BlockScope: Detecting and Investigating Propagated Vulnerabilities in Forked Blockchain Projects

Xiao Yi[1], Yuzhou Fang[1], Daoyuan Wu[1]*, and Lingxiao Jiang[2]
[1]The Chinese University of Hong Kong
[2]Singapore Management University

***Abstract*—Due to the open-source nature of the blockchain ecosystem, it is common for new blockchains to fork or partially reuse the code of classic blockchains. For example, the popular Dogecoin, Litecoin, Binance BSC, and Polygon are all variants of Bitcoin/Ethereum. These "forked" blockchains thus could encounter similar vulnerabilities that are propagated from Bitcoin/Ethereum during forking or subsequently commit fetching. In this paper, we conduct a systematic study of detecting and investigating the propagated vulnerabilities in forked blockchain projects. To facilitate this study, we propose BlockScope, a novel tool that can effectively and efficiently detect multiple types of cloned vulnerabilities given an input of existing Bitcoin/Ethereum security patches. Specifically, BlockScope adopts similarity-based code match and designs a new way of calculating code similarity to cover all the syntax-wide variant (i.e., Type-1, Type-2, and Type-3) clones. Moreover, BlockScope automatically extracts and leverages the contexts of patch code to narrow down the search scope and locate only potentially relevant code for comparison.**

Our evaluation shows that BlockScope achieves good precision and high recall both at 91.8% (1.8 times higher recall than that in the state-of-the-art ReDeBug while with close precision). BlockScope allows us to discover 101 previously unknown vulnerabilities in 13 out of the 16 forked projects of Bitcoin and Ethereum, including 16 from Dogecoin, 6 from Litecoin, 1 from Binance BSC, and 4 from Optimism. We have reported all the vulnerabilities to their developers; 40 of them have been patched or accepted, 66 were acknowledged or under pending, and only 4 were rejected. We further investigate the propagation and patching processes of discovered vulnerabilities, and reveal three types of vulnerability propagation from source to forked projects, as well as the long delay (mostly over 200 days) for releasing patches in Bitcoin forks (vs. ∼100 days for Ethereum forks).

I. Introduction

Blockchain [67] and DeFi (Decentralized Finance) [79] are emerging in recent years. A good development in the blockchain ecosystem is that many projects are open-source. This is particularly true for the public blockchains like Bitcoin and Ethereum. As a result, new blockchains could fork or partially reuse the code of classic blockchains to speed up the development. Notably, Bitcoin is the one with most forked projects — the popular Dogecoin, Litecoin, Dash, Zcash, and Bitcoin Cash/SV are all variants of Bitcoin. In recent years, Ethereum was also forked by a number of EVM (Ethereum Virtual Machine)-compatible chains, such as Binance Smart Chain (BSC), Polygon, Avalanche Contract Chain, and Optimism (Ethereum's Layer-2 rollup network).

However, "forked" blockchains could encounter similar vulnerabilities that appeared in the code of Bitcoin and Ethereum. Specifically, a vulnerability could be propagated from Bitcoin/Ethereum to the forked projects during the initial fork or subsequently when updated commits are fetched from Bitcoin/Ethereum. In this paper, we aim to systematically detect cloned vulnerabilities in forked blockchain projects and investigate how they are propagated and patched.

To facilitate this study and future analysis, we propose BlockScope, a novel tool that can not only automatically detect vulnerable clones but also pinpoint the cases already fixed and their patching process information. To achieve effective and efficient detection on all the syntax-wide cloned vulnerabilities (i.e., Type-1, Type-2, and Type-3 clones, as to be defined in Sec. II-C), BlockScope has two unique designs as compared to typical code clone detection tools, e.g., [43], [47], [50], [57], [68], [82]. First, we adopt similarity-based code match, instead of the hash-based exact match in ReDeBug [43], VUDDY [50], and MVP [82], so that BlockScope is more tolerant to the code lines with no exact "abstracted" hashes. Moreover, we design a new way of calculating code similarity to better handle the code fragments with inserted/deleted/reordered code lines. According to our evaluation with the state-of-the-art ReDeBug tool, our new design greatly reduces false negatives while only slightly increasing false positives for our problem. Second, BlockScope automatically extracts and leverages patch code contexts to locate only potentially relevant code for comparison. This not only dramatically improves the running performance for large projects, e.g., 15.4 times faster than ReDeBug in analyzing Ethereum's forked projects with more lines of code (LOC), but also enhances the detection precision because the context similarity is also being considered.

To evaluate BlockScope, we collect a dataset of 38 security patches — 32 of them are directly from Bitcoin's repository because there were only four CVEs in the recent five years, and the rest six are CVEs of Ethereum reported in the last three years. With this input, we apply BlockScope and ReDebug to test 11 most popular forked projects of Bitcoin and 5 of Ethereum (identified from nearly the top 100 cryptocurrencies), with 4.2M C/C++ LOC and 3.5M Go LOC, respectively. The evaluation shows that BlockScope detects 101 true vulnerabilities in all the 13 forked projects (three projects, Qtum, Avalanche, and Polygon, does not contain any of the tested vulnerabilities), whereas ReDeBug detects

*Corresponding author.

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24222
www.ndss-symposium.org

only 57 vulnerabilities in 11 forked projects. By performing a thorough code review of all the raw detection results, we find that BlockScope achieves good precision and high recall both at 91.8%, whereas ReDeBug's recall is only 51.8% despite its precision at 95%. Among the 101 vulnerabilities automatically detected by BlockScope, we are able to identify serious ones from the top blockchains like Dogecoin, Litecoin, Bitcoin SV, Binance BSC, and Optimism. This demonstrates the real-world impact of our work[1].

We further investigate how the discovered vulnerabilities[2] are propagated from Bitcoin/Ethereum to their forked projects and understand the patching processes of the 138 cases that were already fixed in forked projects before our detection. Specifically, we reveal three types of vulnerability propagation from Bitcoin/Ethereum to their forked projects, including the cases directly forked in the beginning, fetched from vulnerable commits, and infected with no explicitly vulnerable commits. Besides vulnerability propagation, we additionally identify three other propagation that caused false positives and negatives in BlockScope; details in Sec. V-B. As for patch delays, we find that only DigiByte, among the six forked projects of Bitcoin with enough patched cases, can catch up with Bitcoin's patch release schedule. The patch delays for the other five are typically long, mostly over 200 days. Compared with Bitcoin, the result for Ethereum's forked projects is relatively acceptable, with half of the patches released within 100 days.

Contributions. To sum up, we make the following major contributions in this paper:

- *(Methodology)* We propose novel patch-based clone detection for vulnerable code clones in forked projects, in which we design (i) a context-based search with similarity measurement to efficiently locate candidate code clones and (ii) a new way of calculating the similarity between two code fragments that is immune to Type-1/2/3 clones.
- *(Detection)* We apply this methodology to detect 101 previously unknown vulnerabilities in the forked projects of Bitcoin and Ethereum with high precision and recall.
- *(Investigation)* We further conduct a deep investigation of the vulnerability propagation and patching processes of the discovered vulnerabilities, and reveal new findings.

Ethics. As an ethical research and one contribution of this paper, we have spent significant efforts reporting all the 110 vulnerabilities (including nine false negatives manually identified during the evaluation). The details are available in Sec. IV-D and this GitHub repository, https://github.com/VPRLab/BlkVulnReport.

Roadmap. The rest of this paper is organized as follows. After explaining different blockchain projects and code clone types in Sec. II, we first propose the BlockScope tool in Sec. III to effectively detect the propagated vulnerabilities in the forked blockchains. We then evaluate the accuracy and performance of BlockScope and leverage it to discover previously unknown vulnerabilities in Sec. IV. We further analyze how the discovered vulnerabilities are propagated from Bitcoin and Ethereum to the forked projects and understand their patching processes in Sec. V. We then discuss some insights and implications in Sec. VI. Lastly, Sec. VII reviews the related work and Sec. VIII concludes the paper.

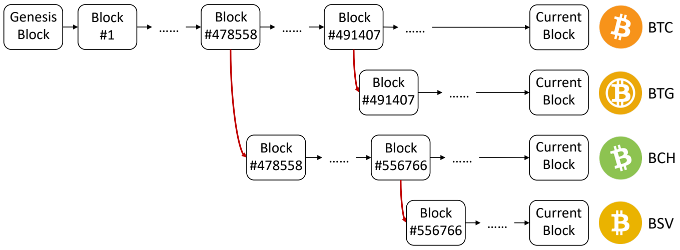


Fig. 1: Representative hard forks of Bitcoin.

TABLE I: The basic information of Bitcoin, Ethereum, and their popular forked projects.

(a) Bitcoin and its forked projects (as of 7 September 2021).

#	Name	Code	Market Cap	Repository	Star
1	Bitcoin	BTC	$749.70B	bitcoin/bitcoin	60.3K
6	Dogecoin	DOGE	$42.55B	dogecoin/dogecoin	13.6K
11	Bitcoin Cash	BCH	$12.02B	Bitcoin-ABC/bitcoin-abc	1.1K
12	Litecoin	LTC	$11.88B	litecoin-project/litecoin	4K
33	Bitcoin SV	BSV	$3.24B	bitcoin-sv/bitcoin-sv	520
55	Dash	DASH	$1.79B	dashpay/dash	1.4K
59	Zcash	ZEC	$1.64B	zcash/zcash	4.5K
75	Bitcoin Gold	BTG	$1.04B	BTCGPU/BTCGPU	611
79	Horizen	ZEN	$935.27M	HorizenOfficial/zen	202
80	Qtum	QTUM	$923.88M	qtumproject/qtum	1.1K
83	DigiByte	DGB	$868.91M	digibyte/digibyte	361
100	Ravencoin	RVN	$693.34M	RavenProject/Ravencoin	932

(b) Ethereum and its forked projects (as of 6 June 2022).

#	Name	Code	Market Cap	Repository	Star
2	Ethereum	ETH	$229.87B	ethereum/go-ethereum	37.7K
5	Binance	BNB	$50.69B	bnb-chain/bsc	1.6K
14	Avalanche	AVAX	$7.65B	ava-labs/subnet-evm	1.6K
17	Polygon	MATIC	$5.15B	maticnetwork/bor	400
78	Celo	CELO	$604.02M	celo-org/celo-blockchain	382
199	Optimism	OP	$263.36M	ethereum-optimism/optimism	1.2K

II. Background

In this section, we first introduce the background of Bitcoin, Ethereum, and their popular forked projects in Sec. II-A and Sec. II-B, and then provide the definition of different code clone types in Sec. II-C.

A. Bitcoin and its Forked Projects

Bitcoin (BTC) [61] is the first cryptocurrency that introduced the blockchain technology to the world. Bitcoin leverages blockchain as a distributed ledger to guarantee the consensus between different peers. Currently, Bitcoin is, without doubt, the dominant cryptocurrency, whose market capitalization takes around 40% of the whole market. Since Bitcoin is open-sourced, it has nourished many blockchain projects. Specifically, among the top 100 cryptocurrencies on CoinMarketCap [15] as of 7 September 2021, we identified that 11 projects directly fork or partially reuse the code of Bitcoin. We list them in Table Ia and refer to them as Bitcoin's forked projects in this paper.

Most forked projects forked only the Bitcoin code, whereas **Bitcoin Cash** (BCH), **Bitcoin SV** (BSV), and **Bitcoin Gold** (BTG) also forked Bitcoin's blockchain, i.e., copying its transaction history, as the basis for their own blockchain [41]. They

[1] Binance acknowledged our vulnerability report with a bug bounty reward.

[2] Besides 101 automatically detected cases, we also analyzed 9 that were false negatives in BlockScope but manually identified during the evaluation.

are known as the "hard forks" of Bitcoin, as each of them creates a permanent fork of the original Bitcoin's blockchain. We present the relationship between Bitcoin and these three projects in Fig. 1. As we can see, Bitcoin Cash is the earliest fork, which aims to reduce the transaction fee and improve the transaction speed of the original Bitcoin. Therefore, they extend the maximum block size to 32MB, while the original Bitcoin's block size limit is 1MB. Bitcoin SV further extends this limit to 128MB, which leads to another hard fork. Bitcoin Gold, on the other hand, claims to solve the original Bitcoin's monopolized mining problem. Specifically, they hope that by enabling mining on commonly available GPUs instead of specialized ASICs, it can democratize and decentralize the mining.

Litecoin gets its name from "the light version of Bitcoin". Its goal is to provide faster transactions than Bitcoin. Notably, instead of using Bitcoin's SHA-256, Litecoin adopts Scrypt [23] as the hash function, which offers a less compute-intensive but more memory-intensive mining process [33]. **Dogecoin** also leverages Scrypt as the hash function. Indeed, it copies both Bitcoin's and Litecoin's code. Although Dogecoin reached a market capitalization of over 40 billion USD, it was initially created as a meme cryptocurrency with an unlimited total supply [16]. **DigiByte** is another fork of Litecoin's code. Besides SHA-256 and Scrypt, it can work with three more mining algorithms [25].

