FORAY: Towards Effective Attack Synthesis against Deep Logical Vulnerabilities in DeFi Protocols

Hongbo Wen
hongbowen@ucsb.edu
University of California, Santa Barbara
Santa Barbara, California, USA

Hanzhi Liu
hanzhi@ucsb.edu
University of California, Santa Barbara
Santa Barbara, California, USA

Jiaxin Song
jiaxins8@illinois.edu
University of Illinois Urbana-Champaign
Champaign, Illinois, USA

Yanju Chen
yanju@cs.ucsb.edu
University of California, Santa Barbara
Santa Barbara, California, USA

Wenbo Guo
henrygwb@ucsb.edu
University of California, Santa Barbara
Santa Barbara, California, USA

Yu Feng
yufeng@cs.ucsb.edu
University of California, Santa Barbara
Santa Barbara, California, USA

Abstract

Blockchain adoption has surged with the rise of Decentralized Finance (DeFi) applications. However, the significant value of digital assets managed by DeFi protocols makes them prime targets for attacks. Current smart contract vulnerability detection tools struggle with DeFi protocols due to deep logical bugs arising from complex financial interactions between multiple smart contracts. These tools primarily analyze individual contracts and resort to brute-force methods for DeFi protocols crossing numerous smart contracts, leading to inefficiency.

We introduce Foray, a highly effective attack synthesis framework against deep logical bugs in DeFi protocols. Foray proposes a novel attack sketch generation and completion framework. Specifically, instead of treating DeFis as regular programs, we design a domain-specific language (DSL) to lift the low-level smart contracts into their high-level financial operations. Based on our DSL, we first compile a given DeFi protocol into a token flow graph, our graphical representation of DeFi protocols. Then, we design an efficient sketch generation method to synthesize attack sketches for a certain attack goal (e.g., price manipulation, arbitrage, etc.). This algorithm strategically identifies candidate sketches by finding reachable paths in *Token Flow Graph* (TFG), which is much more efficient than random enumeration. For each candidate sketch written in our DSL, Foray designs a domain-specific symbolic compilation to compile it into SMT constraints. Our compilation simplifies the constraints by removing redundant smart contract semantics. It maintains the usability of symbolic compilation, yet scales to problems orders of magnitude larger. Finally, the candidates are completed via existing solvers and are transformed into concrete attacks via direct syntax transformation. Through extensive experiments on real-world security incidents, we demonstrate that Foray significantly outperforms Halmos and ItyFuzz, the state-of-the-art (SOTA) tools for smart contract vulnerability detection, in both effectiveness and efficiency. Specifically, out of 34 benchmark DeFi logical bugs that happened in the last two years, Foray synthesizes 27 attacks, whereas ItyFuzz and Halmos only synthesize 11 and 3, respectively. Furthermore, Foray also finds *ten* zero-day vulnerabilities in the BNB chain. Finally, we demonstrate the effectiveness of our key components and Foray's capability of avoiding false positives.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690293

CCS Concepts

• **Security and privacy** → **Vulnerability scanners**; **Logic and verification**; *Economics of security and privacy*.

Keywords

Blockchain; Smart Contract; DeFi; Attack Synthesis

ACM Reference Format:
Hongbo Wen, Hanzhi Liu, Jiaxin Song, Yanju Chen, Wenbo Guo, and Yu Feng. 2024. FORAY: Towards Effective Attack Synthesis against Deep Logical Vulnerabilities in DeFi Protocols. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690293

1 Introduction

Decentralized Finance (DeFi) applications have driven a surge in blockchain adoption by offering real-world financial services like lending, borrowing, and trading on blockchain networks. This has brought in a broader user base and increased interest in blockchain technology, with a total funding amount of more than $90 billion locked in DeFi applications as of March 2023 [24]. Nonetheless, the substantial value of digital assets under the management of DeFis renders them an enticing target for potential attacks. For instance, the recent price manipulation vulnerability [9, 21, 54] allows malicious actors to induce DeFi protocols (a set of smart contracts that realize a certain financial model) to execute transactions that are detrimental to user's funds. Furthermore, attackers can manipulate DeFi protocols to instigate exchanges from lower-valued assets to higher-valued ones or to secure significant loans, often using low-value assets as collateral. This manipulation is achieved by tampering with the circulation of tokens, thus influencing token prices in the process. Statistics from the incomplete hack event

database [23] show that attacks exploiting logical flaws of *financial models* behind DeFis (denoted as deep logical bugs) have resulted in a cumulative loss of up to $200 million over the past two years.

Improving the robustness of DeFi protocols is thus a pressing concern and there has been a flurry of research [10, 19, 31, 42, 57] in the past few years. However, the majority of current detection tools primarily concentrate on code vulnerabilities of a single contract, such as re-entrancy, integer overflow, access control, etc. Therefore, it is unsurprised that these tools cannot be employed effectively to identify DeFi attacks stemming from logic flaws. The complexity of multiple contracts in DeFi and their interactions dramatically increase the search space that goes beyond the capability of existing analyzers. To make things even worse, the smart contracts in DeFis are immutable – once they are deployed, fixing their bugs is extremely difficult due to the design of the consensus protocol.

We introduce Foray, a synthesizer for automatically generating exploits against deep logical bugs in DeFi protocols. Foray introduces an attack sketch generation and completion framework. It first generates incomplete attack sketches written in our DSL. Then, it leverages our proposed *domain-specific symbolic compilation* approach to compile the attack sketches with logical holes into constraints that can be solved by off-the-shelf solvers. Finally, it fills the holes with a SOTA solver and transforms the complete sketches into concrete attacks through a direct syntax transformation.

The key technical challenges are two-fold. First, existing tools cannot strategically generate sketches for DeFi beyond random enumeration. Second, current symbolic compilation tools treat DeFi as a collection of regular smart contracts, disregarding the high-level *financial models* in DeFi protocols. To mitigate the first challenge, given a DeFi protocol, Foray first compiles it into a *Token Flow Graph* (TFG), our proposed high-level semantic representation for DeFi protocols. Here, nodes represent different tokens (USDC, WETH, USDT, etc.) and edges are labeled with constructs from Foray's *abstract financial language*, which provides high-level operators (e.g., lend/borrow/pay/swap) over financial assets. Now, given a particular attack goal (e.g., price manipulation, arbitrage, etc.) in the form of a logical formula, Foray models the attack sketch generation as a reachability problem in TFG. Instead of random enumeration, Foray devises an effective *sketch generation* algorithm that strategically enumerates relevant attack sketches using a *type-directed* graph reachability algorithm over the TFG.

To tackle the second challenge, Foray employs a domain-specific symbolic compilation strategy, which maintains the usability of symbolic compilation, yet scales to problems orders of magnitude larger. For each candidate attack sketch, Foray leverages the *abstract semantics* of our proposed DSL to compile possible completions of the sketch into SMT constraints that can be efficiently solved by off-the-shelf solvers [20]. Here, our domain-specific symbolic compilation can filter out low-level smart contract semantics and thus significantly simplify the constraints. Because both our *sketch generation* and *sketch completion* overapproximate the concrete semantics of DeFis, Foray may generate spurious attacks that fail to achieve the goal. We mitigate this problem by incorporating a CEGIS (Counter Example-Guided Inductive Synthesis) loop that iteratively adds the root cause of the failed attempt to Foray's *knowledge base*, which avoids similar mistakes in future iterations.

We implement Foray and compare it against Halmos [2] and ItyFuzz [52], the state-of-the-art tools for analyzing smart contracts and DeFi protocols. Our experiment shows that our tool is efficient and effective. On the set of 34 security incidents in the past two years, Foray manages to synthesize attacks for 79% of the benchmarks with an average synthesis time of 105.9 seconds. On the other hand, Halmos can only solve 10% of the benchmarks with an average running time of 8085.0 seconds, which demonstrates that Foray's domain-specific symbolic compilation accelerates synthesis several orders of magnitude compared to the general-purpose compilation to an SMT solver. Furthermore, we also apply Foray to DeFi protocols on the BNB chain [7] and uncover *ten* zero-day vulnerabilities with concrete attacks. Finally, we verify the effectiveness of sketch generation and completion through an ablation study and demonstrate Foray's capability in alleviating false positives. Overall, Foray provides a novel attack synthesis technique against various types of deep logical bugs in DeFis protocols.

In summary, this paper makes the following contributions:

- We propose *Abstract Financial Language*, a DSL that describes high-level financial operators in DeFis. We also design *Token Flow Graph*, a semantic representation that summarizes the financial model of a DeFi protocol.
- We propose an effective CEGIS framework for DeFi attack synthesis. In particular, our sketch generation leverages a type-directed graph reachability over a token flow graph and our sketch completion designs a domain-specific symbolic compilation strategy that results in easy-to-solve constraints.
- We implement the proposed ideas in a tool called Foray and demonstrate that it achieves several orders of magnitude speed-up compared to general-purpose symbolic compilation. Furthermore, Foray not only generated 80% security incidents in the past two years (2022-2023) but also detected ten zero-day DeFi vulnerabilities from popular blockchains.

2 Background

2.1 Blockchain basis.

Ethereum. Blockchain functions as a decentralized record-keeping platform that chronicles and disseminates transaction data among multiple users. It is an expand-only chain of interconnected blocks, managed by a consensus mechanism, where each block contains a collection of transactions. Among various blockchain systems, Ethereum [61] is the first blockchain capable of storing, managing, and running Turing-complete scripts, termed *smart contracts*. Ethereum operates on a comprehensive state system updated via transaction execution. The transactions are initiated by and received by users through their accounts. Ethereum has two principal types of accounts: those owned by users and those governed by smart contracts, each associated with a distinct *address*. Besides making transactions, users can also develop customized smart contracts that are programmed to execute transactions autonomously.

Tokens and cryptocurrencies. Among different types of smart contracts, Tokens are a specific type that represents cryptocurrencies. Each Token contract must adhere to standardized interfaces like ERC20 [59], ERC721 [27], and ERC1155 [51], which define how users interact with the corresponding token. For Ethereum, ERC20

is the most widely adopted interface. To tether the value of cryptocurrencies to fiat currency, *stablecoins*—like USDT [55], which is implemented as an ERC20 token–have been created. They are pegged to the dollar reserves held by the issuer, providing a stable reference point for the value of other cryptocurrencies.

2.2 Decentralized Finance (DeFi)

Decentralized Finance (DeFi) refers to a set of financial applications built on blockchain technology. They aim to recreate traditional financial systems, such as banking and lending, but without the need for intermediaries like banks or brokers. Instead, each DeFi service is implemented as a protocol that amalgamates various smart contracts. Users access a DeFi service by engaging with the corresponding protocol through transactions. According to a recent survey [22], over 200 DeFi applications have been launched on the Ethereum platform. Here we list three major DeFi applications:

Lending. platforms (such as Aave [3], MakerDAO [43]) enable users to obtain on-chain cryptocurrencies as loans by depositing collateral into the system. The interest rates for borrowing are set by the DeFi protocols while maintaining transparency for users. As market conditions fluctuate, the collateral's value may fall below or rise above a certain threshold. When this happens, either the application or other users can liquidate or sell the collateral to gain profits.

Flash loans. (e.g., dYdX [25], Uniswap [58]) represent a collateral-free borrowing model. This enables the borrower to run custom code through a callback function, with the stipulation that the loan must be repaid within the same transaction. If the borrower fails to return the loaned tokens, the lender will automatically reverse the lending transaction, ensuring that no permanent changes (to storage variables) are made by this transaction.

