Approve Once, Regret Forever: On the Exploitation of Ethereum's Approve-TransferFrom Ecosystem

Nicola Ruaro[†], Fabio Gritti[†], Dongyu Meng[†], Robert McLaughlin[†], Ilya Grishchenko[‡], Christopher Kruegel[†], and Giovanni Vigna[†]

[†]University of California, Santa Barbara, [‡]University of Toronto
{ruaronicola,degrigis,dmeng,robert349,chris,vigna}@cs.ucsb.edu, ilya.grishchenko@utoronto.ca

Abstract

Smart contracts are immutable programs hosted on the blockchain that power decentralized applications. With the growth of decentralized finance (DeFi), many services interact with contracts that must be trusted to manage digital assets. To this end, several Ethereum standards (e.g., ERC20, ERC721) introduced an approval mechanism that allows decentralized applications to trade digital assets (or "tokens") on behalf of others. After receiving an approval, the (approved) application can invoke the token's `transferFrom` function to trade the approved tokens. Unfortunately, approved applications often contain vulnerabilities. If an attacker maliciously controls the parameters of a `transferFrom` call, they can steal not only the application's assets but also the assets of any user who previously approved the application. We refer to this widespread issue as **A**pproved **C**ontrollable **T**ransferFrom (ACT), which has already led to losses exceeding 65 million USD.

We present OSPREY, an end-to-end system that detects ACT vulnerabilities and automatically generates proof-of-concept attacks. Our evaluation across the entire Ethereum ecosystem identified 32,582 potentially vulnerable contracts, with 410 confirmed exploitable at the time of writing. Our findings reveal previously unknown attack vectors threatening digital assets worth over 3.4 million USD.

1 Introduction

Ethereum [18] is a global, decentralized blockchain that allows both peer-to-peer transfers of Ether (its native currency) and on-demand execution of decentralized programs (smart contracts) on the Ethereum Virtual Machine (EVM), laying the foundations for complex decentralized applications. Ethereum hosts a variety of such applications, including games [48], loan providers [2], artwork auctions [44], and financial derivatives [3]. Users can interact with applications in a trustless and transparent way thanks to Ethereum's public ledger: a smart contract's state is entirely public, its code is immutable, and its execution rules are clearly defined.

Ethereum is the largest programmable blockchain by market capitalization – $398 billion at the time of writing [11]. This exceptional growth has been driven by the excitement around *decentralized finance* (DeFi): a rich ecosystem of digital currencies and financial services such as lending platforms, decentralized marketplaces, and governance systems. DeFi services often incorporate several interoperating smart contracts, collectively referred to as "DeFi protocols," and hundreds of digital currencies that operate independently from Ether and can be tailored to the protocol's needs. Since transacting with diverse assets could become cumbersome and inefficient, Ethereum standards such as ERC20 [16] provide a standardized way to create digital currencies (i.e., "tokens").

Ethereum users interact with a DeFi protocol on-demand by signing and broadcasting a transaction. An Ethereum user might want to buy a certain amount of tokens, swap them for different tokens, or lend them to other users for a small fee. To accomplish this, DeFi protocols commonly make use of coordinated interactions between several smart contracts. For example, a user may consult an on-chain oracle for exchange rates, swap ERC20 tokens on a decentralized exchange, and then lend them to others via a lending protocol.

The composable nature of DeFi protocols allows for the creation of sophisticated services and financial transactions. However, these composable interactions also give rise to greatly increased logical complexity and a much broader attack surface. DefiLlama [12] estimates that DeFi protocols lost more than 5 billion USD due to attacks since January 2021, making smart contract vulnerability hunting a hot topic in both industry and academia [6, 19, 22, 43].

One vulnerability that has lately emerged lies in the access control mechanisms of Ethereum's token ecosystem. In Ethereum, access control traditionally relied on the identity of the transaction's sender (the signer). If the transaction sender is the owner of the tokens, then they are allowed to transfer their tokens – otherwise, access is denied. However, this type of access control quickly became insufficient to support the needs of protocol developers. For example, a lending protocol must be able to move a user's tokens when another user requests to borrow them, on-demand, without interacting with the user(s) who provided the tokens in the first place. Of

course, this cannot work with traditional (transaction sender-based) access control, since the owner is not sending the transaction. One option is to require that the lender simply transfers (upfront) all their assets to the lending protocol, but this would subject the lender to increased risk: if the lending protocol is compromised, the protocol's users might lose all their funds. To bridge this gap, the ERC20 standard, and other standards such as ERC721 [15] and ERC1155 [14], introduced *approvals* as an alternative access control mechanism [10]. In brief, instead of transferring assets directly to the lending protocol, the tokens' owner (i.e., the "approver") can invoke the token's `approve` function to allow a different user or smart contract (i.e., the "spender") to trade an allowance (amount) of tokens on their behalf. The spender can then directly invoke the token's `transferFrom` function in order to transfer the tokens from the approver's account to another account. In this way, if the lending protocol is compromised, its users can revoke their approval to rescue their tokens. Critically, revoking approval does not require any interaction with the lending protocol, so this action can be performed even if the protocol itself is no longer usable.

Approvals come with inherent risk, as approved spenders may transfer tokens to unintended recipients – perhaps due to a flaw in the smart contract [37, 38, 40], or a malicious trusted administrator [24]. This greatly amplifies an attack's impact: An attacker can steal not only the assets directly owned by the victim contract but also the assets of any user who previously approved the victim contract.

To maliciously transfer approved user assets, an attacker must devise a way to interact with the victim contract and make that contract execute an outbound *function call* (to another smart contract) with attacker-controllable arguments. If the attacker can achieve this, they can force the victim contract to call the `transferFrom` function of any arbitrary token, and thus, request the transfer of tokens through the victim contract (spender) on behalf of a user of choice (approver) to the attacker's wallet. We will refer to this class of vulnerabilities as **A**pproved **C**ontrollable **T**ransferFrom (ACT) vulnerabilities.

Detecting logic vulnerabilities such as ACT is known to be hard [59]. To this day, ACT vulnerabilities are routinely exploited, and we estimate that over 65 million USD have been stolen since 2020 due to this class of vulnerabilities alone [46]. Previous works [23, 60] have proposed generic systems that use taint analysis to detect "fully controllable" function calls – that is, function calls where every argument is attacker-controllable. However, a fully controllable call is neither a necessary nor a sufficient condition for an ACT vulnerability. First, a partially controllable call can still lead to an ACT vulnerability. Unlike generic arbitrary call vulnerabilities, ACT vulnerabilities often emerge from partially controllable calls where the attacker has limited but sufficient influence over critical execution parameters. Second, an ACT vulnerability *only* occurs if the vulnerable contract is approved to spend other users' funds. Thus, detection requires a deep understanding of the DeFi context – that is, the approve-transferFrom mechanism and the on-chain approval states.

In this paper, we describe the design and implementation of a novel end-to-end system (OSPREY) to detect ACT vulnerabilities. To address the limitations of existing systems, our approach uses a combination of static analysis, symbolic execution, and concrete execution to provide (1) a deep understanding and proper handling of on-chain token approvals, (2) an analysis of the constraints at the call site to determine the degree of control that the attacker has over the call parameters, and (3) the automated generation of (semantically correct) interactions between the target contract and external contracts. First, OSPREY analyzes on-chain data to find all contracts that have been granted approval to transfer another account's tokens. Second, OSPREY studies the internal state of the involved tokens to identify (or simulate) potential victim users. Third, OSPREY relies on symbolic execution to model the external interactions of the approved contract. OSPREY's analysis of these interactions is specialized to detect and exploit ACT vulnerabilities. In certain cases, reaching the vulnerable call requires not only precise input but also successful interactions with (user-controlled) external "accessory" contracts. OSPREY identifies these cases and automatically generates accessory contracts that shape the execution environment and enable the deep execution required to reach the vulnerable call. Upon detection, OSPREY raises warnings (along with an associated confidence level) and, if possible, automatically crafts an end-to-end exploit against the victim contract.

Although approvals are a common mechanism across several token standards, the following sections focus on the ERC20 standard. We refer the interested readers to Section 5.6 for an in-depth discussion of other token standards. In this paper, we make the following contributions:

- We present a novel approach leveraging static analysis, symbolic execution, and concrete execution to find ACT vulnerabilities and automatically craft proof-of-concept attacks. We implement our approach in OSPREY.
- We compile a labeled dataset of historical attacks that exploited ACT vulnerabilities. Our dataset includes information such as the exploit transactions, the vulnerable contracts, and the vulnerable functions.
- We evaluate OSPREY against (1) our dataset of known attacks and (2) 424,676 Ethereum contracts that have historically received ERC20 approvals. We identify 32,582 previously-unreported contracts that are potentially vulnerable to ACT vulnerabilities. OSPREY automatically crafts proof-of-concept attacks for 410 of them and estimates the financial damage to be over 3.4 million USD.
- We demonstrate that our conceptual approach is applicable to arbitrary token standards with approvals. Our evaluation against ERC721 uncovered 18 previously unreported vulnerable contracts with confirmed exploits.

2 Background

Smart Contracts. Ethereum smart contracts are programs typically written in a high-level language – such as Solidity [49] – and then compiled to EVM bytecode. Smart contracts can be deployed either (directly) by a blockchain user or (indirectly) during the execution of another smart contract.

Once deployed on the blockchain, smart contracts can be executed on demand. More precisely, a smart contract executes when a blockchain user requests to execute one of the contract's (public) functions by sending a signed *external* transaction – optionally including input data and some value of Ether. The EVM then processes the transaction and sequentially executes the contract's EVM instructions. During execution, smart contracts make use of two types of temporary storage – addressable memory and a stack – and one type of persistent storage, which is simply known as "storage." Storage persists across transactions and allows, for example, to keep track of a user's balance over time.

A smart contract can interact with other smart contracts by invoking one of their functions (which likewise optionally includes input data and a quantity of Ether to transfer) – an interaction also known as an *internal* transaction.

