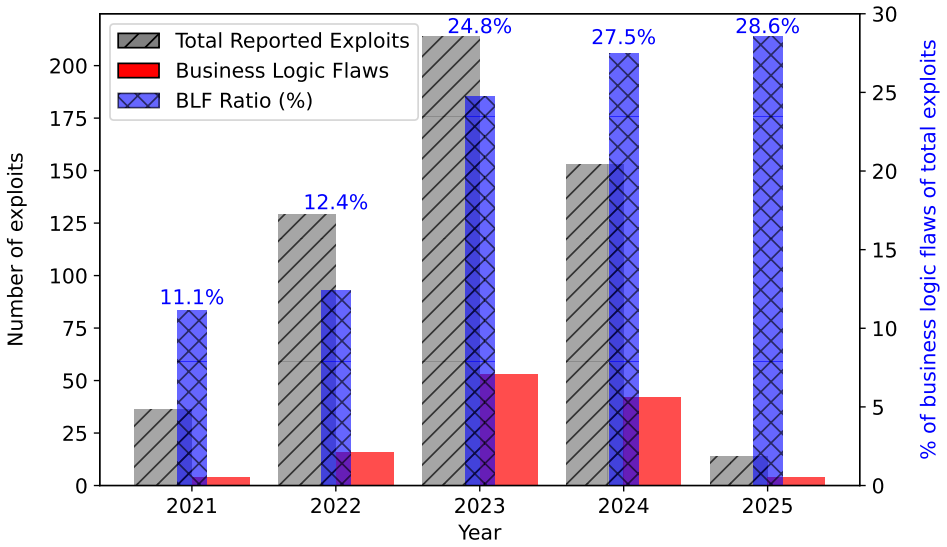


Figure 1.2: Emerging business logic flaw (BLF)-related incidents statistics 2021–January and Feburary 2025

1.3.1 Statistics of Business Logic-Related Incidents

To show the prevalence of BLF-related attacks, we rely on two sources: 1) "OWASP Smart Contract Top 10" [16] that relies on 1.42 billion USD lost across 149 documented incidents in 2024 and 2) DeFiHackLabs [19], an active repository containing a list of reproducible attacks and their exploit script updated regularly by 120 contributors (at the time of writing this thesis).

OWASP [16] lists the top ten vulnerability types by significance based on the amount of losses inflicted by the vulnerability type. According to their 2025 report, business logic vulnerabilities have risen from rank seven in 2023 report to rank three in the 2025 report.

DeFiHackLabs [19] contains 558 incidents at the time of writing this thesis. For each incident, the listing contains its vulnerability type, amount lost due to the attack (if available), and the exploit reproduction script. Figure 1.2 shows the proportion of incidents marked as business logic flaw in this listing compared to the total incidents reported for each year. As this figure shows, the ratio of business logic flaws to total exploits in this listing has steadily increased from 11.1% in 2021 to 27.5% in 2024 (and 28.6% for the limited information of the first two months of 2025).

Addressing the mentioned business logic issues requires a shift from purely code-level audits to process-aware verification approaches that formally model and monitor the correct execution of business workflows in the contract imple-

mentation.

1.3.2 Motivating Example: A $600 Million USD Cross-Chain Business Logic Attack

Smart contracts that facilitate cross-chain asset transfers rely on a well-defined sequence of operations to ensure that funds are securely locked, verified, and then released. In this section, we use the Poly Network incident in August 2021 to present an attack scenario involving business logic flaw.

Incident Description In the Poly Network incident [40, 41], an adversary exploited a vulnerability in the cross-chain messaging mechanism. The attacker subverted the protocol by overriding the list of authorized bookkeepers. As a result, the critical `Bookkeeper Verification` step was subverted, and the `Release Funds` event was executed without the necessary precondition being met. This flaw enabled the adversary to withdraw approximately 610 million USD worth of assets from the protocol.

Under normal operation, the protocol enforces that the release of funds is contingent on proper verification. However, by tampering with the verification mechanism, the attacker caused a deviation from the intended process, ultimately compromising the security of the asset transfer.

Business Process Behind the Protocol. The intended cross-chain process can be captured using a business process modeling formalism such as Dynamic Condition Response (DCR) graphs [42] (introduced in Section 2.3). In this model, each key function is represented as an event node, and directed relations enforce the proper ordering and constraints. In particular, a *condition relation* (denoted by $\rightarrow\bullet$ in the DCR model) is established from the `Bookkeeper Verification` event to the `Release Funds` event. This relation serves as a safeguard by ensuring that the `Release Funds` event can only be executed when the `Bookkeeper Verification` event has been performed and its associated guard condition is satisfied. Essentially, this condition relation prevents funds from being released unless all necessary verifications are in place.

In the Poly Network incident, the attacker manipulated the protocol so that this safeguard was subverted. Although the verification step was not properly executed, the protocol erroneously allowed the `Release Funds` event to trigger, leading to a security breach.

Process-Aware Detection of the Attack A monitoring tool that relies on the business process model proceeds as follows:

1. *Mapping Transactions to Activities:* Each incoming transaction is mapped to its corresponding activity in the DCR graph.

2. *Evaluating the Current State:* The tool maintains a state that captures whether each activity has been executed, is pending, or is excluded according to the DCR relations in the smart contract model (Section 2.3).

3. *Checking Execution Prerequisites:* Before allowing the execution of the `Release Funds` activity, the tool verifies that all condition relations (including the one from `Bookkeeper Verification`) are satisfied.

4. *Flagging Violations:* If the execution trace shows that `Release Funds` is triggered while the necessary verification step is missing, the tool flags this as a deviation from the intended business process.

In summary, by formalizing the intended cross-chain transfer process with a process-aware formalism and continuously comparing the observed transaction trace against the model execution trace, any deviation such as the unauthorized execution of `Release Funds` without prior verification can be detected. This example shows how monitoring based on business process models can serve as a mechanism for identifying attacks that exploit logic flaws in smart contract implementations.

1.4 Research Gaps

Despite the rapid evolution of smart contract security techniques, several interrelated research gaps persist that hinder the development of truly reliable decentralized applications. Our work addresses multiple of such gaps by integrating process-aware formalization, semantic invariant analysis, and holistic blockchain oracle security. We present four gaps in the literature and investigate nine research questions in Figure 1.3.

RQ1

How can a formalism from business process modeling be leveraged to systematically capture the intended business logic of decentralized applications and represent their high-level design patterns?

RQ2

How can we use modeling before the development of the contract to discover design flaws in decentralized applications?

RQ3

How does integrating a machine-readable oracle—via formal business logic specifications—into LLM prompting influence the automatic synthesis of (*vulnerability*, *exploit*) pairs for smart contracts compared to relying solely on natural language?

RQ4

Despite their syntactic variations across Solidity versions how can we perform quick and reliable semantic invariant differencing for smart contracts?

RQ5

How can we use semantic invariant analysis to refine automatically mined invariants to prepare them further for downstream analysis tasks for smart contracts?

RQ6

How well can a machine learning model act as a general monitor to detect vulnerabilities in Ethereum smart contracts without requiring the source code of the contract, instrumentation, or manual effort to define the vulnerability?

RQ7

How effectively does a monitoring framework based on the formal DCR specification of a smart contract, detects deviations in its business logic?

RQ8

How well can a DCR-based monitor detect the cross-chain business logic vulnerabilities?

RQ9

Does the data obtained form blockchain oracles flow through different stages and how does each stage affect the attack vectors on oracles?

Figure 1.3: Overview of all research questions addressed in this thesis.

1.4.1 Gap 1: Lack of High-Level, Process-Aware Specifications and Verification Tools for Smart Contracts

Many existing security techniques for smart contracts focus primarily on low-level vulnerabilities—such as reentrancy, arithmetic overflows, and unchecked external calls—using static analysis [43, 44, 45], formal verification [37, 38, 46], or monitoring [47, 48]. However, the majority of these approaches neglect the high-level business logic that underpins smart contract functionality. Traditional formalisms (e.g., finite state machines [49, 50]) have been useful for low-level analysis but often overlook high-level decentralized application workflows such as time-based patterns, role-based access control, event ordering, and temporal constraints.

Without a formal specification of the business logic, vulnerabilities rooted in design flaws may remain hidden until they are exploited post-deployment. This gap is critical because the intended behavior of a smart contract is not self-evident from the source code alone. Moreover, without a process-aware specification, it is difficult to develop automated verification or runtime monitoring tools that can detect deviations from the intended workflow. In our work, we propose leveraging a declarative formalism from business process modeling–DCR graphs–to specify smart contract behavior at a high level. By doing so, we can capture the intended design patterns in a platform-agnostic manner (**RQ1**) and provide a precise basis for automated exploit synthesis (**RQ3**).

Part of our work in this thesis that formalizes smart contract design using DCR graphs [15] demonstrates that this formalism provides a natural and declarative means to specify the intended business processes. Yet, the full potential of DCR models—especially for detecting design flaws before implementation—is largely untapped. In our *Model to Mitigate* approach, we show how pre-development DCR modeling can expose ambiguities (e.g., entangled access control checks) that might otherwise lead to vulnerabilities or inefficiencies (such as unnecessary gas consumption). In short, there exists a gap in bridging the high-level process specification with secure-by-design implementations that we address in our contributions (**RQ2**).

As mentioned in Section 1.3, business logic flaw-related incidents are on the rise during the past few years. The current static and dynamic analysis tools [7, 22, 23] have mainly overlooked this class of vulnerabilities. Moreover, detecting a business logic vulnerability relies heavily on expecting the process workflow of the contract. In our HighGuard contribution presented in Section 4.3.2 we address this issue through investigating the answer to **RQ7** and **RQ8** (Figure 1.3).

1.4.2 Gap 2: Inadequate Invariant Analysis and Mining

Invariants, enforced via `require` and `assert` statements in Solidity source, are a primary means of ensuring that a smart contract remains in a valid state. Many tools such as InvCon [51], InvCon+ [52], Trace2Inv [53], and PropertyGPT [54] generate smart contract invariants from execution traces or source code. The

produced invariants that are typically in the form of boolean expressions written in Solidity, are typically not readily available for downstream analysis [55, 56], fuzzing [57, 58, 59], and automated contract repair [60, 61, 62] tasks as they contain a considerable amount of noise. These noisy invariants require another level of analysis and cleaning to filter out noises such as weak invariants (`x >= 0` when `x > 1` provides a stronger guarantee) and redundant ones (e.g., `x != false` and `x` appearing together). Furthermore, invariants may appear in different syntactic forms despite being semantically equivalent, making this task more challenging. Therefore, relying solely on syntactic checks or even purely textual similarity [63] will not have the desired results. In our contribution in Section 4.2.2 we address **RQ4** and **RQ5** based on this gap with tools and research on invariant synthesis and mining.

1.4.3 Gap 3: Fragmented Understanding of Oracle Security

Blockchain oracles are specialized data relays that fetch and verify real-world information (e.g., asset prices, weather metrics, or IoT sensor readings) from off-chain sources and deliver it on-chain in a secure, tamper-evident format. Despite the pivotal role that oracles play as the bridge between on-chain smart contracts and off-chain data sources, research on oracle security has largely treated their vulnerabilities and incidents as isolated phenomena, focusing on smart contract issues such as flash loan attacks or reentrancy rather than addressing them as parts of an integrated ecosystem. Prior studies and tools have concentrated on low-level protocol flaws or single aspects of oracle operation without considering how the whole system operates throughout the entire lifecycle of oracle data [64, 65]. This fragmented perspective leaves a gap in our understanding of oracle security. In practice, attacks on blockchain oracles might not be confined to a single isolated component on chain (smart contract). Rather, they might target the main functionality of the oracles–providing data securely–since the time data is created until its deprecation on blockchain. For instance, vulnerabilities in data creation (such as compromised sensors) may go unnoticed if analysis tools focus solely on preventing smart contract vulnerabilities such as reentrancy [1]. Section 4.4.1 presents our contribution addressing **RQ9** regarding this gap.

1.4.4 Gap 4: Lack of Source Code-Agnostic Transaction Monitoring

While existing smart contract monitoring tools [1, 66, 67] often depend on source code instrumentation or manually defined vulnerability patterns, there is a notable gap in non-intrusive, scalable monitoring solutions that rely solely on transaction metadata. Current methods require either access to contract source code or modifications to the contract or blockchain client, limiting their applicability in real-world environments where many deployed contracts have closed-source or legacy implementations. Moreover, manual rule engineering is both time-

Table 1.1: Overview of publications (chronologically).

#	Paper	Venue	Year	Summary
1	Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning [1]	BSEW, EASE	2021	Section 4.3.1
2	Capturing Smart Contract Design with DCR Graphs [15]	SEFM	2023	Section 4.1.1
3	From Creation to Exploitation: The Oracle Lifecycle [14]	IWBOSE, SANER	2024	Section 4.4.1
4	HighGuard: Cross-Chain Business Logic Monitoring of Smart Contracts [66]	ASE	2024	Section 4.3.2
5	Oracle-Guided Vulnerability Diversity and Exploit Synthesis of Smart Contracts Using LLMs [68]	ASE	2024	Section 4.2.1
6	Formalizing Smart Contract Design Patterns with DCR Graphs	SoSyM (under review)	2025	Section 4.1.1
7	Model to Mitigate: Using DCR Graphs to Prevent Vulnerabilities in Smart Contracts	JLAMP (under review)	2025	Section 4.1.2
8	SInDi: Semantic Invariant Differencing For Solidity Smart Contracts	TBD	2025	Section 4.2.2

consuming and prone to obsolescence as smart contract patterns evolve. Addressing this gap is essential to develop a general monitor for smart contract security post-deployment that operates in a code-agnostic manner—eliminating the need for instrumentation or manual intervention. Therefore, we systematically address **RQ6** in Section 4.3.1.

Together, our systematic efforts in answering research questions related to each gap paves the way for a more secure decentralized application ecosystem where vulnerabilities are mitigated from the very inception of a smart contract's design to its live operation on blockchain platforms.

1.5 List of Papers

Table 1.1 provides an overview of the publications resulting from this research. In all of the mentioned papers, I have been the first author as I derived the design of the method, implementation, and writing the papers.

During different stages of each study, I received critical feedback from my co-authors and engaged in regular discussions that helped shape the investigations. Furthermore, I received feedback and help from co-authors during the writing stage.

Table 1.2 summarizes open-source reusable research tools I developed during the course of this thesis according to their security stage, purpose, and evaluation dataset. Together, these publications and tools form the technical foundation for the methodologies and contributions described in this thesis.

Table 1.2: Research tools developed during the course of this thesis.

Tool	Stage[a]	Purpose	Evaluation Dataset
DYNAMIT [1]	3	Monitoring	105 transactions
HIGHGUARD [66, 69]	3	Monitoring	54 exploits
XPLOGEN [68, 70]	2	Exploit synthesis	104 contracts
SINDI [71]	2	Invariant Differencing	200 invariant pairs
KIWI [72]	2	Invariant Noise Reduction	111 contracts

[a] DApp security stage from Section 1.2.

1.6 Thesis Organization

The remaining chapters of this thesis are organized as follows:

- **Chapter 2 (Background):** Fundamentals of blockchain, smart contracts, and DCR graphs.
- **Chapter 3 (State of the Art):** Review of related work on smart contract formalization, design patterns, invariants, and runtime monitoring.
- **Chapter 4 (Contributions):** Detailed presentations of our contributions on formal specification, exploit synthesis, invariant analysis, and monitoring of smart contracts.
- **Chapters 5 and 6: (Future Works and Conclusion)** Present the interesting research directions to follow based on this thesis and conclude the thesis.

1.7 Summary

This chapter introduced the key gaps in security of smart contracts. It explained how immutable smart contracts, while enabling trustless transactions, expose high-stakes vulnerabilities—illustrated by incidents like the DAO hack [8] and the Poly Network incident [40]. We identified recurring issues such as business logic flaws, and presented research gaps in process-aware specifications, invariant analysis, oracle security, and monitoring of smart contract behavior. Our work aims to address these gaps through formal modeling with DCR graphs, semantic invariant analysis, and holistic security measures across the DApp lifecycle. This thesis establishes a comprehensive framework that bridges design, analysis, and runtime monitoring, paving the way toward more resilient DApps.

2

Background

To fully understand the contributions of this thesis, it is essential to grasp the underlying technologies and theoretical foundations of decentralized systems. This chapter provides an overview of blockchain technology, smart contracts, and blockchain oracles. In addition, it outlines the key vulnerability types that impact decentralized applications and introduces our formal framework, DCR graphs that underpin our analysis methods.

2.1 Blockchain and Smart Contracts

A blockchain is an append-only distributed ledger system that enables trustless interactions among participants in a peer-to-peer network. Its core principles, decentralization, immutability, and transparency ensure that data recorded on the blockchain cannot be easily altered or deleted once confirmed [73]. In this system, data is grouped into blocks that are cryptographically linked, and consensus mechanisms (such as Proof-of-Stake [74]) ensure that all network participants agree on the ledger's current state [6, 75].

2.1.1 Smart Contracts and Ethereum

Ethereum, one of the most prominent blockchain platforms, extends the concept of blockchain by introducing *smart contracts*. These are reactive programs stored on blockchain that automatically enforce the rules and agreements defined by their code [4]. Smart contracts eliminate the need for intermediaries by enabling decentralized applications (DApps) to function in a trustless environment [4, 76].

At the heart of Ethereum is the Ethereum Virtual Machine (EVM). EVM is a stack-based virtual machine responsible for executing smart contract code. As Figure 2.1 shows, it processes transactions by reading instructions, performing computations using a stack architecture, and updating the contract's storage. An important component of the EVM's operation is the concept of *gas*. Gas

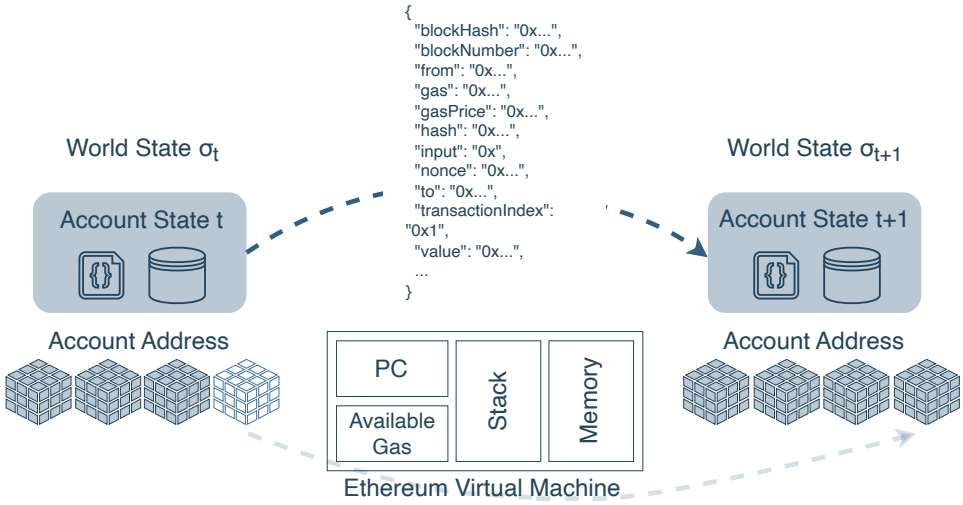


Figure 2.1: Overview of EVM Semantics: A stack-based virtual machine that executes instructions from transactions and updates smart contract storage.

is a unit that quantifies the computational work required to execute operations. Every instruction in a smart contract consumes a specified amount of gas [6]. The gas fee of a transaction is paid by the blockchain account that initiates the transaction, using Ether—Ethereum's native cryptocurrency. This mechanism compensates blockchain miners for their computational resources. Miners are network participants who validate transactions and add new blocks to the chain.[1] Additionally, this mechanism protects the network by making inefficient code execution or infinite loops prohibitively expensive. As a result, gas ensures that smart contract execution remains both efficient and secure.