Dash is not only a cryptocurrency but also a decentralized autonomous organization run by a subset of its users called "masternodes". Specifically, anyone with 1,000 Dash can become a masternode in the Dash network and share the block reward. Besides the standard node functions, the masternodes can vote on proposals to improve the ecosystem and provide two additional kinds of transactions, i.e., "InstantSend" and "PrivateSend" for instant transactions and private transactions, respectively [19].

Zcash and **Horizen** are designed to enhance the privacy for their users. As the original Bitcoin is pseudo-anonymous, it is possible to decipher the patterns and connections involved, which may expose all information related to the sender and the receiver [62]. To tackle this problem, Zcash applies Zero-Knowledge proof algorithms (called zk-SNARKs) to "shield" the transactions so that it will not disclose the information about the coin holders. Similarly, Horizen (formerly known as ZenCash) is a derivative of Zcash. On top of the zk-SNARKs system, Horizen adopts a different funding model, which shares the block reward among miners, developers, and secure/super node operators, while Zcash just rewards miners and developers [62].

Qtum is a hybrid blockchain that combines the characteristics of Bitcoin and Ethereum. It introduces an Account Abstraction Layer to integrate Bitcoin's Unspent Transaction Output model with the Ethereum Virtual Machine for smart contracts to operate [74]. Besides, Qtum adopts Proof-of-Stake (PoS) consensus mechanism instead of Bitcoin's Proof-of-Work (PoW) to simplify the mining process since PoW is resource-intensive, i.e., it wastes enormous amounts of electricity on mining coins [37].

Ravencoin is unique in terms of that it was designed for users to tokenize assets on-chain and transfer ownership via blockchain transactions [34]. Such assets can be physical or digital, including gold, in-game items, copyrights, etc [71].

B. *Ethereum and its Forked Projects*

Ethereum [80] is the first blockchain system with the capability of constructing Turing-complete *smart contracts*, which contain a set of pre-defined rules and regulations for self-execution. Ether (ETH) is the native cryptocurrency for maintaining the operations on Ethereum, which is the second largest cryptocurrency with a market capitalization of around 230 billion USD as of June 2022. As an open-sourced project, Ethereum also nourished many blockchain projects. Specifically, we analyzed all the projects listed on Blockscan [10] and selected five of the most popular projects that directly fork or partially reuse the code of Ethereum. Table Ib presents the basic information of these forked projects as of 6 June 2022.

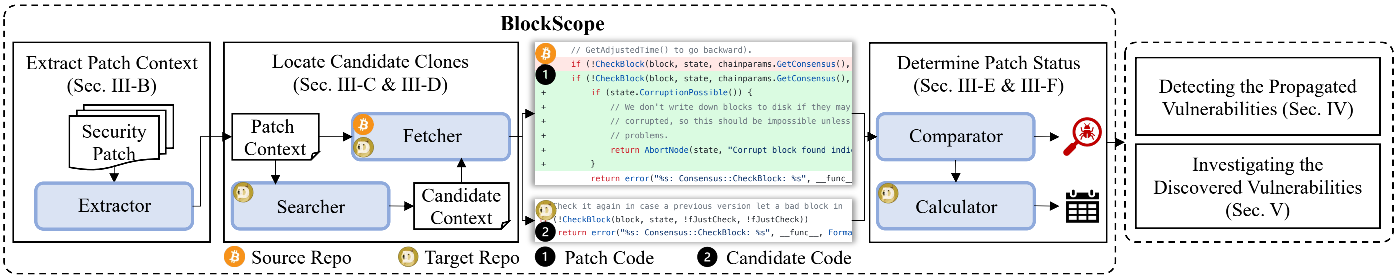
Binance is the largest cryptocurrency exchange in the world. As of 27 July 2022, its 24-hour trading volume reaches 11.7 billion USD [13]. Originally, Binance developed Binance Chain to provide a marketplace for trading cryptocurrency in a decentralized manner, with BNB being the native token. However, as Binance Chain is not EVM-compatible, users cannot develop decentralized applications (DApps) using smart contracts [11]. Binance initiated Binance Smart Chain (BSC) with EVM compatibility to solve this problem. On February 15, 2022, Binance Chain and Binance Smart Chain united into BNB Chain [18]. Currently, BNB Chain holds around 3.4 million transactions daily, with 2.0 million active wallets [14].

Avalanche aims to solve Ethereum's issues regarding transaction fee, scalability, and programmability, by leveraging a multi-chain approach [40]. Specifically, Avalanche combines three separate blockchain networks, i.e., X-Chain: for issuing digital assets, C-Chain: for converting Ethereum's DApps to Avalanche, and P-Chain: for validating the states of subnets. **Celo** is also EVM-compatible. Notably, it provides a client designed for mobile phone users. Moreover, while the transaction fee is paid with the native asset (ETH) on Ethereum, Celo allows users to pay transaction fees with the native asset (CELO) and stable coins (cUSD and cEUR) [35].

Polygon and **Optimism** are Ethereum's layer-2 networks, which also target on Ethereum's scalability and transaction fee issues. Layer-2 solutions refer to infrastructures or simple protocols built on top of the Ethereum main chain [21], i.e., layer-1. Typically, they handle off-chain transactions and send only compact data to layer-1. Polygon is technically a sidechain of Ethereum, as it uses its own consensus algorithms and runs in parallel with the main chain. However, different from sidechains, Optimism uses Optimistic Rollups [20] to interact with the main chain and use smart contracts that reside within Ethereum [24].

C. *Definition of Code Clone Types*

Due to the nature of open-source projects, it is common for projects to reuse parts of code from others. However, vulnerabilities are always reintroduced due to the casual code reuses, namely code clones. While code clone detections are widely studied among the famous open-source projects, e.g., Linux Kernel, detections for cloned vulnerabilities in the forked blockchain projects are much less explored. In


BlockScope
Extract Patch Context (Sec. III-B)
Security Patch
Extractor
Locate Candidate Clones (Sec. III-C & III-D)
Patch Context
Fetcher
Searcher
Candidate Context
Determine Patch Status (Sec. III-E & III-F)
Comparator
Calculator
Detecting the Propagated Vulnerabilities (Sec. IV)
Investigating the Discovered Vulnerabilities (Sec. V)
Source Repo
Target Repo
1 Patch Code
2 Candidate Code

```
    // GetAdjustedTime() to go backward).
    if (!CheckBlock(block, state, chainparams.GetConsensus(),
    if (!CheckBlock(block, state, chainparams.GetConsensus(),
+       if (state.CorruptionPossible()) {
+           // We don't write down blocks to disk if they may
+           // corrupted, so this should be impossible unless
+           // problems.
+           return AbortNode(state, "Corrupt block found indi
+       }
        return error("%s: Consensus::CheckBlock: %s", __func_
```

```
heck it again in case a previous version let a bad block in
(!CheckBlock(block, state, !fJustCheck, !fJustCheck))
return error("%s: Consensus::CheckBlock: %s", __func__, Forma
```

Fig. 2: The overall workflow of BlockScope and our study.

this study, it is essential to analyze the cloned code among the forked blockchain projects. Therefore, we adopt the type definitions of code clones from [59] as follows:

- Type-1 clones refer to two identical code fragments with variations in whitespaces, layouts, and comments.
- Type-2 clones include Type-1 clones and extend the variations to identifiers, literals, and types, e.g., variable renaming.
- Type-3 clones further extend these variations to syntactically similar code with inserted, deleted, or updated statements.
- Type-4 clones refer to semantically equivalent code fragments but syntactically different, which is out of the scope of this paper.

In this paper, we focus on the detection of Type-1, Type-2, and Type-3 code clones. Detecting Type-4 code clones requires code semantic learning or understanding, which is out of the scope of typical clone detection tools including BlockScope.

III. BlockScope

A. Design Choices and System Overview

To detect the propagated vulnerabilities from the existing security patches of Bitcoin/Ethereum, we design BlockScope as a patch-based code clone detection tool. This makes BlockScope, by nature, more similar to security-oriented clone detection tools (e.g., ReDeBug [43], VUDDY [50], MVP [82], and VGraph [29]) rather than the traditional clone detection tools (e.g., CCFinder [47], CPMiner [57], DECKARD [44], and SourcererCC [68]) that do not differentiate vulnerable and patched code inputs. Moreover, since we aim to test all different blockchain projects, we design BlockScope to be language-agnostic as similar to ReDeBug. As a result, we do not perform "program analysis-alike" preprocessing, such as variable/type/function abstraction in VUDDY, program slicing in MVP, and code property graph [83] in VGraph, before the similarity measurement between source and target code.

Besides the choices above, BlockScope offers two unique designs that are also the major novelty of our methodology:

- *Leveraging patch code contexts to search and locate only potentially relevant code.* Since our detection targets are the propagated vulnerabilities in the forked projects, it is reasonable to assume that they have similar contexts as the original patch code in the source repositories. BlockScope thus leverages the extracted patch code contexts to search for potentially relevant code in the target repositories and employs code similarity to finalize the contexts of candidate code clones. This not only helps BlockScope avoid the whole-repository analysis as in typical code clone detection tools but also improves the precision because the context similarity is also being considered.
- *Adopting similarity-based code match for being more tolerant to variant code clones.* To cover all the syntax-wide Type-1, Type-2, and Type-3 clones, we adopt similarity-based code match, instead of the hash-based exact code match in ReDeBug [43], VUDDY [50], and MVP [82]. This allows BlockScope to be more tolerant to the code lines with no exact "abstracted" hashes (i.e., Type-2 clones). Moreover, we design a new way of calculating code similarity to better handle the code fragments with inserted/deleted/reordered code lines (i.e., Type-3 clones).

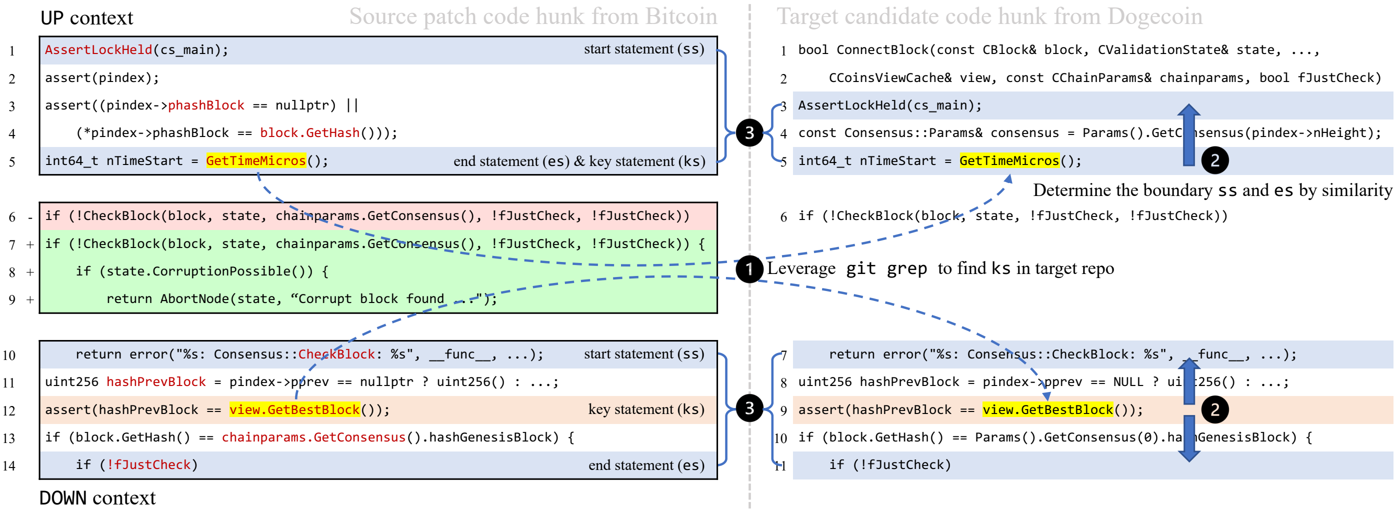
Fig. 2 presents the overall workflow of BlockScope in five major steps. Firstly, Sec. III-B describes how the `Extractor` component or `Extractor`[3] extracts the code contexts from patches in the source repositories. Secondly, in Sec. III-C, `Searcher` leverages the extracted patch contexts to search for candidate contexts in the target repositories. Thirdly, `Fetcher` in Sec. III-D retrieves the patch and candidate code hunks in the source and target repositories, respectively. Fourthly, `Comparator` in Sec. III-E employs a new similarity-based code matching technique to determine the propagated vulnerabilities from `Fetcher`'s outputs. Lastly, for the vulnerabilities already patched, `Calculator` in Sec. III-F measures their patch delays in the target repositories.

