FlashSyn: Flash Loan Attack Synthesis via Counter Example Driven Approximation

Anonymous Author(s)

ABSTRACT

In decentralized finance (DeFi), lenders can offer flash loans to borrowers, i.e., loans that are only valid within a blockchain transaction and must be repaid with fees by the end of that transaction. Unlike normal loans, flash loans allow borrowers to borrow large assets without upfront collaterals deposits. Malicious adversaries use flash loans to gather large assets to exploit vulnerable DeFi protocols.

In this paper, we introduce a new framework for automated synthesis of adversarial transactions that exploit DeFi protocols using flash loans. To bypass the complexity of a DeFi protocol, we propose a new technique to approximate the DeFi protocol functional behaviors using numerical methods (polynomial linear regression and nearest-neighbor interpolation). We then construct an optimization query using the approximated functions of the DeFi protocol to find an adversarial attack constituted of a sequence of functions invocations with optimal parameters that gives the maximum profit. To improve the accuracy of the approximation, we propose a novel counterexample-driven approximation refinement technique. We implement our framework in a tool named FlashSyn. We evaluate FlashSyn on 16 DeFi protocols that were victims to flash loan attacks and 2 DeFi protocols from Damn Vulnerable DeFi challenges. FlashSyn automatically synthesizes an adversarial attack for 16 of the 18 benchmark cases. The artifact can be found at https://github.com/FlashSyn-Artifact/FlashSyn-Artifact-ICSE24 or https://zenodo.org/records/10458602.

1 INTRODUCTION

Blockchain technology enables the creation of decentralized, resilient, and programmable ledgers on a global scale. Smart contracts, which can be deployed onto a blockchain, allow developers to encode intricate transaction rules that govern the ledger. These features have made blockchains and smart contracts essential infrastructure for a variety of decentralized financial services (DeFi). As of April 1st, 2023, the Total Value Locked (TVL) in 1,417 DeFi smart contracts had reached 50.15 billion [19].

However, security attacks are critical threats to smart contracts. Attackers can exploit vulnerabilities in smart contracts by sending malicious transactions, potentially stealing millions of dollars from users. Particularly, a new type of security threat has emerged, exploiting design flaws in DeFi contracts by leveraging large amounts of digital assets. These attacks, commonly referred to as *flash loan attacks* [48, 58, 69], typically involve borrowing the required large amount of assets from flash loan contracts. Among the top 200 costliest attacks recorded in Rekt Database, the financial loss caused by 36 flash loan attacks exceeds 418 million USD [48].

A typical flash loan attack transaction consists of a sequence of actions, or function calls to smart contracts. The first action involves borrowing a substantial amount of digital assets from a flash loan contract, while the last action returns these borrowed assets. The sequence of actions in the middle interacts with multiple DeFi contracts, using the borrowed assets to exploit their design flaws. When a DeFi contract fails to consider corner cases created by the large volume of the borrowed assets, the attacker may extract prohibitive profits. For example, many flash loan attacks use borrowed assets to temporarily manipulate asset prices in a DeFi contract to trick the contract to make unfavorable trades with the attacker [12, 53]. Although researchers have developed many automated program analysis and verification techniques [2, 33, 41, 50] to detect and eliminate bugs in smart contracts, these techniques cannot handle flash loan attack vulnerabilities. This is because such vulnerabilities are design flaws rather than implementation bugs. Moreover, these techniques typically operate with one contract at a time, but flash loan attacks almost always involve multiple DeFi contracts interacting with each other.

FlashSyn: We present FlashSyn, the first automated end-to-end program synthesis tool for detecting flash loan attack vulnerabilities. Given a set of smart contracts and candidate actions in these contracts, FlashSyn automatically synthesizes an action sequence along with all action parameters to interact with the contracts to exploit potential flash loan vulnerabilities. Additionally, FlashSyn can analyze past blockchain transaction history, assisting users in identifying candidate actions for synthesis.

The primary challenge FlashSyn faces is that the underlying logic of DeFi actions is often too sophisticated for standard solvers to handle. Even if the action sequence was already known, a naive application of symbolic execution might not be able to find action parameters because it may need to extract overly complicated symbolic constraints causing the solvers to time out. Moreover, FlashSyn synthesizes the action sequence and the action parameters together and therefore faces an additional search space explosion challenge.

FlashSyn addresses these challenges with its novel *synthesis-via-approximation* technique. Instead of attempting to extract accurate symbolic expressions from smart contract code, FlashSyn collects data points to approximate the effect of contract functions with numerical methods. FlashSyn then uses the approximated expressions to drive the synthesis. FlashSyn also incrementally improves the approximation with its novel *counterexample-driven approximation refinement* techniques, i.e., if the synthesis fails because of a large deviation caused by the approximations, FlashSyn collects the corresponding data points as counterexamples to iteratively refine the approximations. The combination of these techniques allows the underlying optimizer of FlashSyn to work with more tractable expressions. It also decouples the two difficult tasks, finding the action sequence and finding the action parameters. When working with a set of coarse-grained approximated expressions, FlashSyn can filter out unproductive action sequences with a small cost.

Experimental Results: We evaluate FlashSyn on 16 DeFi benchmark protocols that were victims to flash loan attacks and 2 DeFi benchmark protocols from Damn Vulnerable DeFi challenges [16].

FlashSyn synthesizes adversarial attacks for 16 out of the 18 benchmarks. For comparison, a baseline with manually crafted accurate action summaries only synthesizes attacks for 7 out of the 18.

Contributions: This paper makes the following contributions:

- **FlashSyn:** The first automated end-to-end program synthesis tool for detecting flash loan attack vulnerabilities. It enables approximate attack synthesis without diving into sophisticated logics of DeFi contracts.
- **Synthesis-via-approximation:** A novel synthesis-via-approximation technique to handle sophisticated logics of DeFi contracts.
- **Counterexample Driven Approximation Refinement:** A novel counterexample driven approximation refinement technique to incrementally improve the approximation during the synthesis process.
- **Experimental Evaluation:** We implemented FlashSyn in a tool and evaluated it on 16 protocols that were victims to flash loan attacks and 2 fictional flash loan attacks.

Our solution, FlashSyn has been adopted and further developed by a leading smart contract auditing company for the detection of flash loan vulnerabilities in DeFi contracts [34].

2 BACKGROUND

Blockchain: Blockchain is a distributed ledger that broadcasts and stores information of transactions across different parties. A blockchain consists of a growing number of blocks and a consensus algorithm determining block order. Each block is constituted of transactions. Ethereum [11, 65] is the first blockchain to support, store, and execute Turing complete programs, known as smart contracts. Many new blockchains use the Ethereum virtual machine (EVM) for execution due to its popularity among developers.

Smart Contracts: Each smart contract is associated with a unique address, a persistent account's storage trie, a balance of native tokens, e.g., Ether in Ethereum, and bytecode (e.g., EVM bytecode [11, 65]) that executes incoming transactions to change the storage and balance. Users interact with a smart contract by issuing transactions from their user accounts to the contract address. Smart contracts can also interact with other smart contracts as function calls. Currently, there are several human-readable high-level programming languages, e.g., Solidity [35] and Vyper [36], to write smart contracts that compile to the EVM bytecode.

Decentralized Finance (DeFi): DeFi is a peer-to-peer financial ecosystem built on top of blockchains [66]. The building blocks of DeFi are smart contracts that manage digital assets. A few DeFi protocols dominate the DeFi market and serve as references for other decentralized applications: stable coins (e.g., USDC and USDT), price oracles, decentralized exchanges, and lending and borrowing platforms. In DeFi, a special type of loan called **flash loan** allows lenders to offer loans to borrowers without upfront collaterals deposits. The loan is only valid within a single transaction and must be repaid with fees before the completion of the transaction.

3 ILLUSTRATIVE EXAMPLE

We next present a motivating example to describe the complexity of flash loan attacks and our proposed approach to synthesize them.

Background: On October 26th 2020, an attacker exploited the USDC and USDT vaults of Harvest Finance, causing a financial loss of about 33.8 million USD. In this section, we will focus on the attack on the USDC vault. The attacker repeatedly executed the same attack vector 17 times targeting the USDC vault. Fig. 1 summarizes the attack vector. The attack vector contains a sequence of actions that interact with the following contracts:

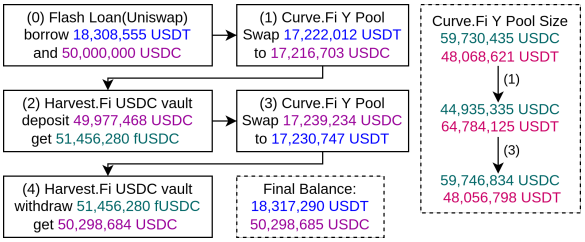

Figure 1: Harvest USDC vault price manipulation attack.