Regardless of the type of interaction (i.e., external or internal), public functions are the only entry point for the contract's functionality. By convention, each public function is associated with a four-byte-long function selector. The transaction sender can provide the function selector – and the function arguments – as part of the input data (i.e., calldata) to the smart contract. A dispatcher routine in the target contract then selects the function to be executed based on the function selector. If matched, the dispatcher executes the function, otherwise it executes a fallback function.

In the EVM, three key instructions enable contract-to-contract function calls (and Ether transfers): `CALL`, `STATICCALL`, and `DELEGATECALL`. In this paper, we study controllable `CALL` instructions as the primary interaction mechanism in ACT vulnerabilities, but we also handle `DELEGATECALL` instructions when the smart contract execution is delegated.

Controllable CALLs. In certain instances, a user may be able to control the contents of an internal transaction's calldata, either in part or in whole. In many cases, this is intentional and necessary to support the regular functionality of smart contracts. For example, when placing a "buy" order on a decentralized exchange, users specify a token and desired quantity. The exchange contract then calls the token's transfer function with the user-supplied quantity parameter.

Unfortunately, the ability of a (malicious) user to control certain call parameters can lead to security vulnerabilities when not adequately restricted. For example, a distributed exchange must only transfer to the user an amount of tokens commensurate to the payment made. Allowing arbitrary, unchecked user data in the "quantity" field of the "transfer" function could lead to severe security issues – allowing an attacker to steal an arbitrary amount of tokens. For this reason, developers must take great care when permitting user-controllable content in their outgoing calls.

```
1  contract ERC20
2      map(addr => uint) balance;
3      map(addr => map(addr => uint)) allowance;
4
5      function transfer(addr to, uint val)
6          require (balance[msg.sender] >= val);
7          balance[msg.sender] -= val;
8          balance[to] += val;
9          emit Transfer(from, to, val);
10
11     function approve(addr spender, uint val)
12         allowance[msg.sender][spender] = val;
13         emit Approval(msg.sender, spender, val);
14
15     function transferFrom(addr from, addr to, uint val)
16         require (allowance[from][msg.sender] >= val);
17         require (balance[from] >= val);
18         allowance[from][msg.sender] -= val;
19         balance[from] -= val;
20         balance[to] += val;
21         emit Transfer(from, to, val);
```

Figure 1: Simplified Solidity code of an ERC20 contract.

Avoiding user-controllable calls is critical because access control regularly relies on the identity of the `msg.sender`. This value represents the entity that initiated the contract call – i.e., the transaction sender for external transactions or (more interestingly) the calling contract for internal transactions. If an attacker can control a contract-originated internal call, they may impersonate the calling contract, bypassing identity-based access control checks. For example, if given sufficient control, an attacker could force the victim contract to execute the "transfer" function of a token, thus moving assets from the contract's account to the attacker's account.

Manipulating a contract to carry out actions on its behalf is a classic example of a *confused deputy* attack. The (victim) smart contract, acting as the deputy, is confused into executing actions on behalf of the attacker. Previous work and best practices emphasize the importance of robust access control [6, 21] and cautious handling of calls and their parameters to mitigate the risk of confused deputy attacks [23, 60].

Token Transfers. The ERC20 standard outlines a common interface for fungible tokens [1], enabling seamless interactions across the Ethereum ecosystem. We show a highly simplified ERC20 token implementation in Figure 1. At its core are a set of state variables, events, and public functions supporting two key functionalities: direct and indirect token transfers.

A direct token transfer occurs when a token holder transfers tokens to another address using the token's `transfer` function (Lines 5-9). In brief, `transfer` is a public function that facilitates the transfer of a specified amount of tokens from the *transaction sender* to the desired recipient – without any intermediaries. Access control is implicit: the sender of the

[1] Other token standards exist, such as ERC721 [15] (non-fungible tokens), ERC1155 [14] (multi-token), and ERC4626 [17] (tokenized vaults).

transaction (`msg.sender`) can only spend their own balance. Upon successful execution, `transfer` will update the token balance (`balance`) of the two involved accounts to reflect the transaction and emit a `Transfer` event – which allows off-chain applications to monitor and index the activity.

An indirect token transfer occurs when a third party (called the "spender") invokes the token's `transferFrom` function (Lines 15-21). In brief, `transferFrom` is a public function that allows a *spender* to transfer tokens from one account (`from`) to another (`to`) on behalf of the token holder, provided the spender was *previously approved*. Access control is still implicit: the spender must be `msg.sender`. However, the token owner can arbitrarily approve any address to become an approved spender. More precisely, the `approve` function (Lines 11-13) allows a token owner to set an allowance for the spender, specifying the maximum amount of tokens they can transfer on their behalf. Importantly, the owner must explicitly grant permission *before* the spender attempts the transfer. Upon successful execution, `transferFrom` updates the token balance (`balance`) and token allowance (`allowance`) of the involved accounts and emits a `Transfer` event. Similarly, upon successful execution, `approve` updates the token allowance of the spender and emits an `Approval` event. This additional layer of indirection enables a more complex but still secure interaction model, allowing the integration of ERC20 tokens into a broader range of applications and services.

In conclusion, when blockchain users grant approvals to a third-party contract, they trust such a contract to operate on their behalf. This is often required in decentralized applications, but creates an opportunity for an attacker to abuse the trust relationship between the trusted contract and its users.

Approval Signatures. Direct and indirect token transfers require the token owner to interact with the token contract – either calling the `transfer` or `approve` function – and to pay a gas fee for such transaction. The token owner must also interact with the token contract to revoke any approvals, which also incurs a gas fee. Permit signatures attempt to solve this problem by allowing the token owner to sign an approval message off-chain. These come in two varieties: EIP-2612 permit signatures [29], and Permit2 signatures [57].

EIP-2612 signatures are simple: the owner creates a signature and, off-chain, sends it to the spender. The spender then uses it as proof of the approval, which the ERC20 token verifies on-chain. This ultimately leads to a change in allowance and is thus *indistinguishable from normal approvals for the purpose of this work*.

Unfortunately, supporting EIP-2612 signatures requires updating the token contract itself, which may be either difficult or impossible depending on the token's configuration. For this reason, Uniswap [56] designed the Permit2 contract as an alternative mechanism. To use Permit2, the token owner must approve the Permit2 contract (instead of the spender), and must sign an approval message off-chain, allowing the spender to transfer tokens *via Permit2* on their behalf. To perform a transfer, the spender then interacts directly with the Permit2 contract (instead of the token) and uses the user's signature as proof that permission is granted. Permit2 signature-based approvals do not require any change in the underlying token contract but rather modify the Permit2 contract's internal allowances.

```
1  contract Lending
2      ERC20 token = 0x[...];
3      map(addr => uint) balance;
4      map(addr => map(addr => uint)) debt;
5
6      function offerLoan(uint val)
7          balance[msg.sender] += val;
8
9      function borrow(uint val)
10         // [...] (ensure valid collateral)
11         //       (find a valid lender)
12         require(balance[lender] >= val);
13         token.transferFrom(lender, msg.sender, val);
14         balance[lender] -= val;
15         debt[msg.sender][lender] += val;
16
17     function repay(address lender, uint val)
18         require(debts[msg.sender][lender] >= val);
19         uint fee = val * 0.1;
20         token.transferFrom(msg.sender, lender, val+fee);
21         debts[msg.sender] -= val;
22
23     function execute(address executor, bytes data)
24         // [...] (missing access control)
25         executor.call(data);
```

Figure 2: Simplified Solidity code of the vulnerable contract. The `execute` function lacks proper access control and allows an attacker to manipulate the call at Line 25.

3 Motivation

The approval mechanism allows decentralized financial services to (indirectly) transfer their users' assets. This mechanism is indispensable for a wide array of DeFi applications. Unfortunately, approvals also open avenues for potential exploitation. In the following paragraphs, we present a scenario that aims to clarify the typical use (and importance) of token approvals while shedding light on the vulnerabilities they potentially introduce. Throughout this section, we will refer to the code of the victim `Lending` contract shown in Figure 2, and the interaction diagram presented in Figure 3.

Benign Flow of Actions. Alice is an active blockchain user, eager to explore Ethereum's decentralized financial services and make the most of her asset holdings. Alice owns some tokens and would like to earn interest on her holdings by loaning them out through a Lending platform. To do this, she first approves the Lending platform to manage 100 of her ERC20 tokens (Ⓐ). This is required by the Lending contract to execute transactions on her behalf. Then, she calls the function `offerLoan` (Lines 6-7) to inform the platform that she is willing to lend 10 of her tokens (Ⓑ). At this stage, the Lending contract updates Alice's internal `balance` (Line 7) but does not transfer any tokens.