B. Extracting Patch Contexts from the Source Repositories

Given a security patch from the source project or code repository (e.g., Bitcoin/Ethereum), BlockScope first extracts its code context. In this paper, we provide an `Extractor` component to *automatically* extract the contexts of patch code and use its output for system evaluation. In reality, BlockScope also supports the manually crafted code contexts from security experts for better accuracy. To distinguish the context of patch code from that of target code, we call the former "*patch context*" and the latter "*candidate context*", as shown in Fig. 2.

Unlike ReDeBug that directly takes the entire part of the nearby code lines (after normalization and tokenization) as context, `Extractor` recognizes important variable and function names as the *context keywords* and uses these keywords to search for candidate contexts in the target repositories (as

[3] We describe different BlockScope components using their names, e.g., `Extractor`, hereafter.


UP context
Source patch code hunk from Bitcoin
Target candidate code hunk from Dogecoin
start statement (ss)
end statement (es) & key statement (ks)
start statement (ss)
key statement (ks)
end statement (es)
DOWN context
3
3
2
2
1 Leverage git grep to find ks in target repo
Determine the boundary ss and es by similarity

```
1     AssertLockHeld(cs_main);
2     assert(pindex);
3     assert((pindex->phashBlock == nullptr) ||
4         (*pindex->phashBlock == block.GetHash()));
5     int64_t nTimeStart = GetTimeMicros();

6  -  if (!CheckBlock(block, state, chainparams.GetConsensus(), !fJustCheck, !fJustCheck))
7  +  if (!CheckBlock(block, state, chainparams.GetConsensus(), !fJustCheck, !fJustCheck)) {
8  +      if (state.CorruptionPossible()) {
9  +          return AbortNode(state, “Corrupt block found ...");

10        return error("%s: Consensus::CheckBlock: %s", __func__, ...);
11    uint256 hashPrevBlock = pindex->pprev == nullptr ? uint256() : ...;
12    assert(hashPrevBlock == view.GetBestBlock());
13    if (block.GetHash() == chainparams.GetConsensus().hashGenesisBlock) {
14        if (!fJustCheck)
```

```
1  bool ConnectBlock(const CBlock& block, CValidationState& state, ...,
2      CCoinsViewCache& view, const CChainParams& chainparams, bool fJustCheck)
3  AssertLockHeld(cs_main);
4  const Consensus::Params& consensus = Params().GetConsensus(pindex->nHeight);
5  int64_t nTimeStart = GetTimeMicros();

6  if (!CheckBlock(block, state, !fJustCheck, !fJustCheck))

7      return error("%s: Consensus::CheckBlock: %s", __func__, ...);
8  uint256 hashPrevBlock = pindex->pprev == NULL ? uint256() : ...;
9  assert(hashPrevBlock == view.GetBestBlock());
10 if (block.GetHash() == Params().GetConsensus(0).hashGenesisBlock) {
11     if (!fJustCheck)
```

Fig. 3: Illustrating BlockScope's context-based search process for finding candidate contexts in a target repository.

in Sec. III-C). As a result, we do not require each extracted keyword to be precise because *as long as one of the context keywords can find the correct candidate context*, context similarity measurement (in Sec. III-C) will automatically exclude the search results of other incorrect context keywords.

We use the left patch code of Fig. 3 to illustrate the process of extracting context keywords. After normalizing and tokenizing each patch code line, `Extractor` uses the following heuristics to automatically recognize at most one context keyword per code line. Specifically, we consider the tokens with both lower and upper case letters (including some special characters like ".") and select the longest one as the most important variable or function name of one code line. In this way, BlockScope automatically selects nine context keywords, as highlighted in red color, from the patch code context in Fig. 3. As mentioned above, we do not require each extracted keyword to be precise, and according to our evaluation in Sec. IV, this simple strategy of automatically extracting context keywords works well for our problem.

C. Searching for Candidate Contexts in the Target Repositories

The `Searcher` component of BlockScope then uses the extracted context keywords to search for candidate contexts in the target repositories. The basic idea is to first search for the key statements in target code (via patch context keywords), then recover the corresponding boundary of each potential code context, and finally determine the candidate contexts via the similarity measurement with the original patch context. To illustrate this context-based search process, we use Bitcoin's patch of checking corrupted blocks and its vulnerable clone in Dogecoin as a running example. As shown in Fig. 3, the left-hand side is the patch code hunk (commit `0e7c52dc`) from Bitcoin, while the right-hand side shows the cloned version in Dogecoin [4]. It also illustrates the following three steps.

1) Searching for the key statements. The first step is to find the key statements (`ks`) that are the code statements in the target code with the searched context keywords. Specifically, `Searcher` first leverages `git grep` to search for all the code statements that contain the patch context keyword(s) in the target repositories, and then finalize the search result by measuring the similarity between the searched `ks` with the original `ks`. If the measured similarity is higher than the threshold configured in BlockScope, we consider it one potential candidate `ks`. To minimize the misses and avoid causing false negatives to the subsequent steps, this step uses a relatively low threshold (0.25) based on the Normalized Levenshtein [53] metric, i.e., strsim() used in equation (1). This is acceptable because among all the searched candidate `kss`, we select the one with the highest similarity as the final candidate `ks`. Specifically, given a patch context $pc = \{(k_1, s_1), (k_2, s_2), ..., (k_m, s_m)\}$, where (k_i, s_i) represents the extracted keyword k_i of the code statement s_i, the search result sr_i for k_i is represented as $sr_i = (k_i, [s'_{i1}, s'_{i2}, ..., s'_{in}])$, where s'_{ij} is the code statement that contains k_i in the target repository. We determine s'_{pq} as the final candidate `ks` according to the equation (1). In the case of Fig. 3, `Searcher` selects line 5 and 9 (both with the highest similarity) of Dogecoin as the final candidate `kss` of the `UP` and `DOWN` contexts, respectively.

$$p, q = \underset{1 \le i \le m, 1 \le j \le n}{\arg\max} \ \text{strsim}(s_i, s'_{ij}) \tag{1}$$

Moreover, in the course of implementing the candidate context search, we adopt three *automatic* optimizations to further improve BlockScope's context search precision and avoid unnecessary analysis in the subsequent steps. First, it excludes the search result with comments and test code. Second, it excludes the search result with the file type different from the patch's file type, e.g., the patch in Fig. 3 is a C/C++ source code file, based on which BlockScope excludes C/C++ header files and non-C/C++ source code files in the search result. Third, BlockScope excludes the search result with different statement types. For example, since line 5 in Fig. 3 is an assignment statement, any search result does not match the same statement type will be automatically discarded.

2) Determining the boundary of candidate contexts.

[4] Note that we only keep the "meaningful" code statements, i.e., empty lines, comments, and single brackets are removed.

Once identified the candidate ks, the next step of Searcher is to retrieve the code statements surrounding it and determine their boundary. Specifically, we need to expand the one-line candidate ks into the multi-line candidate context that has the corresponding boundary as the original patch context. To do so, we first fetch the same number of nearby code statements from target code as that, represented as C_LINES, in the patch context. For example, in Fig. 3, if we set C_LINES=5, Searcher fetches line 1 to 5 and line 7 to 11 for the candidate UP and DOWN contexts in Dogecoin, respectively. Then starting from the ks (i.e., line 5 and 9 of Dogecoin), Searcher compares each code statement upwards and downwards with the start statement (ss) and end statement (es) in the patch context, respectively. It then selects the ones with the highest similarity and also exceeding the aforementioned threshold (0.25) as the boundary ss and es in the candidate context, e.g., line 3 and line 5 for Dogecoin's UP context.

3) Finalizing the candidate contexts via similarity measurement. It is worth noting that ss and es only define the boundary of the candidate context, while the code statements in between remain unchecked. As illustrated in the step 3 of Fig. 3, we thus further check whether the entire candidate context is indeed similar to the patch context via the same multi-line code similarity measurement that will be introduced in Sec. III-E. If the measured similarity between the candidate context *C* and the patch context *P* exceeds a threshold, we consider *C* as the context of a candidate clone for further processing; otherwise, we discard this candidate context. Note that since multiple candidate contexts' similarity could exceed the threshold, all of these candidate contexts will be further processed.

D. Fetching Patch and Candidate Code Hunks from the Source and Target Repositories

With the determined candidate context(s), we leverage Fetcher to retrieve the patch code from the source repository and the candidate code from the target repository, respectively. Note that Fetcher is also used by the earlier Searcher component to retrieve the context of a patch/candidate code hunk. Specifically, a typical code hunk consists of three code fragments, the UP context, the DOWN context, and the middle patch/candidate code, as previously shown in Fig. 3.

For the patch code hunk, Fetcher directly fetches its patch code from the commit history and selects the nearby code statements upwards and downwards (with the line number specified by C_LINES) as the UP and DOWN contexts, respectively. For the candidate code hunk, we fetch its code statements according to the candidate context determined in Sec. III-C and also the original patch context. Specifically, if the original patch contains both UP and DOWN contexts, we regard the code statements between the corresponding candidate contexts as the candidate code. As a result, line 6 of Dogecoin is fetched as the candidate code in Fig. 3. If the patch context contains only the UP context, we regard the code statements below it as the candidate code. Similarly, if the patch context contains only the DOWN context, we regard the code statements above it as the candidate code. Note that for the last two situations, the candidate code is fetched with the same number of code statements as the patch code.

E. Measuring the Similarity between Patch and Candidate Code

With the fetched patch and candidate code, Comparator measures the similarity between their two code fragments and also determine whether the target repository has fixed the vulnerability, if the candidate code is not vulnerable. As mentioned in Sec. III-A, we need a new way of calculating the code similarity that is immune to Type-1/2/3 clones.

We first abstract the code similarity problem in this form: given a source code fragment *S* with p code statements and a target code fragment *T* with q code statements, respectively, we need to design an appropriate measure to determine their similarity. Intuitively, we can compute the similarity between *S* and *T* by first adding up the similarity of each pair of code statements at the same position in *S* and *T* and then normalizing it into $[0, 1]$, i.e., $\frac{1}{p}\sum_{i=1}^{p} \text{strsim}(S_i, T_i)$. While this can handle Type-2 clones because of not using the hash-based exact match per code line, it is still not applicable to measuring Type-3 clones for two reasons. First, as Type-3 clones involve inserted/deleted statements, i.e., $p \neq q$, the extra code statements will not be measured in this way. Second, because of the inserted/deleted statements, the ordering of the same code statement in *S* and *T* might be also different.

To solve the problems above, we determine two principles: (i) all the code statements in *S* and *T* should be considered; and (ii) the influence of the ordering issue should be adjustable. For the first principle, we identify the most similar code statement in *T* for every code statement in *S*, i.e., for each code statement $S_i \in S$, we find $T_j \in T$, s.t., $j = \arg\max_k \text{ strsim}(S_i, T_k)$. For the second principle, we first define the index *i* and *j* as the relative positions of the code statements in *S* and *T* if S_i's most similar statement is T_j. The basic idea is that the greater the difference between *i* and *j* is, the less similarity between S_i and T_j should be. Therefore, we introduce a parameter $r \in [0, 1]$, and $r^{|i-j|}$ to indicate the reward of the similarity between S_i and T_j. By multiplying this reward by the original similarity, we can adjust the ordering issue's influence on code similarity. To illustrate the impact of r on the similarity measurement, we calculate the similarities of all the patch and candidate code pairs under different r. We present the result in Appendix A. In this paper, we set 0.95 as the default value of r. Once finishing the calculation of such similarity for every code statement in *S*, we sum them up and normalize the result into $[0, 1]$, as shown in the following equation (2).

$$\begin{aligned} \text{SIMILARITY}(S, T) &= \frac{1}{p}\sum_{i=1}^{p} \text{strsim}(S_i, T_j) r^{|i-j|} \\ \text{s.t., } j &= \underset{1 \leq k \leq q}{\arg\max} \ \text{strsim}(S_i, T_k) \end{aligned} \tag{2}$$

While the method above provides a new way of measuring the similarity between two code fragments, we still need to determine whether the target repository has applied a patch or not. Specifically, given the candidate code *C* of the target repository, we compare it with the patch code *P*. Note that there are three types of *P*: (i) DEL-type: contains only the deleted lines, i.e., $P = [dp]$; (ii) ADD-type: contains only the added lines, i.e., $P = [ap]$; and (iii) CHA-type: contains both deleted and added lines, i.e., $P = [dp, ap]$. We thus determine the comparison logic as follows (where t is the threshold):

TABLE II: An example of the output of `git blame`.

src/qt/bitcoin.cpp

```
202d853b   201          }
202d853b   202      }
202d853b   203
a2714a5c   204      static int qt_argc = 1;
797fef7b   205      static const char* qt_argv = "qtum-qt";
a2714a5c   206
a2714a5c   207      BitcoinApplication::BitcoinApplication(...):
a2714a5c   208          QApplication(qt_argc, const_cast<char **>(...)),
9096276e   209          coreThread(nullptr),
71e0d908   210          m_node(node),
9096276e   211          optionsModel(nullptr),
```

- For type (i), if SIMILARITY$(C, dp) \geq t$, we determine that *C* did *not* apply *P*; otherwise, we determine that *C* has applied *P*.
- For type (ii), if SIMILARITY$(C, ap) \geq t$, we determine that *C* has applied *P*; otherwise, we determine that *C* did *not* apply *P*.
- For type (iii), if SIMILARITY$(C, dp) \geq t$ and SIMILARITY$(C, ap) \geq t$ and SIMILARITY$(C, dp) \geq$ SIMILARITY(C, ap), we determine that *C* did *not* apply *P*; otherwise, if SIMILARITY$(C, dp) \geq t$ and SIMILARITY$(C, ap) \geq t$ and SIMILARITY$(C, dp) <$ SIMILARITY(C, ap), we determine that *C* has applied *P*.