- **Uniswap:** Uniswap is an exchange protocol which also offers flash loan services.
- **Curve:** Curve is an exchange protocol for stable coins like USDT, USDC, and DAI, whose market prices are close to one USD. It maintains pools of stable coins and users can interact with these pools to exchange one kind of stable coins to another. For example, Y Pool in Curve contains both USDC and USDT. Users can put USDC into the pool to exchange USDT out. The exchange rate fluctuates around one, which is determined by the current ratio of USDC and USDT in the pool. Note that internally Y pool automatically deposits USDT and USDC to *Yearn*,[1] keeps yUSDT and yUSDC tokens, and retrieves them back when the users withdraw. We omit this complication for simplicity.
- **Harvest:** Harvest is an asset management protocol and the victim contract of this attack. Users can deposit USDC and USDT into Harvest and receive fUSDC and fUSDT tokens which users can later use to retrieve their deposit back. Harvest will invest the deposited USDC and USDT from users to other DeFi protocols to generate profit. Note that the exchange rate between fUSDC and USDC is also not fixed. It is determined by a vulnerable closed source oracle contract, which ultimately uses Curve Y pool ratios to calculate the exchange rate.

Attack Actions: The attack vector shown in Fig. 1 first flash loaned 18.3M USDT and 50M USDC, then called 4 methods (actions) to exploit the design flaw in Harvest. The first action, action 1, swaps 17, 222, 012 USDT for 17, 216, 703 USDC via *Curve.Fi Y Pool*. Action 1 reduces the estimated value of USDT based on the ratio in Y pool as the amounts of USDT in Y Pool considerably increases. This in turn reduces Harvest Finance's evaluation of its invested assets. Action 2 deposits 49, 977, 468 USDC into *Harvest Finance USDC vault* and due to the reduced evaluation of the invested underlying assets, the attacker receives 51, 456, 280 fUSDC back, which is abnormally large. Similar to action 1, action 3 then swaps 17, 239, 234 USDC back to 17, 230, 747 USDT via *Curve.Fi Y Pool*, which normalized the manipulated USDT/USDC rate in the pool. It also brings Harvest Finance's evaluation of its invested underlying asset back to normal. Finally, action 4 withdraws 50, 298, 684 USDC (using 51, 456, 280 fUSDC) from *Harvest Finance USDC vault*. Assuming 1 USDC = 1 USDT = 1 USD, the profit of the above attack vector is 307, 420 USD.

[1]Yearn is a DeFi protocol that generates yield on deposited assets. yTokens of Yearn represent the liquidity provided in a Yearn product.

```
function get_D(uint[] xp) returns (uint):
  uint N_COINS = xp.length;
  uint S = sum(xp);
  // ...
  uint D = S;
  uint Ann = A * N_COINS; // A is a constant
  for (uint i = 0; i < 255; i = i + 1) {
    uint D_P = D;
    for (uint j = 0; j < xp.length; j = j + 1) {
      D_P = (D_P * D) / (xp[j] * N_COINS + 1); }
    uint Dprev = D;
    D = ((Ann * S + D_P * N_COINS) * D) /
      ((Ann - 1) * D + (N_COINS + 1) * D_P);
      if (abs(D - Dprev) <= 1) { break; }   }
    return D;
```

Figure 2: *get_D* method to compute *D*.

This attack is a typical case of oracle manipulation. The exploiter manipulated the USDT/USDC rate in *Curve.Fi Y Pool* by swapping a large amount between USDC and USDT back and forth, which caused Harvest Finance protocol to incorrectly evaluate the value of its asset, leaving large arbitrage space for the exploiter. The actions sequence and particularly the parameters are carefully chosen by the attacker to yield best profit. There are multiple challenges FlashSyn faces to synthesize this attack.

Challenge 1 - Sophisticated Interactions: The attack involves several smart contracts that interact with each other and with other contracts outside the attack vector. The state changes caused by one action influence the behavior of other actions. This makes the synthesis problem of finding an attack vector more complicated as the effect of an action depends on its predecessor actions thus actions cannot be treated separately.

Challenge 2 - Close Source: Some external smart contracts that a DeFi protocol interacts with are not *open-source*. For instance, the source code of the external smart contract *PriceConverter*[2] of Harvest Finance protocol is not available on Etherscan [31], and it is called by actions 2 and 4 to determine the exchange rate between fUSDC and USDC. This impedes the complete understanding of the DeFi protocol implementation and to reason about its correctness to anticipate attacks vectors.

Challenge 3 - Mathematical Complexity: DeFi contracts use mathematical models that are too complex to reason about. For instance, actions 2 and 4 swap an amount of token i to token j, while maintaining the following *StableSwap invariant* [20]:

$A \cdot n^n \sum_i x_i + D = A \cdot n^n \cdot D + \frac{D^{n+1}}{n^n \prod_i x_i}$ where A is a constant, n is number of token types in the pool(4 for *Curve.Fi Y Pool*)[3], x_i is token i's liquidity, D is the total amount of tokens at equal prices. There does not exist a closed-form solution for D as it requires finding roots of a *quintic equation*. In the actual implementation, D is calculated iteratively on the fly via Newton's method.

To demonstrate the complexity of DeFi protocols, we run an experiment with Manticore [50], a symbolic execution tool for smart contracts, to execute the function *get_D*, for computing D as shown in Fig. 2, with symbolic inputs and explore all possible reachable states. Manticore fails and throws a solver-related exception together with an *out of memory* error. We then simplified *get_D* by removing the outer *for* loop and bounding the length of *xp* to 2, Manticore still fails and throws the same error.

[2]Ethereum address: 0xfca4416d9def20ac5b6da8b8b322b6559770efbf.
[3]Ethereum address: 0x45f783cce6b7ff23b2ab2d70e416cdb7d6055f51.

Table 1: Actions in Harvest USDC Vault attack. IDP and TDP denotes the initial and total number of datapoints. USDT(-) (resp., USDC(+)) denotes USDT (resp., USDC) tokens transferred out (reps., in).

Action	Token Flow	IDP	TDP-Poly	TDP-Inter
exchange (USDT, USDC, v)	USDT(-), USDC(+)	2000	2238	2792
exchange (USDC, USDT, v)	USDC(-), USDT(+)	2000	2148	2888
deposit(v)	USDC(-), fUSDC(+)	2000	2162	2358
withdraw(v)	fUSDC(-), USDC(+)	2000	2364	2876

3.1 Apply FlashSyn

We will now show how FlashSyn synthesizes the Harvest USDC vault attack from the identified set of actions listed in Table 1.[4] The first two input arguments to *exchange* specify the token types to be swapped. The third argument specifies the quantity to swap.[5] Table 1 lists each action's token flow, along with the number of data points collected initially (without counterexamples) and the total number of data points for polynomial and interpolation, respectively. The amounts of tokens transferred in/out for each action are calculated based on its contract's member variables or read-only functions. We refer these variables and functions as **states** of an action. A **prestate** refers to the state before the execution of an action. Executing an action will also alter states, which are denoted as **poststates**. The states not altered by any action are ignored.

For example, `exchange(USDT,USDC,v)` leverages two states, `balances[USDC]` and `balances[USDT]`, to calculate the amounts of token exchanges. Upon execution, this function also modifies these two states. Consequently, these two states act as both the prestates and poststates of `exchange(USDT,USDC,v)`.

Initial Approximation: To generate the initial approximation of the state transition functions of each action, FlashSyn first collects data points where each data point is an *input-output* pair. The *input* is the action's prestates and parameters, and the *output* is its poststates and the outputted values. To collect data points, FlashSyn executes the associated contracts on a private blockchain (a forked blockchain environment) with different parameters to reach *input-output* pairs with different prestates and poststates. FlashSyn then uses the collected data points to find the approximated state transition functions. We consider two techniques to solve the above multivariate approximation problem: linear regression based polynomial features and nearest-neighbor interpolation [49, 55]. The following example is one of *exchange(USDT,USDC,v)*'s state transition functions approximated by polynomials: $s_1' = 0.73244455 \times s_1 - 0.23655202 \times s_2 - 0.85915531 \times v + 27351279.416023515$ where s_1 and s_1' are the prestate and poststate `balances[USDC]`, s_2 is the prestate `balances[USDT]`, and v is the third argument of the action *exchange(USDT,USDC,v)*.