Figure 2.1 provides an overview of the EVM's core semantics. It shows how EVM receives transactions, executes the corresponding instructions, and interacts with the contract storage.

The integration of blockchain with the virtual machines that run smart contract code enables Decentralized Applications (DApps), a new class of applications where complex business logic can be executed in a transparent and tamper-resistant manner.

2.1.2 Blockchain Oracles

Smart contracts are executed in an isolated blockchain environment where they cannot access external information directly. However, many decentralized applications require real-world data to function correctly. For example, decentralized

[1] In Proof-of-Stake systems, they act as validators instead [74].

finance (DeFi) platforms depend on up-to-date asset prices, while applications in supply chain management or IoT scenarios need sensor readings or environmental data. To support such use cases, smart contracts must interact with data sources outside the blockchain [14, 77].

Blockchain oracles serve as bridges between on-chain smart contracts and off-chain data sources. They enable DApps to interact with real-world events and information. By retrieving and delivering external data such as asset prices, sensor readings, or event outcomes, oracles empower smart contracts to execute context-sensitive logic that would otherwise be impossible on an isolated blockchain. These systems can be designed as either permissioned or permissionless networks, with the former relying on trusted, authenticated data sources and the latter leveraging decentralized consensus mechanisms to aggregate multiple data inputs. Such designs address the inherent "oracle problem," ensuring that the data fed into smart contracts remains reliable and tamper-resistant [64, 65].

2.2 Vulnerability Types

Our contributions (Chapter 4) concern two vulnerability types of business logic flaw [66] and reentrancy [1] which we define and review in this section.

2.2.1 Business Logic Flaw Vulnerability

Business Process Models. A *business process* is a structured set of activities or tasks performed by individuals, systems, or organizations to achieve a specific goal. It defines the logical sequence of actions, decision points, and interactions among different participants, ensuring that work is carried out in a coordinated and efficient manner. Business processes can range from simple workflows, such as customer order fulfillment, to complex operations involving multiple stakeholders, automated systems, and regulatory constraints.

A *business process model* is a formal representation of a business process, capturing its structure, behavior, and partial dependencies between events. It serves as a blueprint that defines the flow of activities, the roles of participants, and the conditions under which transitions occur. Business process models can be represented using various formal methods, including Flowcharts, Petri Nets, Finite State Machines (FSMs), Business Process Model and Notation (BPMN), and DCR graphs.

Business Logic in Smart Contracts. The business process behind a smart contract defines the overarching workflow that governs interactions between roles and external systems, while the *business logic* is the set of encoded rules within the contract that enforce and automate specific steps of that process. In other words, *business logic*, implements the business process behind the contract, and

ensures that transactions to the contract adhere to the behavior specified using the business process model of the contract.

Business Logic Flaw. Business logic vulnerabilities are flaws in the design and implementation of an application's business processes that allow attackers to manipulate legitimate functionality to achieve unintended and often malicious outcomes. These vulnerabilities arise when developers make incorrect assumptions about user behavior or fail to anticipate how users might misuse application features. Unlike typical security flaws, business logic vulnerabilities exploit the operations of an application in unintended ways, making them challenging to detect with automated tools built for detecting implementation errors [55, 56]. Unlike traditional security vulnerabilities, which stem from implementation errors (e.g., reentrancy or integer overflows), business logic flaws originate from contract design misaligned with its business process specification which usually translates to incorrect assumptions about user behavior or incomplete enforcement of constraints [15, 19]. Because business logic directly dictates how the business process is executed on-chain, flaws in it can lead to severe financial losses, such as improper fund transfers, infinite minting, or bypassing protocol restrictions. Detecting and preventing such flaws requires not just code audits but also *formal modeling and verification of the intended business process* to ensure that all possible interactions align with the expected system behavior.

2.2.2 Reentrancy Vulnerability

Reentrancy vulnerability happens when a function in a target contract is repeatedly reentered when the target makes an external call to another contract before it updates and checks its own state for bookkeeping. Reentrancy allows the receiving contract (attacker) to call back into the target contract, repeating operations before the first execution is completed. In Figure 2.2, the contract in Figure 2.2a contains a reentrancy flaw in its `withdraw` function. When a user withdraws funds, the contract sends the requested amount using a low-level `call` before setting the user's balance to zero. Because the transfer is made before the balance update, an attacker can hijack execution by repeatedly re-entering the function.

The attacker contract in Figure 2.2b exploits this behavior. By initially depositing 1 ether and then calling `withdraw`, the contract transfers funds to the attacker. However, before the balance in `VulnContract` is updated, the attacker's fallback function (line 12 in Figure 2.2b) is triggered. This function calls `withdraw` again, leveraging the fact that the vulnerable contract still believes the attacker has a balance. The cycle continues, allowing the attacker to withdraw multiple times in a single transaction, draining all available funds.

```
contract VulnContract {
mapping(address => uint) public
    balances;
function deposit() public payable {
balances[msg.sender] += msg.value;
}
function withdraw() public {
uint amt = balances[msg.sender];
require(amt > 0);
(bool ok,) = msg.sender.call{value:
    amt}("");
require(ok);
balances[msg.sender] = 0;
}
}
```

(a) Vulnerable contract

```
contract Attacker {
Vuln target;
constructor(address _target) {
target = VulnContract(_target);
}

function attack() public payable {
target.deposit{ value: 1 ether }();
target.withdraw();
}

receive() external payable {
if (address(target).balance > 0)
    target.withdraw();
}
}
```

(b) Attacker contract

Figure 2.2: Reentrancy vulnerability in a contract (a) and an attacker contract exploiting it (b).

2.3 Dynamic Condition Response (DCR) Graphs

DCR graphs is a declarative framework for representing business processes in graphical form. Its formal structure is presented in Definition 1 below, and an illustrative example is shown in Figure 2.3.

DCR graphs offer an alternative to traditional state machines. Instead of modeling events as transitions, DCR graphs represent them as nodes (visualized as boxes). Each event node may be associated with particular roles, and its execution may be influenced by other events through various types of directed arrows.

The nodes in the graph form a set E of events, where each event is annotated with both a role label and an action label. In Figure 2.3, nodes appear as boxes with the action label centered and the role label displayed in the top bar. Nodes are classified into three types:

- **Input actions** (e.g., *Commit* in Figure 2.3) are indicated by a folded paper-corner icon in the top right of the box. These actions receive an external value when executed.

- **Computation actions** (e.g., the *Decide* action) are marked with an =-sign in the top right of the box. Their execution involves evaluating a computation

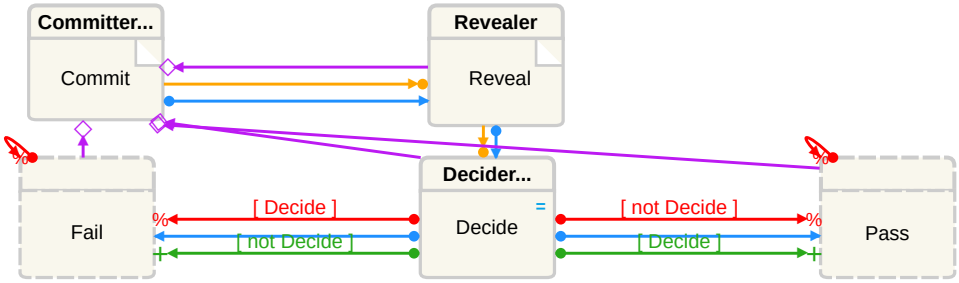


Figure 2.3: Commit and reveal design pattern in DCR (example from [15]).

expression that potentially references their own or other events' values and recording the result.

- **Simple actions** (e.g., the *Fail* and *Pass* actions) do not involve additional input or computation.

For instance, the computation associated with the *Decide* action is given by the Boolean expression *commit* = *hash(reveal)* (although this expression is not depicted in Figure 2.3), which compares the value supplied during the *Commit* action with the hash of the value from the *Reveal* action.

Directed edges in the graph define rules that govern the execution of events. These rules fall into two categories: constraints and effects. A typical constraint is the condition rule, shown in Figure 2.3 as an orange arrow labeled $\rightarrow\bullet$ with a bullet at the target node. This rule requires that the source event (here, *Commit*) must either have been executed at least once or be excluded for the target event (here, *Reveal*) to become executable.

Effect rules include the *exclude*, *include*, and *response* relations. In Figure 2.3, these are depicted as follows: a red arrow $\rightarrow\%$ with a %-sign at the target, a green arrow $\rightarrow+$ with a +-sign at the target, and a blue arrow $\bullet\rightarrow$ with a dot at the source. For example, the exclude rule specifies that when the source event (in this case, the *Decide* action) is executed, the target events (here, *Fail* and *Pass*) are excluded. Excluded events are not executable and are disregarded when evaluating constraints, which facilitates the representation of *defeasible rules* [78].

When DCR graphs incorporate data, rules can be guarded by Boolean expressions that determine whether they should be active in the current state. For example, a guard labeled *Decide* associated with the exclude relation $\rightarrow\%$ from *Decide* to *Fail* implies that *Fail* is excluded if and only if the computed value from *Decide* is true—that is, when the value provided at *Commit* equals the hash of the value supplied at *Reveal*. Likewise, the response rule $\bullet\rightarrow$ indicates that if the source event (e.g., *Commit*) is executed, then the target event (e.g., *Reveal*) must either be executed or become excluded at some point in the future.

The execution state of a DCR graph is captured by a *marking*, which assigns state information to each event. In the original DCR framework [42], each event's marking comprised three Boolean values indicating whether the event had been executed, whether it is required to be executed (or excluded) in the future, and whether it is currently excluded. In this work, we extend DCR graphs to include data, time, and nested sub-processes—a development supported by an online design tool[2]. This extended model also introduces two new effect rules:

- The **value relation** $\rightarrow\!\!\!\!\!=$ (depicted as a gray arrow with an =-sign at the target) updates the data of the target event upon the execution of the source event.
- The **cancel relation** $\bullet\!\!\rightarrow\!\!\!\!\times$ (shown as a brown arrow with a $\times$-sign at the target) removes any pending execution requirement (e.g., due to a previous activation of a response rule) of the target event when the source event is executed.

For DCR graphs with data, the marking additionally records the current data value (if one exists) for each event. In time-extended DCR graphs, the marking also captures temporal details—specifically, the elapsed time since an event was last executed (if it has been executed) and a deadline by which an event must be executed if it remains required.

Definition 1 formally specifies timed DCR graphs with sub-processes and data. This definition unifies previous work on timed DCR graphs with sub-processes [79] and timed DCR graphs with data [80], while also introducing a new edge type that models a value effect.

We assume a set of computation expressions $\mathsf{Exp_E}$ with a subset $\mathsf{BExp_E} \subseteq \mathsf{Exp_E}$ containing the Boolean expressions. For each event $e \in E$, an expression from $\mathsf{Exp_E}$ is associated with it to denote its current value (as recorded in the marking). A discrete-time model is adopted, where time is represented by natural numbers. We denote the set of natural numbers (including 0) by ω and define $\infty = \omega \cup \{\omega\}$, where ∞ represents a non-fixed deadline in accordance with classical liveness properties. (The design tool follows the ISO 8601 standard [81] for date and time representations, supporting years, months, days, and seconds.)

Definition 1 (DCR Graph)**.** A *timed DCR graph with sub-processes, data, and roles* G is defined as a tuple

$$(E, E_G, nest, D, M, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \bullet\!\!\rightarrow\!\!\!\!\times, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\%, \rightarrow\!\!\!\!=, L, l)$$

where

1. E is a finite set of *events*,
2. E_G is a finite set of *event groups*,

[2] Available for academic use at `dcrsolutions.net`

3. $nest\colon E \cup E_G \rightharpoonup E \cup E_G$ is an acyclic *nesting* function (i.e., for all $k > 1$, if $nest(e)$ is defined then $nest^k(e) \neq nest(e)$),

4. $D\colon E \to \mathsf{Exp_E} \uplus \{?\}$ classifies each event as either a *computation event* with expression $d \in \mathsf{Exp_E}$ or an *input event* (denoted by ?),

5. $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va}) \in \big((E \rightharpoonup \omega) \times (E \rightharpoonup \infty) \times \mathcal{P}(E) \times (E \rightharpoonup V)\big)$ is the *timed marking with data*,

6. $\rightarrow\!\!\bullet \subseteq E \times \omega \times \mathsf{BExp_E} \times E$ is the *guarded timed condition relation*,

7. $\bullet\!\!\rightarrow \subseteq E \times \infty \times \mathsf{BExp_E} \times E$ is the *guarded timed response relation*,

8. $\bullet\!\!\rightarrow\!\!\times$, $\rightarrow\!\!\diamond$, $\rightarrow\!\!+$, $\rightarrow\!\!\%$, $\not\rightarrow \subseteq E \times \mathsf{BExp_E} \times E$ are the guarded cancel, milestone, include, exclude, and value relations, respectively,

9. $L = \mathcal{P}(R) \times A$ is the set of labels, with R and A being sets of roles and actions, respectively,

10. $l\colon E \to L$ is a *labelling function* that maps events to their labels.

The nesting function $nest(e)$ establishes a partial hierarchical relation among events and event groups. A key distinction is that when an event e is nested in another event (i.e., $nest(e) \in E$) it has a different interpretation from when it is contained in an event group (i.e., $nest(e) \in E_G$). In the latter case, the event group serves as a shorthand for its contained events. We denote $e < e'$ if $nest(e) = e' \in E_G$ and use $<^*$ to denote the reflexive, transitive closure of $<$. Consequently, we write $(e', k, d, e) \in \rightarrow\!\!\bullet$ to mean that, if $(e'', k, d, e''') \in \rightarrow\!\!\bullet$ and $e' <^* e''$ and $e <^* e'''$, then the condition relation holds; an analogous interpretation applies to the other relations.

If there exists an event $e' \in E$ such that $e' <^* e''$ and $nest(e'') = e \in E$, then the event e is termed a *sub-process* event. A sub-process event is executed when one of its constituent events is executed, and it is considered complete when no events within the sub-process remain pending and included. The state of the process is captured by the marking $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$, where:

- $\mathsf{Ex}(e)$, if defined, indicates the elapsed time since event e was last executed,
- $\mathsf{Re}(e)$, if defined, provides the deadline by which the event is required (thus, the event is pending),
- In is the set of events that are currently included, and
- $\mathsf{Va}(e)$, if defined, holds the current data value of the event.

Enabledness. The condition relation $\rightarrow\bullet$ and the milestone relation $\rightarrow\diamond$ impose constraints on when events may be executed. Specifically, a condition of the form $e'\rightarrow\bullet e$ stipulates that for e to be enabled, the event e' must either have been executed at least once or be excluded. Similarly, a milestone $e'\rightarrow\diamond e$ requires that e' is either excluded or not pending for e to become enabled. In Figure 2.3, the milestone relations ensure, for example, that the *Commit* action cannot be repeated while there remain pending executions of *Reveal*, *Decide*, *Fail*, or *Pass*.

More formally, an event e is enabled in a marking $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$ and may be executed by a role $r \in R$ (with $l(e) = (R', a)$ and $r \in R'$) if:

1. e is included, i.e., $e \in \mathsf{In}$,
2. Every condition rule is satisfied, i.e., for every $e' \in E$ such that $(e', k, d, e) \in \rightarrow\bullet$, if e' is included and the guard d evaluates to true in M, then $\mathsf{Ex}(e') \geq k$,
3. Every milestone rule is met, i.e., for every $e' \in E$ such that $(e', k, d, e) \in \rightarrow\diamond$, if e' is included and the guard d holds in M, then $\mathsf{Re}(e')$ is undefined, and
4. For any event e' with $e <^* e'$ such that $nest(e') = e'' \in E$, the event e'' must itself be enabled and executable by role r.

2.4 Decentralized Application Monitoring

Definition 2 (Monitoring)**.** Monitoring is a *continuous* process of observing smart contract execution in on real blockchain or on simulated blockchain environment. The goal in runtime monitoring is to detect deviation of the *observed behavior* from the *required behavior* [82].

2.4.1 Monitor Placement in Blockchain

The monitor placement is determined by the source of information and the intervention approach. Different sources of runtime information are available in a public blockchain such as Ethereum [6] which vary in terms of accessibility (*on-chain* vs. *off-chain*), granularity, and real-time availability. Some monitoring techniques require explicit instrumentation of contract code, while others passively collect data from blockchain transaction processing. As Figure 2.4 presents, this information belong to one of the protocol layer, smart contract implementation layer, or blockchain layer. We categorize the source of information into these five categories:

- *Contract's storage:* As presented in Figure 2.1, a smart contract has read-/write access to its *storage* section. One should instrument the contract source or bytecode to implement this and inject data collection instructions into the code. Both the instrumentation (contract's code section) and collected data (contract's storage) reside *on chain*. As the data collection method relies on

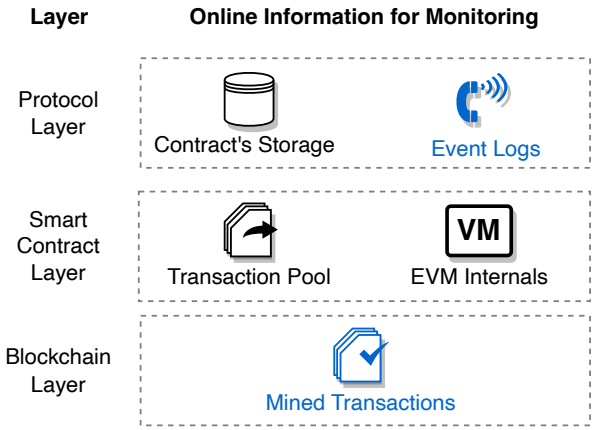


Figure 2.4: Monitorable online information in Ethereum according to their layer in decentralized ecosystem. Blue color represents the information we use.

EVM opcodes such as `SSTORE` [83], it incurs extra *gas* transaction fee that should be paid by the transaction initiator [84]. The monitoring information is *on-chain* and can be accessed both *on-chain* and *off-chain* by reading appended blocks.

- *Event logs:* Smart contracts can emit *events*, which are recorded in the transaction receipt and stored in the blockchain's log system. Monitoring systems can subscribe to events, enabling real-time tracking of specific activities such as fund transfers, function calls, or security-related alerts. Event logs are *on-chain* and accessible to off-chain monitoring tools via JSON-RPC calls [76]. This method also relies on EVM opcodes to write logs, and hence incurs extra gas on transactions [83].