Moreover, as `Searcher` may return multiple candidate contexts in the target repository, leading to multiple candidate code, i.e., $C_i \in [C_1, C_2, ..., C_n]$. For each C_i, we calculate $s_i =$ SIMILARITY(C_i, P), and determine its patch applying status $fv_i \in \{0, 1\}$, where $fv_i = 1$ $(= 0)$ indicates C_i has (not) applied *P*. Here we introduce a factor $conf_i$ to measure the confidence of fv_i on C_i by $conf_i = s_i - t$, i.e., the greater s_i exceeds t the more confident fv_i is on C_i. Finally, we can determine the status of *P* in the target repository by the most confident fv_i, i.e., $i = \arg\max_j conf_j$. If the target repository did not apply *P*, we consider it a vulnerability; otherwise, we consider the vulnerability fixed.

F. Determining Patch Delays for the Vulnerabilities Already Patched in the Target Repositories

For the vulnerabilities already patched in the target repositories, we further leverage `Calculator` to automatically measure their patch delays. We define the *patch delay* as the interval between the patch's commit date in the source project and the patch's release date in the target project because eventually, the release date is the actual time when a patch is available to the blockchain node operators and end users.

Upon receiving a candidate code that is determined as fixed, `Calculator` leverages `git blame` to retrieve the commit that patched the code. Table II illustrate an example output of `git blame`, where the left column shows the commit hash (SHA), the column in the middle shows the line number for the code statements on the right in Qtum's `src/qt/bitcoin.cpp` file. The code from line 204 to line 208 is actually Qtum's patch for fixing the cloned CVE-2021-3401 [12] in its project. It was added by two commits, `a2714a5c69` and `797fef7bee`, where `797fef7bee` only modified line 205. Hence, we still need to determine which commit is the *true* fix. In the Qtum example, after checking both commits, we identify that line 205 in Table II was originally added by `a2714a5c69` on 10 August 2019 as `static const char* qt_argv = "bitcoin-qt";`, where `"bitcoin-qt"` is later replaced by `"qtum-qt"` in `797fef7bee` on 26 June 2020. As a result, if multiple commits modify the candidate code, we consider the earliest one is the *true* fix commit.

Moreover, we need to scrape the release information from GitHub because the local git repository does not contain such information. By analyzing a commit's GitHub webpage, `Calculator` can retrieve all of its release versions and determine the earliest date when the commit was first released. In the Qtum example, the patch commit `a2714a5c69` was first released in the version `mainnet-ignition-v0.19.0` on 22 February 2020, which was delayed from the original Bitcoin commit by 197 days.

IV. Detecting the Vulnerabilities Propagated to Forked Projects

In this section, we aim to detect the vulnerabilities that are propagated from Bitcoin and Ethereum to their forked blockchain projects using BlockScope. To this end, we first benchmark the accuracy and performance of BlockScope (Sec. IV-B) using an experimental setup introduced in Sec. IV-A. We then present the detected vulnerabilities in Sec. IV-C. Finally, we conduct ethical vulnerability reporting and summarize vendors' response/actions in Sec. IV-D.

A. Experimental Setup

To make sure that BlockScope's vulnerability detection results are reliable, we not only run BlockScope in our experiment but also compare it with the open-source state-of-the-art ReDeBug [43] using the same dataset and environment below. Note that we also considered other clone detection tools (e.g., [47], [50], [68], [82]) for more comparison but eventually did not choose them for two reasons. First, MVP [82] was not open-source and it does not support the Go language. While VUDDY [50] released its signature generating scripts, its most important vulnerability search engine was not available. Indeed, we contacted the VUDDY team and confirmed that their cloud version currently supports only one CVE in our dataset. Second, CCFinder [47] and SourcererCC [68] are pure code clone detection tools and are not able to perform patch-based detection in our problem without adjustment.

Dataset. As illustrated in Fig. 2, BlockScope requires two sets of input, the target blockchain code repositories and the security patches of a reference blockchain (i.e., Bitcoin and Ethereum in this paper). As a result, we collect these two sets of data as our dataset. Specifically, for code repositories, we select all the 11 forked projects of Bitcoin from the top 100 cryptocurrencies (based on the market capitalization on CoinMarketCap) and five popular forked projects of Ethereum (picked from Blockscan) as our target blockchains, as previously introduced in Sec. II. The total market capitalization of these 16 blockchains was around 142 billion USD. To build a reproducible dataset, we kept a local copy of the latest version of code repositories at the time of our research on 7 September 2021 and 6 June 2022 for Bitcoin forks and Ethereum forks, respectively. On the other hand, for security patches, an intuitive idea is to use the CVE (Common Vulnerabilities and Exposures) information; however, we found

TABLE III: The experimental result of BlockScope.

(a) The accuracy and performance comparison between BlockScope and ReDeBug.

Forked Project	LOC	BlockScope					ReDeBug				
		TP	FN	TN	FP	Time	TP	FN	TN	FP	Time
Dogecoin	326.9K	16	-	15	1	7.6s	7	9	15	1	12.5s
Bitcoin Cash	607.1K	1	-	30	1	10.5s	-	1	31	-	22.2s
Litecoin	423.3K	6	-	26	-	8.3s	5	1	26	-	16.4s
Bitcoin SV	221.1K	11	1	18	2	10.6s	2	10	19	1	9.9s
Dash	380.3K	9	1	22	-	13.9s	7	3	21	1	17.7s
Zcash	199.4K	9	2	19	2	8.4s	1	10	21	-	10.7s
Bitcoin Gold	381.7K	10	1	21	-	8.8s	10	1	21	-	17.4s
Horizen	178.9K	9	2	20	1	7.7s	1	10	21	-	12.6s
Qtum	569.0K	-	-	31	1	12.0s	-	-	32	-	33.5s
DigiByte	416.3K	10	1	21	-	10.7s	10	1	21	-	15.8s
Ravencoin	504.2K	14	1	16	1	11.4s	10	5	17	-	20.9s
Sum	**4.2M (382.6K)***	**95**	**9**	**239**	**9**	**109.9s (3.4s)**$^\diamond$	**53**	**51**	**245**	**3**	**189.6s (5.9s)**$^\diamond$
Binance	565.3K	1	-	5	-	2.2s	-	1	5	-	30.2s
Avalanche	1070.1K	-	-	6	-	2.5s	-	-	6	-	55.2s
Polygon	592.0K	-	-	6	-	2.3s	-	-	6	-	31.3s
Celo	631.0K	1	-	5	-	2.7s	1	-	5	-	44.5s
Optimism	630.6K	4	-	2	-	3.6s	3	1	2	-	43.3s
Sum	**3.5M (697.8K)***	**6**	**-**	**24**	**-**	**13.3s (2.2s)**$^\diamond$	**4**	**2**	**24**	**-**	**204.5s (34.1s)**$^\diamond$

*: the numbers in (.) of these cells represent the average LOC per *project*.
$^\diamond$: the numbers in (.) of these cells represent the average processing time per *patch*.

(b) The fixed cases detected by BlockScope.

Forked Project	# Fixed Cases		
	Detected	Truth	Err*
Dogecoin	1	1	-
Bitcoin Cash	23	25	(2;-)
Litecoin	22	22	-
Bitcoin SV	1	1	-
Dash	11	10	(-;1)
Zcash	2	1	(-;1)
Bitcoin Gold	14	14	-
Horizen	1	-	(-;1)
Qtum	28	28	(1;1)
DigiByte	14	14	-
Ravencoin	3	3	-
Sum	**120**	**119**	**(3;4)**
Binance	5	5	-
Avalanche	3	3	-
Polygon	6	6	-
Celo	4	4	-
Optimism	1	1	-
Sum	**19**	**19**	**-**

* represents (the number of missed cases; the number of mistake cases).

that there are only 12 CVEs about Bitcoin with explicit patch code and eight of them are out of the recent five years. That said, we could select only four to test if we just use the public CVE information.

To address this problem, we select bug issues/pull requests with notable security impacts (i.e., vulnerabilities) and their patch commits (i.e., patches) directly from Bitcoin's GitHub repository according to three simple principles: (i) the patches should be released within the recent five years since outdated patches had been applied to Bitcoin before it gets forked; (ii) the patches that cover different vulnerability types should have a higher chance to be picked up so that we can evaluate the generality of BlockScope; and (iii) the patches should be applicable to most forked projects, i.e., not specific to one particular Bitcoin component or one fork. As a result, we are able to select 32 patches of Bitcoin from June 2017 to March 2020, including four CVEs. For Ethereum, since its forks are relatively new, we select six CVEs of Ethereum since November 2020 as the patches. These 38 patches involve multiple vulnerability types, including denial-of-service, race conditions, privacy leakage, and etc. While the number of Bitcoin and Ethereum vulnerabilities here is not large, we have to be *selective* to make sure they are actually vulnerabilities. Indeed, Bitcoin and Ethereum have a limited number of vulnerabilities over the years. For example, the VUDDY dataset included only 9 CVEs of Bitcoin, with 8 of them already before 2013 and only one after 2018. Moreover, we have 16 popular forked projects of Bitcoin and Ethereum forked projects to test, which multiplied the total test cases to 382 ($32 \times 11 + 6 \times 5$).

Environment and tool configuration. We evaluate BlockScope and ReDeBug on the same virtual machine running Ubuntu 18.04 with 4GB memory configured, while the host machine is a Macbook Pro with a 3.5GHz dual-core Intel Core i7 CPU and 16GB memory. Note that ReDeBug needs to set a `n-gram` parameter to adjust the number of lines for context code. While the default is four, we tried from one to ten and found that when `n-gram=3`, ReDeBug achieves its best result when analyzing our dataset.

B. Accuracy and Performance

After running BlockScope and ReDeBug on the dataset in Sec. IV-A (i.e., using 32 Bitcoin patches and six Ethereum patches to test the 16 forked projects) and performing a thorough code review of all the raw detection results (including the cases that have no any output), we are able to precisely obtain the accuracy and performance data for both tools. Overall, BlockScope detects 101 true vulnerabilities in 13 forked projects (Qtum, Avalanche, and Polygon do not contain any vulnerability in our dataset as we manually checked), whereas ReDeBug detects only 57 vulnerabilities in ten forked projects, which makes BlockScope's recall 1.8 times higher than that in ReDeBug. For performance, BlockScope is also 1.7 times faster than ReDeBug in analyzing Bitcoin's forked projects and even 15.4 times faster in analyzing Ethereum's forked projects with more code per project.

Table IIIa shows a breakdown of the detailed accuracy and performance results of BlockScope and ReDeBug, where TP, FN, TN, and FP represent true positive, false negative, true negative, and false positive, respectively. According to this table, we can calculate the precision via $TP/(TP+FP)$ and the recall via $TP/(TP+FN)$, respectively. We find that BlockScope achieves good precision and high recall both at 91.8%. In contrast, while ReDeBug has only three false positives in our dataset (mainly because it uses the exact match per code line), its recall is as low as 51.8%. That said, ReDeBug fails to detect many of the vulnerabilities covered by BlockScope. Since we aim to perform a thorough investigation of forked blockchains' vulnerabilities, BlockScope achieves the high recall we need while introducing a low false alarming rate at only 8.18%. Moreover, among the 13 forked projects with vulnerabilities (i.e., no Qtum, Avalanche, and Polygon), BlockScope detects vulnerabilities in all of them, while ReDeBug fully misses the results for two projects, Bitcoin Cash and Binance. In particular, BlockScope successfully detects all the

TABLE IV: # of different vulnerability types in each project.