Enumerate and Filter Action Sequences: After capturing an initial approximation of state transition functions, FlashSyn leverages an enumeration-based top-down algorithm to synthesize

[4]In Section 6, we present FlashFind to automatically find the set of candidate actions that are used here by FlashSyn to synthesize an attack vector.
[5]Note that in the implementation the actual name of the *exchange* method is *exchange_underlying*, 1 and 2 are used to identify the tokens USDC and USDT, respectively, and the method has a fourth argument to specify the minimal quantity expected to receive from the swapping.

different action sequences. FlashSyn applies several pruning heuristics to filter unpromising sequences. For each enumerated action sequence, FlashSyn uses the approximated state transition functions to construct an optimization problem, consisting of constraints and an objective function that represents profit. FlashSyn then applies an off-the-shelf optimizer to obtain a list of parameters that maximize the profit estimated using approximated transition functions.

Counterexample Driven Refinement: After obtaining a list of parameters that maximize the estimated profit of an action sequence, FlashSyn proceeds to verify the synthesized attack vectors by executing them on a private blockchain and check their actual profits. If the difference between the actual profit and the estimated profit of an attack vector is greater than 5%, FlashSyn reports it as a counterexample, indicating inaccuracy of our approximated transition functions. To correct this inaccuracy, FlashSyn employs *counterexample-driven approximation refinement* technique. FlashSyn utilizes the reported counterexamples to collect new data points and refine the approximations. The revised approximations are subsequently used to search for parameters in next loops. For example, FlashSyn with polynomial approximations collects 238 additional data points for the action *exchange(USDT, USDC, v)* throughout 7 refinement loops.

Synthesized Attack: In the Harvest USDC example, FlashSyn successfully found the following attack vector that yields an adjusted profit of 110051 USD using the interpolation technique with the counterexample driven refinement loop.

exchange(USDT, USDC, 15192122) · *deposit(45105321)* · *exchange(USDC, USDT, 11995404)* · *withdraw(46198643)*

4 PRELIMINARY

Labeled Transition Systems (LTS). We use LTS to model behaviors of smart contracts. A LTS $A = (Q, \Sigma, \mathrm{q}_0, \delta)$ over the possibly-infinite alphabet Σ is a possibly-infinite set Q of states with an initial state $\mathrm{q}_0 \in Q$, and a transition relation $\delta \subseteq Q \times \Sigma \times Q$.

Execution. An *execution* of A is a sequence of states and transition labels (*actions*) $\rho = \mathrm{q}_0, \mathrm{a}_0, \mathrm{q}_1 \ldots \mathrm{a}_{k-1}, \mathrm{q}_k$ for $k > 0$ such that $\delta(\mathrm{q}_i, \mathrm{a}_i, \mathrm{q}_{i+1})$ for each $0 \leq i < k$. We write $\mathrm{q}_i \xrightarrow{\mathrm{a}_i \ldots \mathrm{a}_{j-1}}_A \mathrm{q}_j$ to denote the subsequence $\mathrm{q}_i, \mathrm{a}_i, ..., \mathrm{q}_{j-1}, \mathrm{a}_{j-1}, \mathrm{q}_j$ of ρ.

Invocation Label. Formally, an *invocation label* $\mathrm{adr}.m(\vec{u})$ consists of a method name m of a contract address adr, accompanied by a vector $\vec{u}$ containing argument values.

Operation Label. An *operation label* $\ell := \mathrm{adr}.m(\vec{u}) \Rightarrow (I, v) \cup \bot$ is an invocation label $\mathrm{adr}.m(\vec{u})$ along with a return value v , and I is a sequence of operation labels representing the "internal" calls made during the invocation of m. The distinguished invocation outcome $\bot$ is associated to invocations that revert.

Interface. The *interface* Σ_{adr} is the set of non-read-only operation labels in the contract adr. We assume w.l.o.g. that the preconditions are satisfied for all the operations in Σ_{adr}, otherwise, the external invocation $\mathrm{adr}.m(\vec{u})$ reverts. Σ_{adr} is a superset of the set of action candidates of FlashSyn.

Smart Contract. A *smart contract* at an address adr is an LTS $C_{\mathrm{adr}} = (Q_{\mathrm{adr}}, \Sigma_{\mathrm{adr}}, \mathrm{q}_0, \delta_{\mathrm{adr}})$ over the interface Σ_{adr} where Q_{adr} is the set states and δ_{adr} is the transition relation.

Symbolic Actions Vector. We define the notion of a symbolic actions vector $\mathbf{S} = \ell_{\mathrm{adr}1} \ldots \ell_{\mathrm{adr}n}$ s.t. $\ell_{\mathrm{adr}i} \in \Sigma$ for $1 \leq i < n$ as the sequence of operation labels (possibly from different contracts) associated with the execution ρ, i.e., $\rho = \mathrm{q}_1, \ell_{\mathrm{adr}1}, \mathrm{q}_1 \ldots \ell_{\mathrm{adr}n}, \mathrm{q}_n$.

Balance. We define the balance of address adr in a blockchain state q as the mapping $\mathcal{B} : Q \times \mathbf{A} \implies \mathbf{V}$ that maps the pair $(\mathrm{q}, \mathrm{adr}) \in Q \times \mathbf{A}$ to the weighted sum of tokens the address adr holds at q, i.e., $\mathcal{B}(\mathrm{q}, \mathrm{adr}) = \sum_{\mathrm{t} \in \mathbf{T}} \mathbf{M}(\mathrm{q}, \mathrm{adr}, \mathrm{t}) \cdot \mathbf{P}(\mathbf{t})$, where $\mathbf{T}$ represents tokens hold by adr, $\mathbf{M}(\mathrm{q}, \mathrm{adr}, \mathrm{t})$ represents the amount of token t hold by adr at the blockchain state q, and $\mathbf{P}(\mathrm{q}, \mathbf{t})$ represents the price of token t at the blockchain state q.

Attack Vector. An *attack vector* by an adversary adr consists of a symbolic actions vector $\mathbf{S}$ where the symbolic arguments are replaced by concrete values (integer values) and $\mathbf{S}$ transforms a blockchain state q to another state q′ such that $\mathcal{B}(\mathrm{q}', \mathrm{adr}) - \mathcal{B}(\mathrm{q}, \mathrm{adr}) > 0$, i.e., the adversary adr generates profit when the sequence of actions $\mathbf{S}$ is executed with the concrete values.

Problem formulation. Given a specification φ (which contains vulnerable contract addresses or action candidates) and a blockchain state q, the objective is to find an attack vector consisting of a concretization of the symbolic actions vector $\mathbf{S} = \ell_{\mathrm{adr}1} \ldots \ell_{\mathrm{adr}n}$ s.t. $\ell_{\mathrm{adr}i} \in \Sigma \cap \varphi$ for $1 \leq i < n$, transforming the state q to a state q′, and that maximizes the profit of an adversary adr, $\mathcal{B}(\mathrm{q}', \mathrm{adr}) - \mathcal{B}(\mathrm{q}, \mathrm{adr})$.

5 FLASHSYN

5.1 Symbolic Actions Vectors Synthesis

In Algorithm 1, we give the overall synthesis procedure of FlashSyn. FlashSyn first collects initial data points to approximate the actions in **Act** (line 3) where FlashSyn uses the state q as a starting blockchain state. Then, using the sub-procedure Approximate FlashSyn generates the approximations ApproxAct of the actions in **Act** using the collected data points (line 5). FlashSyn uses the sub-procedure ActionsVectors to generate all possible symbolic actions vectors of length less than len (line 6). FlashSyn then iterates over the generated actions vectors and use some heuristics implemented in the sub-procedure IsFeasible to prune actions vectors (line 8). For instance, an actions vector containing two adjacent actions invoking the same method can be pruned to an actions vector where the two adjacent actions are merged. Afterwards, using the actions vector and approximated transition functions, the sub-procedure Construct constructs the optimization framework $\mathcal{P}$ for the actions vector (line 9). Then, FlashSyn uses the optimization sub-procedure Optimize (line 10) to find the optimal concrete values to pass as input parameters to the methods in the actions vector that satisfy the constraints of $\mathcal{P}$. FlashSyn then validate whether the attack vector generated by the optimizer indeed generates the profit. To do this FlashSyn uses the sub-procedure QueryOracle to execute the actions vector with the optimizer generated input parameters on the actual smart contracts on the blockchain. If the query is successful, i.e., the actual profit closely matches the profit found by the optimizer, FlashSyn adds the attack vector to the list of discovered attacks. Otherwise, FlashSyn considers the attack vector to be a counterexample, and uses it to generate new data points to refine the approximation in the subsequent iterations, within the sub-procedure CEGDC

(referenced in line 14 and introduced later in Section 5.4). FlashSyn repeats the process until the number of iterations reaches n (line 4).