Decentralized exchanges (DEXs). function as cryptocurrency exchanges that enable users to trade various tokens through direct interaction with smart contracts. These platforms incentivize users to deposit pairs or multiple tokens into a liquidity pool. As long as the pool maintains sufficient token volume, users can execute token swaps within it. The exchange rate for these trades is determined autonomously by the application's built-in pricing algorithm. Popular DEXs protocols include 1inch [1], PancakeSwap [49].

DeFi vulnerabilities. At a high level, there are two types of vulnerabilities in DeFi protocols. The first type refers to vulnerabilities in individual smart contracts, including assertion failures, arbitrary writes, control-flow hijacking, etc (denoted as common vulnerabilities). These vulnerabilities are similar to traditional software security bugs and are possible to be automatically detected by analyzing the smart contract code. As discussed in Section 10, existing research works propose a number of tools that utilize static and dynamic program analysis to automatically identify such vulnerabilities. The second type of vulnerability exploits logical flaws in a DeFi protocol, which we refer to as **deep logical bugs** in this paper. As demonstrated in Section 3, these deep logical bugs exploit public functions across multiple smart contracts within the DeFi protocol to maliciously increase an attacker's profits. Identifying such vulnerabilities is extremely challenging because it requires a deep understanding of the semantics and business logic of the DeFi protocol, as well as the composition of transaction sequences. As shown in recent studies [65, 67], most existing tools designed for smart contract vulnerabilities fail to detect deep logical bugs.

source code: Mubank

```
...
function _mu_bond_quote(uint256 amount) internal view returns (uint256) {
    (uint112 reserve0, uint112 reserve1, ) = uniswap.getReserves();
    uint256 amountIN = uniswap.getAmountIn(amount, reserve1, reserve0);
    uint256 amountOUT = uniswap.getAmountOut(amount, reserve0, reserve1);
    uint256 mu_bond_amount = (amountIN + amountOUT) / 2;
    return mu_bond_amount;
}
function mu_bond(uint256 amount) public {
    uint256 mu_bond_amount = _mu_bond_quote(amount);
    IERC20(_USDCe).transferFrom(msg.sender, address(this), amount);
    IERC20(_Mu).transfer(msg.sender, mu_bond_amount);
}
...
```

concrete attack

```
contract Attacker {
    ...
    function attack() external {
        lender.flashloan(mu, 24945e18);
        uniswap.swap(mu, usdce, 24944.5e18, 1);
        mubank.mu_bond(13876e18);
        lender.payback(mu, 25019.835e18);
    }
    ...
}
```

source code: Uniswap

```
function swap(address fromToken,
address toToken, uint256 amountIn,
, uint256 minAmountOut) { ... }
```

source code: DeFiLender

```
function flashloan(address token,
uint256 amount) { ... }
```

Figure 1: Illustration of MUMUG and a concrete exploit against it. `IERC20().transferFrom` and `IERC20().transfer` are standard APIs that enable the withdraw and deposit of tokens for one address. `uniswap.getAmountIn` and `uniswap.getAmountOut` are `uniswap` APIs that calculate the required amount to swap one type of token for another based on their current reserves.

3 Problem Definition and Existing Solutions

In this section, we begin by specifying our problem scopes and demonstrating a deep logical bug of a simplified DeFi protocol, MUMUG, which was hacked in 2022, resulting in the loss of nearly all its stablecoins. Then, we formally define DeFi attack synthesis and discuss the limitations of existing solutions.

3.1 Problem Scope and Technical Challenges

Threat model. Our goal is to detect *deep logical bugs* in a DeFi protocol by synthesizing a sequence of attack transactions that can exploit the DeFi protocol to gain profits maliciously. We assume an entirely trustless setup where an attacker can access all public information, including but not limited to on-chain blockchain states and the victim contracts' source code. For contracts with only bytecodes, their source code can be obtained via reserve engineering, which is not our focus. Additionally, beyond directly interacting with the victim contracts, we assume the attacker can deploy their own contract, which can invoke public transactions of the target victim contracts (either directly or through callbacks). The attacker's goal is to synthesize a sequence of transactions that exploit the logical flaws in the target DeFi protocol to gain extra profit. We do not consider the common vulnerabilities.

MUMUG protocol and an attack. As shown in Figure 1, the protocol is composed of three key smart contracts: a) `DeFiLender` provides the `flashloan` function to enable the borrower to get tokens without collateral; b) `Mubank` with two functionalities. The internal function (`_mu_bond_quote`) manages the sale and price of MU tokens based on the current reserves of MU and USDCe. It takes as input the amount of USDCe and outputs the corresponding amount of MU in the same value. The public function (`mu_bond`) enables users to withdraw MU by providing the same value of USDCe

determined by `_mu_bond_quote`. c) `Uniswap` is a popular protocol, which defines swap pairs for two types of tokens (e.g., MU and USDCe). Its `swap` function enables users to exchange tokens in a swap pair. These three smart contracts define the MUMUG DeFi protocol where benign users can borrow, withdraw, and exchange MU with USDCe.

The susceptibility of MUMUG lies in the pricing mechanism in the `Mubank` contract (highlighted in Figure 1). Given that the price of MU is determined by the reserve of USDCe and MU within the swap pair. A significant fluctuation in the reserve level can result in an unexpectedly high volume of MU tokens and significantly lower its price. An attack can leverage the price difference to withdraw the MU bank's stablecoins. A concrete attack is shown in Figure 1. ❶ Borrow a huge amount of MU tokens through the flashloan function in `DeFiLender`. ❷ Swap those MU tokens to a large amount of USDCe at the swap pair. This will dramatically increase the reserve balance ratio of MU to USDCe, devaluing the MU. ❸ Leverage the abnormal reserve balance ratio to swap a tiny amount of USDCe for a huge amount of MU tokens at `MuBank`. ❹ Pay MU tokens back to the flash loan lender, keeping the majority of USDCe acquired at step ❷ as the profit. Through this process, the attacker harvested approximately 57,660 USDCe from the `MuBank`.

Formal definition of attack synthesis for deep logical bugs. Automatic attack synthesis in DeFi is equivalent to finding a sequence of function calls that exploit deep logical bugs of the DeFi protocol. This can be formally defined as

Definition 3.1 (DeFi Attack Synthesis). An attack synthesis for a DeFi protocol D is a tuple (L, S_0, ψ), where L is the domain-specific language (DSL) for constructing the attack program. For instance, a list of public functions is provided by the victim DeFi protocol. S_0 is the initial and public blockchain state, and ψ is the attack goal written in a logical formula. DeFi attack synthesis is equivalent to finding an attack program P written in DSL L, such that $P(S_0) \models \psi$ where $P(S_0)$ denotes the resulting state after executing P on S_0.

Technical challenges. It is extremely challenging for the following two reasons. First, the search space is huge. In fact, MUMUG protocol itself contains 26 public functions and the attackers can freely call public functions of other smart contracts (e.g., `uniswap.swap`). Even when we constrain the length of the function call sequence, the number of possible sequences is still extremely huge. Searching a malicious function call sequence in such a huge search space is equivalent to finding a needle in a haystack. Second, smart contracts and DeFi protocols have complicated semantics. This imposes extra challenges to automatically represent a DeFi protocol with logical representations, making it hard to reason and synthesize attacks.

3.2 Existing Solutions and Limitations

While attack synthesis is a novel concept in DeFi, it has been explored in traditional software security and program synthesis domains [29–31]. Without any heavy customization, we can draw inspiration from traditional program synthesis and try to solve the problem with the following two solutions.

Static analysis and symbolic execution based-sketch generation and completion. Given that synthesizing the entire attack program from scratch is unlikely to scale, existing works in program synthesis usually decompose the synthesis into two phases *sketch generation* and *sketch completion*. Here, an attack sketch refers to a sequence of actions, where each action is a function call to a certain smart contract. Formally, we define an attack sketch $\tilde{P}$ as a sequence of invocations to constructs in L where some of the constructs contain holes or symbolic variables yet to fill in.

To avoid exploring sketches doomed to fail, existing approaches typically leverage the *abstract semantics* to only preserve sketches whose abstract semantics are *consistent* with the attack goal ψ, $\tilde{P}(S_0) \Rightarrow \psi$, where $\tilde{P}(S_0)$ corresponds to the program state by *abstractly* evaluating the sketch $\tilde{P}$ on S_0. Then, the sketch completion step fills in the holes $\diamond$ in each feasible sketch $\tilde{P}$ ($P = \tilde{P}[\mu/\diamond]$) with language constructs μ in L using symbolic execution, such that $P(S_0) \models \psi$. Each hole in Foray represents a function parameter. By resolving these parameters, the attack sketch is transformed into a concrete program and its execution result satisfies the attack goal.

The main challenges of this solution are as follows: First, there are no existing tools in DeFi that can effectively generate feasible attack sketches. The only way is to randomly select and combine function calls, which is extremely inefficient given the huge search space. Second, due to the complex semantics of DeFi protocols, the corresponding symbolic constraints of attack goals are intricate and often beyond the reasoning capacity of SOTA SMT solvers. Specifically, to verify $P(S_0) \models \psi$, existing approaches have to reason about program P by faithfully following the operational semantics of the host language L, which contains language features (e.g., gas consumption and memory models in Solidity.) and low-level details irrelevant to the synthesis goal. As demonstrated in Section 8, it is extremely difficult for Halmos [2], a SOTA symbolic testing tool for Ethereum smart contracts [19, 31, 46], to solve the constraints for common attacks within a feasible time limit.

Fuzzing. SOTA fuzzers (e.g., ItyFuzz [52] and Smartian [18]) in smart contracts support synthesizing sequences of actions that lead to vulnerabilities (violation of DeFi protocol). Fuzzing is more computationally efficient than symbolic execution-based solutions but it relies more on random generations and mutations. In addition, due to DeFis' complex semantics, existing fuzzers do not have fitness functions or testing oracles that correspond to specific attack goals and thus cannot provide proper feedback signals of whether the current input is valid, making it even more difficult to find valid attacks through random mutations.

Note that as discussed in Section 10, there are some recent tools for automatically detecting DeFi protocol vulnerabilities. Most tools rely on summarizing attack patterns from past attack incidents and thus are hindered by the limited scope of these patterns. They can only detect limited types of vulnerabilities and struggle to identify unseen ones. Among existing tools, DeFiPoser [66] adopts the methodology of automatic sketch generation and completion. However, its sketches are generated based on limited heuristics, limiting its ability to synthesize anything beyond arbitrage scenarios.

Overall, due to the lack in *effective searching strategies for attack generation* and *domain-specific attack validation mechanism*, existing tools cannot effectively synthesize complicated DeFi attacks.

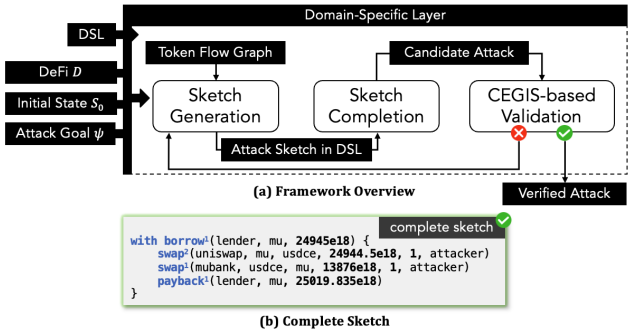

Figure 2: Overview of FORAY with the demonstrated completed sketch for the example in Figure 1. In the sketch, `swap`[1] refers to the `mn_bond` function. `swap`[2] is achieved through the `uniswap.swap` function.