Forked Project	Type-1		Type-2		Type-3		Sum	
	T	**B;R**	**T**	**B;R**	**T**	**B;R**	**T**	**B;R**
Dogecoin	6	(6;4)	-	-	10	(10;3)	16	(16;7)
Bitcoin Cash	1	(1;-)	-	-	-	-	1	(1;-)
Litecoin	5	(5;5)	-	-	1	(1;-)	6	(6;5)
Bitcoin SV	1	(1;-)	-	-	11	(10;2)	12	(11;2)
Dash	7	(7;7)	-	-	3	(2;-)	10	(9;7)
Zcash	1	(1;-)	2	(1;-)	8	(7;1)	11	(9;1)
Bitcoin Gold	9	(9;8)	-	-	2	(1;2)	11	(10;10)
Horizen	-	-	2	(2;-)	9	(7;1)	11	(9;1)
Qtum	-	-	-	-	-	-	-	-
DigiByte	7	(7;7)	1	(1;-)	3	(2;3)	11	(10;10)
Ravencoin	7	(7;7)	-	-	8	(7;3)	15	(14;10)
Sum	**44**	**(44;38)**	**5**	**(4;-)**	**55**	**(47;15)**	**104**	**(95;53)**
Binance	-	-	-	-	1	(1;-)	1	(1;-)
Avalanche	-	-	-	-	-	-	-	-
Polygon	-	-	-	-	-	-	-	-
Celo	1	(1;1)	-	-	-	-	1	(1;1)
Optimism	4	(4;3)	-	-	-	-	4	(4;3)
Sum	**5**	**(5;4)**	**-**	**-**	**1**	**(1;-)**	**6**	**(6;4)**

T, B, and R represent: the total number of vulnerabilities of each clone type, the number of vulnerabilities detected by BlockScope, and the number of vulnerabilities detected by ReDeBug, respectively.

vulnerabilities in Dogecoin, Bitcoin Cash, Litecoin, Binance, Celo, and Optimism with zero false negative.

We further explore the reasons that cause BlockScope to have a much better detection effectiveness than ReDeBug by analyzing the detailed results of detecting different clone types. This is because while ReDeBug claims that it can handle Type-1 and Type-3 clones, the accuracy of each clone type may vary. As shown in Table IV, among the 110 (TP + FN) vulnerabilities in the forked projects of our dataset, 95.5% of them are the Type-1 and Type-3 clones, with the number of Type-3 clones slightly higher than that of Type-1 clones. For these cases, ReDeBug achieves an accuracy of 85.7% for Type-1 clones, but its detection rate for Type-2 and Type-3 clones drops to 0% and 26.8%, respectively. This explains why ReDeBug performs better on six particular projects — the number of Type-1 clones in those six projects (i.e., Litecoin, Dash, Bitcoin Gold, DigiByte, Celo, and Optimism) is larger than that of Type-3 clones. Indeed, if a forked project has more Type-1 clones, it is more similar to the original project. In contrast, BlockScope does not have this limitation. It is able to detect all the Type-1 clones, and misses only one and eight cases for the more complicated Type-2 and Type-3 clones, respectively. This indicates that BlockScope still reaches a high rate of 80% for Type-2 clones and 85.7% for Type-3 clones.

For performance, BlockScope performs much faster on all the projects than ReDeBug. In particular, BlockScope can finish the analysis of 10 forked projects within ten seconds, while ReDeBug just finishes only one project (i.e., Bitcoin SV) within ten seconds. We further analyze whether the project's LOC affects the performance of BlockScope and ReDeBug. For BlockScope, we notice that it takes almost the same time (10.5s vs. 10.6s) to analyze Bitcoin Cash and Bitcoin SV, even though the LOC of Bitcoin Cash is 2.7 times that of Bitcoin SV (607K vs. 221.1K). In contrast, the processing time of ReDeBug for the same two projects is 22.2s and 9.9s, respectively. The difference of 2.2 times is close to the ratio of those two projects' LOC. This indicates that the project's LOC does not explicitly affect the processing time of BlockScope, while it has a significant effect on ReDeBug's performance.

Indeed, when we compare the performance of BlockScope between Bitcoin forks (with fewer LOC) and Ethereum forks (with more LOC), we notice that BlockScope can finish the analysis of Ethereum forks even faster. It suggests that for BlockScope, the number of target patches (32 for Bitcoin vs. 6 for BlockScope) has a more noticeable impact on its performance than LOC. ReDeBug, on the other hand, is the opposite, with LOC having much more impact than the number of target patches on its performance. For example, for Qtum and Binance that have almost the same LOC, the analysis time of ReDeBug is also almost the same (33.5s vs. 30.2s). As we mentioned earlier, typical code clone detection tools like ReDeBug perform a whole-project analysis – so LOC dominates the performance, while BlockScope leverages patch code contexts to search and locate only potentially relevant code for comparison – so LOC has a much limited effect.

C. Analysis of the Detected Vulnerabilities

Since BlockScope detects not only the cloned vulnerabilities but also whether a patch is applied, we perform an analysis on both the detected vulnerabilities and the fixed cases in this subsection. For a deep investigation on the individual vulnerability, we present it later in Sec. V.

As shown in Table IIIa, Bitcoin's forked projects have a total of 104 vulnerabilities. Among the 11 projects, only Bitcoin Cash and Qtum have few vulnerabilities, while eight projects have at least 10 vulnerabilities each out of the 32 patches investigated. In particular, Dogecoin and Ravencoin did not patch around half of the total 32 vulnerabilities. On the contrary, Ethereum's forks present a better result, with only Optimism having four vulnerabilities out of the six patches investigated. The other four projects have at most one vulnerability each, with Avalanche and Polygon fully patched.

For the result of fixed cases, the forked projects of Bitcoin and Ethereum have fixed a total of 138 vulnerabilities (119 for Bitcoin and 19 for Ethereum). While Bitcoin's 11 forked projects have fixed 119 vulnerabilities, five of them, Dogecoin, Bitcoin SV, Zcash, Horizen, and Ravencoin, fixed only six vulnerabilities in total. Three projects, Qtum, Bitcoin Cash, and Litecoin, contribute to 63% of all the fixed cases. Similar to the result above regarding the vulnerable cases, Ethereum's forked projects also perform better in the fixed cases. While Optimism fixed only one vulnerability, the other four projects have fixed at least half of the investigated patches. Indeed, when comparing the ratio of the fixed/vulnerable cases between Bitcoin's and Ethereum's forked projects — 119/104 vs. 19/6, we notice that Ethereum's forks are more active in fixing propagated vulnerabilities. Another aspect for measuring the project's activeness on patching vulnerabilities is the patch delay, which we provide a detailed analysis in Sec. V-C.

D. Vulnerability Reporting and Response

As an ethical research and one contribution of this paper, we have spent significant efforts reporting all the 110 discovered vulnerabilities (including 101 TP automatically detected by BlockScope and 9 FN manually identified by us during evaluation) to the developers of the affected forked projects via multiple channels. In Table V, we summarize the latest developers' response and actions to our vulnerability reports

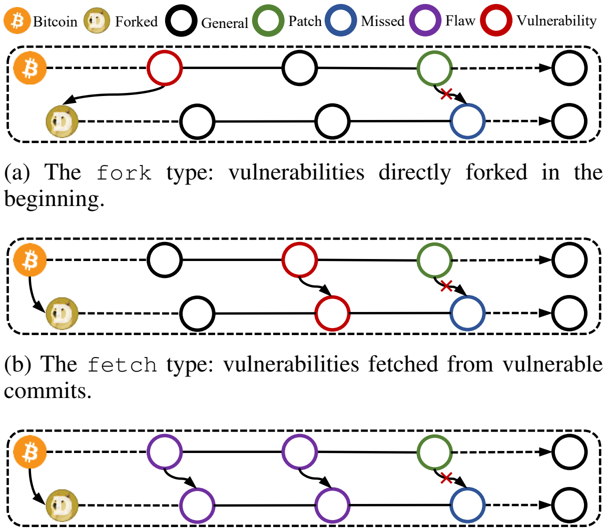
TABLE V: Developers' response to our vulnerability reports.

Forked Project	Fixed	Accepted	ACK	Pending	Reject	Sum
Dogecoin	11	3	2	-	-	16
Bitcoin Cash	-	-	-	1	-	1
Litecoin	2	-	3	1	-	6
Bitcoin SV	-	-	8	2	2	12
Dash	1	5	3	1	-	10
Zcash	-	-	9	1	1	11
Bitcoin Gold	7	-	1	3	-	11
Horizen	-	-	4	7	-	11
Qtum	-	-	-	-	-	-
DigiByte	-	-	-	11	-	11
Ravencoin	9	1	3	1	1	15
Sum	**30**	**9**	**33**	**28**	**4**	**104**
Binance	-	1	-	-	-	1
Avalanche	-	-	-	-	-	-
Polygon	-	-	-	-	-	-
Celo	-	-	1	-	-	1
Optimism	-	-	-	4	-	4
Sum	**-**	**1**	**1**	**4**	**-**	**6**

as of 26 July 2022. Specifically, "Fixed" means that the vendor has adopted our reports to fix the issues, "Accepted" represents that the developers accepted our reports and were exploring appropriate patch migration, "ACK" suggests that the vendor has acknowledged our reports but did not explicitly indicate to fix the issues, "Pending" means that we have not received any response yet, and lastly, "Reject" means that the vendor has denied and refused to fix the vulnerabilities. We can see that around 74 of our 110 vulnerability reports received positive response, which demonstrates that the impact of our work. We further classify developers' response into three categories:

Positive/Active Response. Among the 13 forked projects with vulnerabilities, around half of them responded to our vulnerability report positively, namely Dogecoin, Ravencoin, Dash, Bitcoin Gold, Litecoin, and Binance. Specifically, Dogecoin acknowledged all of our reports and quickly fixed 11 serious vulnerabilities, while the others are scheduled or under the community discussion for appropriate patch migration. Meanwhile, Ravencoin accepted nearly all the reports. The developers fixed nine of them and acknowledged three except one rejection and one pending due to the compatibility consideration. Similarly, the developers of Dash approved nearly all the reports and informed us that they had fixed five vulnerabilities under the development branch, which will be merged into a new release in the future. Bitcoin Gold also fixed seven vulnerabilities in one release after around four months receiving our reports, with another one acknowledged and three under pending, while Litecoin fixed two of the vulnerabilities and claimed that they had noticed the other three. Lastly, Binance immediately acknowledged our report on BSC and rewarded us a bug bounty with the promise of fixing it. During this reporting process, we found that developers are more likely to fix a vulnerability with authoritative proofs, especially those with CVE numbers. For instance, the Dogecoin developers quickly released a new version of the Dogecoin core after they fixed CVE-2021-3401 and CVE-2019-15947. However, for the other vulnerabilities with no CVE assigned, they just acknowledged them and kept them on the to-do list.

Neutral Response. In this category, developers also accepted our reports but did not have intention to fix any of them yet. Specifically, Bitcoin SV's developers quickly acknowledged 8 of the 12 reports, and Zcash similarly acknowledged 9 of the 11 reports. However, both rejected a few (2 for Bitcoin SV and 1 for Zcash) due to incompatibility, and we have not received further updates from them. Meanwhile, Horizen acknowledged four vulnerability reports with the other seven still under pending, and Celo acknowledged the only report.



(a) The `fork` type: vulnerabilities directly forked in the beginning.

(b) The `fetch` type: vulnerabilities fetched from vulnerable commits.

(c) The `mixed` type: vulnerabilities infected with no explicitly vulnerable commits.

Fig. 4: Three types of the vulnerability propagation from Bitcoin to its forked projects.

Negative/Inactive Response. Unfortunately, the response from the rest of three projects is not active and worrisome. Specifically, Bitcoin Cash, DigiByte, and Optimism did not give response to any of our reports. The worst case is DigiByte because it ignored 11 vulnerabilities, including some critical ones like CVE-2021-3401 and CVE-2019-15947.

V. Investigating the Propagation and Patching Processes of Discovered Vulnerabilities

In this section, we conduct a deep investigation of the vulnerabilities discovered in Sec. IV. Specifically, in Sec. V-A, we aim to understand how these vulnerabilities are propagated from Bitcoin and Ethereum to their forked projects. Furthermore, in Sec. V-B, we diagnose some other propagation that caused our detection to fail (both FP and FN). Lastly, we perform a patch delay analysis in Sec. V-C to understand the patching processes of the cases that were already fixed in forked projects before our detection.