Algorithm 1: Attack vectors synthesis procedure. Its inputs are actions **Act**, the maximum length len, a blockchain state q, and a threshold number of iterations n. Its outputs are attack vectors that yield positive profits.

```
1:  procedure Synthesize(Act, len, P, q, n)
2:      for each a ∈ Act
3:        datapoints[a] ← DataCollect(q, a);
4:      for each i ∈ [0; n]
5:        ApproxAct ← Approximate(Act, datapoints);
6:        wlist ← ActionsVectors(ApproxAct, len);
7:        for each p ∈ wlist
8:          if IsFeasible(p)
9:            P ← Construct(p, ApproxAct)
10:           (p*, profit) ← Optimize(p, P);
11:           if QueryOracle(q, p*, profit)
12:             answerlist.add(p*, profit);
13:           else
14:             datapoints := datapoints ∪ CEGDC(p*, q);
15:     return answerlist;
```

5.2 Pruning Symbolic Actions Vectors

The sub-procedure IsFeasible implements some heuristics to prune undesired symbolic actions vectors.

Heuristic 1: no duplicate adjacent actions. Two successive calls to the same method in a DeFi smart contract are usually equivalent to a single call with larger parameters. Thus, we discard actions vectors containing duplicate, successive actions.

Heuristic 2: limited usage of a single action. Using the observation that attack vectors do not contain repetitions, we fix a maximum number of calls to a single method an attack can contain and discard actions vectors that do not satisfy this criterion, e.g., an actions vector of length 4 cannot contain more than 2 calls to the same method.

Heuristic 3: necessary preconditions. Based on the observation that owning certain tokens is a necessary precondition for invoking some actions, FlashSyn prunes symbolic actions vectors that contain actions requiring tokens[6] not owned by the attacker. For example, in Harvest USDC example, invoking *withdraw* method requires users own some share tokens (fUSDC) beforehand. The only action candidate that mints fUSDC for users is *deposit*; thus, this heuristic mandates that *deposit* must be called before invoking *withdraw*. This heuristic establishes a necessary but not sufficient condition to ensure that synthesized attack vectors will not be reverted.

5.3 Optimization

Given a symbolic actions vector and their approximated transit functions, the sub-procedure Construct constructs an optimization framework to find optimal values for the parameters for the actions. Recall that given a blockchain state q and an address adr, the actions vector S transforms q to another state q′. The objective function in the optimization problem targets to increase the tokens values in the balance of the address adr, i.e., $y = \mathcal{B}(q', adr) - \mathcal{B}(q, adr)$. The optimization problem is accompanied by constraints on the symbolic values to be inferred. For instance, the balance of any token **t** for any address adr′ must always be non-negative, i.e., the adversary and the smart contracts cannot use more tokens than what they have in their balances, otherwise the transaction reverts. In the following, we give the definition of the optimization problem.

$$\mathcal{P}: \begin{cases} \max_{p_0,p_1,\ldots,p_n} \; y = \mathcal{B}(q', adr) - \mathcal{B}(q, adr) \\ \text{subject to: } \forall\ \mathbf{t} \in \mathbf{T}, adr' \in \mathbf{A}.\ \mathbf{M}(q', adr', \mathbf{t}) \geq 0 \end{cases}$$

[6]Note a token can be standard tokens (ERC20, BEP20), or any other forms of tokens such as debt tokens or share tokens.

5.4 Counterexample Guided Data Collection (CEGDC)

The optimization sub-procedure might explore parts of the states space not explored during the initial data points collection. This might challenge the accuracy of the approximations and result in mismatch between the estimated and the actual values. Thus, it is necessary to collect new data points based on the counterexamples that show the mismatch between the estimated and the actual values, to refine the approximations. Therefore, we propose counterexample guided data collection (CEGDC), inspired of counterexample guided abstraction refinement [14], to refine approximations when mismatches are identified.

We use **C** to denote the attack vector s.t. $q \xrightarrow{\mathbf{C}} q'$. q'_e and q'_a denote the estimated value for the state q′ found by the optimizer and the actual value obtained when executing **C** on the actual protocol on the blockchain, respectively. $\mathcal{P}_e(\mathbf{C}) = \mathcal{B}(q'_e, adr) - \mathcal{B}(q, adr)$ and $\mathcal{P}_a(\mathbf{C}) = \mathcal{B}(q'_a, adr) - \mathcal{B}(q, adr)$ denote the estimated profit and actual profit, respectively.

Definition 5.1. *A counterexample is an attack vector* **C** *whose estimated profit* $\mathcal{P}_e(\mathbf{C})$ *is different from its actual profit* $\mathcal{P}_a(\mathbf{C})$. *Formally,* $|\mathcal{P}_e(\mathbf{C}) - \mathcal{P}_a(\mathbf{C})| \geq \varepsilon \cdot (|\mathcal{P}_e(\mathbf{C})| + |\mathcal{P}_a(\mathbf{C})|)$, *where* ε *is a small constant representing accuracy tolerance.*

Algorithm 2: Counterexample guided data collection procedure. It takes a counterexample **C** and a state q, and returns datapoints. $k \in [n; 1]$ means that in the first iteration $k = n > 0$.

```
1:  procedure CEGDC(C, q)
2:      datapoints ← [ ];
3:      for each k ∈ [len(C); 1];
4:        q'_e ← Estimate(q, C, k);
5:        q'_a ← Execute(q, C, k);
6:        if IsAccurate(q'_e, q'_a)
7:          returns datapoints ;
8:        else
9:          (a, paras) ← C[k];
10:         datapoints[a] ← (q, paras, q'_a);
11:     return datapoints;
```

In Algorithm 2, we present the sub-procedure CEGDC for collecting new data points from a counterexample. CEGDC takes as inputs a counterexample **C** which is known to have an inaccurate profit estimation, and a blockchain state. The for loop on line 3 is used to locate approximation errors backward from the last action to the first action and collect new data points accordingly. In a loop iteration k, FlashSyn checks if the estimated methods of the action at the index k of **C** are accurate. First, FlashSyn computes the estimated state q'_e reached

by executing C until reaching the action indexed k (line 4) using the approximated transition functions. Second, FlashSyn fetches the actual state q'_a reached by executing C until reaching the action indexed k (line 5) on the actual smart contracts on the blockchain. Then, FlashSyn compares the estimated and actual execution results (line 6). If the estimation is accurate, this indicates that the transition functions of the action at the index k of C and its predecessors are accurate; so the procedure breaks the loop and returns the data points computed in the previous iterations (line 7). Otherwise, it indicates inaccurate transition functions of this action or/and its predecessors. Thus, we add a new data point associated with the action at the index k of C (lines 9 and 10) and proceed to the next iteration of the loop to explore the action predecessors.

6 IMPLEMENTATION

FlashSyn is implemented in Python. Figure 3 shows an overview of our implementation. The components *Runner*, *Synthesizer*, *Approximator*, and *Optimizer* implement the FlashSyn synthesis procedure presented in Algorithm 1. An optional component FlashFind is used to automatically identify action candidates.

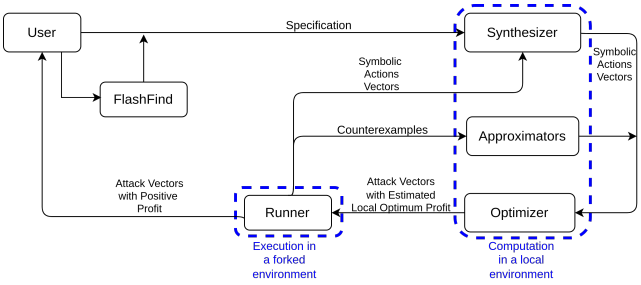

Figure 3: An overview of FlashSyn implementation

Approximator. *Approximator* approximates the transition functions using data points collected by *Runner*. The approximated transition functions are then given to *Optimizer* to construct the optimization framework. In *Approximator*, all transition functions of an action are approximated unless the transition functions are straightforward assignment/addition/subtraction (by simple inspection of smart contract codes), or the action is very common (such as Uniswap that has been widely studied [53, 67, 68]). FlashSyn implements two numerical methods using external libraries. FlashSyn-poly utilizes the sklearn [10, 52] library, and FlashSyn-inter employs the scipy [62] library. The choice of polynomial and interpolation methods is motivated by several considerations. First, FlashSyn requires fast evaluation of approximated functions, as thousands of evaluations are performed in the optimization process. Second, when provided with an input not seen before, the approximation method needs to yield a reasonable estimation based on the nearest points. Lastly, given that a typical FlashSyn process involves learning dozens of approximated formulas, the approximation process for one formula should not exceed a few seconds. Polynomial and interpolation methods are the two most popular approximation approaches that meet all of these criteria and there are off-the-shelf tools like sklearn and scipy that are easy to intergate in FlashSyn.