4 Overview of FORAY

To mitigate the limitations of existing solutions, we design and develop FORAY, a novel DeFi-specific attack synthesis technique to uncover various deep logical vulnerabilities in DeFi applications. At a high level, FORAY follows the attack sketch generation and completion methodology but includes multiple customized designs to enable more effective sketch search and verification. As shown in Figure 2, we design a domain-specific language to lift the low-level smart contracts into their high-level financial semantics and models (e.g., exchanges, lenders, loans). Based on our DSL, we first compile DeFi protocols into abstract representations (token flow graph construction), which filter out low-level semantics and constrain the attack sketch space. We design an efficient sketch generation method based on the graph reachability in the TFG (sketch generation). Then, we complete a sketch by compiling it into symbolic constraints and replacing the symbolic variables with concrete assignments using an off-the-shelf solver [20] (sketch completion). Finally, we conduct direct syntax transformation to transform the complete sketches into concrete attacks. Given that the abstraction process may over-simplify blockchain states and concrete smart contract semantics, we conduct an additional validation step to actually run the synthesized attack. If an attack cannot satisfy the attack goal, our CEGIS loop will add additional constraints corresponding to the root causes to the solver and avoid similar mistakes in future iterations.

Token Flow Graph construction (Section 5). The insight of this component is to lift the low-level semantics of smart contracts to their high-level financial models. This process filters out a significant portion of solidity semantics, reducing the synthesis space and simplifying the validation process. To do so, we first define *Abstract Financial Language*, a domain-specific language for describing high-level *financial operations* commonly used by DeFis such as swap, borrow, payback, transfer, etc. Then given a DeFi protocol, FORAY *lifts* it to a *Token Flow Graph* (TFG). As we will show later, this TFG helps develop effective strategies for attack sketch synthesis. Motivating by prior work [30, 36, 44] in type-directed program synthesis, we design each node to represent a certain type of token in DeFi. To avoid and simplify the complexity due to multi-party communication, we also introduce the ϵ token, a special node that represents tokens from parties other than the current attacker. Each edge refers to an operation in our abstract financial language and its source and target nodes represent the tokens that the operation needs to consume and produce, respectively. Figure 3 shows the TFG of the MUMUG protocol. Here the nodes are MU, USDCe, and ϵ (i.e., lender of flash loan). The edges are possible operations invoking the three smart contracts in MUMUG. For example, the edge `borrow`[1] from ϵ to MU represents one functionality in `flashloan` function, which enables borrowing a certain amount of MU tokens from the lender, i.e., `DeFiLender`.

Sketch generation (Section 6.2).

Given a TFG of a victim protocol, an attack goal ψ and an initial state S_0 are both expressed as first-order logic constraints, with S_0 being satisfied by the initial blockchain state and ψ being expected to be satisfied after the attack program's execution. The goal of this step is to synthesize an incomplete program P in abstract financial language such that $P(S_0) \models \psi$.

Intuitively, an attack sketch P outlines the key financial steps to achieve the attack goal ψ. Given the huge space, we need to develop an effective search strategy that only enumerates the sketches that are likely to be successful. To do so, we model the problem of achieving the attack goal as a readability problem in our TFG. We then design a customized graph readability algorithm to efficiently enumerate candidate sketches that conform with the attack goal.

In our motivating example, the attack goal is:

$$B_{t_2}^{usdce} - B_{t_1}^{usdce} > 0, \tag{1}$$

stating states the attacker's balance of USDCe at the end of the execution (t_2) should be greater than his initial balance (t_1). The details of how to infer the attack goal will be introduced in Section 7.

The attack sketch shown in Figure 2 is a feasible candidate sketch by taking the reachable path of $\epsilon \rightarrow MU \rightarrow USDCe \rightarrow MU \rightarrow \epsilon$ in the TFG.

Sketch completion (Section 6.3). After synthesizing feasible attack sketches, our next step is to complete the feasible attack sketches by substituting all symbolic variables with concrete assignments with constants or storage variables. At a high level, we first design a domain-specific symbolic compilation procedure (motivated by existing solutions [15, 41, 50]) that soundly compiles a candidate sketch into a set of constraints that represent the space of all possible concrete attacks. Then, we conduct the completion by solving the constraints using an off-the-shelf solver [20]. The first challenge in this procedure is to constrain the complexity of symbolic constraints such that they are feasible for existing solvers. As mentioned above, our abstract financial language and token flow graph are proposed for tackling this challenge. Representing the victim protocol and attack sketches in our abstract financial language significantly simplifies the constraints. The second challenge is how to leverage cases that fail to pass the verification. We tackle this by integrating a CEGIS (Counter Example-Guided Inductive Synthesis) loop into the synthesis process. This step first conducts direct syntax transformation to map the synthesized attack from our abstract financial language back to solidity code. It then deploys and executes the attack code using foundry framework [32] to test whether the attack goal is achieved in a simulated environment. It constructs a knowledge base and iteratively adds the root causes of the failed attempts. We will transform root causes as additional constraints

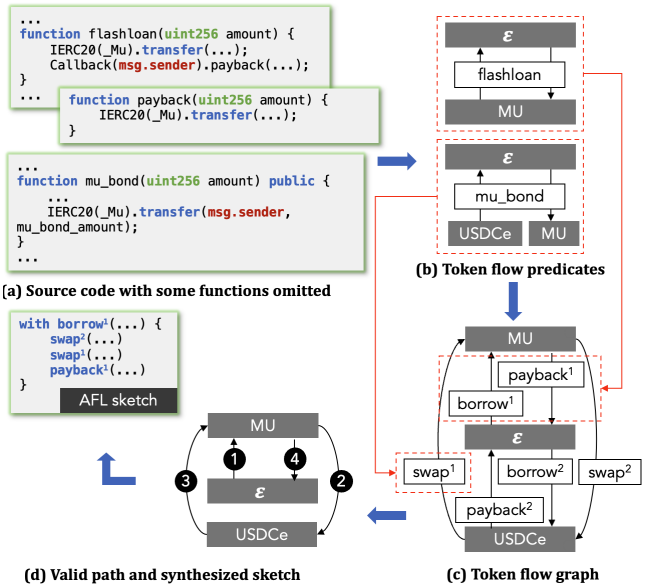

Figure 3: Demonstration of token flow graph construction, graph reachability, and valid attack sketch of MUMUG in Figure 1. In the TFG, the token nodes (except ϵ) represent tokens owned by the attacker and edges are financial operators (constructs in abstract financial language).

⟨*prog*⟩	::=	⟨*stmt*⟩+
⟨*stmt*⟩	::=	⟨*transfer*⟩ | ⟨*burn*⟩ | ⟨*mint*⟩ | ⟨*swap*⟩ | ⟨*borrow*⟩
⟨*transfer*⟩	::=	transfer(token: ⟨*token*⟩, from: ⟨*addr*⟩, to: ⟨*addr*⟩, amt: ⟨*expr*⟩)
⟨*burn*⟩	::=	burn(token: ⟨*token*⟩, from: ⟨*addr*⟩, amt: ⟨*expr*⟩)
⟨*mint*⟩	::=	mint(token: ⟨*token*⟩, from: ⟨*addr*⟩, amt: ⟨*expr*⟩)
⟨*swap*⟩	::=	swap(market: ⟨*addr*⟩, src: ⟨*token*⟩, tgt: ⟨*token*⟩, in: ⟨*expr*⟩, minout: ⟨*expr*⟩, to: ⟨*addr*⟩)
⟨*borrow*⟩	::=	with borrow(lender: ⟨*addr*⟩, to: ⟨*addr*⟩, amt: ⟨*expr*⟩) {⟨*stmt*⟩+ ⟨*payback*⟩}
⟨*payback*⟩	::=	payback(lender: ⟨*addr*⟩, to: ⟨*addr*⟩, amt: ⟨*expr*⟩)
⟨*balance*⟩	::=	balance(token: ⟨*token*⟩, of: ⟨*addr*⟩)
⟨*expr*⟩	::=	⟨*const*⟩ | ⟨*op*⟩(⟨*expr*⟩+) | ⟨*balance*⟩

⟨*const*⟩ ∈ **constants** ⟨*op*⟩ ∈ **operators**
⟨*addr*⟩ ∈ **addresses** ⟨*token*⟩ ∈ **tokens**

Figure 4: Syntax for our abstract financial language.

to avoid failed sketches in future attempts. Figure 2 demonstrates a complete attack sketch given by a constrained solver, where the symbolic variables are filled with concrete values.

As demonstrated Figure 2, FORAY also requires inputs ψ and S_0 written in first-order logic and a final transformation and validation component (See Section 6 for more details of these two parts).

5 Token Flow Graph

In this section, we present a new graph abstraction for modeling flows of tokens within a DeFi environment, which is used to summarize common DeFi behavior, as well as searching for potential program sketches that satisfy a given attack goal.

5.1 Abstract Financial Language (AFL)

As shown in Figure 4, Abstract Financial Language (AFL) is a domain-specific language that is designed to model token flows of common financial operations achieved by DeFi protocols. A program ⟨*prog*⟩ written in AFL corresponds to a sequence of statements composed by the following commonly used financial operators:

- ⟨*transfer*⟩ models a single transfer of a specific amount of a token from one address to another.
- ⟨*burn*⟩ models the destruction of a certain amount of a token from an address.
- ⟨*mint*⟩ models the generation of a certain amount of a token from an address.
- ⟨*swap*⟩ models the exchange of a certain amount of one token to another for an address.
- ⟨*borrow*⟩ models a temporary transfer behavior of a certain amount of a token from a lender to a borrower's address. A ⟨*payback*⟩ statement should always be paired at the end to model the return of the borrowed tokens.

Note that ⟨*burn*⟩ and ⟨*mint*⟩ functions are implemented to control the total token supply and liquidity, aiming to stabilize its price. These operations are restricted to specific authorized users. However, attackers may also leverage these functions via exploitation.

AFL also provides easy syntax and interface for accessing different entities from a DeFi environment, including:

- ⟨*addr*⟩ for referring to one of all available addresses in a given DeFi environment.
- ⟨*token*⟩ for referring to one of all available types of tokens in a given DeFi environment.
- ⟨*balance*⟩ accesses a token's balance in a given address.

Note that AFL can represent both benign and malicious behaviors. We mainly use it to model attackers in this work.

Example 5.1 (AFL attack program). As shown in Figure 3(d), an AFL program may include ⟨*borrow*⟩ and ⟨*payback*⟩, interspersed with several ⟨*swap*⟩ operators in the context. It represents the following attack behavior: initially, borrowing MU tokens from another party ϵ, then exchanging MU tokens for USDCe tokens, subsequently swapping these back via another exchange contract, and finally, repaying the borrowed MU tokens to the environment ϵ.

5.2 Definition of Token Flow Graph

We propose a Token Flow Graph (TFG) to model changes in amounts of abstract tokens owned by the attacker when interacting with public functions of DeFi protocols. It helps filter out low-level semantics of smart contracts and guides the synthesis of attack sketches. To formally define TFG, we first introduce the following domains:

- $\mathbb{F}$ is a set of public DeFi functions accessible by the attacker. We assume all non-public functions are resolved by inlining.
- $\mathbb{P}$ contains all AFL operators, e.g., `borrow`.
- $\mathbb{T}$ is a set of different tokens appearing in a given DeFi protocol, i.e., nodes in TFG.
- $\mathbb{E}$ is a set of edges in TFG.
- Φ is a set of behavioral constraints about logical relations between tokens, addresses, and AFL operators.