Afterward, Bob invokes the function `borrow` (Lines 9-15, Ⓒ), and offers appropriate collateral for the loan he wishes to

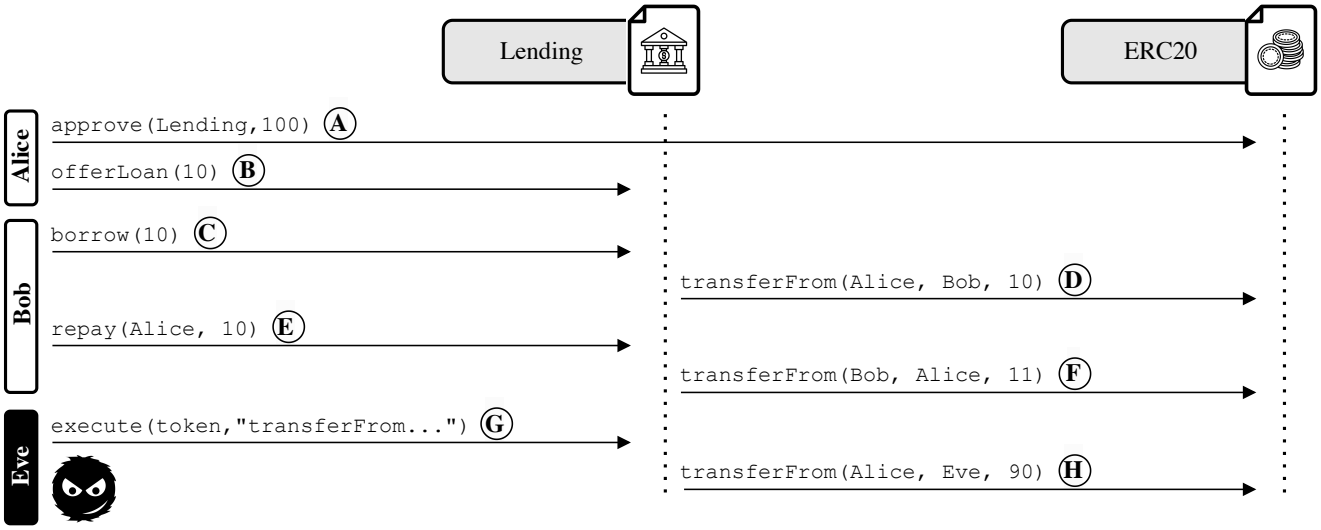

Figure 3: Simplified interaction flow of the Lending contract. Alice approves the Lending platform to manage 100 of her ERC20 tokens (Ⓐ). Then, she lends 10 of her tokens (Ⓑ). Bob requests to borrow 10 tokens (Ⓒ). The Lending contract matches Bob's request with a valid lender (Alice) and transfers 10 tokens from Alice to Bob (Ⓓ). Later, Bob repays his debt – plus an interest fee (Ⓔ/Ⓕ). Eve notices that the `execute` function has improper access control, and calls it (Ⓖ). The Lending contract executes Eve's request and transfers 90 tokens from Alice to Eve (Ⓗ).

secure. The platform matches his request with Alice's offer, and invokes the ERC20 token's `transferFrom` function to transfer 10 tokens from Alice to Bob (Ⓓ). As a result, Bob has borrowed 10 tokens from Alice (and is now the new owner). To get his collateral back, Bob is obliged to return the tokens that he owes – plus some interest, according to the terms set by the Lending platform. Thus, Bob calls the function `repay` (Lines 17-21, Ⓔ). The platform then transfers 11 tokens from Bob to Alice (Ⓕ), closing the loop on the smart-contract-mediated lending-borrowing operation.

Malicious Flow of Actions. Eve has found a flaw in the Lending platform's smart contract. Within the contract's code is a function named `execute` (Lines 23-25). The `execute` function takes two parameters: `executor` and `data`. When invoked, this function will call the `executor` contract with the provided `data` bytes as input. Critically, the `execute` function does not include any access control checks. Thus, Eve invokes the `execute` function (Ⓖ) and provides as arguments (1) the address of the token contract and (2) the input data `transferFrom(Alice, Eve, 90)` (as bytes). The Lending platform complies, proceeding to transfer 90 tokens from Alice to Eve (Ⓗ).

Threat Model. ACT vulnerabilities differ from open-call vulnerabilities in two key ways. First, ACT vulnerabilities can only exist in very particular settings – that is, an approved contract with the ability to interact with other token contracts. Detecting this setting is critical and allows one to significantly reduce false positives. Second, ACT vulnerabilities require valid interactions with the token contract and, thus, require modeling a more powerful attacker with knowledge of the token functionality. With this knowledge, the attacker can craft an exploit even if only a few bytes are controllable. Since ACT is a logic vulnerability, proper modeling of the attacker is critical and allows us to significantly reduce false negatives.

Since 2020, at least 22 high-profile incidents have been made public where attackers exploited ACT vulnerabilities to steal a total of 65 million USD [46]. These attacks have two common properties: (1) DeFi users trusted the victim contract to manage tokens on their behalf, and (2) the attacker found a way to manipulate the victim contract and gain control over a call to `transferFrom`. This allowed the attacker to gain unauthorized access to the funds of all DeFi users who trusted (approved) the victim contract. In this work, we consider a contract vulnerable to an ACT vulnerability if it satisfies the following requirements:

(R1). The contract must have been previously approved as the spender for at least one token.

(R2). The contract must contain a (partially) controllable function call that can be manipulated to (a) call the `transferFrom` function of a valid token and (b) transfer tokens from one user (not the attacker) with active approvals to the attacker's wallet.

(R3). The controllable function call can be executed successfully – without reverting – by any arbitrary blockchain user – e.g., without access control.

Among the 22 attacks that have exploited ACT vulnerabilities, 14 match this threat model. We consider these attacks to be within the scope of our work. The remaining 8 attacks required the attacker to chain additional vulnerabilities. We deem these attacks out-of-scope for our automated analysis.

4 Approach

At a high level, OSPREY aims to detect controllable calls to a `transferFrom` function that – if executed by an attacker – enable the unauthorized transfer of assets from a victim users' account to the attacker. Our approach consists of five analysis stages as shown in Figure 4 (❶-❺): First, we analyze Ethereum's historical transactions to identify all smart contracts that have ever been approved to spend any ERC20 token (**R1**). Second, we use symbolic execution to analyze each approved contract (found in the first stage) and determine whether any of its `CALL` instructions is controllable and can be manipulated to meet the (**R2**) requirements. At this stage, the analysis operates under an ideal (entirely symbolic) blockchain environment. That is, in the real world, the `CALL` instruction might not be reachable. Third, we use a combination of symbolic and concrete execution to verify that the `CALL` instruction is actually reachable and controllable *under a concrete blockchain environment* – and can be executed without reverting (**R3**). If so, we determine that the contract is *vulnerable*. Otherwise, we use heuristics to determine the reason for failure and emit warnings with two degrees of confidence – high-confidence if the execution reverts *after* the `transferFrom` call, or else low-confidence. Fourth, utilizing our knowledge of existing on-chain approvals, we further analyze all vulnerable contracts and attempt to automatically synthesize a proof-of-concept attack (an exploit). Finally, we estimate the current (and historical) impact of all such exploits. In the following sections, we describe each stage in more detail.

4.1 Approved Contracts

ACT vulnerabilities only have a material impact among smart contracts that have received user approvals. In fact, a contract that has not received user approvals does not have authorization to transfer any token other than its own. Thus, we first identify all contracts that have received user approvals, which constitute the underlying dataset for subsequent analysis.

❶ **Approval Filter.** As mentioned in Section 2, when users want a third-party contract to manage their ERC20 tokens on their behalf, they must first interact with the corresponding ERC20 token contract to issue an approval. In turn, the ERC20 token contract will update its internal state to reflect the approval and emit an `Approval` event that informs other parties of this change. The ERC20 specification requires that tokens emit this event upon a successful call to `approve` [16].

In our setup, we use a well-known and locally deployed Ethereum client, Erigon [27], and leverage its record of log events to observe and collect all approval events. We do this for all ERC20 tokens, regardless of their value. In this way, we reconstruct a complete historical picture of all smart contracts that have ever been approved to trade users' funds. We refer to this group of contracts as the APPROVED contracts.

4.2 Controllable TransferFrom

Once we have identified all APPROVED contracts, we analyze them to find ACT vulnerabilities. We do this in two steps. First, we use a lightweight static analysis, based on symbolic execution, to discard all contracts without a controllable call to a `transferFrom` function. Second, we analyze the remaining contracts using a combination of symbolic and concrete execution to understand whether the controllable call can be executed under a specific, concrete blockchain environment. Depending on the outcome of the latter analysis, we either determine that the contract is VULNERABLE or raise a warning if the vulnerability cannot be (concretely) confirmed.

❷ **Off-chain Controllability.** To identify controllable calls to `transferFrom`, we implement a static analysis stage based on the symbolic (simulated) execution of the APPROVED contract. Although this does not guarantee that the call can be controlled in practice, it allows us to isolate a group of contracts that would be vulnerable given the right blockchain state. In other words, we determine whether the call to `transferFrom` would be controllable under an entirely symbolic blockchain environment.

Practically, we statically inspect the APPROVED contract's code to find all `CALL` instructions. For each `CALL` instruction, we study the contract's control flow graph (CFG) to identify all public functions that can reach such instruction – that is, the relevant contract's entry points. Then, we leverage a symbolic execution engine for EVM smart contracts, `greed` [55], to execute the contract's code symbolically. More precisely, using directed symbolic execution [31], we execute the APPROVED contract's code from each relevant entry point (a subset of public functions) to an identified `CALL` instruction. After reaching this `CALL` instruction, we request from `greed` a solution that satisfies all the accumulated path constraints *and* allows an attacker to steal ERC20 tokens belonging to a different user. To this end, we formulate the following additional constraints:

1. The `CALL` target address must be a valid ERC20 token.
2. The function selector must be equal to `transferFrom`.
3. The `transferFrom` "from" parameter (the victim) must be different from the attacker's address.
4. The `transferFrom` "to" parameter must be equal to the attacker's address.
5. The `transferFrom` "value" parameter must be greater than zero.

If such a solution exists, we determine that the `CALL` instruction might be controllable. We refer to a contract with a controllable `CALL` as a CANDIDATE contract.