Optimizer. *Optimizer* automatically builds an optimization problem using the approximated transition functions returned by *Approximator* and the symbolic actions vectors enumerated by *Synthesizer* and performs global optimization on it. The obtained attack vector that yields a positive profit is then executed by *Runner* to confirm the accuracy of the estimation. If the estimation is inaccurate, the attack vector is treated as a counterexample and is used to collect new data points by *Runner* that are used by *Approximator* to refine the approximation. We built *Optimizer* on top of an off-the-self global optimizer **scipy.optimize.shgo** [21, 43, 63], which solves the simplicial homology global optimization algorithm to find the optimal parameters.[7] In FlashSyn-poly, we parallelized *Optimizer* component using up to 18 processes where *Optimizer* is run over multiple symbolic actions vectors in parallel.

Runner. *Runner* executes transactions on a forked blockchain. It performs both initial and counterexample based data collection, and validates results of *Optimizer*. We implemented *Runner* on top of Foundry [37], a toolkit written in Rust for smart contracts development that allow to interact with EVM based blockchains.

Synthesizer. *Synthesizer* first enumerates and prunes symbolic actions vectors using heuristics. Then, during counterexample-guided loops, it employs priority scoring to gradually drop actions vectors based on their scores. *Synthesizer* uses iterative synthesis. *Optimizer* can be configured with different hyperparameters to perform different strengths of parameter search. We designed 3 sets of hyperparameters which represent different strengths of parameter search. *Synthesizer* first conducts a weakest parameter search on all enumerated symbolic actions vectors using *Optimizer*. After *Runner* validates the results, *Synthesizer* ranks symbolic actions vectors and drops the ones with low priority scores. The actions vector with high priority score will be searched with higher strengths. An actions vector will also be dropped when its priority score does not increase between iterations. When all actions vectors are dropped, the whole synthesis procedure stops, and FlashSyn returns all the profitable attack vectors it found.

FlashFind. We also implemented a tool FlashFind as an optional component of FlashSyn to help users automatically identify action candidates from smart contract addresses to be used by FlashSyn synthesis procedure (see Supplementary Material Section 4 for the algorithm implemented by FlashFind). FlashFind first utilizes the Application Binary Interface (ABI) of the smart contracts to identify non-privileged non-read-only action candidates. For action candidates that require non-integer input arguments (e.g., address, bytes, or string), FlashFind uses the transaction history to identify possible values for these arguments. Next, FlashFind executes each action candidate to ensure that the action candidate is executable at the given block and collects storage accesses information. FlashFind then performs Read-After-Write (RAW) like dependency analysis on storage accesses to find intra-dependency relationships between functions of different contracts. Functions that behave independently are

[7] Note that we were not able to use local optimizers in **scipy.optimize** library [62] which require an initial guess of parameters. Under our settings, it is not feasible to find an initial guess for every symbolic attack, as each attack behaves differently.

Table 2: Benchmark of attacks used in the evaluation.

	Benchmark	#C+	LoC*	Vulnerability Type	Tx
ETH	bZx1	6	4964	pump&arbitrage	[22]
	Harvest_USDT	6	6446	manipulate oracle	[25]
	Harvest_USDC	6	4095	manipulate oracle	[26]
	Eminence	2	489	design flaw†	[24]
	ValueDeFi	8	7043	manipulate oracle	[28]
	Cheesebank	12	1246	manipulate oracle	[23]
	Warp	11	13139	manipulate oracle	[29]
	Yearn	5	2200	forced investment	[30]
	inverseFi	7	5734	manipulate oracle	[27]
BSC	bEarnFi	3	3007	asset mismatch	[5]
	AutoShark	6	8052	design flaw†	[4]
	ElevenFi	7	5613	design flaw†	[6]
	ApeRocket	7	1562	design flaw†	[3]
	WDoge	2	788	deflationary token	[8]
	Novo	4	7080	design flaw†	[7]
FTM	OneRing	14	5386	design flaw†	[39]
DVD	Puppet	2	742	manipulate oracle	[17]
	PuppetV2	1	161	manipulate oracle	[18]
Total Financial Loss in History				82.5 million USD	

+: #C denotes number of the victim protocol's contracts invoked in exploits.
∗: LoC denotes total number of lines of code in the contracts identified by #C, excluding closed-source contracts.
†: The logic designs of one or more functions in the victim contracts is flawed, with highly specific case-by-case vulnerabilities.

filtered out. Finally, FlashFind returns a list of action candidates which will be used by FlashSyn to synthesize attack vectors.

FlashFind uses TrueBlocks [56] and Blockchain Explorers (Etherscan [31], BscScan [9], and FtmScan [40]) to collect past transactions of the target contracts. FlashFind employs Phalcon [1] and Foundry [37] to extract function-level trace data from those transactions and perform analysis on storage accesses. Our evaluation shows that FlashFind is able to identify action candidates and helps FlashSyn discovers alternative attack vectors (see RQ4 in Section 7).

FlashSyn does not require prior knowledge of a vulnerable location or contract. Given a set of DeFi "Lego" user interface contracts, action candidates and their special parameters such as strings are given by the users or automatically extracted from transaction history using FlashFind. FlashSyn utilizes these action candidates to synthesize attack vectors and search for optimal numerical values. Note that these action candidates are not necessarily the ones that contain the vulnerability. Rather, they serve as user interfaces for interacting with the protocol. The vulnerability may reside in any contract invoked through nested calls originating from these action candidates. If FlashFind is not utilized, users can consult the protocol documentation to identify the appropriate user-interface contracts and functions (action candidates), as well as how to select special parameters for invoking these functions. Such information is essential for any user interacting with the protocol, and is generally available in the documentations of DeFi protocols.

7 EVALUATION

We aim to answer the following research questions:

RQ 1: How effective is FlashSyn in synthesizing flash loan attack vectors?

RQ 2: How well does the synthesis-via-approximation technique perform compared to precise baselines?

RQ 3: How much does counterexample-driven approximation refinement improve FlashSyn's results?

RQ 4: How effective is the combination of FlashFind and FlashSyn to synthesize attack vectors end-to-end?

Scope: FlashSyn focuses on flash loan attacks that generate positive profit by sequentially invoking functions within existing DeFi contracts. Security attacks that require exploiting other vulnerabilities such as re-entrance or conducting social engineering are outside the scope of FlashSyn and our evaluation. The goal of FlashSyn is to prove the existence and the exploitability of flash loan vulnerabilities. Consequently, activities such as getting and repaying the flash loan are not part of our synthesis task but are discussed in Supplementary Material Section 5.

Benchmarks: We investigated historical flash loan attacks that span from 02/14/2020 to 06/16/2022 across Ethereum, Binance Smart Chain (BSC), and Fantom (FTM) and attempted to reproduce each of them in our environment. In the end, we reproduced 16 attacks that are within our scope and collected them as our benchmark attacks. These attacks invoked 2-14 contracts of the victim protocol in the nested invocation tree per attack, consisting of a total of 489 to 13,139 lines of code, reflecting the multifaceted nature of the DeFi protocols exploited in real-world flash loan attacks. Also, protocols in our benchmark contain up to 15 action candidates from which FlashSyn needs to find an attack vector. Altogether, the 16 historical flash loan attacks in our benchmark have caused over 82.5 million US dollars in losses and include widely-known cases such as Harvest, bZx, and Eminence. Additionally, we include 2 fictional attacks from the Damn Vulnerable DeFi (DVD) challenges [16].

Ground Truth: For historical flash loan attacks, we forked the corresponding blockchain at one block prior to the attack transaction and replayed the attacker's attack vector as the ground truth. For DVD benchmarks, we select community solutions as ground truth. Note that in a flash loan attack, if the same attack vector is repeated multiple times, we remove the loop and only consider the first attack vector as the ground truth.

Precise Baseline: To demonstrate the effectiveness of synthesis-via-approximation techniques, we implemented a baseline synthesizer that works with manual summaries of smart contract actions. Specifically, we manually inspected all benchmarks whose relevant smart contracts that are all open-source and for each benchmark we allocated more than 4 manual analysis hours to extract the precise mathematical summaries. The baseline synthesizer then uses the manually extracted precise summaries to drive the synthesis.

Environment Setup: We assume that the flash loan providers are generally available, and we do not consider the borrow and the return as the part of the synthesis task. To facilitate FlashSyn experimentation, we manually annotated the prestates and poststates for each action. The details of this annotation process are described in Supplementary Material Section 3. Although this manual effort is required, it's worth noting that automation of this step is possible. Techniques such as dynamic taint analysis and forward symbolic execution can be employed to automatically identify which storage variables influence the

Table 3: Summary of FlashSyn results. AC denotes the number of action candidates. AP denotes the number of action candidates to approximate. GL and GP denote the length and the profit of the ground truth attack vector, respectively. IDP and TDP denote the initial and total number of collected of data points, respectively. Time denotes the time spent in seconds.

