Estimating Patch Propagation Times across (Blockchain) Forks

Sebastien Andreina
NEC Labs Europe
Heidelberg, Germany
sebastien.andreina@neclab.eu

Lorenzo Alluminio
NEC Labs Europe
Heidelberg, Germany
lorenzo.alluminio@neclab.eu

Giorgia Azzurra Marson
NEC Labs Europe
Heidelberg, Germany
giorgia.marson@neclab.eu

Ghassan Karame
Ruhr University Bochum
Bochum, Germany
ghassan.karame@rub.de

ABSTRACT

The wide success of Bitcoin has led to a huge surge of alternative cryptocurrencies (altcoins). Most altcoins essentially fork Bitcoin's code with minor modifications, such as the number of coins to be minted, the block size, and the block generation time. As such, they are often deemed identical to Bitcoin in terms of security, robustness, and maturity.

In this paper, we show that this common conception is misleading. By mining data retrieved from the GitHub repositories of various altcoin projects, we estimate the time it took to propagate relevant patches from Bitcoin to the altcoins. We find that, while the Bitcoin development community is quite active in fixing security flaws of Bitcoin's code base, forked cryptocurrencies are not as rigorous in patching the same vulnerabilities (inherited from Bitcoin). In some cases, we observe that even critical vulnerabilities, discovered and fixed within the Bitcoin community, have been addressed by the altcoins tens of months after disclosure. Besides raising awareness of this problem, our work aims to motivate the need for a proper responsible disclosure of vulnerabilities to all forked chains prior to reporting them publicly.

1 INTRODUCTION

The wide success of Bitcoin has led to an explosion in the number of so-called "altcoins", i.e., cryptocurrencies designed as a fork of the Bitcoin-core code base. Although altcoins share—to a large extent—the same technical foundations of Bitcoin, they feature minor differences to Bitcoin. For instance, some altcoins exhibit a different block-generation time (e.g., Dogecoin and Litecoin), use a different hash function (e.g., Litecoin and Namecoin), or impose a different limit on the supply amount (e.g., Dogecoin, Litecoin) [19]. Often users and practitioners assume that altcoins offer a similar level of security and stability as Bitcoin-core. Indeed, the predominant choice for users to adopt a given altcoin over Bitcoin seems to hinge on the perceived profitability of mining that altcoin [9] rather than its actual security and maintenance effort of the developing team.

Bitcoin (and many of its descendants) have been found vulnerable to a wide variety of attacks. Gervais *et al.* [19] proposed the first quantitative framework to analyze the security and performance of proof-of-work blockchains, hinting that some altcoins might offer weaker security compared to Bitcoin, owing to the various, often ad-hoc parameters that they adopt. Thanks to its strong developing team, the Bitcoin-core software is routinely monitored and promptly patched. However, whether altcoins are also as proactive in fixing disclosed vulnerabilities remains unclear.

In this paper, we investigate the robustness and stability of altcoins from the perspective of code maintenance and patching. More specifically, we identify vulnerabilities reported in Bitcoin, and study how their patches are propagated through various altcoins. Our approach relies on the inspection of GitHub repositories of popular cryptocurrencies, to identify relevant bugs and corresponding patches in the commit history of GitHub-hosted altcoin projects. Concretely, we aim to estimate the time it took to propagate patches from Bitcoin to various altcoins, i.e., to determine whether and how quickly altcoins address disclosed security vulnerabilities.

However, retrieving detailed timing information associated to code changes in GitHub, especially in the context of Bitcoin forks, emerges as a challenging task. The reason is that most patches are taken directly from the Bitcoin-core repository and applied to a Bitcoin fork via a `rebase` operation which only exposes the timestamp of the original patch (applied to Bitcoin) and not the actual time when the patch was ported (to the altcoin) [17]. When patches are ported via rebasing, it is not possible to determine patch propagation times from the information recorded in GitHub. Indeed, every rebasing replaces all current commits in a fork by new commits that apply the same changes but with an updated and fresh timestamp. In addition, the original commits are no longer referenced after `rebase` occurs. As Git prunes unreferenced commits periodically, the timestamps associated to a given patch are lost with every subsequent `rebase` invocation.

To overcome this challenge we propose `GitWatch`, a tool to measure patch propagation times in Git-hosted forked projects even in the case of patches ported using the `rebase` command. `GitWatch` leverages GitHub's event API and GH archive to estimate the time when a given patch is applied to a forked project. `GitWatch` design relies on the following observation: although GitHub follows the same practices as Git, pruning unreferenced commits, internally it keeps a log of metadata information for all commits that ever existed. Importantly, this information can be retrieved through GitHub's API, as long as one can reference the relevant commits. Technically, `GitWatch` makes use of the GH archive service to generate a list of all commits pertaining to a given GitHub project, before querying GitHub to retrieve the commits' metadata. For each forked project under scrutiny, we harvest all events from GitHub using GH archive, and we reconstruct the full graph of commit operations—including unreferenced commits related to the sought patches. By monitoring all past GitHub commits instead

of inspecting the sole commit history of the fork, we are able to estimate realistic patching times despite rebasing.