❸ **On-chain Controllability.** At this stage, we must determine whether the CANDIDATE contract allows a controllable

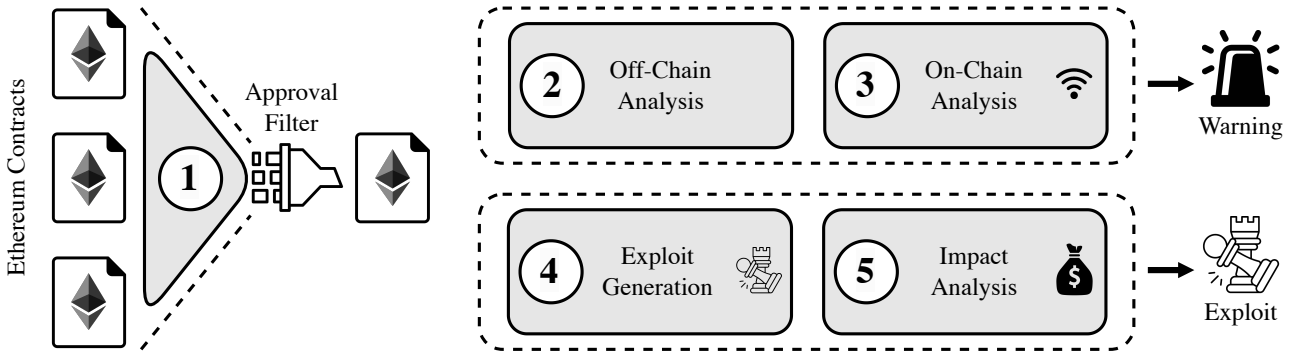

Figure 4: Overview of our approach, implemented in OSPREY. The analysis pipeline follows the order of the circled numbers. First, ❶ we identify smart contracts with user approvals. Then, ❷ we use symbolic execution to determine if the contract would be vulnerable under an ideal blockchain environment. ❸ We verify that the contract is vulnerable under a specific, concrete blockchain environment. If not, we emit a warning. Otherwise, ❹ we automatically craft an exploit and ❺ estimate its impact.

```
1 function execute2(address registry, bytes data)
2     address executor = registry.executor();
3     executor.call(data);
```

Figure 5: The `execute2` function allows manipulating the call at Line 3: the attacker-controlled `registry` contract can be crafted to manipulate the address of the `executor`.

function call to `transferFrom` *under a specific, concrete blockchain environment*. We leverage the same symbolic execution technique presented in ❷ – but with a concrete (non-symbolic) environment – to automatically craft the calldata needed to launch the attack on-chain. We then concretely execute the attack against the same specific blockchain state. More precisely, we simulate the execution of the attack against our locally deployed Ethereum client. If the attack is successful, we report the contract as VULNERABLE. Otherwise, we resume symbolic execution and iteratively retry to synthesize the input data for a different path until the attack succeeds. If the attack generation fails for all attempted paths, we automatically classify the failure and emit warnings where appropriate.

Practical Challenges. In Section 3, we provided a simplified example of a vulnerable Lending contract. Real-world contracts are often considerably more complex. The additional complexity comes from both the contracts' logic and their external interactions. Modeling these external interactions is one of our primary analysis challenges. Previously, during ❷, we modeled the blockchain environment (and all contract interactions) symbolically. In other words, our analysis assumed that any interaction of the CANDIDATE contract with external contracts (1) does not have any side effects and (2) returns exactly what is needed to continue the symbolic exploration – that is, to satisfy the symbolic constraints. Of course, these assumptions might not hold during concrete execution.

We identify two primary challenges: First, when invoking the controllable `transferFrom` our analysis must provide both a valid user address and an ERC20 token with an allowance. However, there may currently be no valid victim users (e.g., the contract is approved to manage users' funds, but the users do not currently hold a token balance). When this happens, we must determine whether the attack would work if valid victim users existed.

Second, in certain cases, reaching the vulnerable call requires not only specific inputs, but also successful interactions with external "accessory" contracts. For example, the CANDIDATE contract might call a decentralized exchange's `getCurrentPrice` function, and continue execution only if the price returned is above 10. If the address of the accessory contract is predetermined, we rely on symbolic execution to craft a valid interaction. Instead, if the address is user-controllable, we must identify or craft an appropriate smart contract (that would result in a successful interaction) and use its address in order to make progress in the execution.

We design two mechanisms to (1) automatically deploy a synthetic victim user with arbitrary token balance and allowance and (2) automatically deploy an attacker-controlled synthetic accessory contract to support our analysis.

Synthetic Victim Users. To determine whether the attack would work if victim users existed, we automatically craft synthetic victim users with a positive token balance and enough allowance for the CANDIDATE contract to transfer tokens on their behalf. To do so, OSPREY automatically identifies (a) a valid ERC20 token and (b) the token's storage slot where the user balance resides.

First, after creating a fresh Ethereum address for the victim user, we leverage symbolic execution to identify valid ERC20 token addresses that are compatible with the accumulated constraints. If the address of the ERC20 token is fully controllable, we simply use USDC [7].

Then, we analyze the chosen ERC20 token to determine the storage slot where the user balance is stored. More precisely, we invoke the ERC20 standard function `balanceOf` (in our local, simulated environment) to retrieve the balance of the synthetic victim user. This allows us to dynamically observe all storage accesses – typically few storage reads. By

matching the function's return value with the accessed storage values, we infer where the balance is stored, and overwrite it with a chosen arbitrary value x.

Finally, we invoke the `approve` function on behalf of the synthetic victim user, specifying the CANDIDATE contract as the spender and the same chosen value x as the allowance.

Synthetic Accessory Contracts. Most CANDIDATE contracts do not operate in isolation. Instead, they commonly interact with external contracts such as price oracles and ERC20 tokens. We refer to all such contracts as "accessory contracts".

While the semantics of these interactions are unknown at analysis time, symbolic execution tells us what expected responses should look like (expressed in the form of symbolic constraints). Sometimes, the attacker has little control over the address of the accessory contract. If so, we allow symbolic execution to choose feasible inputs that satisfy the symbolic constraints (without analyzing the target contract). At other times, the address of the accessory contract is itself controllable. For example, in Figure 5, we show a variation of the `execute` function presented in Section 3. The `execute2` function takes two input parameters (`registry` and `data`) and does not allow direct control of the `executor` address. Instead, it queries the `registry` contract to obtain that address.

OSPREY automatically recognizes that the address of the `registry` contract is attacker-controllable, and leverages the accumulated symbolic constraints to concretize the return value of the `CALL` instruction (Line 3). This allows OSPREY to compile and deploy the bytecode of a contract that returns exactly the value that is (symbolically) shown to work. Connecting back to the example in Figure 5, OSPREY (a) determines that the `Lending` contract calls the `executor` contract with arbitrary calldata, (b) determines that the `executor` address is returned by the call to the `registry` contract at Line 2, (c) determines that the address of the `registry` contract is attacker-controllable, (d) compiles and deploys a custom (`registry`) contract that returns the address of an ERC20 token (the "`executor`") and finally (e) calls the `execute()` function to launch the attack.

In summary, the interplay between concrete and symbolic execution allows us to execute deep functionality in the vulnerable contract's code, which is often required to reach the vulnerable `transferFrom` function call and synthesize a successful exploit.

Failure Analysis. In the previous paragraphs, we discussed our approach to modeling both the contract logic (with symbolic execution) and the external interactions (using synthetic victim users and synthetic accessory contracts). Sometimes, the concrete execution does not reach the `transferFrom` function. If the concrete execution otherwise completes successfully (that is, it does not revert), we complete the analysis with no warning. This often happens when specific storage values (from the concrete blockchain state) conflict with the values expected by symbolic execution.

If the execution, on the other hand, *reverts*, we aim to determine whether it reverts "by design" (i.e., there are explicit access control checks) or whether the contract logic and external interactions are simply too complex for our analysis to handle (and the contract might be vulnerable after all). In the following paragraphs, we will discuss how, in practice, we distinguish between these two cases.

Access Control. Often, the execution reverts simply because the (concrete) attacker address is not allowed to execute the smart contract. That is, there are access control checks in place that prevent the attacker from reaching (and invoking) the vulnerable function. The three most common access control patterns in smart contracts are:

- **Identity-based**. The contract compares `msg.sender` to a constant value or a value loaded from storage.
- **Role-based**. The contract looks up `msg.sender` in a mapping variable that associates addresses with roles (e.g., owner) and permissions.
- **Oracle-based**. The contract queries an external oracle (contract) to confirm whether `msg.sender` is authorized.

As shown by previous research [21, 60], identifying these access control patterns can be challenging. For this reason, in our previous analysis stage (❷), we do not attempt to statically (symbolically) model any access control pattern. That is, both the environment and the `msg.sender` values are symbolic, ensuring that any such access control check would succeed. Instead, we defer this analysis until the contract is concretely executed. In doing so, we can observe all instructions that are executed, allowing us to implement simple heuristics to detect access control patterns effectively. We observe that, during an access control sequence, the contract executes the `CALLER` instruction to fetch the address of the `msg.sender`. Moreover, if the access control check fails, the contract executes the `REVERT` instruction. We leverage these observations to determine whether a failure could have resulted from a prior access control check. Our heuristic works as follows: When the execution reverts, we look at the instructions executed directly before the `REVERT` instruction – ignoring the instructions executed by any accessory contract. If these instructions include the `CALLER` instruction (which retrieves `msg.sender`), we classify the failure as access control-related. In Section 5.2, we evaluate the effectiveness of this heuristic. We find that, despite its simplicity, it is effective.

Warnings. If we determine that the execution reverted, but not because of an access control check, we emit a warning. We associate this warning with a confidence level – low if the execution reverts before the `transferFrom`, high otherwise. Our intuition is that most of the contract's logic is expected to execute *before* the `tranferFrom` function call. Therefore, being able to reach (and execute past) the controllable function call suggests that an exploit might be viable with further (manual) effort.

(w_1) **Post-`transferFrom` warning**. In some cases, the execution successfully reaches the controllable `tranferFrom`. Nonetheless, to complete the transaction, the execution must return successfully – that is, not revert. For example, the input data might be structured as a list of actions – that our analysis failed to model. Since we expect the `transferFrom` to be past most of the contract's logic, we consider these high-confidence warnings.

(w_2) **Pre-`transferFrom` warning**. In other cases, the execution reverts before reaching the controllable `tranferFrom`. This often happens because the contract expects a certain input (e.g., an accessory contract address) that our analysis failed to model (e.g., the contract looks up the provided address in a mapping). Since we expect most of the contract's logic to execute *before* the `tranferFrom` function call, and since, in this case, the call could not be reached, we consider these low-confidence warnings.

In Section 5.2, we evaluate the quality of these warnings and confirm that, in some cases, it is possible to manually craft an exploit even if our analysis failed to do so automatically.

4.3 Exploit Generation and Impact Analysis

To estimate the impact of a generated exploit, we leverage our knowledge of all historical approvals and automatically synthesize one targeted exploit for each approver-token pair with both balance and allowance. Then, as detailed below, we use Uniswap [56] (a well-known decentralized exchange) to estimate the monetary impact of the exploits.