- *Transaction Pool:* Before transactions are included in a block, they reside in the *mempool* (transaction pool). Monitoring the mempool allows for early detection of potentially malicious or anomalous transactions, such as front-running or sandwich attacks [85]. Since mempool transactions are not yet confirmed, monitoring them enables a *reactive* security approach, but access to mempool data is dependent on the node configuration.

- *EVM internals:* One could use transaction traces to get EVM runtime information granularly. One way to retrieve this information is to instrument an EVM client to store them [6]. The downside of this method is that the monitoring delays due to the instrumented virtual machine could disrupt the normal execution flow of the smart contract, potentially affecting the resulting list of included transactions in the block [47]

- *Mined transactions:* This source is based on the information of transactions mined and included in the newly appended blocks. Both transaction attributes [86] (i.e., transaction receipt) and full transaction traces from the virtual machine [87] are available at this point. This type of information is available *off-chain*.

Our contributions DYNAMIT (Section 4.3.1) and HIGHGUARD (section 4.3.2) use *event logs* and *mined transactions* respectively as their information source.

2.4.2 Violation Handling (Intervention)

Depending on the monitoring goal, in the event of a violation flagged by the monitor, it will either passively report or react:

1. *Passively report:* Do not interfere with transaction execution but instead log and report flagged transactions for later analysis. This approach is suitable for auditing, compliance checks, and forensic investigations. It ensures minimal impact on the contract's execution while helping off-chain decision-making.

2. *Reactive measures:* Actively intervene when a transaction is flagged by the monitor. This intervention can take various forms: (a) reverting malicious transactions if monitoring is embedded within a smart contract or an EVM node, (b) triggering an alert system to notify stakeholders, or (c) dynamically adjusting contract state or access control policies in response to detected threats, leveraging specific design patterns in the contract architecture [15]. Both of our monitoring contributions (Section 4.3.1 and Section 4.3.2) support reactive measures aligned with (b) and (c).

2.5 Summary

This chapter provided a concise overview of the foundational elements of DApps. We reviewed blockchain and smart contracts highlighting how EVM works as a special constrained execution environment. We also demonstrated blockchain oracles as bridges to off-chain data. We discussed vulnerability types such as business logic flaws and reentrancy which our contributions (Chapter 4) try to prevent as design flaws and detect at later stages of DApp lifecycle. The introduction of DCR graphs as a formalism for modeling business processes in smart contracts helps grasping our contributions. Finally, we surveyed various methods for DApp monitoring. These form the basis for analyses presented in the upcoming chapters.

3

State of the Art

Understanding the current research landscape in decentralized applications is crucial for identifying the challenges and gaps that motivate our contributions. In this chapter, we review key approaches in formalizing decentralized application designs, design patterns tailored for smart contracts, and techniques for vulnerability analysis, including exploit generation, invariant analysis, and blockchain transaction monitoring methods. This overview situates our work within the research landscape.

3.1 Formalizing Decentralized Application Designs

We extensively review the existing research concerning the first step of building a decentralized application: "specifying the behavior" in this section.

Prior research distinguishes modeling formalisms into high-level (business process-level) and low-level (platform-specific) categories [15, 36]. High-level models capture the overall behavior of a smart contract while abstracting away implementation details. Both of our contributions (Section 4.1.1 and Section 4.1.2) in this stage concern high-level design of the DApps.

Behavioral Interaction Priority (BIP) has been employed to visually model smart contract interactions [88]. Graphical formalisms such as Petri Nets and their Colored variants, provide intuitive representations of system dynamics and data dependencies [89, 90, 91]. Timed automata capture temporal constraints inherent in smart contract execution [92, 93]. Probabilistic transition systems (e.g., Markov decision processes) offer insight into nondeterministic behaviors [94, 95, 96] which does not fit the DApp ecosystem. Despite these advances, none of these approaches fully provide the expressive, declarative modeling needed for business process–level design. We address this gap with Dynamic Condition Response (DCR) graphs [80].

DApps commonly involve multiple, interdependent transactions. The nondeterministic ordering of transactions due to miner behavior has been highlighted

by Sergey and Hobor [97]. Other works explore conditions under which transaction interleavings can be exploited or serialized [98, 99]. While finite state automata have been used to model these dependencies, they may lead to "bad states" when many actions become inaccessible [67]. In contrast, DCR graphs offer an elegant, visual mechanism for representing transactional dependencies by capturing the partial ordering of events.

3.1.1 Decentralized Application Design Patterns

Design patterns are well-established, general repeatable solutions to common problems encountered in software design. Originating from the seminal work of Gamma et al. [100], these patterns provide developers with a vocabulary and proven templates that can be adapted to solve recurring design challenges. In traditional object-oriented programming, design patterns such as *Singleton*, *Observer*, and *Factory* have become tools for designing systems. They not only encapsulate best practices but also promote code reuse and ease communication among developers by formalizing common design strategies. In the context of smart contracts, previous studies have focused on categorizing patterns without formalizing or verifying them [33, 39, 101, 102]. Notable exceptions include the work of Wohrer and Zdun [39], who formalize the "pull over push" pattern using a domain-specific language automatically generate Solidity code [103]. Similarly, approaches addressing contract upgradability [104, 105, 106] formalize design patterns for patching, though without guaranteeing the correctness of the patched contracts.

Design patterns for smart contracts must address not only the traditional concerns of code organization and reusability but also incorporate security requirements. For example, reentrancy vulnerability (Section 2.2.2) can be mitigated by adopting the Checks-Effects-Interactions (CEI) pattern. CEI pattern emphasizes updating state before making any external calls [10, 107]. Similarly, the challenge of upgradeability in immutable smart contracts has led to the development of the *Proxy* pattern (and its advanced variants such as the *Diamond* pattern defined in EIP-2535 [106]), which decouples contract logic from data storage to allow controlled upgrades while maintaining state integrity. Role-based access control, another critical security design pattern, is implemented via pseudo-standardized Solidity language modifiers in libraries such as OpenZeppelin [108] that ensure only authorized entities can invoke sensitive functions [109].

By integrating these design patterns specific to decentralized applications, developers are less likely to repeat the same vulnerabilities identified in the literature [5]. Our contribution (Section 4.1.1) intersects with the mentioned works by categorizing design patterns into high-level patterns that we formalize using DCR graphs.

3.1.2 Process-Oriented View of Decentralized Applications

Traditional business process models typically downplay data in favor of control flow, whereas blockchain-based applications are inherently data-centric as a blockchain functions as a replicated database [110]. To reconcile these aspects, we classify smart contract design patterns into high-level and low-level categories. Contract-level patterns akin to those in conventional business processes focus on workflow and interaction rather than low-level implementation details. Frantz and Nowostawski [111, 112] argue for the codification of legal rules directly from natural language, enabling semi-automated contract generation. Moreover, several works propose executing workflows defined in DCR graphs directly on-chain [113, 114, 115] which differs from our work as we capture the high-level design of the contract using DCR graphs rather than executing models on chain.

3.1.3 Formal Analysis of Decentralized Applications Using DCR Models

Capturing and formalizing the design of the contract using DCR specifications serve as a foundation for automated analysis by verifying that a smart contract's implementation adheres to its intended behavior. For example, our runtime monitoring tool HighGuard [66] observes Ethereum transactions while concurrently executing the corresponding DCR graph model. Deviations from the model trigger alerts, facilitating post-deployment audits. This approach leverages runtime information without incurring the overhead typically associated with on-chain instrumentation.

3.1.4 Designing to Deter Vulnerabilities

Correct-by-Construction

Research involving correct-by-construction approach uses FSMs and LTL specifications for smart contracts proposed by Suvorov and Ulyantsev [116]. VeriSolid framework [38] enables automatic generation and verification of Solidity code based on such specifications. Our contributions leverage DCR graphs to model the high-level business logic of smart contracts where many business logic vulnerabilities originate [5, 66], and serve as the first step for a correct-by-construction approach using business process model of the smart contracts.

Bug Hunting and Auditing for Smart Contracts

On-chain bug bounty systems, such as the Hydra contract [117], employ multi-version programming to manage multiple contract versions via a proxy. However, these systems face limitations including increased gas consumption, limited opcode support, and challenges in code delegation. As Zhou et al. [118] note,

incorporating defenses and bug hunting for smart contracts alone is insufficient without a holistic design; our contribution in Section 4.1.2 (**RQ2**) demonstrates that using DCR graphs during the pre-development can prevent design flaws and vulnerabilities that typically require post-development analysis of source code or transaction data.

3.2 Exploit Generation for Smart Contracts

Automated exploit generation for smart contracts has been approached from several angles. In general, prior work can be classified into two broad categories: those relying on symbolic (or concolic) execution and those using fuzzing-based, dynamic methods. More recently, hybrid frameworks have emerged that combine dynamic mutation with oracle support to overcome the inherent limitations of each individual technique.

Symbolic execution–based approaches. Early work such as TeEther [119] exploits vulnerabilities by performing symbolic execution directly on the contract bytecode. TeEther focuses on generating input transactions that trigger specific low-level vulnerable operations (e.g., suicidal or call injection vulnerabilities) by formulating and solving path constraints using SMT solvers. ExGen [120] extends this idea to a cross-platform setting by first translating contracts (from Ethereum [6]) into an intermediate representation and then symbolically generating multi-transaction exploits. Although these techniques can precisely pinpoint vulnerability-triggering conditions, they often struggle with complex constraints such as cryptographic checks or with modeling inter-transaction dependencies.

Fuzzing-based approaches. To address the limitations of symbolic execution, fuzzing-based techniques have been proposed. EthPloit [29] adopts a fuzzing strategy tailored for smart contracts. EthPloit combines static taint analysis with a dynamic seed strategy that leverages runtime feedback from an instrumented Ethereum Virtual Machine. By generating and mutating transaction sequences, EthPloit can overcome hard-to-solve constraints and better simulate blockchain effects (such as timestamps and gas limits), thereby improving exploit coverage for vulnerabilities like those involving cryptographic functions. ContraMaster [27] uses specialized mutation operators that account for control-flow, data-flow, and the dynamic state of the contract. It then monitors execution to validate candidate exploits against a semantic oracle. This dynamic, oracle-supported framework is particularly adept at generating multi-transaction exploits and has been shown to generate exploits with high precision.

In contrast to the aforementioned techniques which mainly focus on either low-level, symbolic constraint solving or on dynamic fuzzing, our work (Section 4.2.1) leverages LLMs with an expressive formal oracle to specify the intended business logic of a smart contract. This formal guidance enables systematic injection

of vulnerabilities and the generation of exploit transactions that are verifiable against a precise specification. This capability is embodied in our XploGen contribution (see Section 4.2.1), which demonstrates how formal oracles can be integrated into dynamic exploit generation frameworks to improve accuracy and generalizability of exploit synthesis.

3.3 Invariant Analysis for Solidity Smart Contracts

Invariants have been central to program verification since the early work of Floyd [121] and Hoare [122]. These foundational works introduced the idea of annotating programs with *pre-* and *post*-conditions, which later evolved into formal frameworks for verifying program correctness [123, 124]. Moreover, the concept of enforceable security policies, where execution monitoring can prevent unsafe states, has further underscored the significance of invariants in maintaining system integrity [125].

Dynamic invariant generation tools typically work by instrumenting a program to record execution traces at strategic program points such as loop entries and function calls, and then statistically analyzing these traces to infer properties (or candidate invariants) that appear to hold consistently. For instance, Daikon's approach initializes a large set of invariant templates and then discards any candidate that is falsified by the observed runtime data, ultimately reporting the remaining likely invariants as the program's dynamic specification [123, 124]. Recently these dynamic approaches have been extended to smart contracts where blockchain transaction histories serve as a rich source of execution data [51, 52]. Static methods, on the other hand, transform source code into an analyzable representation.

Recent advancements incorporate machine learning to enhance invariant inference. Neural network models trained on annotated code datasets, as well as fine-tuned LLMs[1], have shown promising improvements over classical invariant mining approaches, particularly in scenarios with limited runtime data [127, 128].

3.3.1 Smart Contract Invariant Mining and Synthesis

Invariant mining research, such as InvCon [51], InvCon+ [52] and Trace2Inv [53], and OpenTracer [129] attempt to infer invariants from contract execution traces. PropertyGPT [54] employs a retrieval-augmented generation strategy based on source code to synthesize properties, but it does not address invariant refinement or semantic differencing. Our contribution in Section 4.2.2 (**RQ4** and **RQ5**) introduces a hybrid method for further refining the dynamically mined invariants to prepare them for downstream tasks such as fuzzing [27, 28, 29].

[1] A fine-tuned LLM is a pre-trained model further trained on a specific dataset to adapt its responses to a particular domain or task [126].

3.3.2 Invariant Differencing

Semantic invariant differencing has been mainly done through clustering. PatchPart [63] clusters invariants into different semantic partitions. It operates on invariants generated by Daikon [123, 124]. It calculates the implication distance to determine how semantically close two invariants are. It checks whether invariants in one group imply the invariants in the other by using an SMT solver (Z3 [130]). Alternatively, it also utilizes Levenshtein edit distance [131] to calculate the heuristic distance between the two invariants. Using an SMT solver is more exact but has more computation cost than Levenshtein distance [131]. PRECIS [132] is another work around invariant clustering. This tool clusters based on input, output, and path conditions. After creating a test suite with execution traces, it clusters the traces based on similar evaluated conditions. A linear regression is then applied to these predicate groups to check whether a linear relationship exists between their input and output. If yes, they form a seed cluster. With a heuristic similar to the Hamming-distance [133], neighboring groups are put together that share the same output, ensuring that invariants with the same output are clustered together. These clusters then represent the invariants.

3.3.3 Semantic Equivalence Checking of Smart Contracts.

Semantic equivalence checking examines whether two versions of a smart contract preserve identical behavior—a task related to, but distinct from, semantic invariant differencing. For tasks such as automated program repair, semantic equivalence checking ensures that patches do not alter the intended functionality. For example, tools like ReenRepair [134] leverage def-use chain analysis to verify that patched contracts remain functionally equivalent. Although effective, such methods often rely on runtime traces, which can limit their applicability for undeployed contracts.

Clone detection is a complementary technique that evaluates similarity between code fragments by distinguishing superficial syntactic changes from genuine behavioral equivalence. General clone detection techniques have been applied in smart contracts to identify type 3 clones (modified copies with minor changes) and type 4 clones (different implementations achieving the same functionality). For instance, Khan et al. [135] applied the NiCad clone detector [136] on 33 073 verified smart contracts and found that type 3 clones constitute approximately 2.05% of the Solidity code corpus. In a similar vein, SmartEmbed [137] employs deep learning and code embedding similarity to detect both type 3 and type 4 clones, with their analysis revealing nearly 6.6 million lines of cloned code—a clone ratio of about 90%. These clone detection techniques are not only valuable for measuring semantic similarity but also for verifying patch safety. By comparing pre- and post-patch versions of a contract, clone detection can help ensure that modifications preserve the contract's original semantics, thus safeguarding

Table 3.1: Comparison of available runtime analysis tools for smart contracts

Tool	Monitor Placement	Evaluation Datasets	Target Vulnerabilities	Year
Single-Chain Tools				
ContractLarva [67]	On-chain	1 contract [67]	—	2018
Solythesis [138]	On-chain	23 contracts [139]	—	2020
ContraMaster [27]	Instrumented EVM	218 contracts [27]	Reentrancy Exception Disorder Integer Over/underflow	2020
SCMon [47]	Instrumented EVM	1 contract [47]	—	2020
Annotation [140]	On-chain	50 contracts [140]	Reentrancy Type Casting Transaction Order Non-determinism Exception Disorder	2022
Scribble [141]	On-chain	—	—	2021
Transaction Monitors [142]	Instrumented EVM	—	—	2022
Forta [143, 144]	Off-chain	—	—	2022
HAL Streams [145]	Off-chain	—	—	—
OpenZeppelin Monitors [146]	Off-chain	—	—	—
Dynamit [1]	Off-chain	25 contracts [147]	Reentrancy	2021
Cross-Chain Tools				
Xscope [48]	Off-chain	4 cross-chain bridges [148]	Unrestricted Deposit Emitting Inconsistent Event Parsing Unauthorized Unlocking	2023
HighGuard [66]	Off-chain	52 Exploits [149]	Business Logic Flaw Cross-chain Business Logic Flaw	2024

against unintended behavioral changes.

3.4 Research on Smart Contract Monitoring

Monitoring decentralized applications is critical for ensuring contract safety after deployment. A variety of techniques are used to observe and verify smart contract behavior ranging from on-chain instrumentation to off-chain analysis. These tools differ in how they integrate with the blockchain, the level of instrumentation they require, and their operational overhead. In the following, we group these solutions into two main categories: (i) on-chain instrumentation approaches and (ii) off-chain and hybrid monitoring approaches. Table 3.1 provides an overview of the tools and research, their attributes such as monitor placements, publication or

release year, target vulnerabilities, and the dataset the tool is evaluated on.

3.4.1 On-Chain Instrumentation Approaches

On-chain monitoring tools embed additional code into smart contracts to check for compliance with specified properties at runtime. These tools typically modify the contract's source code (or bytecode) to inject assertions or tracking mechanisms. For example:

- *ContractLarva* [67] instruments the contract's source code before deployment using an automaton-based specification. It injects tracking code that monitors executions and reverts the last transaction when a violation is detected. While effective, ContractLarva can incur significant overhead—up to 1100% during contract initialization, though normal function calls usually see overhead below 5%.
- *Solythesis* [138] adopts a source-to-source compilation approach to embed runtime invariant checks. It mitigates overhead by employing *delta update* and *delta check* techniques that reduce redundant storage accesses. Its performance overhead is measured as a 24% slowdown in transactions per second (TPS) without consensus, reducing to 0.1% when consensus is applied.
- *Annotation-based Approaches* such as those proposed by Shayamasundar [140] rely on developer-provided annotations (e.g., marking functions as non-reentrant) that a pre-processor uses to inject safeguards. This method abstracts the monitoring logic away from the core business code but, like other on-chain solutions, introduces additional gas costs.

Collectively, these on-chain approaches ensure that violations are detected and handled immediately by reverting transactions. However, the increased gas consumption and potential performance degradation make it essential to apply such techniques selectively.

3.4.2 Off-Chain and Hybrid Monitoring Approaches

Off-chain monitoring solutions and hybrid approaches aim to reduce on-chain overhead by performing much of the analysis externally or by combining on-chain data collection with off-chain processing. These techniques often leverage blockchain event logs, transaction traces, or external scan nodes, and include:

- *Scribble* [150] utilizes annotations embedded in contract comments to generate runtime assertions. It transforms the original contract into an instrumented version that emits events when an assertion fails. This output can then be analyzed off-chain (e.g., using fuzz testing) to detect violations without imposing high gas costs.

- *Formalized Transaction Monitors* (e.g., by Capretto et al. [142]) augment smart contracts with methods that validate pre- and post-conditions across a transaction's sequence. Although this approach requires heavy instrumentation or even new blockchain instructions, it provides fine-grained validation of multi-operation transactions.

- *Hybrid Fuzzing Approaches*, such as ContraMaster [27], integrate grey-box fuzzing with a semantic test oracle. ContraMaster generates mutation-based transaction sequences and executes them on an instrumented Ethereum Virtual Machine (EVM), and continuously validates contract states against invariants.