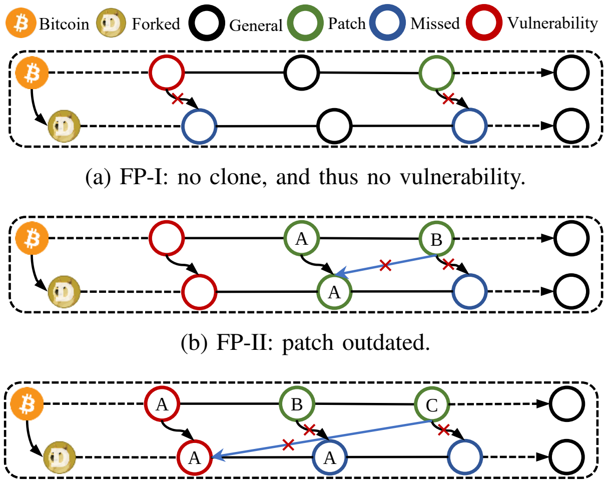
A. Revealing the Vulnerability Propagation from Bitcoin/Ethereum to Their Forked Projects

To reveal how a vulnerability is propagated from Bitcoin and Ethereum to the forked projects, we manually check all the 110 vulnerabilities, including 104 from Bitcoin forks and 6 from Ethereum forks, respectively, and categorize them into three types, as shown in Fig. 4. To simplify the description in this section, we apply "Bitcoin" to represent both Bitcoin and Ethereum, unless explicitly specified. The first type, as illustrated in Fig. 4a, refers to the vulnerabilities that were introduced when the project was initially forked from Bitcoin. For better understanding and simplicity, we call it the `fork` type. The second type, as depicted in Fig. 4b, is similar to the first type except that it fetched and merged vulnerable commits of Bitcoin afterwards. We call it the `fetch` type. The third type, as shown in Fig. 4c, is an advanced version of the `fetch` type. The major difference is that vulnerabilities of this type were infected with no explicitly vulnerable commits of Bitcoin. Typically, they are caused by the defective program design or inappropriate functionality implementation that involves multiple code commits. We call this type the `mixed` type. In total, we identify 41 `fork`-type, 25 `fetch`-type, and 44 `mixed`-type vulnerabilities, respectively. We conduct case studies about these three types as follows.

Vulnerabilities directly forked in the beginning. In the `fork` type, vulnerabilities were propagated into the forked projects when they forked from Bitcoin. Many vulnerabilities, such as CVE-2022-29177 and CVE-2021-41173 from Ethereum, or CVE-2021-3401 from Bitcoin, are the classic cloned vulnerability cases to explore `fork`-type vulnerabilities and study their propagations. Take CVE-2021-3401 as an example. This vulnerability first appeared in Bitcoin, but we found that it also affected three other forked projects (Dash, Ravencoin, and Bitcoin Gold) since they were initially forked from Bitcoin. As detailed in [32] and the patch code in [12], it was caused by the misuse of the Qt-framework built-in arguments. Specifically, Bitcoin and its forked projects leverage the Qt-framework [22] to design their own GUI programs. However, Qt suffered from argument misinterpretation, in which attackers can inject dangerous built-in Qt arguments, e.g., `-platformpluginpath`, into a normal Qt command to load and execute their malicious plugin code remotely.

Vulnerabilities fetched from vulnerable commits. In the `fetch` type, vulnerabilities were introduced when forked projects fetch commits from Bitcoin to update their functionalities without verifying whether a commit is vulnerable or neglecting a patch from Bitcoin afterwards. Dogecoin and DigiByte (forked from Bitcoin) were also affected by the aforementioned CVE-2021-3401 yet in this way, and Optimism (forked from Ethereum) were similarly affected by the CVEs including CVE-2020-26265, CVE-2020-26264, and CVE-2020-26260. Taking Dogecoin as example, it fetched the vulnerable commit `202d853b` [2] of CVE-2021-3401 from Bitcoin that sets inappropriate arguments in the class `BitcoinApplication`, but failed to pose any security check, causing a typical `fetch` vulnerability. This is different from the `fork` vulnerabilities because Dogecoin fetched the vulnerable code *actively* instead of *passively* including it. Unfortunately, there are no specification for the developers of forked projects to use the upstream code so that it is easy to skip the security patches and fetch a vulnerable commit only.

Vulnerabilities infected with no explicitly vulnerable commits. Different from the `fork` and `fetch` vulnerabilities, it is hard to locate the specific vulnerable commits that introduced vulnerabilities in the `mixed` type. It usually contains a few consecutive or discrete commits instead of the specific one(s). Only when *all* the buggy commits were included together, a vulnerability would then appear. Typically, in the `mixed` type, the program would still run correctly at the code level, but attackers can exploit the logical flaws. For instance, Bitcoin PR#16512 [9] fixed a logical flaw where the `joinpsbts` function did not shuffle its inputs and outputs, causing a privacy leak that attackers could easily identify which outputs belong to which inputs. This vulnerability was originated from the defect of the `joinpsbts` function implementation, instead of a certain commit that made the function vulnerable.



(a) FP-I: no clone, and thus no vulnerability.

(b) FP-II: patch outdated.

(c) FN: target code outdated.

Fig. 5: Three types of propagation from Bitcoin to its forked projects that caused BlockScope to fail in terms of FP and FN.

B. Diagnosing Some Other Propagation that Caused Our Detection to Fail

During our investigation of vulnerability propagation, we also identified some other propagation that evaded BlockScope's detection (FN) or caused false positives (FP). We carefully analyze all the 18 failed detection cases (9 FPs and 9 FNs) that are listed in Table VI, and summarize them into three types, FP-I, FP-II, and FN, as shown in Fig. 5.

FP-I: no clone, and thus no vulnerability. As shown in Fig. 5a, the forked project sometimes keeps its outdated code and does not clone the vulnerable commit. As a result, it has no need to fetch the patch commit either. However, for certain vulnerabilities, there may have multiple ways to write a security patch — some fix the root cause while others close the attack surface. Since BlockScope detects the vulnerable clone based on the similarity measurement with one specific patch, it is possible that it gives false alarming if the vulnerable code could be avoided in other ways.

One notable example is CVE-2018-17145 [4], which caused BlockScope to generate four same false positives, as shown in Table VI. We conducted a deep analysis of this DoS vulnerability. We found that the root cause is a susceptible variable `m_callbacks_pending`, which was introduced in Bitcoin at the commit `08096bbb` [3] (but forked projects like Dogecoin did not fetch this vulnerable commit). The size of this variable would grow unlimitedly and run out of all the host memory if attackers create flooding transactions to execute an interface function called `Inventory(inv.hash)`. Unfortunately, Bitcoin patched this vulnerability only by deleting the unrestricted `Inventory` function. Since Dogecoin did not clone both vulnerable and patch commits, BlockScope identifies the unrestricted `Inventory` function and thus determines that the forked vulnerable is also vulnerable. While the interface function is still there, there was no victim

TABLE VI: All the 18 failed detection in BlockScope.

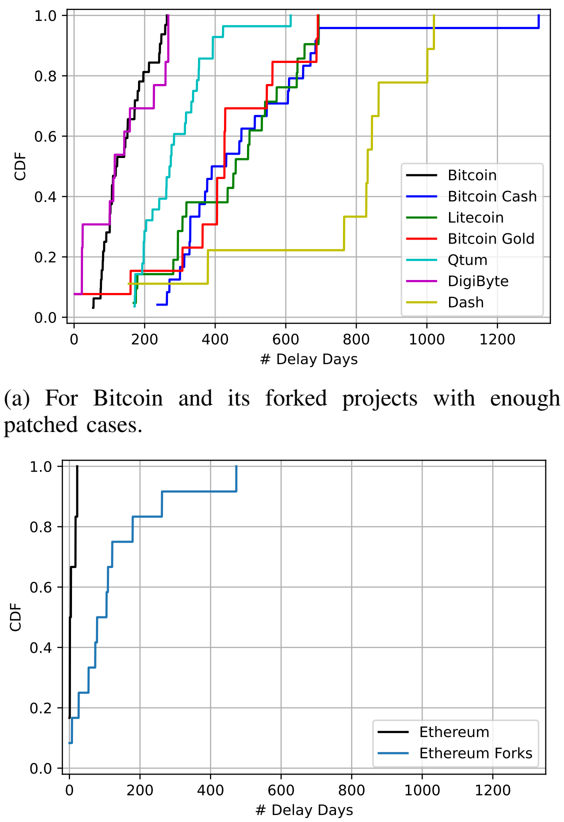
SHA	Source	Project	Cause	FP/FN
beef7ec4	CVE-2018-17145	Bitcoin SV	No Clone	FP-I
beef7ec4	CVE-2018-17145	Dogecoin	No Clone	FP-I
beef7ec4	CVE-2018-17145	Horizen	No Clone	FP-I
beef7ec4	CVE-2018-17145	Zcash	No Clone	FP-I
d8318318	CVE-2019-15947	Bitcoin SV	No Clone	FP-I
0e7c52dc	Bitcoin PR#12561	Zcash	No Clone	FP-I
b8f80196	Bitcoin PR#14249	Ravencoin	No Clone	FP-I
0e7c52dc	Bitcoin PR#12561	Qtum	Outdated Patch	FP-II
18f690ec	Bitcoin PR#13808	Bitcoin Cash	Outdated Patch	FP-II
76f74811	Bitcoin PR#10345	Bitcoin SV	Outdated Target	FN
37886d5e	Bitcoin PR#11568	Horizen	Outdated Target	FN
37886d5e	Bitcoin PR#11568	Zcash	Outdated Target	FN
e254ff5d	Bitcoin PR#13907	Zcash	Outdated Target	FN
4433ed0f	Bitcoin PR#15305	Horizen	Outdated Target	FN
effe81f7	Bitcoin PR#15323	Dash	Outdated Target	FN
effe81f7	Bitcoin PR#15323	Ravencoin	Outdated Target	FN
e6c58d3b	Bitcoin PR#15325	Bitcoin Gold	Outdated Target	FN
e6c58d3b	Bitcoin PR#15325	DigiByte	Outdated Target	FN

`m_callbacks_pending` variable in Dogecoin, making attackers cannot exploit the `Inventory` function. There are a total of seven false positives like this, as shown in Table VI.

FP-II: patch outdated. An outdated patch means that the forked projects had fetched a patch commit but neglected its further update. As shown in Fig. 5b, suppose there was a vulnerability in both Bitcoin and its forked project. Bitcoin released two different version of the patch at the point A and B, respectively. The first patch is for instant fixing while the latter for the patch update. However, the forked project just accepted the first patch. When BlockScope applied the final patch (i.e., the second) to detect clones, it cannot not match the target code and trigger a false positive.

For example, BlockScope generated a false positive for Bitcoin PR#13808 when testing Bitcoin Cash. Bitcoin fixed this vulnerability by using the `shuffle` [5] function of the C++ standard library, which is the first patch. However, Bitcoin later substituted the patch with the `Shuffle` [6] function created in Bitcoin PR#14624. It is an updated patch to fix the issue in a more appropriate way. Since Bitcoin Cash adopted the first patch only and neglected the update, it caused a FP of BlockScope. More specifically, it means that Bitcoin Cash still uses the `shuffle` function of the C++ standard library. When BlockScope used the updated patch for detection, it failed because BlockScope cannot match the patch from PR#14624. Similarly, BlockScope failed in another FP-II type vulnerability in Qtum from Bitcoin PR#12561.

FN: target code outdated. BlockScope could also encounter false negatives when the target code where the patch applies to is outdated. In the example of Fig. 5c, point A indicates an underlying vulnerability in a Bitcoin function. This vulnerability is further inherited along with the development of Bitcoin at point B, and Bitcoin creates a patch at point C to fix the vulnerability located at point B. A forked project suffers from the same vulnerability because it includes a copy of the vulnerable commit at point A. However, the Bitcoin patch at point C can not be directly applied to the vulnerability at point A due to the inconsistent code, causing a FN. Specifically, BlockScope uses the patch code at point C to search the potentially vulnerable code segments in a forked project. If BlockScope cannot identify any related code segments, it reports nothing and poses a false negative.



(a) For Bitcoin and its forked projects with enough patched cases.

(b) For Ethereum and its forked projects as a whole.

Fig. 6: CDF plots of # the delay days per security patch.

Taking Bitcoin PR#15305 as an example, it specifies the behavior of Bitcoin nodes to disconnect a block correctly when the Bitcoin program hits exceptions. However, when BlockScope applied the patch code of PR#15305 [8] to detect clones in Bitcoin SV, nothing outputted and a false negative appeared. This is because that Bitcoin SV keeps the outdated code cloned from Bitcoin. Indeed, we checked the history of the outdated code in Bitcoin SV and found that it was a copy of an old version of Bitcoin code. This outdated code in Bitcoin SV makes the Bitcoin patch cannot be directly applied. In total, BlockScope made nine such false negatives due to the outdated target code, as shown in Table VI.