❹ **Automatic Exploit Generation.** Given a successful attack against a VULNERABLE contract, we leverage our knowledge from ❶ to retrieve the current allowances and balances of all users that have historically approved the VULNERABLE contract as a spender. We identify as victims all those users with both a positive allowance and a positive balance. More precisely, we calculate the maximum number of tokens that can be stolen from each user as `min(allowance, balance)`. We combine this information with the successful attack reported in ❸ to construct a real execution scenario that leads to the theft of all approved user assets.

❺ **Impact Analysis.** To estimate the practical market value of the stolen assets, we interact with the Uniswap v2 and Uniswap v3 exchanges and trade the stolen assets for an easily priceable asset – Ether. Importantly, even when the stolen assets have no practical estimated market value, one should always keep in mind that (1) the assets could have been valuable in the past, (2) the assets might be valuable in the future, (3) a market for the assets may exist elsewhere, but not on Uniswap, and (4) the assets (and the VULNERABLE contract) could be highly valuable regardless of their estimated market value [2].

	Number of Contracts
VULNERABLE	410
w_1	117
w_2	2,566
Access Control	14,053
Non-Controllable	14,441
Failed	995
Total	32,582

Table 1: Overview of the On-chain analysis results (❸).

5 Evaluation

For all our experiments, we use one server equipped with 512GB of RAM and dual Intel(R) Xeon(R) Gold 6330 CPUs. We use GNU Parallel [51] to parallelize our tasks, and always limit each task to 5GB of RAM and either 10 or 60 minutes of CPU time, depending on the task's complexity.

Live evaluation. To demonstrate its practicality, we also deploy OSPREY as a live detection system (Appendix C).

Preparation. We consider all Ethereum blocks and contracts from its genesis block (July 2015) to block 19,380,000 (March 2024). We rely on a local deployment of the Erigon v2.56.0 client [27] to inspect transactions. Since there are more than two billion external transactions in the Ethereum network at the time of writing, inspecting and indexing these transactions requires a substantial amount of time – approximately 30 days. We consider this as a necessary one-off preparation step for any work that aims to study the Ethereum blockchain.

5.1 Approved Contracts

❶ **Approval Filter.** As discussed in Section 4.1, we leverage the standard ERC20 `Approval` logging mechanism to observe and collect all on-chain approvals. In total, we observe 78,620,333 approvals. We determine that 424,676 accounts (out of which 410,938 are contracts) were historically approved as spenders. We refer to these contracts as the APPROVED contracts.

Results Discussion. To understand the evolution of the approvals ecosystem, we look at past approvals and study how they change over time (see Appendix A). We find that the majority of spenders only have a few approvals. Although some spender contracts receive several million approvals, less than 10% of the spenders receive more than 10 approvals.

We observe that approvals started being adopted at the beginning of 2016 (block 1,022,258), just a few months after the introduction of ERC20 in Ethereum (see Appendix B). Since then, the number of ERC20 approvals has been steadily growing. The growth of this ecosystem calls for automated tools like OSPREY to ensure the security of the involved contracts and their users.

[2] In one of the most significant known attacks [39] (shown in Table 3) the stolen assets are financial derivatives whose value is hard to estimate.

5.2 Controllable TransferFrom

❷ **Off-chain Controllability.** We first run our Off-chain Controllability analysis against all the APPROVED contracts to identify the CANDIDATE contracts. That is, we use symbolic execution to simulate the execution of the APPROVED contracts. Given a `(contract, function, call)` tuple, OSPREY determines whether `call` can be controlled by an attacker who invokes the public `function` of the `contract` (with appropriate arguments) under an ideal (entirely symbolic) blockchain state. For each tuple `(contract, function, call)`, we set a CPU time limit of 10 minutes. We observe a median execution time of 19 seconds.

Among all the APPROVED contracts, we identify 32,582 contracts with a controllable `transferFrom` call – i.e., the CANDIDATE contracts – and 339,957 contracts without a controllable `transferFrom` call. For the remaining 38,399 contracts (9% of all APPROVED contracts), our analysis fails. Specifically, in 38,176 cases, the analysis times out. In 223 cases, the analysis exhausts the memory.

❸ **On-chain Controllability.** We then run our On-chain Controllability analysis against all 32,582 reported CANDIDATE contracts. To do this, we use a combination of symbolic and concrete execution to simulate the execution of the CANDIDATE contracts. For each CANDIDATE contract, we check all symbolically controllable `(contract, function, call)` tuples. For each tuple, OSPREY attempts to determine whether the call is controllable under a specific, concrete blockchain state (the reference block 19,380,000). We limit each task to 60 minutes of CPU time and observe a median execution time of 41 seconds.

We find 410 contracts with a controllable `transferFrom` call – i.e., the VULNERABLE contracts. We manually inspect a random sample of 50 VULNERABLE contracts and confirm that all the manually inspected contracts are indeed vulnerable. Moreover, we find 14,053 contracts with access control checks (which prevent us from reaching the target `call` instruction), 2,683 contracts with a warning (w_1 or w_2), and 14,441 contracts without a (concretely) controllable `transferFrom` call. In the latter case, a `transferFrom` call might not be controllable in practice when the concrete environment is incompatible with OSPREY's symbolic model. For example, the symbolic execution may have assumed that an external call could return the value `true`, but in practice, it always returns `false`. For the remaining 995 contracts (3% of all CANDIDATE contracts), our analysis fails. In 896 cases, the analysis times out. In 99 cases, the analysis exhausts the memory limit. In Table 1, we present an overview of our results.

Access Control. We manually inspect a random sample of 50 (out of 14,053) contracts with access control checks and confirm that, in all cases, the execution was indeed guarded by an access control policy. Note that some (more complex) access control patterns might not be captured by our heuristic and are instead (mistakenly) reported as low-confidence warnings. However, since our heuristic captures the majority of access control patterns, we consider this an acceptable error.

Warnings. Our analysis produces warnings for 2,683 contracts. These contracts are worth reporting, but they need further (manual) scrutiny as the automated analysis could not directly synthesize a working exploit. Looking at the two types of warnings introduced in Section 4.2, we observe 2,566 contracts with a pre-`transferFrom` warning (w_2, low confidence), and 117 contracts with a post-`transferFrom` warning (w_1, high confidence). We manually inspect a random sample of 50 warnings from each of these categories to assess the quality of our results. Our results (presented below) confirm that the distinction between high-confidence and low-confidence warnings is meaningful in predicting likely exploitable contracts.

For 19 out of 50 (38%) of the high-confidence (w_1) warnings, we could manually craft a fully working exploit. We observe that OSPREY's automatic exploit generation failed to model some of the contract's external interaction with accessory contracts – for example, when one of the arguments must be a valid ERC20 token – and consequently failed to craft valid input data. For 21 out of 50 (42%) w_1 warnings, we determine that the exploit might work under some circumstances. For example, the affected contract is currently not functional (paused) but would be exploitable if resumed. Finally, for the remaining 10 out of 50 (20%) w_1 warnings, we determine that the contract is likely not exploitable. In these cases, *before* the attack transaction, the attacker must perform additional actions that make the attack unprofitable. For example, the attacker must `transfer` tokens to the affected contract.

For 16 out of 50 (32%) of the low-confidence (w_2) warnings, we determine that the exploit might work under some circumstances. For example, in several cases, the token address is checked against a whitelist, but the attacker might be able to manipulate the whitelist. Such a scenario was exploited in at least one known real-world attack that caused substantial losses [42]. For the remaining 34 out of 50 (68%) w_2 warnings, we determine that the contract is likely not exploitable. We observe that, in some of these cases, the access control pattern was not properly captured by OSPREY. In other cases, as mentioned before, an attack would require additional actions that make it unprofitable.

5.3 Exploit Generation and Impact Analysis

❹ **Automatic Exploit Generation.** We run our Automatic Exploit Generation analysis against all VULNERABLE contracts: First, for each VULNERABLE contract, we retrieve all user approvals (from ❶), involved tokens, and approved amounts. Then, we combine this information with the successful attacks (concrete executions) reported in ❸ to construct a real execution scenario that leads to the theft of all approved user assets – that is, an *exploit*. In some cases, we generate multiple exploits against the same VULNERABLE contract to

Contract	OSPREY	JACKAL	PS	Block	Impact (USD)
					Found (❺)
3271	VULN	C	·	14,018,000	1.4M
78e8	VULN	C	·	14,043,200	1.3M
cf9c	VULN	·	·	13,946,000	328K
18e1	VULN	C	·	14,658,800	100K
bb43	VULN	C	·	16,563,200	90K
2aaa	VULN	C	·	16,718,000	75K
405e	VULN	·	·	18,006,800	56K
b677	VULN	·	·	14,100,800	26K
40cd	VULN	C	·	17,052,800	21K
d27e	VULN	C	·	14,525,600	13K
34ad	VULN	C	·	14,525,600	12K
86eb	VULN	·	·	14,014,400	11K
53b3	VULN	·	·	18,161,600	10K
23e1	VULN	·	·	17,121,200	9K
Total	14	8	-	-	3.4M

Table 2: (Selection of) previously unknown vulnerabilities reported by OSPREY, JACKAL, and PRETTYSMART (PS). C: CANDIDATE, VULN: VULNERABLE.

Contract	OSPREY	JACKAL	PS	Impact (USD)	
				Found (❺)	Actual
Socket [40]	VULN	C	·	4.3M	3.3M
Maestro [38]	VULN	C	·	3.4M	500K
Unibot [37]	VULN	C	·	1.4M	600K
Hashflow [36]	VULN	·	·	214K	214K
Bancor [1]	VULN	·	C	-	-
dYdX [13]	w_1	·	·	2.7M	2.2M
Rubic [35] (1)	w_1	·	·	1.1M	1.1M
Rubic [35] (2)	w_1	·	·	804K	300K
Li-Fi [28]	w_1	·	·	597K	597K
brahTOPG [32]	w_1	·	·	93K	90K
Seneca [39]	w_2	·	·	-	-
Rabby [33]	w_2	·	·	98K	98K
Revert [45]	w_2	·	·	24K	24K
Sorbet [20]	·	·	·	-	-
Total	13	3	1	14.7M	9M

Table 3: Comparison of OSPREY, JACKAL, and PRETTYSMART (PS) on our labeled dataset. C: CANDIDATE, w_2: low-conf. warning, w_1: high-conf. warning, VULN: VULNERABLE.

target multiple users, multiple tokens, or both. The analysis in this stage is lightweight and takes less than 60 minutes to generate all exploits.