Given the above domains, we define a token flow graph as a tuple $\mathbb{G}(\mathbb{T}, \mathbb{P}, \mathbb{E}, \Phi)$. In particular, $\mathbb{E} \subseteq \times\mathbb{T} \times \mathbb{T} \times \mathbb{P} \times \Phi$ is a set of

edges connecting tokens, where each edge is associated with an AFL operator. For clarity in presentation, edges are attached with superscripts, denoting different functions that they are inferred from.

Special node ϵ. Intuitively, the nodes of a token flow graph represent assets of the user currently interacting with the DeFi. To reflect and simplify the interactions of other participants (e.g., contract owners, other users), each token flow graph has a built-in node $\epsilon \in \mathbb{T}$ that represents tokens of all participants other than the one of interest (i.e., attacker in our problem). Such tokens are not directly related to the attacker's goal but are necessary for the construction of an attack.

Example 5.2 (TFG for an attacker). Figure 3(c) depicts a TFG of the MUMUG protocol. For example, an edge labeled with `swap`[1] indicates that the attacker could exchange USDCe for MU through the function `mu_bond` in Figure 1.

5.3 Construction of Token Flow Graph

Given a DeFi protocol, the key to constructing a token flow graph for one specific user is to generate edges among tokens that the user holds or wants to acquire. FORAY employs an edge discovery procedure based on program analysis. It has two steps, first, we define flow predicate and influence rules for generating flow predicates from concrete programs of a DeFi protocol. Then, we generate edges from the predicates using edge inference rules. Each generated edge comes with a semantically equivalent AFL operation with its corresponding constraints. As illustrated in Figure 3, we first identify the flow predicates in the `flashloan` and `mu_bank` function, represented as an initial graph. Then, we apply the edge inference rules to generate the TFG from flow predicates. For example, the `swap`[1] is deduced from two token flows in `mu_bank`. Meanwhile, the `borrow`[1] and `payback`[1] are inferred from the `flashloan` function. To avoid the confusion between AFL statements and actual (solidity) program statements, we use "operator" to represent AFL statements $p \in \mathbb{P}$ and "statement" to represent actual program statements s. In what follows, we elaborate on the procedure for flow predicate and edge construction.

Flow predicate. denoted by $\text{flow}(u, x, a, b)$, indicates x amount of token u flows from address a to address b. A flow predicate serves as a basic building block of AFL operators. Figure 5 shows the rules for generating flow predicates from actual (solidity) programs. First, we define a *flow state* $\mathbb{W}$ that contains a collection of $s_i : w_i$ pairs where each pair $s_i : w_i$ represents a statement s_i together with its flow predicate w_i. Note that $\mathbb{W}$ is different from the blockchain state S. For each public function $f \in \mathbb{F}$, the func rule processes its statements sequentially by performing a sequence of *flow state transitions*. Specifically, given the original state $\mathbb{W}$ and a statement s, we model the state transition via $\mathbb{W} \overset{s}{\rightsquigarrow} \mathbb{W}'$, which indicates that the analysis of statement s results in a new version $\mathbb{W}'$ by adding the flow predicate corresponding to s to $\mathbb{W}$. Similar to classical symbolic executions [19, 31, 42], all loops are bounded and unrolled to their corresponding branch statements. The branch rule then merges updates of $\mathbb{W}$ from both branches. Other rules that update $\mathbb{W}$ are: flow-from, flow-to, flow-mint and flow-burn, which correspond to public functions in standard interfaces (e.g., ERC20):

$$\frac{f \in \mathbb{F} \quad f \equiv s_0; \ldots; s_n \quad \mathbb{W}_0 \overset{s_0}{\rightsquigarrow} \mathbb{W}_1 \quad \ldots \quad \mathbb{W}_n \overset{s_n}{\rightsquigarrow} \mathbb{W}_{n+1}}{\mathbb{W}_0 \overset{f}{\rightsquigarrow} \mathbb{W}_{n+1}} \ (\textbf{func})$$

$$\frac{s \equiv \text{if \_ then } f_0 \text{ else } f_1 \quad \mathbb{W} \overset{f_0}{\rightsquigarrow} \mathbb{W}_0 \quad \mathbb{W} \overset{f_1}{\rightsquigarrow} \mathbb{W}_1}{\mathbb{W} \overset{s}{\rightsquigarrow} \mathbb{W}_0 \cup \mathbb{W}_1} \ (\textbf{branch})$$

$$\frac{s \equiv u.\text{transferFrom}(a, b, x) \quad w \equiv \text{flow}(u, x, a, b)}{\mathbb{W} \overset{s}{\rightsquigarrow} \mathbb{W} \cup \{s : w\}} \ (\textbf{flow-from})$$

$$\frac{s \equiv u.\text{transfer}(b, x) \quad a = \text{this} \quad w \equiv \text{flow}(u, x, a, b)}{\mathbb{W} \overset{s}{\rightsquigarrow} \mathbb{W} \cup \{s : w\}} \ (\textbf{flow-to})$$

$$\frac{s \equiv u.\text{mint}(a, x) \quad w \equiv \text{flow}(u, x, \bullet, a)}{\mathbb{W} \overset{s}{\rightsquigarrow} \mathbb{W} \cup \{s : w\}} \ (\textbf{flow-mint})$$

$$\frac{s \equiv u.\text{burn}(a, x) \quad w \equiv \text{flow}(u, x, a, \bullet)}{\mathbb{W} \overset{s}{\rightsquigarrow} \mathbb{W} \cup \{s : w\}} \ (\textbf{flow-burn})$$

Figure 5: Flow predicates inference rules. • indicates a special address. Note that mint and burn has an implicit constraint that a must belong to a set of authorized addresses.

- The flow-from rule can be triggered by invocations of ERC20's `transferFrom` (or other similar) interface, e.g., `IERC20(u).transferFrom(a,b,x)`, which transfers x amount of token u from address a to address b.
- The flow-to rule can be triggered by invocations of ERC20's `transfer` (or other similar) interface, e.g., `IERC20(u).transfer(b,x)`, which transfers x amount of token u from the current caller (i.e., the address pointed by `this` keyword) to address b.
- The flow-mint rule matches invocations of ERC20's `mint` (or other similar) interface, e.g., `IERC20(u).mint(a,x)`, which produces x amount of u token for address a.
- The flow-burn rule matches invocations of ERC20's `burn` (or other similar) interface, e.g., `IERC20(u).burn(a,x)`, which destroys x amount of u token from address a.

After parsing the programs of a DeFi protocol with rules in Figure 5, we get a set of flow predicates that summarize critical financial behaviors within that protocol. FORAY then constructs the token flow graph on top of these predicates.

Edge construction. Figure 6 shows the rules for constructing edges in a token flow graph. Recall that the nodes in a TFG are the tokens that the user holds or wants to acquire, as well as the *void* node, representing all other parties. The underlying mechanism of the edge construction procedure is to identify semantic patterns of flow predicates for each AFL construct. An edge is represented by $\text{edge}(u, v, p, \Phi)$, where u and v are addresses, $p \in \mathbb{P}$ corresponds to an AFL operator and Φ is a set of p's behavioral constraints. We have six types of edges corresponding to different financial operators in Figure 4. We elaborate on their inference rules as follows:

- The user could exchange tokens with DeFi functions or third-party APIs from Uniswap, decentralized exchanges, etc. The edge-swap rule captures such a pattern by looking for a pair of consecutive *back-and-forth* flows between two addresses. When a `swap` edge is fired, e.g., $\text{edge}(u, v, \text{swap}, \Phi)$, u tokens are sent in exchange for v tokens. We describe such change of tokens for address a using constraints stored in Φ: $\Phi \equiv u[a] \geq x \wedge u'[a] \leq$

$$\frac{f \equiv \ldots; s_1; s_2; \ldots \quad s_1 : \mathsf{flow}(u, x, a, b) \in \mathbb{W} \quad s_2 : \mathsf{flow}(v, y, b, a) \in \mathbb{W} \quad \Phi \equiv u[a] \geq x \wedge u'[a] \leq u[a] \wedge v'[a] \geq y \wedge v'[a] \geq v[a]}{\mathsf{edge}(u, v, \mathsf{swap}, \Phi)} \ (\textbf{edge-swap})$$

$$\frac{f \equiv \ldots; s_1; \ldots; s_2; \ldots \quad s_3 \in g \quad \mathsf{callback}(s_2, g) \quad s_1 : \mathsf{flow}(u, x, a, b) \in \mathbb{W} \quad s_3 : \mathsf{flow}(u, y, b, a) \in \mathbb{W}}{\mathsf{loan}(s_1, s_2, s_3)} \ (\textbf{loan})$$

$$\frac{\mathsf{loan}(s, \_, \_) \quad s : \mathsf{flow}(u, x, b, a) \in \mathbb{W} \quad \Phi \equiv u'[a] \geq x \wedge u'[a] \geq u[a]}{\mathsf{edge}(\epsilon, u, \mathsf{borrow}, \Phi)} \ (\textbf{edge-borrow})$$

$$\frac{\mathsf{loan}(\_, \_, s) \quad s : \mathsf{flow}(u, x, a, b) \in \mathbb{W} \quad \Phi \equiv u[a] \geq x \wedge u'[a] \leq u[a]}{\mathsf{edge}(u, \epsilon, \mathsf{payback}, \Phi)} \ (\textbf{edge-payback})$$

$$\frac{s : \mathsf{flow}(u, x, \bullet, a) \in \mathbb{W} \quad \Phi \equiv u'[a] \geq x \wedge u'[a] \geq u[a]}{\mathsf{edge}(\epsilon, u, \mathsf{mint}, \Phi)} \ (\textbf{edge-mint})$$

$$\frac{s : \mathsf{flow}(u, x, a, \bullet) \in \mathbb{W} \quad \Phi \equiv u[a] \geq x \wedge u'[a] \leq u[a]}{\mathsf{edge}(u, \epsilon, \mathsf{burn}, \Phi)} \ (\textbf{edge-burn})$$

$$\frac{s : \mathsf{flow}(u, x, a, b) \in \mathbb{W} \quad \Phi \equiv u[a] \geq x \wedge u'[a] \leq u[a]}{\mathsf{edge}(u, \epsilon, \mathsf{transfer}, \Phi)} \ (\textbf{edge-transfer})$$

Figure 6: Edge inference rules. We omit the constraint for b in edge-swap, edge-borrow, edge-payback, and a, b in loan.

$u[a] \wedge, v'[a] \geq y \wedge v'[a] \leq v[a]$, where $u[a]$ and $v[a]$ denote a's balances of token u and v respectively, while $u'[a]$ and $v'[a]$ denote corresponding balances after firing the edge. This indicates that a needs at least x amount of u token before swapping, and will get at least y amount of v token after. The invocation of such an operation increases a's balance of token v but decreases its balance of token u.