C. Patch Delay Analysis

As previously mentioned in Sec. IV-C, we identified a total of 138 cases (119 from Bitcoin forks and 19 from Ethereum forks) that were already fixed before our detection. Among the 11 forked projects of Bitcoin, five projects have only a few fixed cases — Dogecoin, Bitcoin SV, and Zcash have one fixed case each, and Horizen even has no fixed case. Therefore, there is no enough data to analyze their patch delay. Moreover, since we only investigated six patches for Ethereum's forked projects, i.e., they do not have many fixed cases, we put them together as "Ethereum Forks". Hence, we focus on the "Ethereum Forks" and six Bitcoin's forked projects with more than ten fixed cases each, i.e., Bitcoin Cash, Litecoin, Dash, Bitcoin Gold, Qtum, and DigiByte. For each forked project, we draw a CDF plot of its patch delay days, as shown in Fig. 6. We also plot the CDF for Bitcoin's and Ethereum's patch delay days, i.e., the intervals between the commit date and the release date of the patch commit in the original projects, using the black line as a reference.

According to the black line in Fig. 6a, Bitcoin released all the selected patches within 300 days, and 80% of its patches were released within 200 days. The patch delay for serious vulnerabilities is even quicker, e.g., within 110 days for the four investigated CVEs. Unfortunately, only DigiByte can catch up with Bitcoin's release schedule, and Qtum's performance on patch delay is the second best, while the remaining projects could release only less than 20% of the patches within 200 days. Dash is particularly slow, with its 80% patches released after 800 days. In some extreme cases, the release delay could even exceed 1,000 days, e.g., in Bitcoin Cash and Dash.

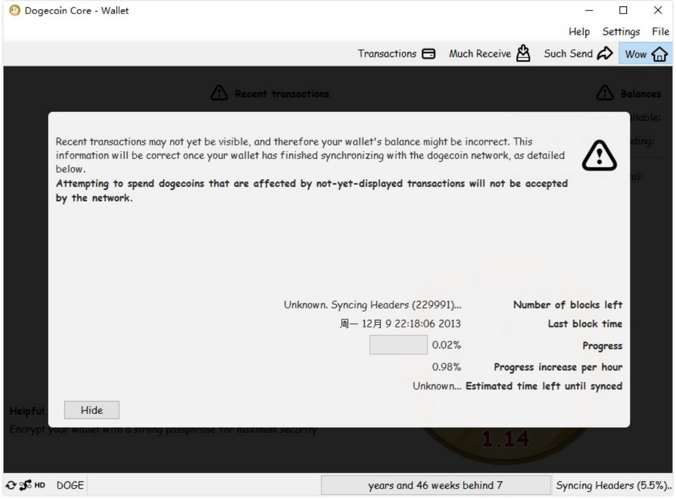
The result for Ethereum and its forked projects is much more acceptable than Bitcoin's forked projects. Note that we exclude Avalanche for the patch delay analysis because three of its fixed cases were included when Avalanche was first initialized. As shown in Fig. 6b, for the investigated six CVEs, Ethereum released all the patches within a short period, at most 22 days to be specific, and four patches were released within four days. Moreover, Ethereum's forked projects released all the investigated patches within 500 days, with more than 80% released within 200 days and half of the patches released around 100 days. Polygon is among the best, as it has six fixed cases whereas all of them were released within 110 days.

VI. Discussion

In this section, we further discuss some insights and implications about the propagated vulnerabilities in forked blockchain projects, as well as their defense and detection.

Attacks against the discovered vulnerabilities. Since the cloned vulnerability typically has a similar code context with the original vulnerability, the vulnerability behavior is also likely to be identical. As a result, an adversary can launch the same attack against the forked project that contains the cloned vulnerability, with only minor alterations. For instance, in the case of CVE-2021-3401 identified in the five forked projects of Bitcoin, we followed a write-up [32] that presented the details of vulnerability behavior and its complete exploitation and successfully exploited the discovered cloned vulnerability in all five projects. Specifically, the root cause of CVE-2021-3401 is that the GUI program of Bitcoin (and its five forked projects) misuses Qt-framework's built-in arguments. For example, by appending the argument `-reverse` to Dogecoin's invoking command with any wallet address, i.e., `dogecoin-qt.exe dogecoin:3E8ociqZa9mZUSwGdSmAEMAoAxBK3FNDcd -reverse`, we demonstrate that adversaries could change the program behavior by showing a reversed GUI in Fig. 7. It is worth noting that real adversaries can use other more dangerous arguments, such as `-platformpluginpath` mentioned in Sec. V-A. Here we use `-reverse` for easier demonstration.

Another similar case is CVE-2019-15947, which introduces the wallet information leakage problem. Since the earlier version of Bitcoin stores `wallet.dat` unencrypted in memory, upon a crash, it may dump a core file that could be used to reconstruct users' `wallet.dat`, including private keys. The original Bitcoin PR#16824 [7] not only reported this vulnerability but also provided a shell script to exploit it. We leveraged this script with only minor alterations to successfully exploit this vulnerability in six forked projects of Bitcoin.



Recent transactions may not yet be visible, and therefore your wallet's balance might be incorrect. This information will be correct once your wallet has finished synchronizing with the dogecoin network, as detailed below.
Attempting to spend dogecoins that are affected by not-yet-displayed transactions will not be accepted by the network.

Unknown. Syncing Headers (229991)... **Number of blocks left**
周一 12月 9 22:18:06 2013 **Last block time**
0.02% **Progress**
0.98% **Progress increase per hour**
Unknown... **Estimated time left until synced**

years and 46 weeks behind 7
Syncing Headers (5.5%)..

Fig. 7: A demo of exploiting CVE-2021-3401 in Dogecoin.

Defense or best practice for developers. According to our investigation in Sec. V-A, there are three types of propagated vulnerabilities from Bitcoin/Ethereum to the forked projects, i.e., the `fork`, `fetch`, and `mixed` types. To avoid them, developers may follow the two principles: (i) try to avoid introducing vulnerable commits to the forked projects; and (ii) if a vulnerability is already introduced, developers should apply the patch code as soon as possible. For (i), before fetching commits from the source project, developers should perform a static detection of the project (e.g., using our tool) and carefully review the commit code as well as the commit message. For (ii), developers should conduct security backports regularly and actively keep up with the source projects' issues/PRs. For instance, although CVE-2021-3401 was disclosed in February 2021, the vulnerability was reported in Bitcoin PR#16578 [12] much earlier on 10 August 2019. This vulnerability would not have existed for such a long period if developers of the forked projects had followed Bitcoin's issue/PR.

Implications on Type-4 clone detection. As Type-4 clones refer to semantically equivalent but syntactically different code, understanding code semantics is necessary for such detection. Specifically, there are two possible directions, static/dynamic program analysis (e.g., [38], [45], [48], [51]) and machine learning on code semantics (e.g., [72], [78], [86]). In particular, [51] and [38] used the representation of isomorphic program dependence graph (PDG) as semantic clones. Jiang et al. [45], on the other hand, regarded that given the same input, if the outputs of two code fragments are the same, they are equivalent. Therefore, they used random testing on different code fragments and found the ones with the same behavior. MeCC [48] proposed a memory-based approach to detect semantic clones, i.e., comparing the abstract memory states of two programs. Sheneamer et al. [72], CDPU [78], and PACE [86] applied machine learning to detect semantic clones. Specifically, Sheneamer et al. [72] applied classification algorithms on the extracted features of abstract syntax trees

(AST) and PDG. CDPU [78] proposed a positive-unlabeled learning model and adversarial training to improve detection performance, while PACE [86] presented an another deep learning approach by applying token-enhanced AST convolution.

VII. Related Work

In this section, we review the related work on blockchain vulnerability detection and clone-based vulnerability detection.

Blockchain vulnerability detection. Existing blockchain vulnerability detection mainly focused on the security of smart contracts. For instance, Oyente [58], Securify [77], ZEUS [46], ETHBMC [36], eThor [69], SmarTest [73], and SAILFISH [28] leveraged static analysis techniques, e.g., symbolic execution, to detect vulnerable smart contracts. On the other hand, Sereum [65] aimed to dynamically detect the reentrancy attacks [17] and protect the deployed smart contracts. Similarly, TXSPECTOR [87] designed a generic and flexible framework for identifying attacking transactions in Ethereum [80], and SODA [30] is another generic framework for attack detection. Lastly, Perez et al. [63] studied the possibility of exploiting the discovered smart contract vulnerabilities. Besides the research about smart contract vulnerability detection, DEFIER [75] automatically investigated the attack incidents of DApps (decentralized apps), which are built on the top of smart contracts. Additionally, EVMPatch [66] proposed a framework for instantly and automatically patching faulty smart contracts. SolType [76] designed a refinement type system for Solidity to prevent arithmetic over- and under-flows.

However, only a few studies focused on the vulnerabilities at the system level. Notably, Kwon et al. [52], Zhang et al. [88], and Yang et al. [84] investigated the consensus reward flaws and the consensus system bugs in the Bitcoin network and Ethereum clients, respectively. Yi et al. [85] systematically mined the existing vulnerabilities from four representative blockchains, Bitcoin, Ethereum, Monero, and Stellar, for security insights. Besides these works, three recent studies focused on the Bitcoin patch delay analysis that is most related to BlockScope's `Calculator` component. Specifically, CoinWatch [42] used four CVEs of Bitcoin to test and analyze the delay of many old Bitcoin's forked projects that are no longer maintained. It used the Simian the clone detector [1], i.e., simple string match, to detect only Type-1 clones. Similarly, Choi et al. [31] conducted a large-scale empirical analysis on the code maintenance activities of Bitcoin forks, with only limited information about security vulnerabilities. Another technical report, GitWatch [27], tried to accurately determine the patch *commit* delay from Bitcoin to its forked projects. Since `git` lacks reliable commit timestamps due to the `rebase` operation, it leveraged GitHub's event API and GitHub Archive to solve this problem. In contrast, BlockScope focused on the patch *release* delay that does not require the `git` commit timestamp, as explained in Sec. III-F.

Code clone-based vulnerability detection. Code clone detection is an attractive research area of computer security, as it has been shown that many bugs and vulnerabilities could be cloned from one software to another [49]. Unlike the traditional clone detection tools, such as CCFinder [47], CPMiner [57], DECKARD [44], and SourcererCC [68], security-oriented clone detection tools like ReDeBug [43], VUDDY [50], MVP [82], and VGraph [29] considered both vulnerable and patched code inputs. Specifically, ReDeBug [43] was among the most representative works in this direction, and it has been widely used because of its generality and public code. Following ReDeBug, VUDDY [50] added variable/parameter/type/function abstraction as a preprocessing and used the generated fingerprints for more scalable code clone detection. Similarly, MVP [82] and VGraph [29] conducted more "program analysis" in the form of program slicing [81] and code property graph [83] before similarity measurement to improve the detection accuracy. Compared with these three works, BlockScope took a completely different path by proposing more suitable candidate code search for our problem (as in Sec. III-C) and improving the core technique on how to better measure code similarity itself (as in Sec. III-E).

Recently, AI techniques have also been applied in clone-related vulnerability detection. Specifically, CLCDSA [60] utilized deep neural networks to detect cross-language code clones. Gao et al. [39] detected code clones in smart contracts by word embeddings. Ahmadiet et al. [26] leveraged machine learning-based methods to detect functionally-similar inconsistent code. DeepBugs [64], VulDeePecker [56], Devign [90], SySeVR [55], and VulDeeLocator [54] utilized various kinds of code features of known vulnerabilities to train deep learning models to identify new vulnerabilities with similar code features. Additionally, Serrano et al. [70] showed that similar yet different patches could share the same semantic and change patterns, while Zhang et al. [89] investigated the patch delays from Android AOSP to the OEM systems.

VIII. Conclusion

In this paper, we detected and investigated the vulnerabilities propagated from Bitcoin and Ethereum to their forked projects. To this end, we proposed BlockScope that leveraged novel context-based candidate search and a new way of calculating code similarity to efficiently and effectively identify Type-1/2/3 clones. BlockScope allowed us to discover 101 previously unknown vulnerabilities in 13 out of the 16 popular forked projects of Bitcoin and Ethereum, including 16 from Dogecoin, 6 from Litecoin, 1 from Binance, and 4 from Optimism. Moreover, the evaluation showed that BlockScope achieved good precision and high recall both at 91.8% (1.8 times higher recall than that in the state-of-the-art ReDeBug). We further investigated the propagation and patching processes of discovered vulnerabilities, and revealed three types of vulnerability propagation from Bitcoin/Ethereum to their forked projects, as well as the long delay (mostly over 200 days) for releasing patches in Bitcoin's forked projects (vs. $\sim$100 days for Ethereum forks). In the future, we will continue to improve BlockScope and expand its scope to none-blockchain domains, e.g., different Linux distributions.

Acknowledgment

We would like to thank all the reviewers for their valuable comments and constructive suggestions to this paper. This work was partially supported by a direct grant (ref. no. 4055127) from The Chinese University of Hong Kong.

References

[1] "Simian - similarity analyser," https://www.harukizaemon.com/simian/, 2013.

[2] "Vulnerability posing commit for CVE-2021-3401," https://github.com/bitcoin/bitcoin/commit/202d853bb, 2014.