We process all 410 VULNERABLE contracts and automatically generate exploits for all of them. In total, we generate 778 fully working exploits at the reference block (19,380,000). These exploits enable the theft of a variety of user assets (120 unique tokens) from a variety of users (275 unique approvers).

Exploits can become available over time as new users approve a VULNERABLE contract. Similarly, exploits become unavailable when users revoke their approvals. To determine the number of historical exploits, we run our automatic exploit generations at intervals of 12 hours over the last 24 months (from January 1st, 2022, to the reference block). In total, we generate 2,728 fully working exploits – which enable the theft of 625 tokens from 1,491 approvers.

❺ **Impact Analysis.** Finally, we run our Impact Analysis on all the automatically generated exploits. For each VULNERABLE contract, we measure the impact of the vulnerability as its maximum historical financial impact – across all automatically generated exploits. To do so, we use Uniswap to estimate the value of the stolen assets (see Section 4). In Table 2, we show a selection of (previously unreported) VULNERABLE contracts with automatically generated exploits, sorted by decreasing impact. In total, we report more than 3.4 million USD of latent financial impact from previously unreported vulnerabilities. For each vulnerability, we report the code hash of the VULNERABLE contract (truncated for ethical reasons), the maximum historical financial impact, and the corresponding block. The most notable previously unreported vulnerability that we discover (with truncated code hash `3271`) has an impact of more than 1.4 million USD, and several others also have a financial impact of more than 100 thousand USD.

5.4 Known Attacks

As discussed in Section 3, we are aware of 14 well-known attacks against ACT vulnerabilities that fall in scope for this paper. We manually inspect each attack to identify the exact exploit transaction, the address of the vulnerable contract, the signature of the vulnerable public function invoked by the attacker, and the program location (i.e., the EVM program counter) of the vulnerable `transferFrom` call. In doing so, we create a manually curated and labeled dataset of 14 real-world contracts that are known to be exploitable.

To further evaluate the effectiveness of our detection approach, we run all the stages of our analysis against this dataset and present our results in Table 3. As expected, OSPREY finds at least one approval event for each known attack (❶) and, therefore, reports all the contracts as APPROVED. We then run our Off-chain analysis (❷) on all contracts. The analysis executes successfully on 13 of these contracts, which we report as CANDIDATE contracts. For one of the contracts (Sorbet [20]), our symbolic analysis times out. We manually confirm that the contract's vulnerable public function takes multiple dynamically sized arrays as input (encoding different "actions"), which is challenging to model symbolically. OSPREY identifies 5 (out of 13) CANDIDATE contracts as VULNERABLE (❸), and automatically generates exploits for each of these contracts (❹). For the remaining 8 contracts, OSPREY emits 5 high-confidence warnings and 3 low-confidence warnings. Our results highlight the importance of our tiered warning classification system: Automatically generating an end-to-end exploit is often hard because of the complexity of these real-world contracts. Nonetheless, OSPREY is able to detect issues worthy of review. In summary, OSPREY successfully detects issues for all but one of the known attacks.

Finally, we assess the financial impact of the generated exploits (❺) and the potential financial impact of the reported

warnings. We then compare our estimate with the amount that was stolen during the attack – the "Actual" impact reported in Table 3. In some cases (Bancor [1] and Seneca [39]), OSPREY could not estimate the financial impact. In all such cases, we manually confirm that the stolen assets are not traded on Uniswap and, therefore, cannot be automatically priced. Surprisingly, this comparison allows us to observe that, in many other cases, the real-world attacks were actually not optimal. In total, we find 5.7 million USD of additional financial damage that could have resulted from the optimal attacks (and that were automatically generated by OSPREY).

5.5 Comparison with Existing Systems

We compare OSPREY against two state-of-the-art systems that perform generic detection of controllable calls: PRETTYSMART [60] and JACKAL [23]. We compare their performance against our curated dataset of known attacks and against all Ethereum smart contracts – as shown in Table 2 and Table 3. In our experiments, OSPREY demonstrates detection efficacy far exceeding prior work.

PrettySmart. PRETTYSMART implements a static taint analysis built on top of the Gigahorse [22] binary lifting framework. While this analysis can flag potential controllable calls, it is unable to verify the actual feasibility of the associated execution paths. Thus, we consider its reports (of controllable calls) as CANDIDATE contracts and compare them with the results of our stage ❷. From the results reported in the paper [60], PRETTYSMART flags 268 smart contracts with potentially controllable call statements. Unfortunately, the authors do not apply any filtering to identify APPROVED contracts (❶) – or to restrict their reports to `transferFrom` calls. We find that only 10 of the reported vulnerable smart contracts have received user approvals. Moreover, PRETTYSMART does not flag any of the attacks reported in Table 2. We manually inspect the 10 contracts flagged by PRETTYSMART and confirm that 2 of them are vulnerable to ACT vulnerabilities: Bancor [1] and TransitSwap [34]. Bancor is also detected by OSPREY. TransitSwap is out-of-scope for our work because it requires chaining multiple vulnerabilities, nevertheless, we confirm that OSPREY identifies the ACT vulnerability.

Since the source code of PRETTYSMART is publicly available, to provide a fair comparison, we modify its analysis to use the same time limits reported in Section 5. Then, we run PRETTYSMART against all APPROVED contracts in our dataset. PRETTYSMART flags 17,669 smart contracts with potentially controllable call statements (as opposed to 32,582 CANDIDATE contracts reported by OSPREY). We manually inspect 100 of the flagged contracts and find that 92 are false positives. For 78 of these false positive contracts, the `transferFrom` "from" parameter (the victim's address) is equal to the attacker's address – that is, the attacker could only exploit themselves. For 5 of these contracts, the `transferFrom` call is guarded by an access control policy. For 9 contracts, the attack is unfeasible due to other logic checks – for example, `transferFrom` can only be executed after creating an "order." The remaining 8 detections are true positives. We confirm that OSPREY also identifies these contracts as potentially vulnerable.

Jackal. JACKAL uses static taint analysis to flag potentially controllable calls, and a combination of symbolic execution and concrete execution to synthesize valid inputs (calldata) that execute the target call. This allows JACKAL to verify that the target call is reachable (dynamically) through concrete execution. This difference sets JACKAL apart from other purely static approaches such as PRETTYSMART. Nonetheless, JACKAL does not model ACT vulnerabilities and instead implements a more generic analysis to detect function calls where every argument is attacker-controllable.

First, we evaluate JACKAL's static analysis reports, which we consider comparable to the results of our stage ❷. Since the source code of JACKAL is publicly available, we modify JACKAL's analysis to use the same time limits reported in Section 5. Then, we run this analysis against all APPROVED contracts in our dataset. JACKAL flags 2,214 smart contracts with potentially controllable call statements (as opposed to 32,582 CANDIDATE contracts reported by OSPREY). Similar to PRETTYSMART, the authors do not apply any filtering to identify APPROVED contracts (❶) – or to restrict their reports to `transferFrom` calls.

Second, we evaluate JACKAL's dynamic analysis reports, which we consider comparable to the results of our stage ❸. JACKAL dynamically confirms the reachability of the call for 325 out of 2,214 reported CANDIDATE contracts. Importantly, JACKAL does not verify that the `transferFrom` was executed successfully. For this reason, we consider the severity of these reports equivalent to OSPREY's w_2 warnings.

In summary, we show that OSPREY is significantly more effective in detecting ACT vulnerabilities. The reasons are twofold. First, OSPREY analyzes all possible paths from a contract's entry point to the candidate `CALL` to assess whether it can be controlled. Instead, JACKAL only analyzes a single representative path for each candidate call. Second, OSPREY does not only look for fully controllable `CALL`s, but instead looks for any degree of controllability that would allow an attacker to execute an ACT attack.

5.6 Other Standards

The conceptual approach presented in this paper is applicable to arbitrary token standards with approvals (e.g., ERC721, ERC1155), arbitrary approval mechanisms (e.g., Permit2), and arbitrary EVM-compatible blockchains (e.g., Binance). To demonstrate this, we evaluate OSPREY against Permit2 approvals and ERC721 approvals.

Permit2. There are some notable differences between Permit2 and the classic approval mechanism. First, when using Permit2, a user does not directly approve a contract as

the designated spender for a token. Instead, the user first approves the Permit2 contract as the designated spender and then approves other contracts to move their tokens *through the Permit2 contract* – by calling `Permit2.approve`. Second, the signature of `Permit2.transferFrom` differs from the respective ERC20 function and takes an additional argument to specify the desired token. With the following considerations, our approach works the same on Permit2 approvals as it does for traditional approvals: (1) The approval filter (❶) must observe approvals directed to the `Permit2` contract. (2) The symbolic analysis (❷/❸) must constrain the target of the `transferFrom` call to be the Permit2 contract and verify that the additional argument "token" is a valid ERC20 token. (3) The procedure used to create synthetic victim users must approve the Permit2 contract.

We run our analysis pipeline to detect and exploit controllable `Permit2.transferFrom` function calls. We leverage the `Approval` logging mechanism to observe and collect 95,485 `Approval` events. Then, we study these events and identify 1,178 APPROVED contracts. We run our off-chain analysis (❷) on all APPROVED contracts, and report 86 CANDIDATE contracts with a symbolically controllable `transferFrom` function call. Finally, we run our on-chain analysis (❸) on all reported CANDIDATE contracts and identify one VULNERABLE contract – with a controllable `transferFrom` call. Upon manual verification, we confirm that the contract is indeed vulnerable at the time of writing.