- *Decentralized Monitoring Platforms* such as Forta [143, 144] operate off-chain using a network of scan nodes and detection bots. Forta monitors on-chain activity via JSON-RPC calls, processes events according to bot specifications, and emits alerts. Similarly, HAL Streams [145] allow users to define blueprints that filter blockchain data to capture specific events, and OpenZeppelin Monitors [146] provide alerting capabilities through APIs.

The off-chain solutions reduce direct gas costs by shifting most of the computational burden off-chain. On-chain approaches (ContractLarva [67], Solythesis [138], annotation-based methods) offer immediate, in-situ detection but at the cost of increased gas consumption and potential performance impact. In contrast, off-chain/hybrid approaches (Scribble [141], formalized transaction monitors [142], Forta [143], HAL Streams [145], OpenZeppelin monitors [146]) minimize on-chain overhead by leveraging external data sources and analysis frameworks. They trade off immediate reaction for scalability and lower operational cost, which is especially advantageous for large-scale or continuous monitoring scenarios.

3.4.3 Cross-Chain Smart Contract Monitoring.

In 2016, Vitalik Buterin [151], a co-founder of Ethereum proposed the idea of "chain interoperability". The intention was to enable the easy transfer of assets between different blockchains. Three different types of interoperability (bridging) were proposed: 1) *Centralized schemes*[2] depend on the involved parties agreeing to carry out actions on the target blockchain when an event takes place on the source blockchain. 2) *Sidechains* are systems inside a blockchain that can validate events and states in another blockchain. 3) *Hash-locking* consists of operations on multiple blockchains that have the same trigger, such as a hashed password being used to lock an asset, and the plaintext password being used to unlock it [151]. These different types of interoperability come with their advantages and trade-offs. Centralized systems are generally easier to implement, but give

[2]Like the one used in HighGuard [66].

power to whoever is entrusted to carry out transactions on the other chain. Hash-locking methods rely on communication between participants to set terms and conditions and to fulfill delivery through the sharing of the key. Sidechains rely on decentralized relayers to transmit relevant transaction information between chains [152].

The complexity and varying trust assumptions of cross-chain bridges create new attack vectors, making cross-chain monitoring crucial. Since cross-chain bridges often rely on external relayers or multi-signature custodians, they introduce vulnerabilities related to transaction validation, replay attacks, and bridge oracle manipulation [153]. Despite the increasing adoption of cross-chain protocols, the security aspects of business logic verification across chains remain underexplored in the research community.

Recent work, such as Xscope [48], has sought to address these challenges by combining static analysis and runtime monitoring to detect cross-chain vulnerabilities. Xscope operates by integrating monitoring within off-chain relayers that fetch events on the source chain and coordinate actions on the target chain. By pre-executing transactions before they are finalized, Xscope can detect security violations, halt malicious transactions, and prevent bridge exploits. This hybrid monitoring approach has already led to the discovery of three new classes of cross-chain security bugs [48]. Ganguly et. al. [154] suggest that runtime verification in a multi-blockchain context can be reduced to a *distributed runtime verification* problem where a monitor observes the behavior of a distributed system that does not share a global clock. A difficulty stems from the fact that the individual blockchains do not share a global clock: the ordering of events. A technique has been proposed that assumes partial synchrony between the blockchains. The computation can then be divided into segments and rearranged with an algorithm, to verify the correctness of the system [154].

3.5 Summary

This chapter surveyed the current research in DApp security. We reviewed methods for formalizing smart contracts. We discussed the limitations of each of the formalisms in capturing the expressive specifications for DApps. We then presented techniques for exploit generation, invariant analysis, and monitoring. We highlighted both static, dynamic, and hybrid methods to conduct the mentioned analysis tasks. Finally, we discussed on-chain, off-chain, and cross-chain monitoring solutions. Overall, researchers have made significant progress, but gaps still remain in process-aware specification, vulnerability detection, and process-aware monitoring for smart contracts. These challenges motivate the contributions in next chapter.

4

Thesis Contributions

We aim to enhance the security of DApps with a holistic perspective. For this purpose, we address the research questions in the introduction (Section 1.4) in relation to the stage of the security lifecycle of the DApps. Instead of presenting the contributions in chronological order, we organize them according to the three stages of smart contract maintenance illustrated in Figure 4.1.

- **Stage 1: Addressing the DApp specification challenge:** Formalizing smart contracts to enable structured verification (**RQ1** and **RQ2**).

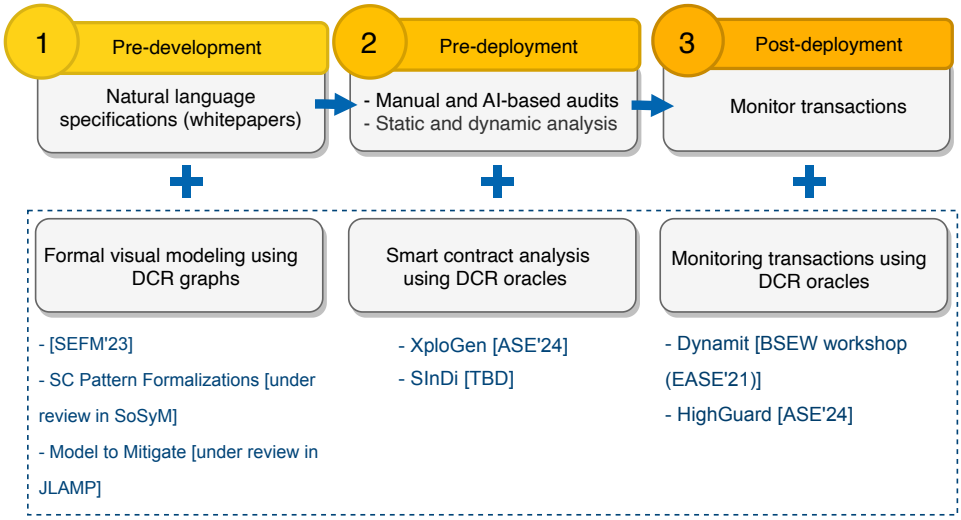


Figure 4.1: Overview of thesis contributions: upper row is the prior works and the bottom row presents the contributions of the this thesis.

- **Stage 2: Pre-deployment analysis:** Detecting business logic vulnerabilities before an application is deployed (**RQ3** and **RQ4**).
- **Stage 3: Post-deployment monitoring:** Ensuring runtime security through business logic monitoring (**RQ6**, **RQ7**, and **RQ8**).

Oracle Lifecycle (Beyond Stage 1–Stage 3): Addressing fragmented view of blockchain oracle security in decentralized ecosystems (**RQ9**).

In practice, some of these contributions are applicable in more than one stage. For instance, XploGen (Section 4.2.1) is useful in both the second and the third stage.

4.1 Stage 1: Addressing the DApp Specification Challenge

Precise and expressive formal specifications are essential for reliable design-time and runtime analyses [155]. However, existing approaches often fall short when applied to smart contracts—programs that implement complex business processes but lack notions of business processes such as roles and temporal dependencies between actions. This gap is particularly acute in early design phases, where the absence of concrete specifications and reliance on ambiguous natural language specifications (i.e., protocol whitepapers) leads developers to introduce complex and hard-to-detect business logic-related bugs into the DApp. Furthermore, formal specifications are expected to guide both design-time verification and runtime monitoring to fully capture system behavior.

Based on the mentioned gap, this thesis contributes to the specification of DApps through two key advancements: formalizing the design of smart contracts using DCR graphs (Section 4.1.1) and preventing faults through DCR modeling of smart contracts (Section 4.1.2).

4.1.1 Contribution 1: Formalizing Smart Contract Design Patterns with DCR Graphs

Introduction and Motivation

Smart contracts are blockchain-based programs that often encode complex business processes, tying together roles, time-based restrictions, and intricate ordering of actions. Mainstream platforms such as Ethereum give developers considerable flexibility to build applications using smart contracts, but they provide only low-level primitives for these features in languages like Solidity. As a result, specifying and enforcing high-level logic such as who may call a certain function (roles), which actions must precede others (ordering constraints), or how to handle delays and deadlines (time) requires ad-hoc `require` or `assert` statements dispersed throughout the smart contract source code and state variables that are difficult to verify and maintain.

In the two papers related to this contribution, we address this gap by proposing Dynamic Condition Response (DCR) graphs as a process-aware approach to modeling and analyzing smart contract designs. DCR graphs offer a natural way to capture dependencies and constraints through relations (directed arrows) between activities (see Section 2.3). By explicitly representing action sequencing, role-based access control, and time intervals, DCR graphs provide a clear, visual specification of contract behavior that can be more easily validated, composited into larger workflows, and monitored post-deployment.

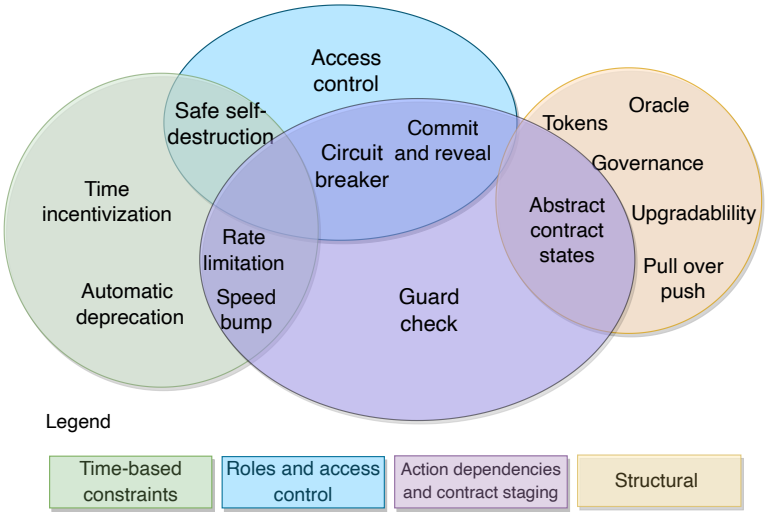


Figure 4.2: Classification of business logic design patterns in smart contracts.

Formalization of Design Patterns

DCR graphs, help us specify *business logic of the contract* and more specifically their reusable design patterns. These type of design patterns are also referred to as high-level design patterns as they present whole or part of the main functionality of the contract and are platform-agnostic.

We present a systematic view of each business logic design pattern, organized in a manner reminiscent of the Gang of Four's design patterns [100]. For each pattern, our presentation comprises:

- **Motivation:** The rationale behind the pattern and the specific challenges it addresses.
- **Formal model:** A generalized DCR graph that abstracts the key elements of the pattern, including events, relations (such as condition $\rightarrow\bullet$, milestone $\rightarrow\diamond$, and response $\bullet\rightarrow$), time constraints, and data guards.

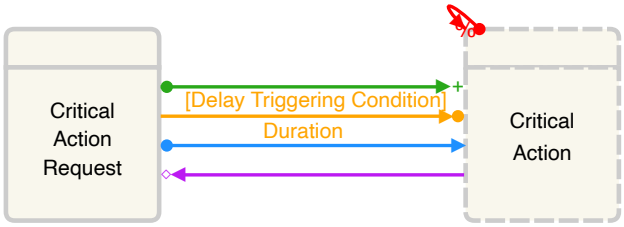


Figure 4.3: Speed bump design pattern DCR model.

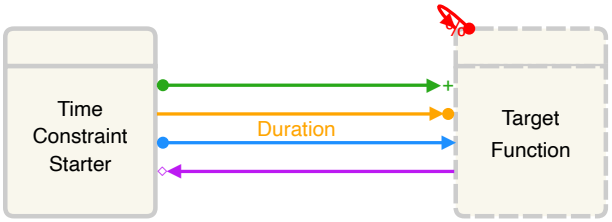


Figure 4.4: Time constraint (specialized version of speed bump) DCR model.

- **Example:** A concrete example demonstrating how the pattern can be applied to a smart contract.

Our formalization enhances clarity and reduces redundancy by generalizing and merging similar patterns. For instance, the previously treated as distinct patterns, *speed bump* and *time constraint*, are now unified into a single model. The time constraint pattern is modeled as a specialized instance of the speed bump design pattern (see Figures 4.3 and 4.4), where the guard on the condition relation is set to always true. Overall, this consolidation reduces the number of high-level patterns from previously identified 19 to 15, as illustrated by the classification in Figure 4.2.

We identify 15 design patterns in four categories from literature [33, 39, 76, 108, 156, 157, 158] as business process-level design patterns. The mentioned categories have overlaps in that each design pattern includes multiple aspects of time, access control, dependencies, or structure. Thus, we present the category assignment of design patterns as a Venn diagram in Figure 4.2. These categories are as follows:

1. **Time-Based Constraint Patterns:** These design patterns impose deadlines or delays, incentivize certain actions through time-based fines, or limit the frequency of certain operations (e.g., withdrawals in many DApps) during a given period.

2. **Access-Control Patterns:** These patterns limit access to functionality to only users holding certain roles.
3. **Action Dependency Patterns:** These patterns impose certain order over the actions.
4. **Structural Patterns:** These patterns impose a certain structure for the contract. In our models, this structure is imposed through the actions and relations of the model.

Case Studies

Beyond individual patterns, we offer three full contract models that demonstrate how multiple design patterns can be used together to form complete decentralized applications. Use of DCR graphs for modeling full contracts facilitates:

- Dynamic inclusion or exclusion of specific patterns without altering the overall contract structure.
- Clear visualization of complex interactions and dependencies inherent in smart contract logic.

Conclusion

Formalizing smart contract design patterns with DCR graphs provides a process-aware design and specification paradigm for DApps which answers to **RQ1**.

RQ1

How can a formalism from business process modeling be leveraged to systematically capture the intended business logic of decentralized applications and represent their high-level design patterns?

Result for RQ1

A declarative formalism like DCR graphs enables us to abstract smart contract functions into high-level activities—each tagged with roles, time constraints, and data dependencies—and to interconnect them via constraints (conditions $\rightarrow\bullet$, responses $\bullet\rightarrow$, inclusions $\rightarrow+$, exclusions $\rightarrow\%$) that mirror business rules. This approach not only encapsulates individual design patterns (e.g., commit and reveal, time incentivization, etc.) but also allows for combining patterns into complete, verifiable contract models that faithfully represent the intended business logic (Section 4.1.1).

Our contributions in this area offer:

- A clear, high-level specification of contract behavior that enhances comprehension and facilitates formal verification.

- A systematic framework for integrating and reusing business logic design patterns in full smart contracts.

4.1.2 Contribution 2: Model to Mitigate: Using DCR Graphs to Prevent Vulnerabilities in Smart Contracts

Motivation

Our second contribution proposes the *design-first* methodology of *Model to Mitigate* for preventing design flaws in DApps. DCR graphs have successfully been used in modeling business processes [79], and their applicability to specifying high-level smart contract logic has been demonstrated (Section 4.1.1). Our current work demonstrates how modeling the contract behavior using DCR graph enables early detection of design flaws and vulnerabilities in smart contracts.

The *key objective* of our contribution is to show, through extensive case studies on real-world exploited and audited contracts that having an explicit declarative model of smart contracts before deployement to the chain leads to:

- *Early discovery of design flaws:* Modeling logic at a high level exposes hidden complexities in access control and order-of-events dependencies before any code is finalized.
- *Efficiency optimizations:* The DCR representation highlights opportunities to reduce gas usage through better structuring of control flows.
- *Improved maintainability:* Even minor clarifications, such as explicitly initializing states, can avoid confusion and mistakes, particularly for large multi-contract systems.

Proposed Method

As Figure 4.5 shows, our methodology follows an iterative *Model to Mitigate* approach that consists of: (1) understanding a contract's logic, (2) identifying user roles, (3) mapping functions onto DCR activities, (4) modeling access control using DCR constraints and activity roles, (5) representing state transitions of the business logic using DCR relations (arrows) (6) adding time constraints (delays and deadlines).

The modeler will refine the model to match the intended business logic in multiple iterations of the loop in Figure 4.5. The exit point can be at any step after the first iteration.

Using this method, we built DCR models for extensively used and audited DApps, such as Sygma [159], as well as exploited contracts like Nomad [160], Deus DAO [161], and IOU [162]. By doing so, we demonstrate how modeling can uncover subtle flaws.

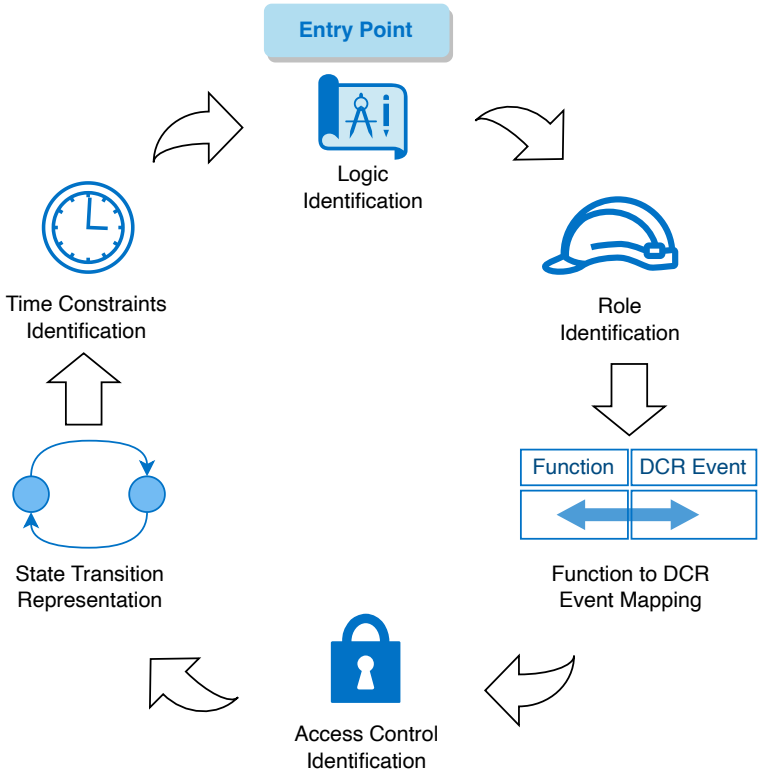


Figure 4.5: The proposed "model to mitigate" method.

Case Studies and Results

We answer **RQ2** by showing the effectiveness of our DCR-based design-first approach for analyzing several real-world contracts:

(1) Sygma Bridge contract: Although it had undergone professional manual audits, we discovered two design flaws: entangled role checks spread between multiple functions and an implicitly assumed `enum` initialization. Additionally, we identified ways to restructure a function to reduce gas usage by selectively updating contract state after re-implementing the Solidity code from DCR model.

(2) Nomad Bridge exploit: The missing condition for initializing and validating message authorization could be captured succinctly via a DCR relation. By enforcing a *condition* ($\rightarrow\bullet$) or *guarded milestone* ($\rightarrow\diamond$) on the `process` event, the exploit scenario where `confirmAt` was never correctly set, would be preempted by design.

(3) Deus DAO's Burn Exploit: Placing a correct *guard check* for the `burnFrom` function is straightforward in a DCR model, highlighting that the transaction

must fail if the function call does not satisfy the correct "balance allowance" Solidity `mapping`. This prevents the attacker from burning unauthorized balances.