						FlashSyn-poly			FlashSyn-inter			Precise
Benchmark	**AC**	**AP**	**GL**	**GP**	**IDP**	**TDP**	**Profit**	**Time**	**TDP**	**Profit**	**Time**	**Profit**
bZx1	3	3	2	1194	5192	5849	2392	422	6373	2302†	441	cs
Harvest_USDT	4	4	4	338448	8000	9325	110139†	670	10289	86798†	7579	cs
Harvest_USDC	4	4	4	307416	8000	8912	59614†	677	10914	110051†	8349	cs
Eminence	4	4	5	1674278	8000	8780	1507174	1191	8104	/	/	1606965
ValueDeFi	6	6	6	8618002	12000	19975	8378194†	4691	15758	6428341†	11089	cx
CheeseBank	8	3	8	3270347	2679	2937	1946291†	4391	2715	1101547†	10942	2816762†
Warp	6	3	6	1693523	6000	6000	2773345†	1164	6000	/	/	2645640†
bEarnFi	2	2	4	18077	4000	4854	13770	470	4652	12329	688	13832
AutoShark	8	3	8	1381	2753	2753	1372†	5484	2753	/	/	cx
ElevenFi	5	2	5	129741	4000	4070	129658	409	4326	85811	898	cx
ApeRocket	7	3	6	1345	6000	6402	1333†	733	6235	1037†	3238	cs
Wdoge	5	1	5	78	2000	2001	75	272	2080	75	289	75
Novo	4	2	4	24857	4000	4164	20210	702	4031	23084	861	cx
OneRing	2	2	2	1534752	4000	4710	1814882	585	4218	1942188	367	cx
Puppet	3	3	2	89000	6000	6301	89000†	1203	6452	87266†	1238	89000†
PuppetV2	4	3	3	953100	4491	4836	747799†	2441	5061	362541†	2835	647894†
						Solved:16/18 **Avg. Time**: 1594			**Solved**:13/18 **Avg. Time**: 3754			

†: FlashSyn's results include at least one attack vector that differs from the ground truth.

change in token balances, thereby streamlining the annotation of prestates and poststates. The experiments are conducted on an Ubuntu 22.04 server, with an AMD Ryzen Threadripper 2990WX 32-Core Processor and 128 GB RAM.

Experiment Overview. To answer RQ 1, we apply FlashSyn to the 18 benchmarks with the same set of candidate actions in ground truths. For each candidate action, the prestates and poststates are annotated for FlashSyn to drive the approximated formula for this action. We set a timeout of 3 hours for FlashSyn-poly and 4 hours for FlashSyn-inter. FlashSyn does not know a priori whether a benchmark has an attack vector with a positive profit, and it does not set any bounds on the profit. It tries iteratively to synthesize an attack vector with a maximum profit. FlashSyn's refinement loop is guided by intermediate results and FlashSyn stops when it cannot improve the profit or the above timeouts are reached. To answer RQ 2, we replace *Approximator* component of FlashSyn with manually extracted precise mathematical summaries, and conduct the same experiment with 4 hours timeout. To answer RQ 3, we evaluate FlashSyn with different initial data points and with CEGDC enabled/disabled. To answer RQ 4, we first use FlashFind to identify candidate actions from given contract addresses which the hacker used in history, manually annotate them as in RQ 1, and then apply FlashSyn with this new set of candidate actions to synthesize attack vectors under the same setting as in RQ 1. The results for RQ 1+RQ 2, RQ 3, RQ 4 are summarized in Table 3, Table 4, and Table 5, respectively.

RQ1: Effectiveness of FlashSyn. Table 3 summarizes the results of the experiment. The first five columns of Table 3 list benchmark information including the number of actions to be approximated and the length of the ground truths. The four columns under FlashSyn-poly list data concerning the synthesis using polynomial approximations. The four columns under FlashSyn-inter list data concerning the synthesis using interpolation based approximation.

Our results in Table 3 show that FlashSyn can effectively synthesize flash loan attack vectors. FlashSyn-poly (resp., FlashSyn-inter) synthesizes profitable attack vectors for 16 (resp., 13) benchmarks with an average normalized profit (w.r.t. the ground truth profit) of 0.945 (resp., 0.641). In particular, for three benchmarks (*ApeRocket*, *ElevenFi*, and *AutoShark*) the profits found by FlashSyn-poly are within 99% of the profits in the original attacks vectors. Surprisingly in another three benchmarks (*bZx1*, *Warp*, and *OneRing*) the profits found by FlashSyn are bigger than the profits in the original attacks vectors. For instance, in the *Warp* benchmark the profit is roughly double the ground truth profit (see Supplementary Material Section 2 for Warp case study). On average, FlashSyn-poly is ×2 faster than FlashSyn-inter, because we used parallelism in FlashSyn-poly which is not possible for FlashSyn-inter.

For 10 benchmarks, FlashSyn successfully discovers new profitable symbolic actions vectors that are different from the ground truths. These vectors either exploit the same vulnerability but in a different order of actions, or represent arbitrage opportunities that were not exploited by the original attackers. For the remaining 6 benchmarks, FlashSyn discovers exactly the same symbolic actions vectors as the ground truths but with different parameters. Note that FlashSyn is not able to solve *Yearn* and *InverseFi* which are not shown in Table 3. These two benchmarks put high requirements on the precision of the approximation and small miss-approximation errors caused FlashSyn to miss finding attack vectors and accurate parameters.

Table 4: FlashSyn results summary under different settings. n+x: n denotes the settings of initial number of data points and +x denotes whether FlashSyn uses counterexample-driven loops.

	FlashSyn-poly								FlashSyn-inter							
	200	**200+x**	**500**	**500+x**	**1000**	**1000+x**	**2000**	**2000+x**	**200**	**200+x**	**500**	**500+x**	**1000**	**1000+x**	**2000**	**2000+x**
Avg. Time (s)	632	893	1120	1747	842	1397	982	1594	2601	3509	3180	3917	3022	3845	3200	3754
Avg. Data Points	584	1042	1432	2376	2795	3571	5445	6367	584	1338	1432	2450	2795	3656	5445	6248
Avg. Norm. Profit	0.793	0.829	0.846	0.922	0.762	0.786	0.717	0.945	0.539	0.555	0.630	0.634	0.535	0.580	0.594	0.641
Benchmarks Solved	15	15	15	16	15	15	15	16	13	13	14	14	13	13	13	13

To evaluate the efficacy of the pruning heuristics introduced in Section 5.2, we conduct experiments comparing the search space sizes when using FlashSyn with and without the application of some of the heuristics. Our results indicate that Heuristic 1 leads to an average reduction of 57% in the search space size. Subsequently, Heuristic 2 further reduces the remaining search space by an additional 34%, and Heuristic 3 contributes an additional reduction of 65% to the remaining search space.

To compare FlashSyn with existing static analyzers, we manually select contracts containing vulnerabilities in the benchmarks and apply the popular smart contracts static analyzer Slither [32] to them. In the experiments, we identify contracts that contain the root cause of the vulnerabilities as the target contracts for Slither to analyze. Note that in practice, identifying target contracts for Slither is much harder than that for FlashFind. For FlashFind, the target contracts are simply user-interface contracts. In contrast, identifying the contract with the actual vulnerability, such as a contract invoked in a deeply nested call chain, can be tedious. Slither fails to detect vulnerabilities for all 18 benchmarks, among them Slither fails to parse 2 benchmarks (Novo and Yearn). The possible reasons include: (i) Slither's inability to reason across multi-contract interactions, common in flash loan attacks; and (ii) its lack of context awareness, such as not detecting Uniswap [61] when used as an oracle.

RQ2: Comparison with Precise Baseline. The last column of Table 3 lists data when the approximation component of FlashSyn is replaced with precise mathematical summaries for actions. Note that 4 benchmarks are partially closed-source (**cs**), and 5 benchmarks are too complicated (**cx**), thus we are not able to extract mathematical precise summaries for them. For others, we list the profit generated using the manually extracted mathematical expressions in the synthesizer and optimizer.

Our results in Table 3 show that the synthesis-via-approximation approach performs well compared to precise baselines. For the 9 cases that the precise baseline failed due to either close source (**cs**) or complicated contract logics (**cx**), FlashSyn found attack vectors that generate positive profits. On average, for the 7 cases that the precise baseline succeeds, the best profit from FlashSyn is 0.97 of the profit returned by the precise baseline. In particular, for *Warp* and *PuppetV2* FlashSyn synthesizes an attack vector with a profit higher than that obtained by the precise approach. This is because the approximations used in FlashSyn are simpler than the mathematical summaries used in precise baseline. This enables the optimizer to converge faster and find better parameter values within the fixed time budget.