ERC721. ERC721 is a widely adopted standard for non-fungible tokens (i.e., NFTs) on Ethereum. The standard implements an approval mechanism equivalent to the ERC20 approval mechanism. More precisely, ERC721 implements a function `approve(spender, tokenid)` that sets the allowance for the NFT `tokenid` and a function `setApprovalForAll` that sets the allowance simultaneously for all NFTs in a collection. The function `ERC721.transferFrom` transfers the NFT `tokenid` only if `msg.sender` was previously approved. Additionally, ERC721 implements a `safeTransferFrom` function that only transfers the NFT if the receiver can handle ERC721 transfers – i.e., if it implements the function `onERC721Received`.

We run our analysis pipeline to detect and exploit controllable `ERC721.transferFrom` function calls. First, we observe and collect 32,056,048 approval events.Then, we automatically study these events and identify 1,143,747 APPROVED contracts. We run our off-chain analysis (❷) on all APPROVED contracts and find 23,569 CANDIDATE contracts with a symbolically controllable `transferFrom` function call. We run our on-chain analysis (❸) on all the reported CANDIDATE contracts, and find 18 VULNERABLE contracts with a controllable `transferFrom`.

We manually confirm that 17 of the 18 vulnerable contracts are indeed vulnerable in practice and are all exploitable at the time of writing – that is, with active user approvals. One of the vulnerable contracts is not exploitable because it constrains (forces) the "from" address to be equal to the "to" address, thus rendering the attack pointless.

We also report 116 high-confidence (w_1) warnings and 219 low-confidence (w_2) warnings. We manually inspect a random sample of 50 warnings from each of these categories to assess the quality of our results. For 38 out of 50 (76%) of the high-confidence warnings, we determine that the contract is indeed vulnerable. For example, in some cases the attack calldata was simply misaligned because of symbolic analysis imprecisions. For the remaining 12 out of 50 (24%) w_1 warnings, we determine that the contract is likely not exploitable. For example, in some cases the attack requires payment of Ether in exchange for the NFT – effectively purchasing the NFT. For 8 out of 50 (16%) of the low-confidence warnings, we determine that the contract is indeed vulnerable. For example, some attacks require setting a public storage variable beforehand. For the remaining 42 out of 50 (84%) w_2 warnings, we determine that the contract is likely not exploitable. In most of these cases, the attack requires exchanging Ether or other assets as a form of payment for the NFT.

The outcome of our manual verification aligns with the results discussed in Section 5.2, and further demonstrates the effectiveness of our warning tiers in predicting likely exploitable contracts.

6 Discussion and Limitations

Access Control. In this work, we consider only victim contracts that can be called by an arbitrary blockchain user (i.e., we discard CANDIDATE contracts that include some form of access control). However, it may still be possible for an advanced attacker to craft an exploit against these contracts.

Smart contracts use access control policies to establish a group of "trusted accounts" that are allowed to execute privileged functionality. A trusted account can be either a smart contract or a blockchain user. Sometimes, it is possible for an attacker to circumvent this mechanism. For example, the attacker might be able to manipulate the contract's internal state to escalate their own privileges. Alternatively, it might be possible to compromise one of the trusted accounts, and consequently gain access to the contract's privileged functionality using the trusted account's identity. Researchers have proposed several techniques to identify sensitive functions with missing access control checks [6, 21], manipulate the contract's state to add new privileged users [60], and confuse a trusted account (contract) to execute privileged functionality on the attacker's behalf [23, 60]. In summary, an attacker could find a way (e.g., using a second vulnerability) to bypass an access control policy and exploit an ACT vulnerability that we currently consider unreachable.

Synthetic Accessory Contracts. When the address of an accessory contract is controllable, we leverage symbolic execution to automatically synthesize the contract, allowing our analysis to progress (see Section 4.2). OSPREY operates under

the assumption that accessory contracts are invoked via CALL instructions rather than STATICCALL. This allows for stateful interactions, that is, the accessory contract might respond differently when called multiple times. OSPREY assumes that stateful interactions are always possible and automatically simulates these interactions to verify the exploit. On the other hand, when a contract is invoked via a STATICCALL, it cannot make changes to its state. To support stateful interactions in such a setting, one would need to incorporate other mechanisms such as measuring GAS consumption to automatically differentiate between multiple STATICCALL invocations. We defer the implementation of advanced synthetic accessory contracts as a promising avenue for future work.

Other Accessory Contracts. When the *address* of the accessory contract is not controllable, it might still be possible to craft an *input* (for the accessory contract) that executes the desired behavior. For example, symbolic execution could be used to study the accessory contract and derive an appropriate input that furthers our analysis. However, symbolically exploring all accessory contracts is computationally demanding and would limit the scalability of our approach. Hence, we defer this to future work.

7 Related Work

Controllable Calls. PRETTYSMART [60] employs a static taint analysis to flag potential controllable calls. However, it cannot verify the feasibility of the associated execution paths (i.e., it lacks automatic synthesis of example attack transactions). AVVERIFIER [50] also uses static analysis to detect improper sanitization of the *destination address* of an outbound call. In contrast, the main objective of our work focuses on the degree of control over the *arguments* of the outbound call (in addition to the destination address). JACKAL [23] incorporates symbolic and concrete execution and, thus, can validate its findings, similar to OSPREY. Despite that, JACKAL seeks to detect controllable calls generically, whereas OSPREY is specialized to find ACT vulnerabilities. We compare our work against both JACKAL and PRETTYSMART in Section 5. SECURIFY 2.0 [53] also employs static analysis to detect unsafe calls, but only reports unsafe DELEGATECALL instructions, which is not the focus of this paper. Several other works (SECURIFY [54], MAIAN [41], and TEETHER [26]) identify "stealing Ether" vulnerabilities: where an attacker could cause a smart contract to execute a CALL with nonzero value (thus, transferring the native currency, Ether). This is different from our work, as we analyze the complex arguments to a call (not just the basic value) to understand whether the degree of control is sufficient to trigger the approved ERC20 token transfer (not a native Ether transfer). CONFUZZIUS [52], a smart contract hybrid fuzzer, detects unsafe DELEGATECALL instructions, but only does so with naive invariants, and does not consider the degree of control over the parameters themselves.

Access Control. ACHECKER [21] detects access control vulnerabilities by combining static analysis (which identifies control variables) and symbolic execution (which searches for paths bypassing checks over these control variables). ETHAINTER [6] focuses on static taint propagation to detect access control on paths to selfdestruct in publicly callable functions. Both ACHECKER and ETHAINTER generate warnings that require manual validation. Unfortunately, neither of these techniques can detect all forms of access control patterns. Our access control detection heuristic is simple, yet effective enough to suit our needs.

ERC20 Bugs. CPMM-EXPLOITER [25], a grammar-based fuzzer, identifies ERC20 tokens that allow an attacker to manipulate balances of other accounts unexpectedly (either increasing or decreasing) in order to perform DeFi exploits. These vulnerabilities are outside the scope of this work, which focuses on ERC20 token approvals. CLOCKWORK FINANCE FRAMEWORK [4] is a model-checking framework that synthesizes arbitrary money-making attacks against DeFi devices, including ERC20 tokens. However, each involved smart contract must be manually translated into its specification language, which does not scale.

Non-ACT Vulnerabilities. Industry-standard tools such as MYTRHIL [9] are widely used by professional smart contract auditors. Ma et al. [30] proposed a symbolic analysis method to explore inter-contract control-flow graphs and detect classic bugs like re-entrancy and arithmetic issues. Frank et al. introduced ETHBMC [19], a bounded model checker supporting symbolic execution and automatic exploit generation for smart contract bytecode, even with external interactions. Fuzzing-based systems have also been developed, such as XFUZZ [58], which uses machine learning on the contract's source code to filter benign cross-contract execution paths, and EF/CF [47], which requires only contract bytecode.

8 Conclusions

In this paper, we introduce a novel system, OSPREY, to detect Approved Controllable TransferFrom (ACT) vulnerabilities in Ethereum smart contracts. Our system uses a combination of static analysis, symbolic execution, and concrete execution to (1) enable a deep understanding of on-chain token approvals, (2) study the constraints at the call site to determine the degree of control that the attacker has over the execution, and (3) automatically generate semantically correct interactions to exploit the ACT vulnerability.

We evaluate OSPREY against 424,676 Ethereum smart contracts with user approvals. OSPREY identifies 32,582 contracts with potential ACT vulnerabilities, and successfully crafts proof-of-concept attacks for 410 of these contracts. The latent financial impact of our (previously unknown) automatically generated exploits exceeds 3.4 million USD.

Acknowledgments

This material is based upon work supported by NSF under Award No. CNS-2334709. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the NSF.

A Approvals Among Spenders

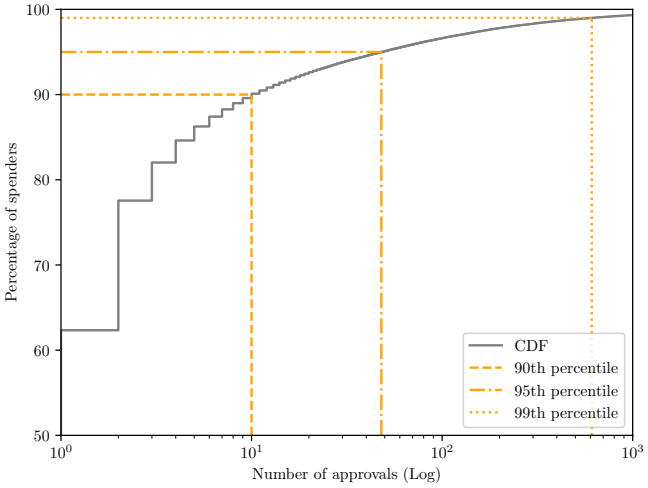

Figure 6: Cumulative distribution of number of approvals (log) among spenders (percentage).