- As mentioned in Section 2, many DeFis provide *flash loans*, a unique feature that enables a (malicious or benign) user to borrow tokens without collateral, as long as the user pays back the loan and its interest within one single transaction. To understand the edge-borrow and edge-payback rules, we first introduce an auxiliary predicate $\mathsf{loan}(s_1, s_2, s_3)$ for identifying flash loan patterns in DeFi. In particular, the loan rule first looks for a statement s_1 together with its corresponding flow. Following s_1, a `callback` statement s_2 is then invoked to register a callback function g, which allows the borrower to execute dedicated business logic and produce another flow (from statement s_3) that pays the original loan. Once a loan pattern is established, the edge-borrow and edge-payback will be triggered simultaneously and generate corresponding `borrow` and `payback` edges. As tokens borrowed could come from different sources, we model the type of token to borrow from and return to using the special node ϵ.
- Flows of tokens from the special address $\bullet$ are directly translated into `mint` edges via the edge-mint rule. The edge goes from ϵ to u token with constraints ensuring sufficient u tokens after the call. Similarly, flows of tokens to the special address $\bullet$ directly construct `burn` edges via the edge-burn rule.
- Other flows that do not fall into the above categories will generate `transfer` edges via the edge-transfer rule. Specifically, give a flow predicate $\mathsf{flow}(u, x, a, b)$, the rule generates a token flow edge (from token u to other participants' token clustered in ϵ) labeled with the `transfer` operator. The constraint on the edge asserts that ① the sender should have sufficient tokens and ② the sender's remaining u tokens decrease after the call.

Algorithm 1 Attack Synthesis

```
1:  procedure AtkSyn(D, S_0, ψ)
2:      Input: DeFi D, Initial State S_0, Attack Goal ψ
3:      Output: Attack Program P or ⊥
4:      κ ← ⊤                                          ▷ initialize knowledge base
5:      𝔾 ← GraphGen(D, S_0)                           ▷ construct token flow graph
6:      while P̃ ← SketchGen(S_0, ψ, 𝔾, κ) do           ▷ enumerate AFL sketch
7:          δ ← CnstGen(φ, R)                          ▷ generate constraints from sketch
8:          while μ ← solve(S_0 ∧ ψ ∧ κ ∧ δ)) do        ▷ get model
9:              if P ← complete(S_0, P̃, μ) then        ▷ attack instantiation
10:                 if P(S_0) ⊨ ψ then                 ▷ validate attack program P
11:                     return P
12:                 else
13:                     κ ← κ ∧ ¬muc(P(S_0) ⊨ ψ)       ▷ update KB
14:     return ⊥
```

$$\begin{aligned} \psi &::= e \mid \neg\psi \mid \psi \wedge \psi \\ e &::= p(\vec{x}, \vec{c}) \mid e_1 \diamond e_2 \mid e_1 \odot e_2 \end{aligned}$$

$$x \in \textbf{variables} \quad c \in \textbf{constants} \quad p \in \textbf{predicates}$$
$$\diamond \in \{+, -, *\} \quad \odot \in \{=, \geq, <\}$$

Figure 7: Syntax for attack goal language. $\vec{x}$ and $\vec{c}$ represent none or more parameters.

6 Attack Synthesis

Like prior sketch-based synthesizers [29, 53, 56], Foray synthesizes candidate attacks through sketch generation and completion. The core insight behind Foray's synthesis algorithm is two-folded. The search space of sketch generation is constrained by graph reachability over a DeFi's TFG (Section 6.2), and the state explosion problem in sketch completion is mitigated by our domain-specific compilation rules over AFL's properties (Section 6.3). In what follows, we first give an overview of Foray's synthesis algorithm (Section 6.1), followed by our attack sketch generation (Section 6.2) and sketch completion (Section 6.3) algorithms.

6.1 Overview of the Synthesis Algorithm

Algorithm 1 shows Foray's top-level attack synthesis algorithm. Given a DeFi protocol, its initial state, and an attack goal (in first-order logic), the synthesis algorithm incorporates a two-phased loop, where phase one (line 6) enumerates attack sketches and phase two (line 8) completes concrete attack programs.

Initial state and attack goal. Figure 7 shows our specification language for expressing initial states and attack goals. Initial states and attack goals are expressed through logical expressions over storage variables x_i or constants c in the DeFi environment, e.g., user balances ($B^{usdce}_{t_2}$), blockchain timestamps, `msg.sender` etc. A complex logical expression e can be composed by arithmetic and logical operators over atomic expressions and custom predicates. Foray converts attack goals into their corresponding first-order logic formulas via syntax-directed translation. For queries that refer to symbols and quantifiers in the program, Foray uses skolemization to make them quantifier-free or reject them otherwise.

Algorithm 2 Attack Sketch Enumeration

```
1:  procedure SketchGen(S_0, ψ, 𝔾, κ)
2:      Input: Initial State S_0, Attack Goal ψ, TFG 𝔾, Knowledge Base κ
3:      Output: Attack Sketch P̃ or ⊥
4:      Assume: 𝔾 = (𝕋, ℙ, 𝔼, Φ)
5:      R ← {}                          ▷ initialize reachable path as ordered set
6:      T, Ω ← init(𝔾, S_0)             ▷ initialize token worklist T and constraint
        store Ω
7:      while choose t ∈ T do           ▷ choose and remove a token from T
8:          E ← {e | ∀e ∈ 𝔼 . e ≡ edge(t, *, *, *)}   ▷ neighboring edges
9:          for each e ∈ E do
10:             if unsat(Ω ∧ κ ∧ e.Φ) then continue
11:             T ← T ∪ {e.out}         ▷ include output node to worklist
12:             Ω ← Ω ∧ e.Φ             ▷ update constraint store
13:             R ← R ∪ {e}             ▷ add edge to reachable path
14:             if α(ψ) ⊆ T then
15:                 P̃ ← (e.op | ∀e ∈ R)   ▷ convert graph to sketch
16:                 return P̃
17:     return ⊥
```

The main loops. Using the rules in Figure 6, the algorithm first constructs a token flow graph from the given DeFi protocol and initial state (line 5). It then invokes an enumeration procedure SketchGen (Section 6.2) that iteratively searches for candidate attack sketches $\tilde{P}$ (line 6). Each sketch $\tilde{P}$ is then compiled by CnstGen into constraints δ that form SMT queries whose solution corresponds to the choices of missing arguments in the attack sketch (line 7). Foray enumerates the solution (a.k.a. *model*) of these queries (line 8). Then, Foray completes the attack sketch $\tilde{P}$ and transforms it into a concrete attack program P through direct syntax transformation (line 9). The algorithm then validates the effectiveness of the attack, by executing it from the initial state and checking whether the attack goal is satisfied (line 10). It returns the concrete attack program P upon passing the validation; otherwise, it invokes a conflict-driven clause learning (CDCL) call (line 13) and moves to the next available candidate.

Conflict-driven learning and knowledge base. To avoid past mistakes, the algorithm also incorporates a knowledge base κ (line 4) that keeps track of constraint clauses that are responsible for each failed validation (line 13).[1] Similar to previous works on conflict-driven program synthesis [16, 29], this allows Foray's synthesis algorithm to avoid previously failed cases (by associating the "root cause" with corresponding constructs in a candidate program) and refine them for better candidates. As such, the knowledge base κ is passed as the argument of sketch generation (line 6).

6.2 Attack Sketch Generation via Graph Reachability Analysis

To generate an attack sketch, Foray performs reachability analysis over the TFG and enumerates a reachable *path* that consists of multiple edges in the TFG. The path points from some initial token node (typically *void*, indicating the attacker does not hold that token) to a target token node that the attacker aims to acquire. Here, each edge is attached with an AFL operator p and a behavioral constraint Φ that encodes the pre- and post-condition of triggering p (Figure 6).

Goal-directed reachability analysis. An attack goal ψ in Figure 7 specifies a logic formula over account balances with *target token(s)* of interest to the attacker. To satisfy the goal, a feasible sketch has to end up with states that "produce" the target token(s) in ψ, by firing a sequence of AFL operators in a path R. Formally speaking, a feasible sketch corresponds to a path in the token flow graph that satisfies the following conditions:

(1) Satisfiability condition: whether the behavioral constraints Φ along the path R can be satisfied, and
(2) Coverage condition: whether the path R covers the target token(s) in the attack goal (denoted by $\alpha(\psi)$).

Sketch enumeration. Given a token flow graph along with its initial state, attack goal, and knowledge base, the algorithm returns an attack sketch $\tilde{P}$ corresponding to a reachable path. It consists of a sequence of AFL operators on tokens defined in the TFG. The algorithm's main loop (line 7-16) is based on a worklist mechanism that gradually refines the current path until a reachable one is constructed. Initially an empty path R, together with the token worklist T and constraint store Ω is created (line 5-6), where T is initialized as tokens that the attacker holds, and Ω stores constraints converted from initial state S_0. If the attacker does not hold any tokens in the TFG, we initialize T with ϵ.

At each step of the main loop, a token t is first chosen from the worklist T (line 7). Then, for each edge e that starts from t (lines 8-9), the algorithm ensures the satisfiability condition is met by checking the conjunction of three sets of constraints using the Z3 solver (line 10); otherwise, it continues with the next available edge. For a satisfiable edge e, the algorithm updates the token worklist by adding its output token $e.o$, the constraint store by adding its constraint $e.\phi$ (the constraint of triggering its corresponding operator), and the reachable path set R by adding e (lines 11-13). Then, it checks for the coverage condition by seeking the existence of target tokens from R (line 14). The path R is finally converted into an attack sketch $\tilde{P}$ and return if the coverage condition is met (line 15); otherwise, the algorithm keeps trying for the next pair token t and edge e until it finds a satisfiable one or terminate by exhaustion. Note that every time a valid sketch $\tilde{P}$ is found and returned, the following lines in Algorithm 1 will be invoked. If $\tilde{P}$ fails to achieve the attack goal, the corresponding root cause will be added to κ and fed back to SketchGen. The R, T, Ω will be reinitialized for generating a new sketch and κ ensures that the algorithm avoids the previously failed sketches.

Example 6.1 (Attack sketch generation). In Figure 3(d), a reachable path on the token flow graph begins at the ϵ node, representing a common scenario where the attacker initially possesses no tokens and must borrow from other entities (❶). Navigating through the graph (❷ - ❸), the attacker is then required to repay the borrowed tokens to prevent execution failure by ending with calling payback and going back to the start node . The sequence of corresponding operators (`borrow -> swap -> payback`) along this generated path constitutes a viable sketch candidate for executing the attack.

[1] muc stands for "minimum unsat core". This corresponds to the feature of *unsat core* computation, which is broadly available in modern SMT solvers.

6.3 Sketch Completion via Domain-Specific Compilation

We aim to compile the sketch into a constraint system whose solution results in the completion of an attack program. In particular, using our AFL semantics, we derive a domain-specific compilation that translates the invocation of each AFL operator into high-level constraints. Our constraints are much easier to solve as they only track the side effects of AFL operators over the attacker's account balances and filter out low-level semantics of the original DeFi.

Figure 8 shows the inference rules for generating constraints of different AFL operators defined in Figure 4. The rules derive judgments of the form $p \Downarrow C$, where C corresponds to the set of constraints obtained by *symbolically evaluating* an AFL operator p. For simplicity, we use two macros $\uparrow(u, a, x)$ and $\downarrow(u, a, x)$ to denote the constraints for describing a balance increase and decrease of amount x of the token u at address a, which compiles to $u'[a] = u[a] + x$ and $u'[a] = u[a] - x$, where $u[a]$ and $u'[a]$ denotes the balance of token u for address a before and after evaluating the corresponding operator p.