[3] "Vulnerability posing commit for CVE-2018-17145," https://github.com/bitcoin/bitcoin/commit/08096bbb, 2017.

[4] "Bitcoin Inventory Out-of-Memory Denial-of-Service attack," https://invdos.net/, 2018.

[5] "Patch for Bitcoin PR#13080," https://github.com/bitcoin/bitcoin/pull/13808/commits/18f690ec, 2018.

[6] "Patch for Bitcoin PR#14624," https://github.com/bitcoin/bitcoin/pull/14624/commits/3db746be, 2018.

[7] "Bitcoin PR#16824," https://github.com/bitcoin/bitcoin/pull/16824, 2019.

[8] "Patch for Bitcoin PR#15305," https://github.com/bitcoin/bitcoin/pull/15305/commits/4433ed0f, 2019.

[9] "Patch for Bitcoin PR#16512," https://github.com/bitcoin/bitcoin/pull/16512/commits/6f405a1d, 2019.

[10] "Blockscan," https://blockscan.com, 2020.

[11] "Binance Chain vs Binance Smart Chain," https://coincodecap.com/binance-chain-vs-binance-smart-chain, 2021.

[12] "Patch for CVE-2021-3401," https://github.com/bitcoin/bitcoin/pull/16578/commits/a2714a5c, 2021.

[13] "Binance exchange," https://coinmarketcap.com/exchanges/binance/, 2022.

[14] "BscScan," https://bscscan.com/charts, 2022.

[15] "CoinMarketCap," https://coinmarketcap.com, 2022.

[16] "Dogecoin vs. Bitcoin: Key differences," https://cointelegraph.com/dogecoin-for-beginners/dogecoin-vs-bitcoin-key-differences, 2022.

[17] "Ethereum smart contract best practices - Reentrancy," https://consensys.github.io/smart-contract-best-practices/known_attacks/, 2022.

[18] "Introducing BNB Chain: The evolution of binance smart chain," https://www.binance.com/en/blog/ecosystem, 2022.

[19] "Is Dash better than Bitcoin," https://cryptonews.com.au/guides/dash-vs-bitcoin, 2022.

[20] "Optimistic Rollups," https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/, 2022.

[21] "Polygon vs Arbitrum vs Optimism: May the best Ethereum layer 2 win," https://academy.youngplatform.com/en/blockchain/ethereum-layer-2-polygon-vs-arbitrum-vs-optimism/, 2022.

[22] "The Qt framework," https://www.qt.io/, 2022.

[23] "Scrypt," https://en.wikipedia.org/wiki/Scrypt, 2022.

[24] "Selecting layer 2: Polygon vs Arbitrum vs Optimism," https://pixelplex.io/blog/polygon-vs-arbitrum-vs-optimism-comparison/, 2022.

[25] "What is DigiByte cryptocurrency," https://crypto-explained.com/services/what-is-digibyte-cryptocurrency/, 2022.

[26] M. Ahmadi, R. M. Farkhani, R. Williams, and L. Lu, "Finding bugs using your own code: Detecting functionally-similar yet inconsistent code," in *Proc. USENIX Security*, 2021.

[27] S. Andreina, L. Alluminio, G. A. Marson, and G. Karame, "Estimating patch propagation times across (blockchain) forks," *CoRR arXiv*, vol. abs/2205.07478, 2022.

[28] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "SAILFISH: Vetting smart contract state-inconsistency bugs in seconds," in *Proc. IEEE Symposium on Security and Privacy*, 2022.

[29] B. Bowman and H. H. Huang, "VGraph: A robust vulnerable code clone detection system using code property triplets," in *Proc. IEEE EuroS&P*, 2020.

[30] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, Y. Tang, X. Lin, and X. Zhang, "SODA: A generic online detection framework for smart contracts," in *Proc. ISOC NDSS*, 2020.

[31] J. Choi, W. Choi, W. Aiken, H. Kim, J. H. Huh, T. Kim, Y. Kim, and R. Anderson, "Attack of the clones: Measuring the maintainability, originality and security of Bitcoin 'forks' in the wild," *CoRR arXiv*, vol. abs/2201.08678, 2022.

[32] A. Chow, "URI argument injection vulnerability in Bitcoin Core 0.18 and earlier," https://achow101.com/2021/02/0.18-uri-vuln, 2021.

[33] K. Dwyer, "Litecoin vs. Bitcoin," https://coinmarketcap.com/alexandria/article/litecoin-vs-bitcoin, 2021.

[34] B. Fenton and T. Black, "Ravencoin: A peer to peer electronic system for the creation and transfer of assets," https://ravencoin.org/, 2018.

[35] C. Foundation, "DeFi with Celo and Ethereum," https://medium.com/celoorg/defi-with-celo-and-ethereum-f978d9dc547f, 2021.

[36] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *Proc. USENIX Security*, 2020.

[37] J. Frankenfield, "Qtum," https://www.investopedia.com/terms/q/qtum.asp, 2021.

[38] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proc. ACM ICSE*, 2008.

[39] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, 2020.

[40] D. Hamilton, "Avalanche Vs. Ethereum - what's the difference?" https://www.securities.io/avalanche-avax-vs-ethereum-eth-everything-you-need-to-know/, 2022.

[41] C. Hoffman, "What's the difference between Bitcoin, Bitcoin Cash, Bitcoin Gold, and others," https://www.howtogeek.com/349263/, 2022.

[42] Q. Hum, W. J. Tan, S. Y. Tey, L. Lenus, I. Homoliak, Y. Lin, and J. Sun, "CoinWatch: A clone-based approach for detecting vulnerabilities in cryptocurrencies," in *Proc. IEEE Blockchain*, 2020.

[43] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire OS distributions," in *Proc. IEEE Symposium on Security and Privacy*, 2012.

[44] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. ACM ICSE*, 2007.

[45] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proc. ACM ISSTA*, 2009.

[46] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. ISOC NDSS*, 2018.

[47] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, 2002.

[48] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: Memory comparison-based clone detector," in *Proc. ACM ICSE*, 2011.

[49] S. Kim and H. Lee, "Software systems at risk: An empirical study of cloned vulnerabilities in practice," *Elsevier Computers & Security*, 2018.

[50] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *Proc. IEEE Symposium on Security and Privacy*, 2017.

[51] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proc. Springer SAS*, 2001.

[52] Y. Kwon, D. Kim, Y. Son, E. Vasserman, and Y. Kim, "Be selfish and avoid dilemmas: Fork After Withholding (FAW) attacks on Bitcoin," in *Proc. ACM CCS*, 2017.

[53] Y. Li and B. Liu, "A normalized Levenshtein distance metric," *IEEE Transactions On Pattern Analysis and Machine Intelligence*, 2007.

[54] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "VulDeeLocator: A deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[55] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[56] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. ISOC NDSS*, 2018.

[57] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *Proc. USENIX OSDI*, 2004.

[58] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM CCS*, 2016.

[59] S. Mancoridis, "Code clone introduction," https://courses.cs.vt.edu/cs5704/spring16/handouts/5704-10-CodeClones.pdf, 2016.

[60] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "CLCDSA: Cross language code clone detection using syntactical features and API documentation," in *Proc. ACM ASE*, 2019.

[61] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *White paper*, 2008.

[62] C. News, "ZenCash vs. Zcash: All you need to know," https://medium.com/@importprivkey/zencash-vs-zcash-all-you-need-to-know-300065c9d0d3, 2018.

[63] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," in *Proc. USENIX Security*, 2021.

[64] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," *Proc. ACM on Programming Languages*, 2018.

[65] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proc. ISOC NDSS*, 2019.

[66] ——, "EVMPatch: Timely and automated patching of Ethereum smart contracts," in *Proc. USENIX Security*, 2021.

[67] M. Saad, J. Spaulding, L. Njilla, C. Kamhoua, S. Shetty, D. Nyang, and A. Mohaisen, "Exploring the attack surface of blockchain: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, 2020.

[68] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big code," in *Proc. ACM ICSE*, 2016.

[69] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "eThor: Practical and provably sound static analysis of Ethereum smart contracts," in *Proc. ACM CCS*, 2020.

[70] L. Serrano, V.-A. Nguyen, F. Thung, L. Jiang, D. Lo, J. Lawall, and G. Muller, "SPINFER: Inferring semantic patches for the Linux kernel," in *Proc. USENIX ATC*, 2020.

[71] SFOX, "Ravencoin vs. Bitcoin: How to transfer truth," https://www.sfox.com/blog/ravencoin-vs-bitcoin-how-to-transfer-truth/, 2020.

[72] A. Sheneamer and J. Kalita, "Semantic clone detection using machine learning," in *Proc. IEEE ICMLA*, 2016.

[73] S. So, S. Hong, and H. Oh, "SmarTest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution," in *Proc. USENIX Security*, 2021.

[74] C. Staff, "Qtum (QTUM): A hybrid blockchain merging Bitcoin and Ethereum," https://www.gemini.com/cryptopedia/qtum-crypto-and-blockchain-evm, 2021.

[75] L. Su, X. Shen, X. Du, X. Liao, X. Wang, L. Xing, and B. Liu, "Evil under the sun: Understanding and discovering attacks on Ethereum decentralized applications," in *Proc. USENIX Security*, 2021.

[76] B. Tan, B. Mariano, S. K. Lahiri, I. Dillig, and Y. Feng, "SolType: Refinement types for arithmetic overflow in Solidity," in *Proc. ACM POPL*, 2022.

[77] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM CCS*, 2018.

[78] H.-H. Wei and M. Li, "Positive and unlabeled learning for detecting software functional clones with adversarial training," in *Proc. IJCAI*, 2018.

[79] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt, "SoK: Decentralized finance (DeFi)," *CoRR arXiv*, vol. abs/2101.08778, 2021.

[80] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Yellow paper*, 2022.

[81] D. Wu, D. Gao, R. H. Deng, and R. K. C. Chang, "When program analysis meets bytecode search: Targeted and efficient inter-procedural analysis of modern Android apps in BackDroid," in *Proc. IEEE DSN*, 2021.

[82] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou, and W. Shi, "MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures," in *Proc. USENIX Security*, 2020.

[83] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symposium on Security and Privacy*, 2014.

[84] Y. Yang, T. Kim, and B.-G. Chun, "Finding consensus bugs in Ethereum via multi-transaction differential fuzzing," in *Proc. USENIX OSDI*, 2021.

[85] X. Yi, D. Wu, L. Jiang, Y. Fang, K. Zhang, and W. Zhang, "An empirical study of blockchain system vulnerabilities: Modules, types, and patterns," in *Proc. ACM ESEC/FSE*, 2022.

[86] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *Proc. IEEE ICPC*, 2019.

[87] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, "TxSpecTor: Uncovering attacks in Ethereum from transactions," in *Proc. USENIX Security*, 2020.

[88] R. Zhang and B. Preneel, "Lay down the common metrics: Evaluating Proof-of-Work consensus protocols' security," in *Proc. IEEE Symposium on Security and Privacy*, 2019.

[89] Z. Zhang, H. Zhang, Z. Qian, and B. Lau, "An investigation of the Android kernel patch ecosystem," in *Proc. USENIX Security*, 2021.

[90] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *NeurIPS*, 2019.

Appendix

A. r's Impact on Similarity Measurement

As illustrated in Sec. III-E, we introduced the reward factor r to adjust the ordering issue's influence on code similarity. By calculating all the patch and candidate code's similarities with different r, we can evaluate the impact of r on the similarity measurement. In Fig. 8, we plot the CDF of similarity with r from 0.15 to 0.95. As we can see, r has a more significant influence on the similarity when the similarity is low. Moreover, since we try to minimize false negatives, we need to include more candidate code in the analysis. As such, we should exclude fewer candidate code that has similarity below the threshold. According to Fig. 8, when $r = 0.95$, it has the least candidate code with similarity below 0.4. Therefore, we set 0.95 as the default value of r.

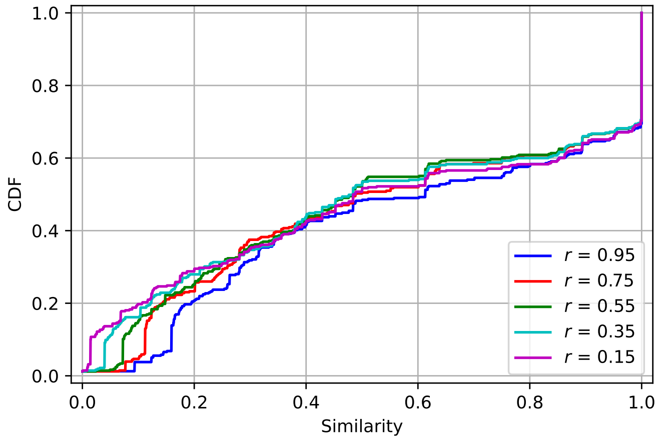


Fig. 8: The CDF plot of similarity with different r.