B Approvals Over Time

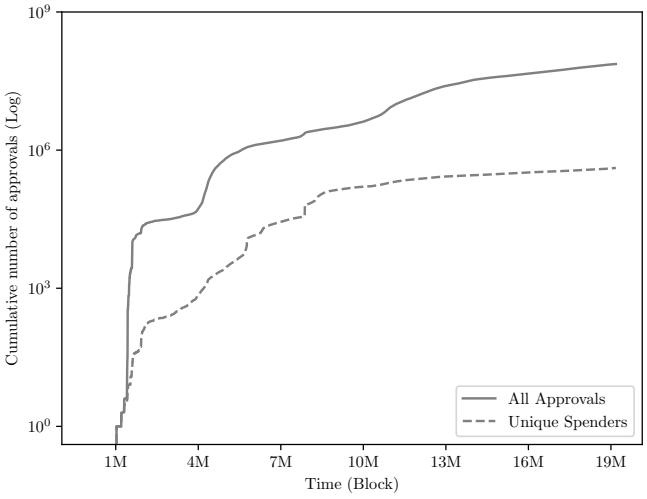

Figure 7: Cumulative number of approvals (log) over time (block).

C Live Evaluation

To further evaluate OSPREY 's practicality, we deployed it to continuously monitor the live Ethereum blockchain. In this setup, we run every stage of OSPREY 's analysis on a single server with the same configuration discussed in Section 5. In practice, we observed both minimal processing delay – that is, each block is processed as soon as it is synced to our local client – and minimal resource consumption – that is, the observed CPU load is below 10% on average, the observed RAM memory usage is below 50GB on average.

Over the course of two weeks of continuous monitoring, OSPREY automatically generated one new exploit, produced two new high-confidence warnings, and identified additional at-risk approvals in contracts previously flagged as vulnerable. Our results demonstrate OSPREY 's ability to scale to real-time workloads while maintaining the in-depth analysis required to detect ACT vulnerabilities at scale.

Ethics Considerations

Our study targets Approved Controllable TransferFrom (ACT) vulnerabilities in Ethereum smart contracts and does not involve human subjects. All our experiments were conducted on a private blockchain fork to avoid any potential harm to existing real-world systems. Similarly, all the discovered exploits have been verified exclusively on our private blockchain fork, and never on the public Ethereum mainnet. To prevent malicious use of our findings, all the vulnerabilities reported in this submission have been anonymized, and no identifiable information about affected contracts (or users) is disclosed.

In compliance with responsible disclosure practices, we reported all confirmed vulnerabilities to the Cybersecurity and Infrastructure Security Agency [8], giving stakeholders the opportunity to address these flaws before any public release of our system. Our actions align with the principles of beneficence and respect for the law and public interest outlined in the Menlo Report [5], as our work aims to enhance security in the broader decentralized finance (DeFi) ecosystem without causing harm to individuals or organizations. By demonstrating the feasibility of exploits solely in a controlled setting and by alerting the relevant authorities, we have taken all reasonable measures to ensure ethical conduct throughout this research.

Open Science

In the spirit of transparency and reproducibility, we will release our system and supporting dataset to the community [3, 4]. Any sensitive details pertaining to the identified vulnerabilities will remain anonymized, and we will include instructions and necessary scripts to replicate our analyses in a manner that poses minimal risk to the affected contracts.

[3] https://github.com/ucsb-seclab/osprey
[4] https://zenodo.org/records/15599087

References

[1] 1inch Network. Bancor network hack. https://medium.com/1inch-network/bancor-network-hack-2020-3c71444fd59d, 2020.

[2] Aave Protocol. Aave. https://aave.com/, 2024.

[3] ApeX. ApeX Protocol. https://www.apex.exchange/, 2024.

[4] Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. Clockwork finance: Automated analysis of economic security in smart contracts. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2499–2516. IEEE, 2023.

[5] M. Bailey, E. Kenneally, D. Maughan, and D. Dittrich. The menlo report. pages 71–75, Los Alamitos, CA, USA, mar 2012. IEEE Computer Society.

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

[7] Circle. Fully backed digital dollars. https://www.circle.com/en/usdc, 2024.

[8] CISA. Cybersecurity and infrastructure security agency. https://www.cisa.gov, 2024.

[9] ConsenSys. Mythril. https://github.com/ConsenSys/mythril, 2022.

[10] Consensys, Inc. What is a token approval? https://support.metamask.io/es/transactions-and-gas/transactions/what-is-a-token-approval/, 2024.

[11] DefiLlama. Ethereum - defillama. https://defillama.com/chain/Ethereum, 2024.

[12] DefiLlama. Hacks - defillama. https://defillama.com/hacks, 2024.

[13] dYdX. Deposit contract post mortem. https://dydx.exchange/blog/deposit-proxy-post-mortem, 2021.

[14] Ethereum. Erc-1155 token standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-1155/, 2023.

[15] Ethereum. Erc-721 token standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-721/, 2023.

[16] Ethereum. Erc-20 token standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-20/, 2024.

[17] Ethereum. Erc-4626 token standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-4626/, 2024.

[18] Ethereum. Ethereum. https://ethereum.org/en/, 2024.

[19] Joel Frank, Cornelius Aschermann, and Thorsten Holz. Ethbmc: A bounded model checker for smart contracts. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 2757–2774, 2020.

[20] Gelato Network. Sorbet finance vulnerability post mortem. https://medium.com/gelato-network/sorbet-finance-vulnerability-post-mortem-6f8fba78f109, 2022.

[21] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. Achecker: Statically detecting smart contract access control vulnerabilities. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 945–956. IEEE, 2023.

[22] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. Elipmoc: advanced decompilation of ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27, 2022.

[23] Fabio Gritti, Nicola Ruaro, Robert McLaughlin, Priyanka Bose, Dipanjan Das, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. Confusum contractum: confused deputy vulnerabilities in ethereum smart contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1793–1810, 2023.

[24] Halborn. Explained: The stablemagnet rug-pull. https://www.halborn.com/blog/post/explained-the-stablemagnet-rugpull-june-2021, 2021.

[25] Sujin Han, Jinseo Kim, Sung-Ju Lee, and Insu Yun. Automated attack synthesis for constant product market makers, 2024.

[26] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, 2018.

[27] ledgerwatch. Erigon. https://github.com/ledgerwatch/erigon, 2024.

[28] Li.Finance. Li.fi smart contract vulnerability post mortem. https://blog.li.fi/20th-march-the-exploit-e9e1c5c03eb9, 2022.

[29] Martin Lundfall. Permit extension for eip-20 signed approvals. https://eips.ethereum.org/EIPS/eip-2612, 2020.

[30] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. Pluto: Exposing vulnerabilities in inter-contract scenarios. *IEEE Transactions on Software Engineering*, 2021.

[31] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, 2011.

[32] Neptune Mutual. Decoding brahtopg smart contract vulnerability. https://neptunemutual.com/blog/decoding-brahtopg-smart-contract-vulnerability/, 2022.

[33] Neptune Mutual. Decoding rabby's smart contract vulnerability. https://neptunemutual.com/blog/decoding-rabbys-smart-contract-vulnerability, 2022.

[34] Neptune Mutual. Decoding transit finance's contract vulnerability. https://neptunemutual.com/blog/decoding-transit-finances-contract-vulnerability/, 2022.

[35] Neptune Mutual. How was rubic protocol hacked? https://neptunemutual.com/blog/how-was-rubic-protocol-hacked/, 2022.

[36] Neptune Mutual. How was hashflow exploited? https://neptunemutual.com/blog/how-was-hashflow-exploited/, 2023.

[37] Neptune Mutual. Taking a closer look at unibot exploit. https://neptunemutual.com/blog/taking-a-closer-look-at-unibot-exploit/, 2023.

[38] Neptune Mutual. Understanding the maestro exploit. https://neptunemutual.com/blog/understanding-the-maestro-exploit/, 2023.

[39] Neptune Mutual. How was seneca protocol exploited? https://neptunemutual.com/blog/how-was-seneca-protocol-exploited/, 2024.

[40] Neptune Mutual. How was socket protocol exploited? https://neptunemutual.com/blog/how-was-socket-protocol-exploited/, 2024.

[41] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 653–663, New York, NY, USA, 2018. Association for Computing Machinery.

[42] Peckshield. Dexible hack. https://x.com/peckshield/status/1626493024879673344, 2023.

[43] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*, pages 1661–1677. IEEE, 2020.

[44] Rarible, inc. Rarible. https://rarible.com/, 2024.

[45] Revert. An attack on v3utils. https://mirror.xyz/revertfinance.eth/3sdpQ3v9vEKiOjaHXUi3TdEfhleAXXlAEWeODrRHJtU, 2023.

[46] Revoke.cash. Revoke. https://revoke.cash, 2024.

[47] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi. Ef/cf: High performance smart contract fuzzing for exploit generation. *arXiv preprint arXiv:2304.06341*, 2023.

[48] Skrice Studios. Heroes of Mavia. https://www.mavia.com/, 2024.

[49] Solidity Team and Ethereum Foundation. Solidity programming language. https://soliditylang.org/, 2024.

[50] Tianle Sun, Ningyu He, Jiang Xiao, Yinliang Yue, Xiapu Luo, and Haoyu Wang. All your tokens are belong to us: Demystifying address verification vulnerabilities in solidity smart contracts. In *The 33rd USENIX Security Symposium*, August 2024.

[51] Ole Tange. Gnu parallel-the command-line power tool. *Usenix Mag*, 36(1):42, 2011.

[52] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 103–119. IEEE, 2021.

[53] Petar Tsankov. Securify 2.0. https://github.com/eth-sri/securify2, 2020.

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

[55] UCSB SecLab. greed. https://github.com/ucsb-seclab/greed, 2024.

[56] Uniswap. Uniswap info. https://v2.info.uniswap.org/home, 2024.

[57] Uniswap Labs. Introducing permit2 & universal router. https://blog.uniswap.org/permit2-and-universal-router, 2022.

[58] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2022.

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

[60] Zhijie Zhong, Zibin Zheng, Hong-Ning Dai, Qing Xue, Junjia Chen, and Yuhong Nan. Prettysmart: Detecting permission re-delegation vulnerability for token behaviors in smart contracts. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.