Each inference rule in Figure 8 models the change of account balances caused by the corresponding AFL operator. For instance, the c-transfer rule generates constraints to assert the increased and decreased amounts of recipient and sender, respectively. The c-swap rule states that from a sender's view (address a), the balance of its source token will decrease and its target token will increase. The recipient's (address b) case is the inverse.

In addition to modeling balance changes, the rules also model financial features for certain operators. For example, for swap operator, besides the macro $\varsigma(a, u, v, x, y, b)$ that describes mutual balance changes between address a and b,[2] we introduce $\rho(x, y)$ to model the *invariant* between token pairs in modern automated market makers (e.g., $x \cdot y = k$ in Uniswap).

Meanwhile, for tokens that provide flash loans, the constraint of an additional fee is modeled via $\vartheta(x, y)$, where x is the amount of flash loan and y is the amount of repayment, and $y > x$ in most cases means additional interest is charged in payback.

Such constraints are inferred in a data-driven way via analysis of massive amounts of real-world transaction data. Since the arguments of an AFL operator may refer to local variables, we leverage off-the-shelf pointer analysis to resolve their actual locations.

Given a sketch $\tilde{P} = (p_1, p_2, ...)$, the constraints of $\tilde{P}$ are obtained by 1) applying the inference rule on each p_i and then 2) conjoining all the resulting constraints together: $\textsc{CnstGen}(S_0, \tilde{P}) = \mathsf{foldl}(S_0, \mathsf{map}(\tilde{P}, \Downarrow), \wedge)$.

7 Implementation

We have implemented Foray in Python with a back-end constraint solver (Z3 [20] version 4.12.2). To fetch the concrete state and verify the feasibility of the attack sketches, Foray integrates Foundry [32] to interact with the blockchain. In what follows, we elaborate on various aspects of our implementation.

Attack goal generation. In real-world cases, an attack who seeks for financial gains would try spending no assets when launching an attack (i.e., launching an attack with no cost), which requires

[2]This compiles to $\downarrow(u, a, x) \wedge \uparrow(u, b, x) \wedge \downarrow(v, b, y) \wedge \uparrow(v, a, y)$.

$$\frac{p \equiv \mathsf{transfer}(u, a, b, x)}{p \Downarrow \downarrow(u, a, x) \wedge \uparrow(u, b, x)}\ \textbf{(c-transfer)} \qquad \frac{p \equiv \mathsf{burn}(u, a, x)}{p \Downarrow \downarrow(u, a, x)}\ \textbf{(c-burn)}$$

$$\frac{p \equiv \mathsf{mint}(u, a, x)}{p \Downarrow \uparrow(u, a, x)}\ \textbf{(c-mint)} \qquad \frac{p \equiv \mathsf{swap}(a, u, v, x, y, b)}{p \Downarrow \varsigma(a, u, v, x, y, b) \wedge \rho(x, y)}\ \textbf{(c-swap)}$$

$$\frac{p \equiv \mathsf{borrow}(u, a, b, x)}{p \Downarrow \downarrow(u, a, x) \wedge \uparrow(u, b, x)}\ \textbf{(c-borrow)}$$

$$\frac{p \equiv \mathsf{payback}(u, a, b, y)}{p \Downarrow \downarrow(u, a, y) \wedge \uparrow(u, b, y) \wedge \vartheta(x, y)}\ \textbf{(c-payback)}$$

Figure 8: Domain-specific constraint compilation rules.

appropriate choices of an attack goal. To achieve this, Foray automatically gathers all *stablecoins* involved in the target DeFi protocol from their on-chain storage variables, and includes them as potentially hackable assets into the attack goal. Foray then tries to solve a feasible attack program for each hackable asset.

For example, in the MUMUG protocol mentioned in Section 3, users could spend USDCe to buy MU without any incentivization. We abbreviate the beginning and ending balance of USDCe of an attacker as $B^{usdce}_{t_1}$ and $B^{usdce}_{t_2}$, accordingly. The contract invariant can then be formalized as:

$$B^{usdce}_{t_2} - B^{usdce}_{t_1} \leq 0,$$

and the attack goal, formalized as (1), is to find a concrete exploit that violates the above invariant.

Inference of token flow. Foray compiles a DeFi protocol (e.g., in Solidity) to its AFL representation via the following steps:

- A static analysis procedure (e.g., provided by Slither [28]) is invoked to first generate machine-readable intermediate representation (IR), e.g., Slither IR, of the DeFi protocol.
- Token flows can then be identified from the generated IR via standard interfaces, e.g., ERC20 [59], ERC721 [27], and ERC1155 [51] in Solidity/EVM, and extracted on statement level, as described by Figure 5.
- Foray then infers the corresponding AFL functions from the identified token flows via rules defined in Figure 6.

8 Evaluation

All experiments are conducted on an Amazon EC2® instance with an AMD EPYC 7000® CPU, 8 Cores, and 64G of memory running on Ubuntu 20.04. We set the default timeout for the solver as 3 hours. This number is obtained by observing the performance of Halmos. In most cases, it either finishes the process at around 2-3 hours or fails completely. Our evaluation plans to answer the following research questions:

- (RQ1): How does Foray perform compared to SOTA tools?
- (RQ2): Is Foray effective in detecting known vulnerabilities?
- (RQ3): How effective are the two key designs of Foray and whether Foray will introduce false positives?
- (RQ4): Can Foray be useful in detecting zero-day vulnerabilities?

8.1 Detecting Known Vulnerability (RQ1&RQ2)

Benchmark. To evaluate Foray on known vulnerabilities, we select our benchmarks from the DeFiHackLabs dataset [23], which keeps track of all DeFi hack incidents in the past. The DeFiHackLabs dataset records 389 incidents (at the time of the submission).

We consider a subset of 200 benchmarks from Jan 2022 to July 2023 and exclude old benchmarks before 2022 because they depend on outdated versions of the Solidity compiler. Furthermore, we exclude benchmarks from one of these categories: a) closed source, b) common vulnerabilities (as referred in Section 2) such as integer overflow, reentrancy, access controls, etc., and c) insider hacks due to losing primary keys or misconfiguration. Our dataset ends up with 34 representative benchmarks. To get better insights into the root causes of the benchmarks, we also categorize them into four types of logical flaws: ❶ *Token Burn* (TB), where the attack can indirectly mint or burn the victim's tokens by calling the corresponding mint or burn function through other public functions (similar to privilege escalation); ❷ *Pump & Dump* (P&D): inflating the price of a token through abnormal financial transactions (e.g., spitefully inflates the token price through substantial purchases); ❸ *Price Discrepancy* (PP), which allows the attack to generate profits based on the price difference of the same token pair in different smart contracts (e.g., MUMUG); ❹ *Swap Rate Manipulation* (SR): the attack can directly or indirectly influence the swap rate between multiple token pairs in the same smart contract. In total, the selected benchmark vulnerabilities have cost > $21M of losses.

Baseline. As discussed in Section 3, there are two possible existing solutions for our problem. We select one SOTA tool for each solution as our baseline method. For sketch generation and completion, we use our sketch generation method (given that no existing tool can strategically generate sketches) and use Halmos [2], the SOTA symbolic reasoning tool for DeFi, for sketch completion. We select ItyFuzz [52], the SOTA tool for cross-contract fuzzing, as our baseline method for the fuzzing solution. Note that an existing DeFi security tool, DeFiPoser [66], also follows the sketch generation and completion methodology but can only be applied to arbitrage (PP in our benchmark). Due to its limited scope and lack of open-source implementation, we do not include it as our comparison baseline.

We run Foray, Halmos, and ItyFuzz on the selected benchmarks using the same computational resource and timeout limit mentioned above. We report the runtime needed for each method to detect each selected vulnerability. We also report the average run time over the success cases (the vulnerabilities that are detected within the time limit) and the overall success rate to assess the effectiveness and efficiency of each tool.

Results. Table 1 shows the main results of the three tools on the selected 34 benchmarks. Here, the first two columns represent the name and category of each benchmark. Columns 3-5 show the running time of Foray, Halmos, and ItyFuzz, respectively. We treat "TO" and "NA" as failure cases. Foray successfully synthesizes the attack programs for 79% benchmarks whereas Halmos and ItyFuzz only solve 9% and 32% benchmarks, respectively. This result demonstrates that by modeling financial logic, Foray is significantly more effective in synthesizing DeFi logical bugs compared to SOTA tools. These tools often struggle to capture application logic and rely on brute-force solutions. Furthermore, Foray is also more efficient than baseline approaches in that it takes an average time of 105.9 seconds to solve 27 benchmarks. In comparison, Halmos takes an average time of 8,085.0 seconds to solve three benchmarks and ItyFuzz takes an average time of 307.1 seconds to solve 11 benchmarks from our dataset. Foray's high efficiency benefits from its strategical sketch generation, which improves the search efficiency, and its domain-specific compilation, which simplifies the constraints. The "# of Sketches" column shows the number of sketches generated by Foray. On average, 31.7 sketches are generated, with a maximum of 180 for benchmark SellToken02 and a minimum of 16.

Table 1: Running time of Foray vs. Halmos and ItyFuzz on the selected benchmark. "TO" means the tool cannot find a valid attack for the corresponding vulnerability within the time limit, and "NA" means the benchmark is not supported.

Name	Category	Foray	ItyFuzz	Halmos	# of Sketches
AES	TB	25.1s	27.0s	TO	16
BGLD	TB	25.1s	172.0s	TO	16
BIGFI	TB	25.1s	511.0s	TO	16
BXH	P&D	350.5s	TO	TO	38
Discover	PP	325.1s	NA	10251.3s	16
EGD	P&D	25.6s	2.0s	TO	56
MUMUG	PP	300.2s	NA	7681.7s	16
NOVO	TB	25.1s	81.0s	TO	16
OneRing	P&D	25.3s	TO	TO	32
RADTDAO	TB	25.1s	627.0s	TO	16
RES	SR	25.2s	3.0s	TO	16
SGZ	SR	25.2s	TO	TO	30
ShadowFi	TB	25.1s	1757.0s	TO	16
Zoompro	SR	25.2s	TO	TO	36
NXUSD	P&D	TO	TO	TO	–
NMB	P&D	626.3s	TO	TO	21
Lodestar	P&D	TO	TO	TO	–
SafeMoon	TB	25.1s	TO	TO	16
Allbridge	PP	TO	NA	TO	–
Swapos V2	SR	25.7s	182.3	6322.0s	80
Axioma	P&D	25.7s	TO	TO	22
0vix	PP	TO	NA	TO	–
NeverFall	P&D	100.7s	TO	TO	16
SellToken02	P&D	26.0s	TO	TO	180
LW	PP	25.1s	NA	TO	16
UN	TB	25.1s	10.1s	TO	16
CFC	TB	625.2s	TO	TO	90
Themis	P&D	TO	TO	TO	–
Bamboo	TB	25.1s	5.2s	TO	16
LUSD	P&D	25.1s	TO	TO	16
RodeoFinance	PP	TO	NA	TO	–
Carson	PP	TO	TO	TO	–
XAI	TB	25.1s	TO	TO	16
Hackathon	TB	25.1s	TO	TO	16
	Succ. rate	79% (27/34)	32% (11/34)	9% (3/34)	Avg. #
	Avg. Time	105.9s	307.1s	8085.0s	31.7

We took a closer look at Foray's performance regarding different categories of benchmarks and realized that Foray performs better on TB and SR than PD and PP. Compared to other types of vulnerabilities, PD and PP usually require a series of repeated arbitrages and flash loans to reach the preset profit in the attack goal. Therefore, the number of parameters and steps in those benchmarks is larger and it takes longer for the synthesizer to enumerate and verify candidate attacks.