(4) IOU Race Condition: As documented in [162], the contract logic for `approve` and `transferFrom` is susceptible to front-running which lets a spender quickly call `transferFrom` before `approve(0)` is processed. Our DCR graph showed the concurrency potential. Imposing a delayed condition relation $\rightarrow\bullet$ overcame the race and clarified the correct sequence of events.

RQ2

How can we use modeling before the development of the contract to discover design flaws in decentralized applications?

Result for RQ2

DCR modeling exposes design flaws by making implicit assumptions explicit. It reveals fragmented access control, unclear state initialization, and redundant or misordered state changes that can lead to both security vulnerabilities and inefficient gas usage. In short, by enforcing explicit dependencies, timing constraints, and unified authorization checks, DCR models provide clear, actionable insights to refine and secure smart contract design (Section 4.1.2).

Conclusion

Our study shows the feasibility and practical value of our *model to mitigate* method for smart contracts using DCR graphs. In our paper, we show in detail, that we can detect:

- *Hidden access control entanglements:* Granting or revoking privileges in multiple places complicate debugging and can create gas inefficiencies.
- *Under-specified or implicit states:* Relying on default `enum` values or hidden guard assumptions can cause confusing code, degrade auditability, and occasionally open the door to exploits.
- *Potential race conditions:* By decoupling or explicitly ordering critical steps, one can remove concurrency hazards or reduce them to acceptable system-level risks.
- *Implementation inefficiencies:* Misordered actions or redundant state changes needlessly consume gas—important in contract systems that may handle a large volume of transactions.

4.2 Stage 2: Pre-Deployment Analysis for Smart Contracts

Despite advancements in smart contract testing [163, 164], analysis [7, 22, 23] and repair [165, 166, 167] existing research primarily focus on low-level vulnerabilities such as reentrancy, integer overflow, and unchecked transfer operations [5, 14]. While these vulnerabilities are critical, business logic flaws—where contract implementation deviates from intended design—are increasingly responsible for substantial financial losses in DApps and are largely overlooked by the current research [168]. Addressing this gap requires a paradigm shift towards reasoning about business logic of the contract. To address this gap, we propose XploGen (Section 4.2.1). XploGen benefits the code understandability and synthesis power of LLM to synthesize exploits for the contract that target its business logic.

Our other pre-deployment stage contribution (Section 4.2.2), SInDi, helps developers and tool builders with semantic differencing of smart contract invariants. This task is crucial when building analysis and repair tools that require fast verdicts for a pair of given invariants in Solidity language to know whether they are semantically equivalent or stronger/weaker in comparison.

4.2.1 Contribution 3: Oracle-Guided Vulnerability Diversity and Exploit Synthesis of Smart Contracts Using LLMs

Introduction and Motivation

A significant portion of recent incidents in DApps stem from *business-logic flaws* where the implementation deviates from the contract's intended design [168]. Past research on exploit synthesis have largely focused on low-level vulnerabilities [29] leaving a gap in the systematic *exploitation* of business logic vulnerabilities for academic and benchmarking purposes. Hence, a rigorous methodology is needed to generate both diverse code variations that contain business logic vulnerabilities and corresponding *executing exploits*. This will present a representative dataset as a benchmark for both researchers and industrial tool builders.

In this contribution, we address the mentioned concerns by leveraging LLMs (GPT-4 [169]) to synthesize vulnerability-exploit pairs for smart contracts. Instead of relying exclusively on natural language descriptions or static analyzers, our approach guides the code generation with a *machine-readable specification of the contract's business logic*. The chosen specification formalism is DCR graphs [15]. The DCR model is used in two ways. First, it is embedded as context in the prompt to *inject vulnerabilities* that violate the declared logic. Second, a runtime monitor (Section 4.3.2) verifies whether the exploit truly causes a violation of the DCR-specified rules. By providing both a *precise definition* of "correct" behavior and a real-time check for malicious transactions, this design ensures that the synthesized exploits are valid evidence of how the contract's state can be driven out of its intended boundaries.

```
I have modeled a smart contract using DCR graphs.
You are provided:  semantics of DCR graphs, the contract in Solidity, its
DCR model, the exploit that abuses a vulnerability in the contract.

DCR semantics summary in natural language

```solidity
// Complete code of the contract
```

```xml
// Summarized DCR model
```

```javascript
// Exploit function source code
```

Task: To instrument the contract to change the vulnerability and create
another business-logic vulnerability. Only the new contract and exploit
are to be returned, with no further explanation. The exploit should try
to break the properties defined in the DCR model.
```

Figure 4.6: Oracle-guided prompt used to inject vulnerabilities into contracts and synthesize exploits.

Methodology

We selected five representative smart contracts, each consisting of tens to one hundred lines of Solidity code. These contracts cover governance, escrow, auctions, and inventory management scenarios. All of them come with associated DCR models that capture their business logic.

The XploGen's methodology consists of generating vulnerable contract variants and exploits. For each of the mentioned set of five contracts, a "seed contract" is embedded in a prompt to GPT-4, together with the semantics of the DCR model in natural-language form, the machine-readable DCR specification of the contract, and an example exploit (Figure 4.6). The LLM is asked to *instrument* the original contract minimally to inject a vulnerability into it and provide an exploit function for the injected vulnerability. The vulnerability introduces a logic flaw given the logic specification, such as removing or weakening critical `require` checks.

Each pair of instrumented contract and its exploit is deployed locally, then the exploit is executed. We use our HighGuard tool (contribution in Section 4.3.2) to monitor this contract at runtime, and check the DCR constraints. A *violation* flagged by HighGuard indicates that the exploit indeed forced the contract to deviate from its intended logic.

The rest of our paper presents details on how many of these generated contract

```
1 function applyDiscount() public
      {
2 require(block.timestamp <=
      discountEndTime, "Discount
      period ended."
3 );
4 if (price > 0) {
5 price -= 1;
6 }
7 discountEndTime = block.
      timestamp + 1 minutes;
8 }
```

(a)

```
1 async function exploit(web3,
      envInfo, contractAddress) {
2 for (let i = 0; i < 10; i++) {
3 await contractInstance.methods.
      applyDiscount().send({
4 from: web3.eth.accounts.wallet
      [0].address,
5 gas: 3000000
6 });
7 await sleep(500);
8 }
9 }
```

(b)

Figure 4.7: (a): Injected vulnerability into `ProductOrder` contract (line 7) (b): The accompanying synthesized exploit for the injected vulnerability into `ProductOrder`

variants were compilable, and among them, how many had successful exploit transactions. Notably, out of the 104 synthesized contracts, 91 compiled, and 52 were shown to be "exploitable" (i.e., at least one transaction truly violated the DCR business logic specification).

In a separate experiment on purely *exploit diversification* (from an exploit seed), the we investigate the performance of LLM-based exploit generation without giving the model a formal DCR specification (only natural-language cues in source code comments). Although the LLM could produce multiple exploits, they tended to generate lengthier transaction sequences with fewer successful property violations.

An Injected Vulnerability and Synthesized Exploit Demonstration

The `ProductOrder`[1] smart contract suffers from a business logic flaw in the function `applyDiscount`. As Figure 4.7(a) shows, after checking that the discount period is still valid, the function subtracts 1 from `price` (line 5) and then erroneously resets the discount period (line 7). This reset allows repeated applications of the discount. The synthesized exploit abuses this design flaw by repeatedly calling `applyDiscount`. Figure 4.7(b) contains the core of the exploit script where the function `exploit` (line 1) loops through multiple calls to `applyDiscount`, continuously renewing the discount period and ultimately manipulating the order price.

[1] `https://github.com/mojtaba-eshghie/HighGuard/blob/main/contracts/src/regular/ProductOrder.sol`

Running the mentioned exploit against its injected vulnerability resulted in the model trace presented in Figure 4.8. As the trace shows, the first transaction to `applyDiscount` does not incur any damage. However, the next transactions are violating the properties of the protocol specified by the oracle (DCR specification of the of contract).

Key Contributions

- *Vulnerability Injection and Exploit Synthesis:* Using the DCR specification, we automated the vulnerability injection tied to real requirements, addressing the gap in existing benchmarking datasets [29, 170]. The synthesis of the exploits is what sets our work apart, as the concrete exploit helps in tasks such as analysis [55, 56] and automated code repair tool [60, 61, 62] evaluations.

- *Integration of DCR Graphs as Oracle:* We build upon prior contributions (Sections 4.1.1 and 4.1.2) on modeling contract logic with DCR graphs [15], applying it here as both a guiding oracle for exploit creation and an online monitor for violation detection [66].

- *Empirical Evaluation of XploGen:* Five distinct contracts are used, each systematically mutated 25 times (except one with 4 variants) to produce a total of 104 contract-exploit pairs. The experiment underscores how DCR-based guidance boosts exploit precision and reduces the number of transactions the exploit needs to harm the contract.

- *Comparison of Formal DCR Model with Comment-Based Exploit Generation*: A side-by-side evaluation highlights that formal DCR prompts yield fewest transactions per exploit compared to purely comment-based instructions.

- *Open-Source Artifact:* All generated exploit-contract pairs are made available for reproducibility, enabling further testing, training, and evaluation of automated security tools[2].

Results and Observations

Our experiments show the following results which answer **RQ3**:

- *Compilability vs. Edit Distance:* The LLM sometimes over-edits the contract, causing compilation failures. However, the overall success rate of producing compilable code was fairly high (91 out of 104). We note that the *Jaccard similarity* [171] between the original and modified contract code does not reliably indicate whether a variant is compilable.

[2] `https://github.com/mojtaba-eshghie/XploGen`

Activity ID	Time	Violation	Simulation
applyDiscount	2024-05-19T13:07:13.907Z	false	2011378
applyDiscount	2024-05-19T13:07:15.147Z	true	2011378
applyDiscount	2024-05-19T13:07:15.324Z	true	2011378
applyDiscount	2024-05-19T13:07:16.026Z	true	2011378
applyDiscount	2024-05-19T13:07:17.273Z	true	2011378
applyDiscount	2024-05-19T13:07:17.414Z	true	2011378
applyDiscount	2024-05-19T13:07:18.018Z	true	2011378
applyDiscount	2024-05-19T13:07:19.393Z	true	2011378
applyDiscount	2024-05-19T13:07:19.433Z	true	2011378
applyDiscount	2024-05-19T13:07:19.600Z	true	2011378

Figure 4.8: Model traces generated by our monitoring tool while observing exploit transactions for Figure 4.7

- *Efficient vs. Lengthy Exploit Sequence:* When guided by DCR specifications, the LLM tends to produce *fewer* transactions to achieve violations. On average, the generated exploits required about 3.5 transactions. By contrast, when the exploit is generated without a formal oracle, scripts may contain large sequences of 10–20 steps with partial or no success.

RQ3

How does integrating a machine-readable oracle—via formal business logic specifications—into LLM prompting influence the automatic synthesis of (*vulnerability*, *exploit*) pairs for smart contracts compared to relying solely on natural language?

Result for RQ3

Integrating formal, machine-readable business logic specifications (via DCR graphs) into LLM prompts directs the synthesis process to produce precise, context-aware *vulnerability–exploit* pairs. This approach enables the LLM to generate exploits that more directly violate the intended contract behavior—achieving a 57% success rate with an average of just 3.5 transactions per exploit, compared to 20.1 when the prompt does not include the formal oracle (Section 4.2.1).

Conclusion and Future Directions

Our work on XploGen demonstrates that LLMs can effectively generate both *contract variations* with targeted logic flaws and *matching exploits*, particularly

when guided by a machine-readable specification. These findings suggest several research directions, including deeper integration with formal verification, where exploit sequences could be linked with automated model checkers or symbolic analysis to identify minimal or guaranteed exploit paths. Another avenue involves expanding the approach to larger systems and cross-contract interactions, enabling the modeling of more complex real-world DApps, including multi-contract and cross-chain frameworks. Additionally, the same prompting paradigm could be leveraged for automated repair, where LLMs suggest fixes for the vulnerabilities they introduce [172, 173, 174].

4.2.2 Contribution 4: SInDi: Semantic Invariant Analysis For Smart Contracts

Motivation and Introduction

Invariants are enforced through `require/assert` statements in Solidity [13] code. They play a critical role in maintaining security of contracts by reverting harmful transactions. These invariants can can be statically synthesized from protocol specifications [175] or source code and mined dynamically [52] using the transaction history of the DApps on blockchain. Prior works on invariant mining generate many noisy invariants and are not readily useful in tasks such as fuzzing for vulnerability detection [57, 58, 59].

Furthermore, the Solidity statements that enforce these invariants may be refactored or evolve across Solidity versions (e.g., the move from using `SafeMath` to native arithmetic in Solidity 0.8.0) without changing their semantic meaning [53]. The same issue persists for tasks such as automated vulnerability repair which relies on injecting `require` statements in the source code of the contracts, and the semantic equivalence or strength of the injected statements should be checked against the ground truth or previous invariants in the same source location.

In this contribution, we introduce SInDi, a tool that performs semantic invariant differencing on Solidity smart contracts. Unlike traditional syntactic analyses [63], SInDi focuses on the underlying semantics of `require` and `assert` statements to determine whether two invariants are equivalent, one is strictly stronger than the other, or if they are unrelated according to Definition 3. SInDi transforms Solidity predicates into canonical symbolic forms, and leveraging a hybrid algorithm that combines symbolic simplification and selective SMT solving [130, 176], it produces a verdict of their semantic relation. This accelerates invariant comparison (averaging 0.09 seconds per pair).

We further developed another tool, KIWI that translates Daikon-style invariants [123, 124] into Solidity expressions, clusters them into groups, and by performing invariant differencing using SInDi, it reduces noises such as weak and redundant invariants from each cluster.

Definition 3 (Invariant Strength Operator)**.** Let S denote the set of all possible program states of a smart contract, and let

$$P, Q : S \to \{\texttt{true}, \texttt{false}\}$$

be two invariants—that is, Boolean predicates over S (typically appearing in `require` or `assert` statements). We define the *invariant differencing* operator Δ as mapping pairs of predicates to their logical relationship:

$$\begin{aligned}\Delta : (S \to \{\texttt{true}, \texttt{false}\}) \times (S \to \{\texttt{true}, \texttt{false}\}) \\ \to \{\texttt{Equivalent}, \texttt{P stronger}, \texttt{Q stronger}, \texttt{Incomparable}\}\end{aligned}$$

The operator is defined by the following cases:

$$\Delta(P, Q) = \begin{cases} \texttt{Equivalent}, & \text{if } \forall s \in S,\ P(s) \iff Q(s), \\ \texttt{P stronger}^{3}, & \text{if } \forall s \in S,\ P(s) \implies Q(s) \\ & \quad \text{and } \exists s \in S \text{ with } Q(s) \wedge \neg P(s), \\ \texttt{Q stronger}, & \text{if } \forall s \in S,\ Q(s) \implies P(s) \\ & \quad \text{and } \exists s \in S \text{ with } P(s) \wedge \neg Q(s), \\ \texttt{Incomparable}, & \text{otherwise.} \end{cases}$$

Remark. The definition characterizes the semantic relationship between predicates based on the entire state space S. To determine $\Delta(P, Q)$ computationally for smart contracts, where S is typically intractably large, SInDi relies on symbolic reasoning techniques (such as logical simplification and potentially Satisfiability Modulo Theories (SMT) solving on the canonical expressions) rather than state enumeration.

Evaluating SInDi

Our evaluation of SInDi on 200 real-world invariant pairs demonstrated both high accuracy and impressive performance. In every case, SInDi correctly classified the relationship between invariant pairs whether one was stronger, equivalent, or unrelated.

The average verdict production time of 0.09 seconds highlights its potential for integration into other software engineering pipelines such as automated program repair systems [165, 166, 167].

Furthermore, we evaluate KIWI on a dataset of dynamically mined invariants of 111 real-world contracts[4]. KIWI showed an effective rate of removing up to 41.8% unuseful invariants on average.

[3] We say P is *stronger* than Q if the set of states satisfying P is a proper subset of the states satisfying Q; that is, P is true less often or is more restrictive than Q.

[4] We utilize the transaction history of the contracts.

RQ4

Despite their syntactic variations across Solidity versions how can we perform quick and reliable semantic invariant differencing for smart contracts?

Result for RQ4

By converting invariants into canonical symbolic forms, we can perform semantic invariant differencing that eliminates syntactic variations (e.g., between `SafeMath.add(a, b)` and `a + b`). We use symbolic simplification and logical implication checks (with SMT solvers when necessary) to determine whether one invariant guarantees another one to identify if one invariant enforces a stronger property or if they are equivalent in SInDi (Section 4.2.2).

RQ5

How can we use semantic invariant analysis to refine automatically mined invariants to prepare them further for downstream analysis tasks for smart contracts?

Result for RQ5

We cluster similar predicates in dynamically mined invariants and eliminate redundant, weak, or conflicting ones through semantic invariant differencing. This process reduces the invariant set by up to 41.8% with 100% precision in removals (Section 4.2.2).

Conclusion and Future Directions

In summary, SInDi helps distinguish behavioral equivalence from mere syntactic variation. Its integration into analysis tools such as KIWI, further demonstrates its practical utility. Future work can focus on extending the symbolic reasoning capabilities with runtime information, and accommodating new domain-specific languages (DSL) for specifying invariants.

4.3 Stage 3: Post-Deployment Monitoring

Static analyses and pre-deployment audits help with preventing incidents, but they may miss unforeseen scenarios that emerge once the contract is deployed on the adversarial environment of trustless blockchains. Runtime monitoring (see Definition 2) continuously observes contract activity, detecting deviations from the intended behavior whether due to unforeseen edge cases, evolving attack strategies, or complex multi-transaction interactions. This continuous oversight provides the basis for interventions to minimize financial losses.

4.3.1 Contribution 5: Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning

Introduction and Motivation

DAO attack [7] in 2016 is a classic example of reentrancy attack (Section 2.2.2) that went unnoticed during the design, development, and testing of the DApp. Real-time detection detection of the attack transaction and intervention by either reverting the transaction or pausing the contract [15] could prevent the huge financial loss as a result of this incident [177].

Traditional analysis solutions often rely on contract code to examine the logic of the contract [24, 26]. Moreover, traditional monitoring and runtime analysis solutions may rely on overfitting on the expected contract behavior to detect upcoming malicious interactions [67, 177]. However, the wide variety of attack scenarios even for the same attack type, motivates the development of a runtime detector that can generalize to a wider range of attacks from the same category without the need to specify the exact attack behavior, overfit on the normal contract behavior, or access to the source code of the contract. Our paper on Dynamit, introduces a machine learning-based monitor that detects malicious interactions with the contract (with a focus on reentrancy) by only inspecting transaction metadata from the blockchain. This approach is also interesting because it forgoes code instrumentation or the need for detailed domain-specific modeling which we expand upon in our next contribution (Section 4.3.2), aiming instead to learn malicious behavioral patterns from labeled transactions.