RQ3: Counterexample-Driven Approximation Refinement. Table 4 summarizes the evaluation of FlashSyn under different settings. In particular, we evaluated FlashSyn with 200, 500, 1000, and 2000 initial data points threshold per action to be approximated without and with counterexample loop. The **Avg.** rows in Table 4 are calculated based on the 16 benchmarks excluding *Yearn* and *InverseFi*. The **Avg. Norm. Profit** is calculated as the average of normalized profits, i.e., profit / ground truth profit.

For FlashSyn-poly, the results in Table 4 show that only with counterexample loop we are able to solve the 16 benchmarks (Columns **500+x** and **2000+x**). Also, the maximum average of normalized profits is achieved with counterexample loop (Column **2000+x**) which improved from 0.717 (Column **2000**) without counterexample loop to 0.945 with counterexample loop. For FlashSyn-inter, the maximum average of normalized profits is also achieved with counterexample loop (Column **2000+x**) which improved from 0.594 (Column **2000**) without counterexample loop to 0.641 with counterexample loop.

RQ4: Effectiveness of FlashFind. In this experiment, we evaluate the combination of FlashFind and FlashSyn on the 14 benchmarks that FlashSyn was able to synthesize a profitable attack vector in Table 3 excluding the two fictional DVD benchmarks.[8]. In particular, only contract addresses are provided to FlashFind and FlashFind identifies candidate actions for FlashSyn to synthesize attack vectors with the 2000 initial data points threshold per action configuration. Table 5 presents the results.

As Table 5 shows, FlashFind successfully identifies a reasonable number of action candidates for 11 out of the 14 benchmarks from given contract addresses. Among them, FlashFind identifies additional candidate actions for 7 benchmarks. For instance, FlashFind identifies 6 additional candidate actions for *OneRing*. The remaining 3 benchmarks contains action candidates whose arguments are non-primitive types, and FlashFind identifies an excessive and impractical number of choices from transaction history.[9]

Even with the extra candidate actions FlashSyn was able to synthesize profitable attack vectors for all 11 benchmarks. Surprisingly in 6 benchmarks, FlashSyn finds attack vectors that contains new action candidates from FlashFind that are not in the ground truth. There are two possibilities: First, the new action candidates identified by FlashFind are functionally similar to one action in ground truths (e.g, *withdraw* and *withdrawSafe* for *OneRing*). Replacing old actions with new ones gives new attack vectors. Second, the new action candidates represent another way of draining assets which the attacker failed to identify. For example, in the *Warp* benchmark, the attacker only invoked *borrowSC(USDC, v)* and *borrowSC(DAI, v)* to drain USDC

[8]DVD benchmarks do not have historical transactions that FlashFind can use.
[9]In such cases, we believe experienced security analysts could manually identify special parameters and further reduce the number of parameter choices.

Table 5: Summary of the evaluation results of combining FlashSyn with FlashFind. AC is the number of action candidates. AP is the number of action candidates to approximate. GL and GP are the length and the profit of the ground truth attack vector, respectively. IDP and TDP are the initial and total number of collected of data points, respectively. Time is measured in seconds.

			FlashFind + FlashSyn-poly						FlashSyn-poly					
Benchmark	**GL**	**GP**	**AC**	**AP**	**IDP**	**TDP**	**Profit**	**Time**	**AC**	**AP**	**IDP**	**TDP**	**Profit**	**Time**
bZx1	2	1194	3	3	5192	5849	2392	422	3	3	5192	5849	2392	422
Harvest_USDT	4	338448	15	15	30000	34052	85593‡	5514	4	4	8000	9325	110139†	670
Harvest_USDC	4	307416	15	15	30000	51726	33645‡	3630	4	4	8000	8912	59614†	677
Eminence	4	1674278	4	4	8000	8780	1507174	1191	4	4	8000	8780	1507174	1191
ValueDeFi	6	8618002	6	6	12000	19975	8378194†	4691	6	6	12000	19975	8378194†	4691
Warp	6	1693523	8	5	7772	7772	2776351‡	3129	6	3	6000	6000	2776351†	1164
bEarnFi	4	18077	2	2	4000	4854	13770	470	2	2	4000	4854	13770	470
ApeRocket	6	1345	11	5	10000	10706	1179‡	3064	7	3	6000	6402	1333†	733
Wdoge	5	78	7	2	4000	4107	75‡	769	5	1	2000	2001	75	272
Novo	4	24857	6	2	4000	4172	15183	791	4	2	4000	4164	20210	702
OneRing	2	1534752	8	8	16000	16614	1814877‡	1104	2	2	4000	4710	1814882	585

†: FlashSyn's results include at least one attack vector that differs from the ground truth.
‡: FlashSyn's results include at least one attack vector that contains an action not present in the ground truth.

and DAI [29, 54], however, FlashFind identifies *borrowSC(USDT, v)* as another candidate action, which could have been used to drain USDT as well in the same transaction.

Impact. One author of this paper has collaborated with a leading auditing company for applying FlashSyn. We discovered one zero-day flashloan vulnerability in a protocol under audit.

Threats to Validity: The *internal* threat to validity mainly lies in human mistakes in the study. Specifically, we may understand results of FlashSyn incorrectly, or make mistakes in the implementation of FlashSyn. All authors have extensive smart contract security analysis experience and software engineering expertise in general. To further reduce this threat, we manually check the balance changes for the best results given by FlashSyn in each benchmark. We verify that with the help of on-chain exchanges, these attack vectors can generate a post-balance strictly larger than initial capital (see Supplementary Material Section 5). The *external* threat to validity mainly lies in the subjects used in our study. The flash loan attacks we study might not be representative. We mitigate this risk by using diverse and reputable data sources, including academic papers [12, 53] and an industrial database [58].

Limitations: Like most synthesis tools, FlashSyn faces scalability challenges. The search space grows exponentially with the number of actions and attack vector length. A practical approach is to assess protocols on a module by module basis. By focusing only on inter-dependent actions within, we can maintain both the number of actions and the attack vector length at manageable levels, thereby mitigating the scaling issue.

8 RELATED WORK

Parametric optimization: For some flash loan attack cases, researchers [12, 53] manually extracted math formulas of function candidates, manually defined related parameter constraints, and used an off-the-shelf optimizer to search for parameters which yield the best profit. However, this technique requires significant manual efforts and expert knowledge of the underlying DeFi protocols. Consequently, it becomes impractical for checking a large number of potential attack vectors. Note that our benchmark set contains significantly more flash loan attacks than prior work [12, 53], i.e., 18 versus 2 in [53] and 9 in [12].

Static Analysis: Slither [32], Securify [60], Zeus [44], Park [70] and SmartCheck [59] apply static analysis techniques to verify smart contracts. There are also several works that use symbolic execution [45] to explore the program states of a smart contract, looking for an execution path that violates a user-defined invariant, e.g., Mythril [15], Oyente [47], FairCon [46], ETHBMC [38], SmartCopy [33], and Manticore [50]. These techniques tend to operate with one contract at a time and therefore cannot handle flash loan attacks that involve multiple contracts. These techniques also may suffer from the complicated logics of the DeFi contracts, and cause path explosion. FlashSyn uses its novel synthesis-via-approximation techniques to avoid these issues.

Fuzzing: ContractFuzzer [42], sFuzz [51], ContraMaster [64], SMARTIAN [13] and ItyFuzz [57] introduce novel fuzzing techniques to discover vulnerabilities in smart contracts. However, these techniques either only work on one contract or focus on specific vulnerabilities like re-entrancy. Moreover, flash loan attack vectors can contain up to 8 actions and 7 integer parameters, which is unlikely to be found by random fuzzing.

9 CONCLUSION

We have proposed an automated synthesis framework based on numerical approximation to generate the flash loan attack vectors on DeFi protocols. Our results of FlashSyn show that the proposed framework is practical and FlashSyn can automatically synthesize attack vectors for real world attacks. Our results also highlight the effectiveness of the synthesis-via-approximation approach. The approach helps FlashSyn to overcome the challenges posed by complicated functions in DeFi protocols and our results demonstrate that using approximations of these functions is sufficient to drive the synthesis process. FlashSyn has been adopted by a top smart contract auditing company to detect flash loan vulnerabilities. The paper also points out a new promising direction for solving trace synthesis problems when facing complicated functions.

REFERENCES

[1] BlockSec. 2023. Phalcon: Powerful Transaction Explorer Designed for DeFi community. https://phalcon.xyz/ Accessed: 2023-03-27.

[2] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469.

[3] BscScan. 2021. ApeRocket Attack Transaction. https://bscscan.com/tx/0x701a308fba23f9b328d2cdb6c7b245f6c3063a510e0d5bc21d2477c9084f93e0.