8.2 Ablation Study and False Positive (RQ3)

Benefit of domain-specific compilation. Given that Halmos and Foray use the same sketch generation procedure, Halmos is equivalent to the ablative version of Foray without domain-specific compilation. In other words, the difference in their performance as shown in Table 1 is mainly caused by the different mechanisms of symbolic compilation. Halmos uses a general-purpose compilation to symbolically evaluate each benchmark using *concrete semantics* of solidity. It only solves the three easiest benchmarks. Halmos generates 1,360 and 2,179 constraints for Discover and MUMUG,

```
function DeFiLender.flashloan(...) public {}
function DeFiLender.payback(...) public {}

function Uniswap.swap(...) public {}

// transfer and swapBack of victim
function transfer(to, amt) public {
    ...
    if (...) { swapBack(); }
    ...
}

function swapBack() internal {
    ...

    Uniswap.swap(victim, wBNB, amt_1, 0);
    ...
}
```

(a) The (partial) vulnerable DeFi protocol

```
function exploit() {
    // flashloan wBNB
    DeFiLender.flashloan(wBNB, 6.3e16);

    // trigger the bug by transfer
    Victim.transfer(address(0x0), 0);

    // swap: wBNB -> victim -> bUSD -> wBNB

    Uniswap.swap(wBNB, victim, 6.3e16, 1.2e12);

    Uniswap.swap(victim, bUSD, 1.2e12, 16.5e18);

    Uniswap.swap(bUSD, wBNB, 16.5e18, 6.5e16);

    // payback wBNB
    DeFiLender.payback(wBNB, 6.3e16);
}
```

(b) The (simplified) attack program

Figure 9: A zero-day vulnerability detected by FORAY. "victim" stands for the address of the token issued by the Victim contract.

whereas FORAY only generates 64 and 120 constraints, respectively. This confirms that our domain-specific compilation significantly reduces the amount of generated constraints, greatly simplifies the solving process, and thus enables more successful cases.

Benefits of sketch generation. To further evaluate the effectiveness of our attack generation algorithm, we replace it with a straightforward breadth-first search and keep all other comments the same. This method brute forces all operators to have a certain length, starting from a length of one, where each operator is treated as a program sketch. Our result shows that FORAY times out on all benchmarks. This is due to the straightforward solution enumerating a huge number of sketches, causing time out. The result verifies the necessity of our sketch generation method in improving the overall efficiency of the synthesis process.

False positives. We run FORAY on 50 benign DeFi protocols, which contain the 34 benchmarks in Table 1 after fixing the bugs and ten popular DeFi protocols from Defillama (Lido [40], MakerDAO [43], Aave [3], etc.). We treat the 10 popular protocols as benign because they pass commercial auditing. Our results show that FORAY timed out (even after we increased the timeout time to 6 hours) on all those benchmarks and did not find any attacks. This result validates FORAY's capability of avoiding false positives.

8.3 Detecting Zero-day Vulnerability (RQ4)

The BNB chain has gained significant traction recently due to its low transaction fees. However, it also accounts for 30% of recent exploits, according to professional web3 security reports [11, 45]. To explore more potential issues, we applied FORAY to 5,000 high-profile DeFi protocols on the BNB chain and uncovered 10 previously unknown vulnerabilities, ranging from different types of logical flaws (TB/P&D/PP/SR). These vulnerable DeFi protocols have a total TVL of 1.1M USD, with the maximum, minimum, and average TVLs being 398K, 2.7K, and 10.7K, respectively. In terms of transactions, these protocols have a total of 1.4M transactions, with the most popular one having 1.3M transactions and the most recently deployed one having only 28 transactions. On average, these protocols have 140K transactions, indicating their activity levels.

This result confirms FORAY's capability of discovering diverse unseen vulnerabilities, which are challenging for existing pattern matching-based approaches (e.g., DeFiRanger [62] and DeFiTainter [38]). Furthermore, the attack synthesized by our tool typically involves more than five transaction actions, which are challenging for general-purpose symbolic execution (e.g., HALMOS and DeFiPoser [66]) and fuzzing tools (e.g., ITYFUZZ).

All bugs found by FORAY are reported to, confirmed, and fixed by corresponding project developers through private channels. We help project developers avoid financial loss via three ways:

- Use the administrative functions of the protocol to disable the vulnerable public functions.
- Lock the protocol and return assets to users.
- Upgrade their smart contracts if possible.

Here, we illustrate one major vulnerability belonging to SR to show how FORAY synthesizes the exploit. Figure 9 shows the buggy protocol and its exploit generated by FORAY. The victim protocol has a logical flaw in its token swap mechanism, i.e., `swapBack` function that will cause a price change between victim and wBNB. Specifically, as shown in Figure 9(b), FORAY generates a program with six concrete function calls. Here is the logic to trigger the vulnerability: ❶ The attacker takes a flash loan of some WBNB tokens by calling `DeFiLender.flashloan`. ❷ The attacker then calls `Victim.transfer` to trigger the `swapBack`. As shown in Figure 9(a), the internal function `swapBack` swaps a certain amount `amt_1` of victim to wBNB, causing a devalue of victim and increasing value of wBNB in the `Uniswap` contract. ❸ the attacker leverage the price change to swap more victim with the loaned wBNB. ❹ ❺ Attacker sequentially swaps Victim to bUSD and bUSD to wBNB. Given that the attacker gets more victim than usual cases after ❸ This enables the attacker to get more wBNBs than its original loaned amount. ❻ Eventually, attacker calls `DeFiLender.payback` to pay back the flash loan and keep the extra $0.2e^{16}$ wBNB as the profit. The exploit program plunderers approximately 11% of the valuable stablecoins (BUSD) in the liquidity pool as the profit. FORAY spent 318.4 seconds synthesizing this program while neither HALMOS nor ITYFUZZ synthesizes a comparable solution within the allotted time frame.

9 Discussion

Generalizability and scalability. As illustrated in Section 8, FORAY can synthesize attacks for various types of logical bugs that current tools cannot detect. However, we acknowledge that there are more types of deep logical bugs that our tool has not yet addressed [65, 67]. So far, these vulnerabilities have been discovered by highly experienced human auditors. By extending our TFG construction and compilation rules, FORAY can be generalized to address other vulnerabilities as well. For example, we can introduce a higher order operator that conducts individual AFL operators multiple times to handle erroneous accounting [65], which requires accumulating a small computational discrepancy multiple times. Similarly, FORAY can also be generalized to common vulnerabilities although they are not our focus. Our future work will extend FORAY to more types of deep logical vulnerabilities.

Section 8 demonstrates that FORAY significantly outperforms existing tools in synthesizing complicated logical bugs (e.g., the zero-day bug in Section 8.3). However, we also notice that FORAY still fails to synthesize some ultra-complicated cases (Table 1) due to the limited capability of the SOTA solver. In our future work, we will explore hybrid approaches that leverage symbolic execution and fuzzing for sketch completion to improve scalability. Note that

our sketch generation would still be valuable in that it is challenging for fuzzing to generate valid transaction sequences.

Manual efforts. So far FORAY still requires certain manual efforts for the generation of the attack goal and initial state specification, as well as additional function mappings. Here, additional function mappings refer to the auxiliary parameters and extra function calls that must be incorporated when mapping an AFL action back to concrete functions. These manual efforts are still way lower than the amount of effort needed to summarize patterns from historical attacks or manual auditing. In addition, pattern summarization and matching have limited generalizability. Our future works will explore automating these steps, such as leveraging deep learning to generate specifications [39] and data mining to extract additional function mappings [5].

Defense. As an offensive defense work, our ultimate goal is to uncover more attacks before they actually happen and provide such attacks to DeFi developers and users so that they can improve their protocol or transaction safety. FORAY's capability of providing exploits makes it easier for developers to analyze the root cause and apply proper defenses. In general, we can patch the vulnerable protocol or add run-time assertions. For example, we can fix the bug in MUMUG by upgrading the way of deciding converting price between MU and USDCe such that the price is robust against the dramatic changes in their reservations.

DSL design choices and VM compatibility. The design of the existing DSL (Figure 4) as well as the token flow graph (Section 5), considers a balance among generality, efficiency, and the amount of domain knowledge incorporated. As we show the flexibility of FORAY, in practice, one can always lean towards different design choices (e.g., towards more precise domain knowledge) and adjust the DSL and graph structure accordingly. FORAY is instantiated in Solidity in our evaluation, which is a programming language supported by any EVM-compatible VMs. Since our DSL is language-agnostic, with sufficient engineering effort, FORAY can be instantiated with other VMs (e.g., MoveVM [8], SVM [64], etc.) as well.

Extension with data-driven approaches. During synthesis, FORAY has to make decisions on which DeFi protocols and functions to enumerate. While this is still an open problem, compared to a brute-force enumeration, the key insight of FORAY is to leverage the token flow graph and attack goal to avoid enumerating choices doomed to fail. Such a core insight naturally gives a potential future extension that leverages data-driven approaches to explore candidates that maximally align with the application logic. We believe the modularity of FORAY's procedures opens up new room for enhancement and integration of data-driven approaches.

Complex path conditions and statements. Flow predicates are used to construct token flow graphs, which are used by sketch enumeration. Since a token flow graph over-approximates the behavior of a Solidity program, most complex path conditions and statements, including modeling of access-controls are abstracted away conservatively during the sketch enumeration phase and the precision loss will be recovered in a goal-driven way during the sketch completion phase via a CEGIS procedure.

10 Related Work

Smart contract vulnerability analysis. Existing tools for detecting and analyzing smart contract vulnerabilities can be categorized into either static analysis [4, 33, 34] or dynamic analysis [18, 37, 52] approaches. Static tools conduct static analysis or symbolic execution to detect the common vulnerability (mentioned in Section 2) that does not require a deep understanding of a DeFi protocol. Notably, Securify [57] analyzes a smart contract's bytecode and finds pre-defined patterns in its control flow graph corresponding to certain bug types. Slither [28] (also used in FORAY) is the most stable and frequently maintained static analysis framework to analyze smart contracts. Notable symbolic execution tools include Manticore [46], Mythril [19], Solar [31], and HALMOS [2] (the SOTA). As demonstrated in Section 8, without effective sketch generation and domain-specific compilation, solely relying on symbolic execution cannot handle deep logical bugs in DeFi protocols. Most dynamic and hybrid analysis tools are designed to be used within one smart contract [10, 37, 47, 63]. Without an understanding of protocol logic, the fuzzers that support cross-contract fuzzing (e.g., ItyFuzz [52]) cannot maintain their effectiveness in DeFi attack synthesis.