The Proposed Approach

This work primarily targets the *reentrancy* vulnerability [10]. In a reentrancy attack, a malicious contract repeatedly calls back into a vulnerable function before the original invocation completes, enabling the attacker to withdraw more funds or trigger unintended behavior. Our contribution takes a *transaction-centric* view: rather than analyzing contract bytecode or source code, it focuses on runtime metadata such as *gas usage*, *balance changes*, and *call stack depth* to relevant contract pairs, and partial information about call-stack depth during a transaction. The hypothesis is that certain *execution signatures* in these metadata features can reveal malicious activity [178].

Dynamit consists of two key components:

- A **Monitor**, which observes relevant on-chain activity via the Ethereum client's APIs [179].
- A **Detector**, which applies machine learning to classify observed transactions as either *benign* or *harmful*.

The detector is trained with a labeled dataset of transaction metadata extracted from transaction receipts and traces, where each sample is tagged to

indicate whether a vulnerability was exploited. The learned model then predicts, at runtime, if a newly observed transaction is likely malicious. By analyzing only the metadata , Dynamit bypasses the complexities of code-level pattern matching.

We implemented DYNAMIT[5] on a private Ethereum testnet. The **Monitor** subscribes to transaction events using the `Web3-JS` library [179], retrieving transaction receipts and states (e.g., contract balances) before and after a transaction. This yields a concise feature vector for each transaction, including: *gas usage of the transaction.*, *Balance difference of Contract 1* (service contract), *Balance difference of Contract 2* (user contract), *Average call-stack depth during the execution.* These features are then passed to the **Detector**, which implements several classification models from the `scikit-learn` library [180], including: Random Forest, Naive Bayes, Logistic Regression, K-Nearest Neighbors, Support Vector Machine, and a simple multi-layer perceptron network. The training step requires a set of transactions labeled *a priori* as benign or harmful. Once trained, the model is applied online to new transactions, real-time classification of potential attacks.

Results

We evaluate DYNAMIT with an experimental setup consisting of:

- *Service contracts* (some robust, some containing reentrancy flaws).
- *User contracts* (some benign, some malicious attackers).

A total of 25 publicly available (open-source) service contracts are complemented by 20 manually created user contracts, producing transaction pairs in which malicious or benign interactions are enacted. Additionally, a randomization mechanism is introduced to generate 80 further transactions with varied behavior (e.,g., injected loops or random donation amounts), increasing the dataset diversity. In total, the experiment collects 105 transactions: about half labeled benign, half labeled harmful. We train and test different machine learning models via 10-fold cross-validation.

We measured the performance primarily in terms of false positives (benign transactions labeled as harmful), false negatives (harmful transactions labeled as benign), and overall accuracy. Among the detectors, the Random Forest classifier achieves the best results, with on average 94% accuracy on the dataset.Notably, it obtains a balance between false positives and false negatives. This answers **RQ6**.

[5] `https://github.com/mojtaba-eshghie/Dynamit`

RQ6

How well can a machine learning model act as a general monitor to detect vulnerabilities in Ethereum smart contracts without requiring the source code of the contract, instrumentation, or manual effort to define the vulnerability?

Result for RQ6

A machine learning model based on transaction traces can effectively monitor Ethereum smart contracts for reentrancy. In our DYNAMIT tool, a Random Forest classifier achieved 94% accuracy in detecting reentrancy attacks on 105 transactions without needing source code, instrumentation, or manual vulnerability definitions (Section 4.3.1).

Contributions

We summarize the contributions of DYNAMIT as follows:

- *Trace-Centric Monitoring:* By focusing on runtime information from blockchain transactions, the framework eliminates the need for code instrumentation or availability.
- *Generalizable Detector:* Because it needs only transaction traces, the method can be adapted to additional vulnerabilities (e.g., price manipulation attacks [14]) by extendeding feature sets. This eliminates the need for domain knowledge which is often required in rule-based techniques [48, 140].
- *Robust Evaluation Methodology:* Through a combination of real-world exploit transactions and random interactions the authors train the detector on a balanced a dataset.

Conclusion and Future Directions

A dynamic, metadata-based approach can effectively classify harmful contract interactions with respect to known classes of vulnerability such as reentrancy. Our monitoring tool achieves high accuracy without manually crafted rules or static analysis. In practice, this method can be integrated into existing blockchain clients to provide real-time alerts or serve as part of a security information and event management pipeline.

Potential directions for extending this work include:

- *Automatic Benchmark Generation:* Leveraging fuzzing [58, 181] or other test-case generation mechanisms [163, 164] to produce large-scale labeled data to improve the ML model to detect other attack categories. We followed this path of exploit synthesis for business logic vulnerabilities with XPLOGEN (Section 4.2.1).

- *Expanded Feature Set:* Including additional runtime features such as (partial) internal states before and after transaction execution or function-level call sequences to detect more complex and diverse attacks [27].

- *Monitoring Transaction Sequences:* Expanding the detector to observe and detect malicious interactions over sequences of transactions will enable detecting temporally distributed malicious interaction patterns. The dataset generated in our XploGen[6] [68] work contains such multi-transaction attacks. Furthermore, using a dedicated fuzzer [27, 28, 29] could increase the diversity of attack scenarios and increase the capacity of the detector to detect more complex attacks.

4.3.2 Contribution 6: HighGuard: Cross-Chain Business Logic Monitoring of Smart Contracts

HighGuard is our novel runtime monitoring tool that bridges the monitoring gap between a smart contract's intended business logic and its actual execution both in single-chain and cross-chain environment scenarios. By leveraging DCR graphs as formal, process-aware specifications, HighGuard checks that each transaction to the contract conforms with the designer's original intent without requiring any modifications to the deployed contract code or incurring additional gas costs.

Key Contributions:

- *Chain Agnosticism:* HighGuard's architecture supports monitoring smart contracts across different blockchain platforms. Whether deployed on Ethereum, Avalanche, or any other chain, the tool uses a unified DCR-based specification to validate contract behavior.

- *DCR-Driven Verification:* By mapping contract functions and emitted events to activities in a DCR graph, HighGuard interprets the execution trace in terms of high-level business logic. Any deviation—such as an out-of-sequence transaction or an unauthorized state change—is immediately flagged.

- *Non-Intrusive, Off-Chain Monitoring:* Operating off-chain, HighGuard avoids interfering with contract execution. Its external monitoring engine observes transaction flows in *near* real-time, providing actionable alerts.

Business Logic as an Oracle for Runtime Monitoring

Unlike low-level security vulnerabilities (e.g., reentrancy, integer overflow), business logic vulnerabilities stem from violations of the contract's intended workflow.

[6] https://github.com/mojtaba-eshghie/XploGen/tree/main/synthesized_exploits/synthesized-main-prompt

These flaws often manifest as unintended function call sequences, role misassignments, or failures to enforce crucial constraints. Since such issues are inherently contract-specific, generic static analysis tools struggle to detect them.

HighGuard addresses this by treating a contract's DCR graph model as an **oracle** for correct behavior. Transactions executed on-chain are mapped to DCR events, and their validity is checked against the evolving contract state. If an execution deviates from the expected process flow (e.g., skipping a required approval step, violating a timing constraint), HighGuard flags the transaction as a violation.

Cross-Chain Monitoring Capability

With the growing adoption of cross-chain DApps verifying their security and compliance with the intended business logic is crucial. HighGuard is designed to be *chain-agnostic*, meaning it can monitor smart contracts deployed on different blockchains while ensuring that cross-chain transactions conform to their intended logic. This capability is particularly useful for applications such as cross-chain asset bridges, where funds are locked on one chain and released on another, DeFi protocols, or multi-chain governance frameworks.

Evaluation and Findings

To validate HighGuard's effectiveness, we tested it on multiple smart contract scenarios with injected business logic vulnerabilities, including an escrow contract, a governance contract, and a cross-chain decentralized exchange (DEX). In each case, HighGuard successfully detected violations where:

- Unauthorized role assignments allowed unintended function executions.
- Cross-chain transactions were processed without verifying the source state.
- Time-based constraints (e.g., grace periods) were bypassed.

Across 54 exploit scenarios, HighGuard achieved a 100% detection rate with no false positives or negatives, demonstrating its reliability in identifying contract-specific logic violations. This answers **RQ7** and **RQ8**:

RQ7

How effectively does a monitoring framework based on the formal DCR specification of a smart contract, detects deviations in its business logic?

Result for RQ7

Our monitoring tool HighGuard(Section 4.3.2) grounded in formal DCR specifications, achieved flawless detection of business logic deviations in smart contracts. In our single-chain evaluation involving 52 exploit scenarios, the system flagged every instance of deviation without any false positives or negatives (Section 4.3.2).

RQ8

How well can a DCR-based monitor detect the cross-chain business logic vulnerabilities?

Result for RQ8

By applying the DCR-based monitor to a decentralized exchange spanning Ethereum and Avalanche, HighGuard successfully detected vulnerabilities—such as issues with transaction expiration and double-payouts—with perfect precision, confirming its effectiveness in identifying cross-chain business logic flaws (Section 4.3.2).

Conclusion

HighGuard represents a shift towards *process-aware security* for smart contracts. By integrating formal business logic models with runtime monitoring, it enables precise and efficient detection of contract-specific vulnerabilities without requiring intrusive code modifications or additional gas costs. Its cross-chain capability further extends its applicability to multi- and cross-chain applications.

4.4 Beyond Stage 1–Stage 3

While our contributions in S1–S3 tackle gaps related to formal specification and pre-deployment analysis to post-deployment monitoring, they primarily address incidents related to smart contracts. At the same time, decentralized applications that rely on blockchain oracles require a more holistic view of DApp security. Our next contribution addresses this gap.

4.4.1 Contribution 7: From Creation to Exploitation: The Oracle Lifecycle

Introduction and Motivation

Blockchain oracles feed external data into on-chain smart contracts (and vice versa). Therefore, they remain a prime target for attacks that threaten trust in DApps that rely on external data such as exchange rates, sensor readings, or

off-chain events [64, 65, 182]. The current work proposes a holistic view of oracle security by examining data *throughout* its lifecycle in the oracle ecosystem rather than focusing only on the final price or on the oracle interface.

Several real-world exploits have highlighted that oracle failures particularly in the DeFi protocols can lead to irreparable financial damage and eroding user trust in decentralized platforms [168]. Existing solutions focus on specific defenses on the smart contract layer (e.g., multi-signature submissions or time-weighted average prices), but attackers exploit weaknesses that may appear in multiple stages: data *creation*, *submission*, *consensus*, *election*, or *deprecation*. Motivated by a desire to design oracles that remain resilient across these stages, this contribution presents a *generalized lifecycle model* to map and systematically mitigate vulnerabilities. In doing so, the authors not only contribute a structured classification of attack surfaces but also propose defensive measures, including a detailed examination of *bond systems* as a cost-imposing mechanism on potential attackers.

Proposed Oracle Data Lifecycle Model

Drawing on four major oracle systems: Chainlink [64], Tellor [65], Coinbase [183], and Uniswap V2 [184], we propose a five-stage lifecycle (Figure 4.9) that captures how data *evolves* inside these oracles:

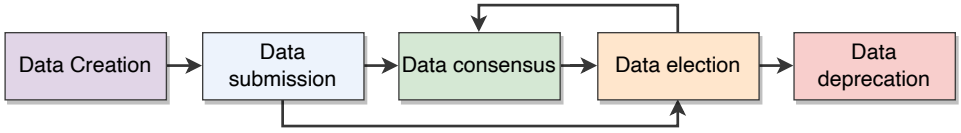


Figure 4.9: The generalized lifecycle model for oracle data stages

1. **Data Creation** (*Stage 1*): Data originates either as a measurement (e. g., sensor readings), the result of a market event (e. g., an exchange updating a trading pair), or an on-chain transaction (e. g., a DEX trade).

2. **Data Submission** (*Stage 2*): The freshly created data is relayed (often by external submitters or automatically by node operators) to the oracle contract.

3. **Data Consensus** (*Stage 3*): For oracles that require multiple submissions or a dispute process (e. g., m-of-n or crowdsourced oracles), a consensus stage consolidates disparate data points or addresses conflicts before a final value is approved.

4. **Data Election** (*Stage 4*): At this stage, the oracle finalizes and picks a single piece of data as *truth*. Other system components will consume this *elected* data.

5. **Data Deprecation** (*Stage 5*): Outdated or invalid data is eventually superseded. In practice, price feeds and time-sensitive metrics must be updated or replaced before they become stale.

Attack Types and Mitigation

To ground the lifecycle model in practice, we classify and discuss nine distinct attack types, their prevention/mitigation mechanism, and the relevant lifecycle stages they align with. These attacks include:

1. **Key Compromise**: Attackers obtain or coerce the private key(s) of permissioned oracles (often at *creation* or *submission*), manipulating data at will. Recommended defenses include multi-signature thresholds and partial anonymity of key holders.

2. **Supply Chain Attack on Oracle Data Instruments**: Physical or firmware tampering with sensors. Mitigations involve multiple vendors/instruments and device inspections.

3. **Denial of Serving the Data for Permissioned Oracles**: Attackers disrupt the network or service that publishes the data feed. Introducing robust update timers and distributed architectures lessen this risk.

4. **Targeted Attacks on Nodes in m-of-n Consensus**: Denial-of-Service, bribery, or legislative pressure on critical nodes can break quorum in consensus or sabotage data. Proxy layers and bond systems can help defend honest participants.

5. **Sybil Attacking m-of-n Consensus**: Attackers artificially inflate their voting power by spinning up new nodes. Requiring non-trivial *bonded stakes* for node registration reduces Sybil vectors.

6. **Denial of Service for On-Chain Components**: Raising block gas fees or controlling block validators can prevent timely data election or disputing. Adequate dispute intervals and careful monitoring are recommended.

7. **Reentrancy Attacks on Oracles**: An attacker reenters the oracle logic mid-transaction, tricking read-only queries or partial state updates. The paper highlights the significance of reentrancy locks, checks-effects-interactions, and caution with `view/pure` functions.

8. **Price Manipulation**: Same-block or multi-block manipulations (e. g., via flash loans) on Automated Market Makers (AMMs) that are used as oracles is an example of this attack. Time-weighted average price mechanisms, circuit breakers, and sufficient liquidity reduce exploitability.

9. **Mass Disputing Elected Data**: If each dispute temporarily freezes the data, spamming disputes can deny oracles altogether. A fee system that scales with repeated dispute attempts is proposed to thwart spammers.

We match each of the seven reported, high-profile DeFi exploits (worth $ 187 million combined) to one or more of these nine attack categories.

Bond System Evaluation

Among the various mitigation strategies proposed by our paper, *bond systems* receive special attention as a cost-based deterrent. A bond system is an economic deterrence mechanism in which participants in an oracle's consensus process are required to stake a certain amount of tokens. If they behave maliciously such as by manipulating data, they forfeit their stake [65]. We model a scenario similar to the Tellor oracle [65] where participants must stake tokens to submit or dispute data. By forcing attackers to acquire an economically significant stake, the overall cost of corrupting or halting consensus rises. Through simulations with historical price and liquidity data, the paper demonstrates:

- **Cost Exponentiality**: For a malicious actor to introduce enough nodes (or votes) to dominate an m-of-n bond system, each bond purchase drives the token price higher, steadily increasing the attacker's cost.
- **Deterrence vs. Accessibility**: If the bond price is set too high, honest participants may be deterred from joining the oracle. If it is set too low, well-capitalized adversaries can cheaply capture consensus.
- **Realistic Attack Barriers**: Under dynamic liquidity constraints, the actual cost for a large-scale Sybil or bribery attack can become prohibitive, significantly reducing the net profit potential of exploitation.

Results and Conclusion

By identifying five data stages and systematically mapping nine possible attacks, we provide an answer to **RQ9**. The lifecycle perspective clarifies how vulnerabilities at each stage (data creation, submission, consensus, election, deprecation) compound to produce catastrophic exploits if left unaddressed.

To illustrate the efficacy of a life-cycle approach, we examine multiple real incidents and connecting them to distinct vulnerabilities and lifecycle stages.

RQ9
Does the data obtained form blockchain oracles flow through different stages and how does each stage affect the attack vectors on oracles?

Result for RQ9

Blockchain oracle data flows through several stages—*creation*, *submission*, *consensus*, *election*, and *deprecation*—and that each stage presents unique security challenges. For instance, the creation stage can be exposed to supply chain attacks on data instruments, while the submission and consensus stages are vulnerable to key compromises and targeted node attacks. Recognizing this segmented lifecycle allows for the development of stage-specific mitigation strategies (such as tailored bond systems), enhancing the resilience of the oracle ecosystem (Section 4.4.1).

Future directions can explore more oracle architectures (e. g., Band [185], Augur [186], DIA [187]). Further evaluations of the bond system could benefit improved economic models (e. g., hedging strategies and advanced attacker cost modeling).

5

Future Works

This thesis set out to advance security of smart contracts. Through the development of new techniques for behavioral specification, monitoring, invariant differencing and generation, and exploit synthesis, we explored how to reason about and enforce correct behavior in an environment where immutability and financial stakes make flaws costly. A key theme running through the thesis is the use of *formal behavioral models* such as DCR graphs. By integrating these models into various stages of contract analysis from specification to runtime monitoring and exploit generation we demonstrated how business logic-level specifications can play a role in improving smart contract resilience.

The research presented here lays the groundwork for future investigations. One natural extension involves introducing a new elegant upgradability pattern for DApps. When stakeholders opt in to a deployed DApp, they implicitly agree to the terms and behaviors encoded in its smart contract at the time of deployment. However, as vulnerabilities are discovered or requirements evolve, updates to the contract's functionality may become necessary. The current approach to such changes often relies on the upgradability design pattern [15]. While effective to some extent, this pattern lacks the ability to perform fine-grained upgrades that preserve the invariants and constraints of the core business logic. To address this limitation, one promising direction is to introduce a dedicated manager contract that encodes all DCR-imposed relations representing the application's business logic. Each DCR activity would then be implemented as an individual smart contract, enabling modular and granular control over upgrades. Updates could be seamlessly applied by either modifying the address of an activity contract or introducing new relations within the manager. This architecture not only supports precise upgrades but also establishes a novel and principled approach to upgradability in decentralized systems.

When using automated invariant mining tools for smart contracts, there exists a noticeable discrepancy between the large number of automatically generated invariants by tools such as InvCon+ [52] and the relatively small set of developer-

enforced invariants expressed through `require` and `assert` statements in the source code. This imbalance raises questions about the relevance and validity of many tool-suggested invariants, especially when they are not aligned with developer intent or domain logic. Future work should explore human-in-the-loop approaches to assess whether a proposed invariant meaningfully strengthens the specification or merely overfits to observed traces, test cases, or transaction histories. Furthermore, while our current work focuses on transpiling Daikon-style invariants [123, 124] into Solidity-compatible predicates, the broader smart contract verification ecosystem makes increasing use of domain-specific languages (DSLs) such as the Certora Verification Language (CVL) [188]. Expanding transpilation support to other DSLs can increase compatibility and encourage adoption of inferred invariants in practical verification workflows.

Another underexplored area is the use of large language models (LLMs) for *exploit synthesis*. In our work, we leveraged GPT-4 [169] in conjunction with DCR-based guidance to synthesize exploits. A more systematic evaluation of different LLMs and their effectiveness across diverse vulnerability classes particularly DeFi-specific attacks [5] could further illuminate the strengths and limitations of this approach.