[4] BscScan. 2021. AutoShark Attack Transaction. https://bscscan.com/tx/0xfbe65ad3eed6b28d59bf6043debf1166d3420d214020ef54f12d2e0583a66f13.

[5] BscScan. 2021. bEarnFi Attack Transaction. https://bscscan.com/tx/0x603b2bbe2a7d0877b22531735ff686a7caad866f6c0435c37b7b49e4bfd9a36c.

[6] BscScan. 2021. ElevenFi Attack Transaction. https://bscscan.com/tx/0x16c87d9c4eb3bc6c4e5fbba789f72e8bbfc81b3403089294a81f31b91088fc2f.

[7] BscScan. 2022. Novo Attack Transaction. https://bscscan.com/tx/0xc346adf14e5082e6df5aeae650f3d7f606d7e08247c2b856510766b4dfcdc57f.

[8] BscScan. 2022. WDoge Attack Transaction. https://bscscan.com/tx/0x4f2005e3815c15d1a9abd8588dd1464769a00414a6b7adcbfd75a5331d378e1d.

[9] BscScan. 2023. The BNB Smart Chain Explorer. https://bscscan.com/

[10] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

[11] Vitalik Buterin. 2014. Ethereum: A next-generation smart contract and decentralized application platform. https://ethereum.org/en/whitepaper/.

[12] Yixin Cao, Chuanwei Zou, and Xianfeng Cheng. 2021. Flashot: a snapshot of flash loan attack on DeFi ecosystem. *arXiv preprint arXiv:2102.00626* (2021).

[13] 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)*. IEEE, 227–239.

[14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*. Springer, 154–169.

[15] ConsenSys. 2022. Mythril. https://github.com/ConsenSys/mythril. Accessed: 2022-06-06.

[16] Damn Vulnerable DeFi. 2023. https://www.damnvulnerabledefi.xyz/. Accessed: 2023-01-31.

[17] Damn Vulnerable DeFi. 2023. Challenge #8 - Puppet. https://www.damnvulnerabledefi.xyz/challenges/8.html. Accessed: 2023-01-31.

[18] Damn Vulnerable DeFi. 2023. Challenge #9 - Puppet v2. https://www.damnvulnerabledefi.xyz/challenges/9.html. Accessed: 2023-01-31.

[19] DefiLlama. 2023. DefiLlama. https://defillama.com/. Accessed: 2023-04-01.

[20] Michael Egorov. 2019. StableSwap-efficient mechanism for Stablecoin liquidity. *Retrieved Feb* 24 (2019), 2021.

[21] Stefan C Endres, Carl Sandrock, and Walter W Focke. 2018. A simplicial homology algorithm for Lipschitz optimisation. *Journal of Global Optimization* 72, 2 (2018), 181–217.

[22] Etherscan. 2020. bZx1 Attack Transaction. https://etherscan.io/tx/0xb5c8bd9430b6cc87a0e2fe110ece6bf527fa4f170a4bc8cd032f768fc5219838.

[23] Etherscan. 2020. Cheesebank Attack Transaction. https://etherscan.io/tx/0x600a869aa3a259158310a233b815ff67ca41eab8961a49918c2031297a02f1cc.

[24] Etherscan. 2020. Eminence Attack Transaction. https://etherscan.io/tx/0x3503253131644dd9f52802d071de74e456570374d586ddd640159cf6fb9b8ad8.

[25] Etherscan. 2020. Harvest_USDC Attack Transaction. https://etherscan.io/tx/0x35f8d2f572fceaac9288e5d462117850ef2694786992a8c3f6d02612277b0877.

[26] Etherscan. 2020. Harvest_USDT Attack Transaction. https://etherscan.io/tx/0x0fc6d2ca064fc841bc9b1c1fad1fbb97bcea5c9a1b2b66ef837f1227e06519a6.

[27] Etherscan. 2020. inverseFi Attack Transaction. https://etherscan.io/tx/0x958236266991bc3fe3b77feaacea120f172c0708ad01c7a715b255f218f9313c.

[28] Etherscan. 2020. ValueDeFi Attack Transaction. https://etherscan.io/tx/0x46a03488247425f845e444b9c10b52ba3c14927c687d38287c0faddc7471150a.

[29] Etherscan. 2020. Warp Attack Transaction. https://etherscan.io/tx/0x8bb8dc5c7c830bac85fa48acad2505e9300a91c3ff239c9517d0cae33b595090.

[30] Etherscan. 2021. Yearn Attack Transaction. https://etherscan.io/tx/0xf6022012b73770e7e2177129e648980a82aab555f9ac88b8a9cda3ec44b30779.

[31] Etherscan. 2023. The Ethereum Blockchain Explorer. https://etherscan.io/

[32] 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, 8–15.

[33] Yu Feng, Emina Torlak, and Rastislav Bodik. 2019. Precise attack synthesis for smart contracts. *arXiv preprint arXiv:1902.06067* (2019).

[34] Anonymized for double blind. 2023. Anonymized for double blind.

[35] Ethereum Foundation. 2023. Solidity Programming Language. https://docs.soliditylang.org/en/v0.8.14/. Accessed: 2023-03-24.

[36] Ethereum Foundation. 2023. Vyper Programming Language. https://vyper.readthedocs.io/en/stable/. Accessed: 2023-02-24.

[37] Foundry Contributors. 2023. Foundry. https://github.com/foundry-rs/foundry/. Accessed: 2023-01-31.

[38] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. {ETHBMC}: A Bounded Model Checker for Smart Contracts. In *29th USENIX Security Symposium (USENIX Security 20)*. 2757–2774.

[39] FtmScan. 2022. OneRing Attack Transaction. https://ftmscan.com/tx/0xca8dd33850e29cf138c8382e17a19e77d7331b57c7a8451648788bbb26a70145.

[40] FTMScan. 2023. The Fantom Blockchain Explorer. https://ftmscan.com/

[41] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 557–560.

[42] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 259–269.

[43] Stephen Joe and Frances Y Kuo. 2008. Constructing Sobol sequences with better two-dimensional projections. *SIAM Journal on Scientific Computing* 30, 5 (2008), 2635–2654.

[44] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[45] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[46] Ye Liu, Yi Li, Shang-Wei Lin, and Rong Zhao. 2020. Towards automated verification of smart contract fairness. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 666–677.

[47] 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*. 254–269.

[48] Jack McKay. 2022. DeFi-ing Cyber Attacks. (2022).

[49] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. 2021. *Introduction to linear regression analysis*. John Wiley & Sons.

[50] 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)*. IEEE, 1186–1189.

[51] 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*. 778–788.

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[53] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. 2021. Attacking the defi ecosystem with flash loans for fun and profit. In *International Conference on Financial Cryptography and Data Security*. Springer, 3–32.

[54] Rekt. 2020. WARP FINANCE - REKT. https://rekt.news/warp-finance-rekt/.

[55] Olivier Rukundo and Hanqiang Cao. 2012. Nearest neighbor value interpolation. *arXiv preprint arXiv:1211.1768* (2012).

[56] Thomas Jay Rush. 2023. TrueBlocks. https://trueblocks.io/.

[57] 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*. 322–333.

[58] SlowMist. 2023. The 10 most common attacks. https://hacked.slowmist.io/en/statistics/?c=all&d=all. Accessed: 2023-02-01.

[59] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 9–16.

[60] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.

[61] Uniswap Labs. 2023. Uniswap Protocol. https://uniswap.org/. Accessed: 2023-11-08.

[62] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa,

Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[63] DJ Wales. 2015. Perspective: Insight into reaction coordinates and dynamics from the potential energy landscape. *The Journal of chemical physics* 142, 13 (2015), 130901.

[64] Haijun Wang, Ye Liu, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. 2020. Oracle-supported dynamic exploit generation for smart contracts. *IEEE Transactions on Dependable and Secure Computing* (2020).

[65] Gavin Wood. 2012. Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.github.io/yellowpaper/paper.pdf.

[66] Karl Wüst and Arthur Gervais. 2018. Do you need a blockchain?. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE, 45–54.

[67] Pengcheng Xia, Haoyu Wang, Bingyu Gao, Weihang Su, Zhou Yu, Xiapu Luo, Chao Zhang, Xusheng Xiao, and Guoai Xu. 2021. Trade or Trick? Detecting and Characterizing Scam Tokens on Uniswap Decentralized Exchange. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 5, 3 (2021), 1–26.

[68] Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. 2021. Sok: Decentralized exchanges (dex) with automated market maker (AMM) protocols. *arXiv preprint arXiv:2103.12732* (2021).

[69] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying Exploitable Bugs in Smart Contracts. ICSE.

[70] Peilin Zheng, Zibin Zheng, and Xiapu Luo. 2022. Park: accelerating smart contract vulnerability detection via parallel-fork symbolic execution. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 740–751.