DeFi Security. The key challenge for DeFi security lies in the larger size and broader scope beyond individual smart contracts as well as the complicated semantics and logic involved. Aside from Zhou et al. [67] which conducts a comprehensive summary of existing DeFi attacks, existing works in this domain mainly follow the methodology of summarizing patterns from existing attack instances and building attack detection tools via pattern matching. Specifically, DeFiRanger [62] lifts the low-level smart contract semantics to high-level ones and uses them to summarize and express patterns. FlashSyn [17] leverages numerical approximation to extract patterns from attack transaction sequences and detect suspicious transactions during run time. UnifairTrade [13] identifies fragile swap pair implementations as patterns. DeFiTainter [38] conducts taint analysis with taint source and target summarized from standard smart contract API templates. The capability and scalability of these approaches are constrained by the pattern extraction step. In fact, the above approach can only detect a certain type of price manipulation vulnerability that leverages `swap` to manipulate token prices (e.g., MUMUG). More recent tools also extend this methodology to other vulnerabilities. For example, DeFiCrisis [35] introduces strategies for exploiting DeFi governance mechanisms by arranging funding to gain profits. TokenScope [14] is designed to detect any inconsistent and phishing behaviors in token applications. The technique that most aligned with FORAY is DeFiPoser [66], which proposes two strategies to facilitate the generation of exploit for profit. The first strategy creates sketches using heuristics and then completes them with an SMT solver, while the second strategy identifies potential trades through a method known as negative cycle arbitrage detection. Due to the limitation in sketch generation, this tool can only work with arbitrage detection, whereas FORAY can be applied to a variety of DeFi protocols, detect different financial flaws, and synthesize complex trading sequences.

Attack synthesis and exploit generation. The synthesis of cyber-attacks and the automated generation of exploits have been subjects of significant research interest, aiming to understand and mitigate security vulnerabilities. The seminal work, AEG [6], used

symbolic execution techniques to generate the exploit for the shell program. Attack synthesis techniques have been applied to many domains, such as Mayhem [12] using concolic execution for Linux Kernel, Intellidroid [60] using dynamic analysis and fuzzing for Android, HeapHopper [26] using bounded model checking for Memory allocator, AASFSM [48] using NLP techniques for TCP and DCCP protocols and etc. Symbolic execution is a well-adopted technique to generate a specific exploit, which creates a set of constraints based on the original program and then solves them by delegating SMT solvers. Compared with a general symbolic execution technique, Foray first benefits from general financial knowledge to eliminate the search space of synthesis efficiently, then do the domain-specific compilation to generate more lightweight constraints for existing SMT solvers to solve, eventually becoming scalable in the DeFi attack synthesis domain.

11 Conclusion

We present Foray, a highly effective attack synthesis framework against deep logical bugs in DeFi protocols. Different from existing tools that only detect common vulnerabilities in individual smart contracts, Foray effectively models the financial logic in DiFi protocols and synthesizes exploits against logical flows accordingly. Our evaluation of 34 benchmark DeFi security attacks demonstrates the advantage of Foray over existing smart contract bug-haunting approaches. We further show that Foray can uncover ten zero-day vulnerabilities from the BNB chain. Finally, we demonstrate the effectiveness of Foray's two key designs (sketch generation and completion) and its capability of avoiding false positives. From extensive evaluation, we can safely conclude that with domain-specific modeling and compilation, symbolic reasoning can be an effective approach for exploit synthesis against deep logical bugs in DeFi protocols.

12 Acknowledgements

We are truly grateful for the time and effort that the anonymous reviewers invested in reviewing our work and offering valuable feedback. This work is supported in part by Google Faculty Research Award, Ethereum Foundation Academic Award, NSF 1908494, and DARPA N66001-22-2-4037. The views and conclusions contained in this document are those of the authors. They should not be interpreted as representing the official policies, expressed or implied, of the funding agencies.

References

[1] 1inch. 2023. One-stop access to decentralized finance. https://1inch.io/.
[2] a16z. 2023. Halmos: A symbolic testing tool for EVM smart contracts. https://github.com/a16z/halmos.
[3] AAVE. 2023. Aave: Open Source Liquidity Protocol. https://aave.com/.
[4] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020. Taming Callbacks for Smart Contract Modularity. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 209 (nov 2020), 30 pages. https://doi.org/10.1145/3428277
[5] Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 4–16.
[6] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84.
[7] Binance Smart Chain Developers. 2017. Binance Smart Chain Whitepaper. https://github.com/bnb-chain/whitepaper/blob/master/WHITEPAPER.md.
[8] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. 2019. Move: A language with programmable resources. *Libra Assoc* (2019), 1.
[9] blockworks. 2023. Mango Markets Mangled by Oracle Manipulation for $112M. https://blockworks.co/news/mango-markets-mangled-by-oracle-manipulation-for-112m/.
[10] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, and Christopher Kruegel. 2022. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*.
[11] CertiK. 2023. Hack3d: The Web3 Security Report 2023. https://www.certik.com/resources/blog/7BokMhPUgffqEvyvXgHNaq-hack3d-the-web3-security-report-2023. Accessed: 2024-07-24.
[12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 380–394.
[13] Jiaqi Chen, Yibo Wang, Yuxuan Zhou, Wanning Ding, Yuzhe Tang, XiaoFeng Wang, and Kai Li. 2023. Understanding the Security Risks of Decentralized Exchanges by Uncovering Unfair Trades in the Wild. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 332–351. https://doi.org/10.1109/EuroSP57164.2023.00028
[14] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1503–1520. https://doi.org/10.1145/3319535.3345664
[15] Yanju Chen, Junrui Liu, Yu Feng, and Rastislav Bodik. 2022. Tree Traversal Synthesis Using Domain-Specific Symbolic Compilation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1030–1042.
[16] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. 2020. Program Synthesis Using Deduction-Guided Reinforcement Learning. In *Computer Aided Verification*, Shuvendu K Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 587–610.
[17] Zhiyang Chen, Sidi Mohamed Beillahi, and Fan Long. 2022. FlashSyn: Flash Loan Attack Synthesis via Counter Example Driven Approximation. arXiv:2206.10708 [cs.PL]
[18] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 227–239. https://doi.org/10.1109/ASE51524.2021.9678888
[19] ConsenSys. 2020. Mythril: Security Analysis Tool for Ethereum Smart Contracts. https://github.com/ConsenSys/mythril.
[20] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 337–340.
[21] decrypt. 2023. Zunami Protocol Loses Over $2.1 Million in Price Manipulation Hack. https://decrypt.co/152366/zunami-protocol-curve-finance-hack/.
[22] DeFi Prime. 2023. Ethereum DeFi Ecosystem. https://defiprime.com/ethereum
[23] DeFiHackLabs. 2023. DeFi Hacks Reproduce - Foundry. https://github.com/SunWeb3Sec/DeFiHackLabs.
[24] defillama. 2023. DefiLlama - DeFi Dashboard. https://defillama.com/.
[25] DYDX. 2023. dYdX: Trade Perpetuals on the most powerful trading platform. https://dydx.exchange/.
[26] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2018. {HeapHopper}: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium (USENIX Security 18)*. 99–116.
[27] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. 2018. ERC-721: Non-Fungible Token Standard. Ethereum Improvement Proposals. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-721.
[28] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE. https://doi.org/10.1109/wetseb.2019.00008
[29] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 420–435.
[30] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 422–436.

[31] Yu Feng, Emina Torlak, and Rastislav Bodik. 2021. Summary-Based Symbolic Evaluation for Smart Contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1141–1152. https://doi.org/10.1145/3324884.3416646

[32] foundry team. 2021. Foundry: A Blazing Fast, Portable and Modular Toolkit for Ethereum Application Development. https://github.com/foundry-rs/foundry.

[33] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving out-of-Gas Conditions in Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 116 (oct 2018), 27 pages. https://doi.org/10.1145/3276486

[34] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *Proc. ACM Program. Lang.* 2, POPL, Article 48 (dec 2017), 28 pages. https://doi.org/10.1145/3158136

[35] Lewis Gudgeon, Daniel Perez, Dominik Harz, Benjamin Livshits, and Arthur Gervais. 2020. The Decentralized Financial Crisis. arXiv:2002.08099 [cs.CR]

[36] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.* 4, POPL (Dec. 2019), 1–28.

[37] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

[38] Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. 2023. DeFiTainter: Detecting Price Manipulation Vulnerabilities in DeFi Protocols. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1144–1156. https://doi.org/10.1145/3597926.3598124

[39] Tien-Duy B Le and David Lo. 2018. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 106–117.

[40] Lido. 2023. Lido - Liquid Staking for Digital Tokens. https://lido.fi/.

[41] Junrui Liu, Yanju Chen, Eric Atkinson, Yu Feng, and Rastislav Bodik. 2023. Conflict-Driven Synthesis for Layout Engines. *Proc. ACM Program. Lang.* 7, PLDI (June 2023).

[42] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[43] MakerDAO. 2023. MakerDAO: An Unbiased Global Financial System. https://makerdao.com/.

[44] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 48–61. https://doi.org/10.1145/1065010.1065018

[45] Merkle Science. 2022. Hack Track: Analysis of the BNB Smart Chain Exploit. https://blog.merklescience.com/general/hack-track-analysis-of-the-bnb-smart-chain-exploit. Accessed: 2024-07-24.

[46] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[47] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. SFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 778–788. https://doi.org/10.1145/3377811.3380334

[48] Maria Leonor Pacheco, Max von Hippel, Ben Weintraub, Dan Goldwasser, and Cristina Nita-Rotaru. 2022. Automated Attack Synthesis by Extracting Finite State Machines from Protocol Specification Documents. arXiv:2202.09470 [cs.CR]

[49] PancakeSwap. 2023. Everyone's Favorite DEX. https://pancakeswap.finance/.

[50] Phitchaya Mangpo Phothilimthana, Michael Schuldt, and Rastislav Bodik. 2016. Compiling a Gesture Recognition Application for a Low-Power Spatial Architecture. *SIGPLAN Not.* 51, 5 (jun 2016), 102–112. https://doi.org/10.1145/2980930.2907962

[51] Witek Radomski, Andrew Cooke, Philippe Castonguay, James Therien, Eric Binet, and Ronan Sandford. 2018. ERC-1155: Multi Token Standard. Ethereum Improvement Proposals. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-1155.

[52] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 322–333. https://doi.org/10.1145/3597926.3598059

[53] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404–415.

[54] solidityscan. 2023. ROE Finance hack Analysis — Price Manipulation. https://blog.solidityscan.com/roe-finance-hack-analysis-price-manipulation-6993fbea0d7c/.

[55] Tether Developers. 2014. Tether: Fiat currencies on the Bitcoin blockchain. https://tether.to/en/.

[56] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 530–541.

[57] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

[58] Uniswap. 2023. The Uniswap Protocol. https://uniswap.org/.

[59] Fabian Vogelsteller and Vitalik Buterin. 2015. ERC-20: Token Standard. *Ethereum Improvement Proposals* 20 (Nov 2015). [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-20.

[60] Michelle Y Wong and David Lie. 2016. Intellidroid: a targeted input generator for the dynamic analysis of android malware.. In *NDSS*, Vol. 16. 21–24.

[61] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[62] Siwei Wu, Dabao Wang, Jianting He, Yajin Zhou, Lei Wu, Xingliang Yuan, Qinming He, and Kui Ren. 2021. DeFiRanger: Detecting Price Manipulation Attacks on DeFi Applications. arXiv:2104.15068 [cs.CR]

[63] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A Greybox Fuzzer for Smart Contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. https://doi.org/10.1145/3368089.3417064

[64] Anatoly Yakovenko. 2018. Solana: A new architecture for a high performance blockchain. *Whitepaper* (2018).

[65] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 615–627.

[66] L Zhou, K Qin, A Cully, B Livshits, and A Gervais. 2021. On the just-in-time discovery of profit-generating transactions in DeFi Protocols. 919–936. https://doi.org/10.1109/SP40001.2021.00113

[67] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2444–2461.