We also relied on manually constructed DCR models from smart contract code throughout our evaluations. Automating the generation of smart contract models either from natural language documentation, source code, or through mining transaction history remains an open and impactful area. Success here would enable scalable formalization of protocol logic and expand the applicability of business model-based analysis and verification techniques.

While this thesis models individual smart contract design patterns using DCR graphs and demonstrates their integration through case studies, a compelling direction for future research lies in formalizing and generalizing the compositionality of these patterns. The current approach shows that DCR-modeled patterns can be combined to form end-to-end contract specifications, but it stops short of defining systematic composition rules or exploring the interaction semantics of overlapping or conflicting patterns. Future work could investigate how DCR models representing different design patterns can be securely composed preserving behavioral properties such as response relations, time constraints, and action dependencies. This will enable a pattern composition framework of a modular library of verified and reusable DCR-based behavioral components for smart contracts.

A further direction involves analyzing *cross-contract dependencies and vulnerability propagation*. Many smart contract systems span multiple interacting components such as proxies, libraries, and oracles that create complex interdependencies. Vulnerabilities do not just emerge in isolation but as a consequence of these interactions. Building a framework to model and trace multi-contract call graphs, interprocedural taint flows, and shared state mutations would reveal emergent risks that current tools miss.

6

Conclusion

This thesis systematically addressed security challenges in DApps, by advancing formal models, pre-deployment analyses, and runtime monitoring. We contribute with novel methods, models, and tools that enable formal reasoning based on business processes to secure smart contracts.

We introduced the use of Dynamic Condition Response (DCR) graphs to formalize smart contract business logic, enabling precise high-level specifications of complex contract behaviors. By capturing the intent behind common design patterns, we provided a formal foundation for verifying contract correctness early in the development cycle to reduce the risk of design flaws that could otherwise lead to severe vulnerabilities.

Building on this formalization, our *Model to Mitigate* approach demonstrated how early modeling of contract behaviors using DCR graphs can identify subtle vulnerabilities and inefficiencies. Through multiple case studies on real-world audited and exploited smart contracts, we validated the effectiveness of this methodology.

Recognizing the threat of business logic vulnerabilities, we developed XploGen, a framework leveraging LLMs guided by DCR graph specifications to inject vulnerabilities and synthesize realistic contract exploits. This contribution helps inject and abuse vulnerabilities that static and dynamic analyses typically overlook and paves the way for creating adversarial test cases as well as benchmarks for assessing effectiveness of analysis tools.

To improve invariant analysis, we introduced SInDi, a novel semantic differencing tool for Solidity invariants. SInDi's capability to accurately differentiate invariant semantics beyond syntactic representations was demonstrated by creating a pipeline to denoise dynamically mined invariants from transaction history.

Addressing post-deployment security concerns, we developed two runtime monitoring tools. Dynamit leveraged machine learning to detect malicious contract interactions without intrusive modifications to contracts or instrumenting the underlying blockchain virtual machine, providing a generalizable monitor to

defend against reentrancy. Additionally, HighGuard leveraged formal business logic specification (DCR model of the contract) for monitoring. HighGuard is capable of monitoring the contract workflows in single- and cross-chain settings to ensure compliance with high-level design intents and detect complex business logic violations.

Finally, recognizing the pivotal role of blockchain oracles, we proposed a holistic oracle lifecycle model that systematically maps vulnerabilities to the lifecycle stage they happen at. Through detailed We survey and classify prevention and mitigation strategies for each attack type and evaulate one of the most important deterrents–bond systems.

These contributions collectively advance the security of DApps by providing formalizations, methodologies, and open-source reusable tools spanning the entire DApp lifecycle. By focusing explicitly on high-level business logic, formal specifications, and runtime monitoring, this thesis lays the groundwork for more resilient decentralized systems based on smart contracts.

References

[1] M. Eshghie, C. Artho, and D. Gurov, "Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning," in *Evaluation and Assessment in Software Engineering*, ser. EASE 2021. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 305–312.

[2] A. Back, "Hashcash - A Denial of Service Counter-Measure," Accessed: 2025-03-05. [Online]. Available: https://blog.infocruncher.com/resources/bitcoin-whitepaper-annotated/Hashcash%20(2002).pdf

[3] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Benerals Problem," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 203–226.

[4] N. Szabo, "Smart Contracts: Building Blocks for Digital Markets," *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, no. 2, p. 28, 1996.

[5] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, "SoK: Decentralized Finance (DeFi) Attacks," in *2023 IEEE Symposium on Security and Privacy (SP)*, May 2023, pp. 2444–2461, iSSN: 2375-1207.

[6] G. WOOD, "Ethereum Yellow Paper," Accessed: 2023-08-29. [Online]. Available: https://github.com/ethereum/yellowpaper

[7] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. Vienna Austria: ACM, 2016, pp. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[8] "Hard Fork Completed," Accessed: 26 March 2025. [Online]. Available: https://blog.ethereum.org/2016/07/20/hard-fork-completed

[9] S. Wang, W. Ding, J. Li, Y. Yuan, L. Ouyang, and F.-Y. Wang, "Decentralized Autonomous Organizations: Concept, Model, and Applications," *IEEE Transactions on Computational Social Systems*, vol. 6, no. 5, pp. 870–878, 2019.

[10] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in *Principles of Security and Trust*, M. Maffei and M. Ryan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186.

[11] "The Parity Wallet Hack Explained - OpenZeppelin blog," Accessed: 26 March 2025. [Online]. Available: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[12] web3author, "PARITY Wallet Hack Demystified: All You Need to Know!" Accessed: 26 March 2025.

[13] "Solidity documentation," Aug. 2023, Accessed: 2023-08-29. [Online]. Available: https://docs.soliditylang.org/en/latest/

[14] M. Eshghie, M. Jafari, and C. Artho, "From Creation to Exploitation: The Oracle Lifecycle," in *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2024.

[15] M. Eshghie, W. Ahrendt, C. Artho, T. T. Hildebrandt, and G. Schneider, "Capturing Smart Contract Design with DCR Graphs," in *International Conference on Software Engineering and Formal Methods*. Springer, 2023, pp. 106–125.

[16] "OWASP Smart Contract Top 10 - OWASP Smart Contract Security," Accessed: 26 March 2025. [Online]. Available: https://scs.owasp.org/sctop10/

[17] K. W. Jie, "Why You Should Verify the Tokens You Own: A Deep Dive into Two Vulnerable ERC20 contracts| The CryptoJobsList Blog," Accessed: 26 March 2025. [Online]. Available: https://cryptojobslist.com/blog/two-vulnerable-erc20-contracts-deep-dive-beautychain-smartmesh

[18] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding Permission Bugs in Smart Contracts with Role Mining," in *SIGSOFT ISSTA 2022*. ACM, 2022, p. 716–727.

[19] SunWeb3Sec, "DeFi Hacks Reproduce - Foundry," Accessed: 26 March 2025. [Online]. Available: https://github.com/SunWeb3Sec/DeFiHackLabs

[20] Chainalysis, "$2.2 Billion Stolen in Crypto in 2024 but Hacked Volumes Stagnate," Accessed: 10 March 2025. [Online]. Available: https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2025/

[21] Shashank, "Level Finance Hack Analysis," Accessed: 10 March 2025. [Online]. Available: https://blog.solidityscan.com/level-finance-hack-analysis-16fda3996ecb

[22] C. Diligence, "Mythril: Security Analysis Tool for Ethereum Smart Contracts," Accessed: 2024-10-10. [Online]. Available: https://github.com/ConsenSys/mythril

[23] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 557–560. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3404366

[24] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework For Smart Contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[25] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static Analysis of Ethereum Smart Contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. Gothenburg, Sweden: ACM, 2018, pp. 9–16.

[26] "Semgrep/semgrep," Accessed: 2025-03-09. [Online]. Available: https://github.com/semgrep/semgrep

[27] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-Supported Dynamic Exploit Generation for Smart Contracts," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1795–1809, May 2022, conference Name: IEEE Transactions on Dependable and Secure Computing.

[28] J. Su, H.-N. Dai, L. Zhao, Z. Zheng, and X. Luo, "Effectively Generating Vulnerable Transaction Sequences in Smart Contracts with Reinforcement Learning-Guided Fuzzing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[29] Q. Zhang, Y. Wang, J. Li, and S. Ma, "EthPloit: From Fuzzing to Efficient Exploit Generation Against Smart Contracts," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2020, pp. 116–126.

[30] I. David, L. Zhou, K. Qin, D. Song, L. Cavallaro, and A. Gervais, "Do You Still Need a Manual Smart Contract Audit?" Accessed: 2025-03-09.

[31] W. Ma, D. Wu, Y. Sun, T. Wang, S. Liu, J. Zhang, Y. Xue, and Y. Liu, "Combining Fine-Tuning and LLM-based Agents for Intuitive Smart Contract Auditing with Justifications," Accessed: 2025-03-09.

[32] S. Xia, M. He, S. Shao, T. Yu, Y. Zhang, and L. Song, "SymGPT: Auditing Smart Contracts via Combining Symbolic Execution with Large Language Models," Accessed: 2025-03-09.

[33] M. Wohrer and U. Zdun, "Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity," in *IEEE IWBOSE*, 2018, pp. 2–8.

[34] "Slowmist/SlowMist-Learning-Roadmap-for-Becoming-a-Smart-Contract-Auditor," Accessed: 2025-03-09. [Online]. Available: https://github.com/slowmist/SlowMist-Learning-Roadmap-for-Becoming-a-Smart-Contract-Auditor

[35] "NatSpec Format — Solidity 0.8.29 Documentation," Accessed: 2025-03-09. [Online]. Available: https://docs.soliditylang.org/en/develop/natspec-format.html

[36] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A Survey of Smart Contract Formal Specification and Verification," *ACM Computing Surveys*, vol. 54, no. 7, pp. 148:1–148:38, Jul. 2021.

[37] "Certora/CertoraProver," Accessed: 10 March 2025. [Online]. Available: https://github.com/Certora/CertoraProver

[38] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-Design Smart Contracts for Ethereum," in *Financial Cryptography and Data Security*, I. Goldberg and T. Moore, Eds. Cham: Springer International Publishing, 2019, pp. 446–465.

[39] M. Wöhrer and U. Zdun, "Design Patterns for Smart Contracts in the Ethereum Ecosystem," in *iThings/GreenCom/CPSCom/SmartData*, 2018, pp. 1513–1520.

[40] "Hack Track: An Analysis of Poly Network Hack and Latest Related Events," Accessed: 2025-03-09. [Online]. Available: https://www.merklescience.com/blog/hack-track-an-analysis-of-poly-network-hack-and-latest-related-events

[41] Elliptic, "$600 Million in Crypto Stolen and Returned in 24 Hours," Accessed: 2025-03-09. [Online]. Available: https://www.elliptic.co/blog/the-poly-network-hack-600-million-in-crypto-stolen-and-returned-in-24-hours

[42] T. T. Hildebrandt and R. R. Mukkamala, "Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs," in *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010*, ser. EPTCS, K. Honda and A. Mycroft, Eds., vol. 69, 2010, pp. 59–73. [Online]. Available: https://doi.org/10.4204/EPTCS.69.5

[43] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[44] Raghav, "Duaraghav8/Ethlint," Accessed: 2025-03-09. [Online]. Available: https://github.com/duaraghav8/Ethlint

[45] "Protofire/solhint," Accessed: 2025-03-09. [Online]. Available: https://github.com/protofire/solhint

[46] L. Alt, M. Blicha, A. E. J. Hyvärinen, and N. Sharygina, "SolCMC: Solidity Compiler's Model Checker," in *Computer Aided Verification*, S. Shoham and Y. Vizel, Eds. Cham: Springer International Publishing, 2022, pp. 325–338.

[47] Y. Ding, C. Wang, Q. Zhong, H. Li, J. Tan, and J. Li, "Function-Level Dynamic Monitoring and Analysis System for Smart Contract," *IEEE Access*, vol. 8, pp. 229 161–229 172, 2020, conference Name: IEEE Access. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9300046

[48] J. Zhang, J. Gao, Y. Li, Z. Chen, Z. Guan, and Z. Chen, "Xscope: Hunting for Cross-Chain Bridge Attacks," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 1–4. [Online]. Available: https://doi.org/10.1145/3551349.3559520

[49] Y. Liu, Y. Li, S.-W. Lin, and Q. Yan, "ModCon: A Model-Based Testing Platform for Smart Contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 1601–1605.

[50] Z. Liu and J. Liu, "Formal Verification of Blockchain Smart Contract Based on Colored Petri Net Models," in *IEEE COMPSAC*, vol. 2, 2019, pp. 555–560.

[51] Y. Liu and Y. Li, "InvCon: A Dynamic Invariant Detector for Ethereum Smart Contracts," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 1–4.

[52] Y. Liu, C. Zhang, and Y. Li., "Automated Invariant Generation for Solidity Smart Contracts," Dec. 2023, arXiv:2401.00650 [cs]. [Online]. Available: http://arxiv.org/abs/2401.00650

[53] Z. Chen, Y. Liu, S. M. Beillahi, Y. Li, and F. Long, "Demystifying Invariant Effectiveness for Securing Smart Contracts," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1772–1795, 2024.

[54] Y. Liu, Y. Xue, D. Wu, Y. Sun, Y. Li, M. Shi, and Y. Liu, "PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation," in *Proceedings of the 32nd Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, Feb. 2025.

[55] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2189–2207, 2021.

[56] A. Ghaleb, J. Rubin, and K. Pattabiraman, "eTainter: Detecting Gas-Related Vulnerabilities in Smart Contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 728–739.

[57] C. Shou, S. Tan, and K. Sen, "ItyFuzz: Snapshot-Based Fuzzer for Smart Contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, Jul. 2023, pp. 322–333.

[58] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, Sep. 2018, pp. 259–269. [Online]. Available: https://dl.acm.org/doi/10.1145/3238147.3238177

[59] Y. Fu, M. Ren, F. Ma, H. Shi, X. Yang, Y. Jiang, H. Li, and X. Shi, "Evmfuzzer: Detect EVM Vulnerabilities via Fuzz Testing," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 1110–1114.

[60] T. D. Nguyen, L. H. Pham, and J. Sun, "sGUARD: Towards Fixing Vulnerable Smart Contracts Automatically," Accessed: 10 March 2025. [Online]. Available: http://arxiv.org/abs/2101.01917

[61] H. Jin, Z. Wang, M. Wen, W. Dai, Y. Zhu, and D. Zou, "Aroc: An Automatic Repair Framework for On-Chain Smart Contracts," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, p. 4611–4629, Nov 2022.

[62] X. Zhou, Y. Chen, H. Guo, X. Chen, and Y. Huang, "Security Code Recommendations for Smart Contract," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2023, pp. 190–200.

[63] P. Cashin, C. Martinez, W. Weimer, and S. Forrest, "Understanding Automatically-Generated Patches Through Symbolic Invariant Differences," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 411–414.

[64] S. Ellis, A. Juels, and S. Nazarov, "ChainLink: A Decentralized Oracle Network," Accessed: 2025-03-09. [Online]. Available: https://research.chain.link/whitepaper-v1.pdf

[65] Tellor, "Tellor, a Community, an Oracle, Unstoppable," Accessed: 2025-03-09. [Online]. Available: https://tellor.io/whitepaper/

[66] M. Eshghie, C. Artho, H. Stammler, W. Ahrendt, T. Hildebrandt, and G. Schneider, "Highguard: Cross-chain business logic monitoring of smart contracts," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2378–2381.

[67] J. Ellul and G. J. Pace, "Runtime Verification of Ethereum Smart Contracts," in *2018 14th European Dependable Computing Conference (EDCC)*, Sep. 2018, pp. 158–163. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8530777

[68] M. Eshghie and C. 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*, 2024, pp. 2240–2248.

[69] M. Eshghie, "Mojtaba-eshghie/HighGuard," Accessed: 2025-03-09. [Online]. Available: https://github.com/mojtaba-eshghie/HighGuard

[70] "Mojtaba-eshghie/XploGen," Accessed: 26 March 2025. [Online]. Available: https://github.com/mojtaba-eshghie/XploGen

[71] "Mojtaba-eshghie/SINA," Accessed: 26 March 2025. [Online]. Available: https://github.com/mojtaba-eshghie/SINA

[72] "Mojtaba-eshghie/KIWI," Accessed: 26 March 2025. [Online]. Available: https://github.com/mojtaba-eshghie/KIWI

[73] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008, white paper. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[74] V. Buterin and V. Griffith, "Casper the Friendly Finality Gadget," Accessed: 26 March 2025. [Online]. Available: https://arxiv.org/abs/1710.09437v4

[75] "Proof-of-stake (PoS)," Accessed: 2025-03-09. [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/

[76] "Ethereum development documentation," Accessed: 2023-08-29. [Online]. Available: https://ethereum.org/en/developers/docs/

[77] K. Mammadzada, M. Iqbal, F. Milani, L. García-Bañuelos, and R. Matulevičius, "Blockchain Oracles: A Framework for Blockchain-Based Applications," in *Business Process Management: Blockchain and Robotic Process Automation Forum*, A. Asatiani, J. M. García, N. Helander, A. Jiménez-Ramírez, A. Koschmider, J. Mendling, G. Meroni, and H. A. Reijers, Eds. Cham: Springer International Publishing, 2020, pp. 19–34.

[78] D. Nute, *Handbook of Logic in Artificial Intelligence and Logic Programming*. Clarendon Press, Oxford University Press, 1994, vol. 3, ch. Defeasible Logic.

[79] H. Normann, S. Debois, T. Slaats, and T. T. Hildebrandt, "Zoom and Enhance: Action Refinement via Subprocesses in Timed Declarative Processes," in *BPM 2021*. Cham: Springer, 2021, pp. 161–178.

[80] T. T. Hildebrandt, H. Normann, M. Marquard, S. Debois, and T. Slaats, "Decision Modelling in Timed Dynamic Condition Response Graphs with Data," in *Business Process Management Workshops*. Cham: Springer, 2022, pp. 362–374.

[81] Wikipedia, "Iso 8601," Aug. 2023, Accessed: 2023-08-29. [Online]. Available: https://en.wikipedia.org/wiki/ISO_8601

[82] M. Leucker and C. Schallhart, "A Brief Account of Runtime Verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, May 2009.

[83] "EVM Codes," Accessed: 2025-03-09. [Online]. Available: https://www.evm.codes

[84] J. Ellul and G. J. Pace, "Optional Monitoring for Long-Lived Transactions," in *Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution*, ser. VORTEX 2021. New York, NY, USA: Association for Computing Machinery, Jul. 2021, pp. 35–39.

[85] Y. Wang, Y. Tang, K. Li, W. Ding, and Z. Yang, "Understanding Ethereum Mempool Security under Asymmetric {DoS} by Symbolized Stateful Fuzzing," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4747–4764.

[86] "Transactions," Accessed: 2025-03-09. [Online]. Available: https://ethereum.org/en/developers/docs/transactions/

[87] "EVM Tracing," Accessed: 2025-03-09. [Online]. Available: https://geth.ethereum.org/docs/developers/evm-tracing

[88] S. Bliudze, A. Cimatti, M. Jaber, S. Mover, M. Roveri, W. Saab, and Q. Wang, "Formal Verification of Infinite-State BIP Models," in *Automated Technology for Verification and Analysis*, B. Finkbeiner, G. Pu, and L. Zhang, Eds. Cham: Springer International Publishing, 2015, pp. 326–343.

[89] W. Duo, H. Xin, and M. Xiaofeng, "Formal Analysis of Smart Contract Based on Colored Petri Nets," *IEEE Intelligent Systems*, vol. 35, no. 3, pp. 19–30, 2020. [Online]. Available: https://doi.org/10.1109/MIS.2020.2977594

[90] I. Garfatta, K. Klai, M. Graïet, and W. Gaaloul, "Model Checking of Solidity Smart Contracts Adopted for Business Processes," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science. Cham: Springer, 2021, pp. 116–132.

[91] N. Zupan, P. Kasinathan, J. Cuellar, and M. Sauer, "Secure Smart Contract Generation Based on Petri Nets," in *Blockchain Technology for Industry 4.0: Secure, Decentralized, Distributed and Trusted Industry Environment*, R. da Rosa Righi, A. M. Alberti, and M. Singh, Eds. Singapore: Springer, 2020, pp. 73–98.

[92] T. Abdellatif and K.-L. Brousmiche, "Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. New York, NY, USA: IEEE, 2018, pp. 1–5.

[93] C. D. Clack and G. Vanca, "Temporal Aspects of Smart Contracts for Financial Derivatives," , 2018. [Online]. Available: http://arxiv.org/abs/1805.11677

[94] M. Ron, "On the Specification and Verification of Atomic Swap Smart Contracts," arxiv, 2018. [Online]. Available: http://arxiv.org/abs/1811.06099

[95] C. Laneve, C. Coen, and A. Veschetti, "On the Prediction of Smart Contracts' Behaviours," in *From Software Engineering to Formal Methods and Tools, and Back: Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday*, ser. Lecture Notes in Computer Science. New York, NY, USA: Springer International Publishing, 2019, pp. 397–415. [Online]. Available: https://doi.org/10.1007/978-3-030-30985-5_23

[96] G. Bigi, A. Bracciali, G. Meacci, and E. Tuosto, "Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods," in *Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, ser. Lecture Notes in Computer Science, C. Bodei, G. Ferrari, and C. Priami, Eds. : Springer International Publishing, 2015, pp. 142–161. [Online]. Available: https://doi.org/10.1007/978-3-319-25527-9_11

[97] I. Sergey and A. Hobor, "A Concurrent Perspective on Smart Contracts," 2017. [Online]. Available: http://arxiv.org/abs/1702.05511

[98] K. Bansal, E. Koskinen, and O. Tripp, "Automatic generation of precise and useful commutativity conditions," in *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I 24*. Springer, 2018, pp. 115–132.

[99] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding Concurrency to Smart Contracts," in *PODC*. ACM, 2017, pp. 303–312.

[100] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Deutschland GmbH, 1994.

[101] Y. Liu, Q. Lu, X. Xu, L. Zhu, and H. Yao, "Applying Design Patterns in Smart Contracts," in *Blockchain – ICBC 2018*, S. Chen, H. Wang, and L.-J. Zhang, Eds. Cham: Springer International Publishing, 2018, pp. 92–106.

[102] "System Design - Smart Contract Security Field Guide," Accessed: 2025-03-09. [Online]. Available: https://scsfg.io/developers/system-design/

[103] M. Wohrer and U. Zdun, "From Domain-Specific Language to Code: Smart Contracts and the Application of Design Patterns," *IEEE Software*, vol. 37, no. 5, pp. 37–42, Sep. 2020.

[104] P. Antonino, J. Ferreira, A. Sampaio, A. W. Roscoe, and F. Arruda, "A Refinement-Based Approach to Safe Smart Contract Deployment and Evolution," *Software and Systems Modeling*, 2024, Accessed: 2025-03-09.

[105] B. Meyer, "Applying 'Design by Contract'," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[106] "ERC-2535: Diamonds, Multi-Facet Proxy," Accessed: 2025-03-09. [Online]. Available: https://eips.ethereum.org/EIPS/eip-2535

[107] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *ACM Conference on Computer and Communications Security*, ACM. Vienna Austria: ACM, 2016, pp. 254–269.

[108] OpenZeppelin, "OpenZeppelin Contracts," Accessed: 2023-08-29. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts

[109] Consensys, "Ethereum Smart Contract Best Practices," Accessed: 2023-08-29. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/development-recommendations/precautions/

[110] R. Hull, V. S. Batra, Y.-M. Chen, A. Deutsch, F. F. T. Heath III, and V. Vianu, "Towards a Shared Ledger Business Collaboration Language Based on Data-Aware Processes," in *Service-Oriented Computing*, Q. Z. Sheng, E. Stroulia, S. Tata, and S. Bhiri, Eds. Cham: Springer International Publishing, 2016, pp. 18–36.

[111] C. K. Frantz and M. , "From Institutions to Code: Towards Automated Generation of Smart Contracts," in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, Sep. 2016, pp. 210–215.

[112] S. E. Crawford and E. Ostrom, "A Grammar of Institutions," *American Political Science Review*, vol. 89, no. 3, pp. 582–600, 1995.

[113] M. F. Madsen, M. Gaub, M. E. Kirkbro, T. Høgnason, T. Slaats, and S. Debois, "Collaboration Among Adversaries: Distributed Workflow Execution on a Blockchain: Symposium on Foundations and Applications of Blockchain 2018," in *Symposium on Foundations and Applications of Blockchain*, 2018.

[114] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine Agreements for Cryptocurrencies," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.

[115] "The Algorand Virtual Machine (AVM) and TEAL. - Algorand Developer Portal," Accessed: 2025-03-09. [Online]. Available: https://developer.algorand.org/docs/get-details/dapps/avm/teal/specification/

[116] D. Suvorov and V. Ulyantsev, "Smart Contract Design Meets State Machine Synthesis: Case Studies," Jun. 2019, Accessed: 2023-08-29. [Online]. Available: https://arxiv.org/abs/1906.02906

[117] L. Breidenbach, P. Daian, F. Tramer, and A. Juels, "Enter the Hydra: Towards Principled Bug Bounties and {Exploit-Resistant} Smart Contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1335–1352.

[118] S. Zhou, Z. Yang, J. Xiang, Y. Cao, Z. Yang, and Y. Zhang, "An Ever-evolving Game: Evaluation of Real-world Attacks and Defenses in Ethereum Ecosystem," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2793–2810.

[119] J. Krupp and C. Rossow, "{teEther}: Gnawing at Ethereum to Automatically Exploit Smart Contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1317–1333.

[120] L. Jin, Y. Cao, Y. Chen, D. Zhang, and S. Campanoni, "ExGen: Cross-platform, Automated Exploit Generation for Smart Contract Vulnerabilities,"

IEEE Transactions on Dependable and Secure Computing, vol. 20, no. 1, pp. 650–664, Jan. 2023.

[121] R. W. Floyd, "Assigning Meanings to Programs," in *Program Verification: Fundamental Issues in Computer Science*. Springer, 1993, pp. 65–81.

[122] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[123] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, Feb. 2001, conference Name: IEEE Transactions on Software Engineering.

[124] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, Dec. 2007.

[125] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 1, pp. 30–50, 2000.

[126] Y. Yu, G. Rong, H. Shen, H. Zhang, D. Shao, M. Wang, Z. Wei, Y. Xu, and J. Wang, "Fine-Tuning Large Language Models to Improve Accuracy and Comprehensibility of Automated Code Review," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 1, pp. 14:1–14:26, Dec. 2024.

[127] X. Si, A. Naik, H. Dai, M. Naik, and L. Song, "Code2Inv: A Deep Learning Framework for Program Verification," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, vol. 12225, pp. 151–164, series Title: Lecture Notes in Computer Science.

[128] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can Large Language Models Reason about Program Invariants?" in *Proceedings of the 40th International Conference on Machine Learning*. PMLR, Jul. 2023, pp. 27 496–27 520.

[129] Z. Chen, Y. Liu, S. M. Beillahi, Y. Li, and F. Long, "OpenTracer: A Dynamic Transaction Trace Analyzer for Smart Contract Invariant Generation and Beyond," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, Oct. 2024, pp. 2399–2402.

[130] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[131] G. Navarro, "A Guided Tour to Approximate String Matching," *ACM Comput. Surv.*, vol. 33, no. 1, p. 31–88, Mar. 2001. [Online]. Available: https://doi-org.focus.lib.kth.se/10.1145/375360.375365

[132] P. Sagdeo, V. Athavale, S. Kowshik, and S. Vasudevan, "PRECIS: Inferring Invariants Using Program Path Guided Clustering," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 532–535.

[133] R. W. Hamming, "Error Detecting and Error Correcting Codes," *The Bell system technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[134] R. Huang, Q. Shen, Y. Wang, Y. Wu, Z. Wu, X. Luo, and A. Ruan, "ReenRepair: Automatic and Semantic Equivalent Repair of Reentrancy in Smart Contracts," *Journal of Systems and Software*, vol. 216, p. 112107, Oct. 2024.

[135] F. Khan, I. David, D. Varro, and S. McIntosh, "Code Cloning in Smart Contracts on the Ethereum Platform: An Extended Replication Study," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2006–2019, Apr. 2023.

[136] J. R. Cordy and C. K. Roy, "The NiCad Clone Detector," in *2011 IEEE 19th international conference on program comprehension*. IEEE, 2011, pp. 219–220.

[137] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "SmartEmbed: A Tool for Clone and Bug Detection in Smart Contracts through Structural Code Embedding," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2019, pp. 394–397.

[138] A. Li, J. A. Choi, and F. Long, "Securing Smart Contract with Runtime Validation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 438–453. [Online]. Available: https://dl.acm.org/doi/10.1145/3385412.3385982

[139] A. Li, "Solythesis," Accessed: 26 March 2025. [Online]. Available: https://github.com/aoli-al/Solythesis

[140] R. K. Shyamasundar, "A Framework of Runtime Monitoring for Correct Execution of Smart Contracts," in *Blockchain – ICBC 2022*, ser. Lecture Notes in Computer Science, S. Chen, R. K. Shyamasundar, and L.-J. Zhang, Eds. Cham: Springer Nature Switzerland, 2022, pp. 92–116.

[141] "Consensys/scribble," Accessed: 26 March 2025. [Online]. Available: https://github.com/Consensys/scribble

[142] M. Capretto, M. Ceresa, and C. Sánchez, "Transaction Monitoring of Smart Contracts," in *Runtime Verification*, ser. Lecture Notes in Computer Science, T. Dang and V. Stolz, Eds. Cham: Springer International Publishing, 2022, pp. 162–180.

[143] "Forta Litepaper," Accessed: 26 March 2025. [Online]. Available: https://docs.forta.network/en/latest/2022-7-11%20Forta%20Litepaper.pdf

[144] "Getting Started - Forta Docs," Accessed: 26 March 2025. [Online]. Available: https://docs.forta.network/en/latest/getting-started/

[145] "HAL Streams Overview," Accessed: 26 March 2025. [Online]. Available: https://docs.hal.xyz/docs/overview

[146] "Monitor a smart contract for on-chain activity - OpenZeppelin Docs," Accessed: 2023-08-29. [Online]. Available: https://docs.openzeppelin.com/defender/tutorial/monitor

[147] M. Eshghie, "Dynamit Smart Contracts Dataset," Accessed: 2025-02-17. [Online]. Available: https://github.com/mojtaba-eshghie/Dynamit/tree/main/contracts

[148] Xscope-Tool, "Xscope-Tool/Results," Accessed: 26 March 2025. [Online]. Available: https://github.com/Xscope-Tool/Results

[149] "Highguard Exploits," Accessed: 26 March 2025. [Online]. Available: https://github.com/mojtaba-eshghie/HighGuard/tree/main/CI/exploits

[150] "Introduction | Scribble," Jul. 2023, Accessed: 26 March 2025. [Online]. Available: https://docs.scribble.codes

[151] V. Buterin, "Chain interoperability," Accessed: 26 March 2025. [Online]. Available: https://allquantor.at/blockchainbib/pdf/buterin2016chain.pdf

[152] A. Augusto, R. Belchior, M. Correia, A. Vasconcelos, L. Zhang, and T. Hardjono, "SoK: Security and Privacy of Blockchain Interoperability," in *2024 IEEE Symposium on Security and Privacy (SP)*, May 2024, pp. 3840–3865.

[153] L. Li, J. Wu, and W. Cui, "A review of blockchain cross-chain technology," *IET Blockchain*, vol. 3, May 2023.

[154] R. Ganguly, Y. Xue, A. Jonckheere, P. Ljung, B. Schornstein, B. Bonakdarpour, and M. Herlihy, "Distributed Runtime Verification of Metric Temporal Properties for Cross-Chain Protocols," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, Jul. 2022, pp. 23–33, iSSN: 2575-8411. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9912283

[155] K. Y. Rozier, "Specification: The Biggest Bottleneck in Formal Methods and Autonomy," in *Verified Software. Theories, Tools, and Experiments*, S. Blazy and M. Chechik, Eds. Cham: Springer International Publishing, 2016, pp. 8–26.

[156] M. Bartoletti and L. Pompianu, "An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns," in *Financial Cryptography and Data Security*, ser. LNCS. Cham: Springer, 2017, pp. 494–509.

[157] Y. Liu, Q. Lu, L. Zhu, H.-Y. Paik, and M. Staples, "A Systematic Literature Review on Blockchain Governance," *Journal of Systems and Software*, vol. 197, 2023.

[158] Fravoll, "Solidity Patterns." [Online]. Available: https://fravoll.github.io/solidity-patterns/

[159] "Chainbridge-solidity," Aug. 2023, Accessed: 2023-08-29. [Online]. Available: https://github.com/ChainSafe/chainbridge-solidity

[160] "Introduction," Accessed: 2023-08-29. [Online]. Available: https://docs.nomad.xyz/nomad-101/introduction

[161] DEUS, "DEUS Finance," Accessed: 2025-03-09. [Online]. Available: https://www.deus.finance/

[162] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the Laws of Order in Smart Contracts," in *SIGSOFT ISSTA 2019*. ACM, 2019, pp. 363–373.

[163] S. Driessen, D. D. Nucci, G. Monsieur, D. A. Tamburri, and W.-J. van den Heuvel, "Automated Test-Case Generation for Solidity Smart Contracts: The AGSolT Approach and its Evaluation," Accessed: 2025-03-09.

[164] X. Wang, Y. Yang, L. Liu, Z. Chen, and S. Huang, "Test Case Generation for Ethereum Smart Contracts Based on Cross-Contract Data Flow Analysis," *IEEE Transactions on Reliability*, pp. 1–14, 2024.

[165] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury, "Smart Contract Repair," *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 4, pp. 27:1–27:32, Sep 2020.

[166] C. Ferreira Torres, H. Jonker, and R. State, "Elysium: Context-Aware Bytecode-Level Patching to Automatically Heal Vulnerable Smart Contracts," in *25th International Symposium on Research in Attacks, Intrusions and Defenses*. Limassol Cyprus: ACM, Oct 2022, p. 115–128. [Online]. Available: https://dl.acm.org/doi/10.1145/3545948.3545975

[167] M. Rodler, W. Li, G. O. Karame, and L. Davi, "{EVMPatch}: Timely and Automated Patching of Ethereum Smart Contracts," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1289–1306.

[168] SunWeb3Sec, "DeFi Hacks Reproduce - Foundry," Accessed: 2025-03-09. [Online]. Available: https://github.com/SunWeb3Sec/DeFiHackLabs

[169] OpenAI *et al.*, "GPT-4 Technical Report," Accessed: 10 March 2025. [Online]. Available: https://arxiv.org/abs/2303.08774

[170] K. B. Kim and J. Lee, "Automated Generation of Test Cases for Smart Contract Security Analyzers," *IEEE Access*, vol. 8, pp. 209 377–209 392, 2020.

[171] P. Jaccard, "The Distribution of the Flora in the Alpine Zone," *New phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[172] S. So and H. Oh, "Smartfix: Fixing Vulnerable Smart Contracts by Accelerating Generate-and-Verify Repair Using Statistical Models," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, San Francisco, California, United States, December 2023, emails: `sunbeom_so@korea.ac.kr`, `hakjoo_oh@korea.ac.kr`.

[173] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "Smartshield: Automatic Smart Contract Protection Made Easy," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 23–34.

[174] A. Storhaug, J. Li, and T. Hu, "Efficient Avoidance of Vulnerabilities in Auto-completed Smart Contract Code Using Vulnerability-constrained Decoding," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Oct. 2023, pp. 683–693.

[175] S. J. Wang, K. Pei, and J. Yang, "Smartinv: Multimodal Learning for Smart Contract Invariant Inference," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 2217–2235.

[176] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh *et al.*, "SymPy: Symbolic Computing in Python," *PeerJ Computer Science*, vol. 3, p. e103, 2017.

[177] S. Azzopardi, J. Ellul, and G. J. Pace, "Monitoring Smart Contracts: ContractLarva and Open Challenges Beyond," in *Runtime Verification*, ser. Lecture Notes in Computer Science, C. Colombo and M. Leucker, Eds. Cham: Springer International Publishing, 2018, pp. 113–137.

[178] Z. Afzal, A. Brunström, S. Lindskog, and J. Garcia, "Using Features of Encrypted Network Traffic to Detect Malware," in *25th Nordic Conference on Secure IT Systems*, ser. LNCS. Online: Springer, 2020.

[179] Web3JS, "Web3.js - Ethereum JavaScript API — web3.js 1.0.0 documentation," Accessed: 2025-03-09. [Online]. Available: https://web3js.readthedocs.io/en/v1.3.0/

[180] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-Learn: Machine Learning in Python," *the Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[181] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu, "VULTRON: Catching Vulnerable Smart Contracts Once and for All," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, May 2019, pp. 1–4.

[182] "Chainlink Data Feeds | Chainlink Documentation," Accessed: 2023-10-20. [Online]. Available: https://docs.chain.link/data-feeds

[183] "Introducing the Coinbase Price Oracle," Accessed: 2025-03-09. [Online]. Available: https://www.coinbase.com/blog/introducing-the-coinbase-price-oracle

[184] "Uniswap/v4-core," Accessed: 26 March 2025. [Online]. Available: https://github.com/Uniswap/v4-core

[185] "Band Protocol Documentation | Band Protocol," Accessed: 2025-03-09. [Online]. Available: https://docs.bandchain.org/

[186] "Introduction | Augur Docs," Accessed: 2025-03-09. [Online]. Available: https://docs.augur.net/

[187] "Quickstart," Accessed: 2025-03-09. [Online]. Available: https://docs.diadata.org/introduction/readme

[188] "The Certora Verification Language — Certora Prover Documentation 0.0 documentation," Accessed: 10 March 2025. [Online]. Available: https://docs.certora.com/en/latest/docs/cvl/index.html