We validate `GitWatch` in the context of analyzing the security of altcoin projects, and we study the time to propagate patches from Bitcoin to various altcoin projects. We consider 47 patches comprising 11 vulnerabilities reported in academic papers, 23 Bitcoin's Common Vulnerabilities and Exposures (CVEs), 3 major CVEs in libraries used by Bitcoin, 3 Bitcoin improvement proposals and 7 major bugs found on the GitHub repository with tags related to the peer-to-peer network, covering crucial vulnerabilities reported in the last decade (see Table 3). Our study investigates the security of five altcoins, which we selected among existing GitHub-based open-source forks of Bitcoin to ensure diversity in terms of market cap, popularity, and vision.

Our results (cf. Section 4) indicate that, for all of the selected altcoins, most patches have been applied with considerable delay compared to the release time of the Bitcoin fix, thereby leaving users running vulnerable software, that could be exploited by attackers, for several months or even years. In contrast to common belief, these findings suggest that Bitcoin forks—due to their less reactive software maintenance—may contain published vulnerabilities in their code base for several months, up to a few years for the worst case, showing therefore weaker robustness and security compared to Bitcoin Core.

We hope that our results motivate altcoin developers to promptly react to disclosed patches; another important purpose of our work is to motivate the need for a proper responsible disclosure of vulnerabilities to all altcoins prior to any publication of the vulnerability.

Notice that `GitWatch` is applicable to other forks beyond cryptocurrency projects, as it provides a workable means to extract reliable timing information of selective patches when these patches are ported via rebasing public repositories. For instance, `GitWatch` can be applied for the analysis of patch-propagation times in highly forked open source projects such as the Linux kernel [15], with more than 42000 forks, or the Bootstrap library [8], with more than 76000 forks.

2 BACKGROUND & PROBLEM

In this section, we overview preliminary concepts about Bitcoin, altcoins, and Git.

2.1 Bitcoin and Altcoins

Bitcoin is the first peer-to-peer system that implements a fully-decentralized cryptocurrency [27]. The Bitcoin client software, called "Bitcoin Core", is maintained by a large open source developer community and is regularly updated, reviewed, patched and tested [18].

A few years after the inception of Bitcoin in 2009, new cryptocurrencies were created and have since then mushroomed—at the time of writing there are more than 11'000 cryptocurrencies [10]. The reasons for introducing alternative cryptocurrencies are diverse, e.g., to improve Bitcoin's design, to support additional features, or to customize the protocol to specific applications. To this end, many projects are directly derived from Bitcoin by forking the official code base hosted on GitHub.

In this paper, we refer to alternative cryptocurrencies that are based on Bitcoin Core's software as *altcoins*. Prominent altcoins are Dash [11], initially renamed as "Darkcoin" for its intense adoption in dark-net markets; Digibyte [12], a cryptocurrency that was advertised by its creators for improving functionality via real-time difficulty adjustment, nearly instantaneous transactions with 15 seconds confirmation, and enhanced security compared to Bitcoin; Monacoin [4], a cryptocurrency created by an anonymous community in Japan that aimed at developing a national cryptocurrency payment system, as well as Litecoin [6] and Dogecoin [7], two famous cryptocurrencies that have emerged among the most popular first-generation derivatives of Bitcoin.

2.2 Lack of Reliable Patching Timestamps in Git

Git is a distributed version control tool [14]. Git resources are widely adopted for the development of collaborative, open-source projects (including cryptocurrencies), as they enable the various collaborators to easily and concurrently handle different versions of the source code. A Git-hosted project can be managed through various operations, providing users with read and write access to the files stored in the project repository. More concretely, a Git repository records the history of changes made by users in the form of a sequence of *snapshots*, so that one can inspect the repository's content at any point in time by retrieving the corresponding snapshot. To join a collaborative project, a user starts by cloning the content of the project's repository and synchronizing its local copy with the remote version (by invoking `pull`). This operation allows incorporating changes made by other users and obtaining the sequence of existing snapshot—akin to a "read" operation. In order to "write", a user can `commit` changes to its local directory and then `push` the commits to the remote repository.

Cloned projects typically port software patches (that have been applied to the parent project) via `rebase` operations. Unfortunately, every `rebase` invocation modifies the history of the fork's repository, in particular altering the timestamps of all commits re-applied to the fork, with the effect of erasing the timestamps of all previously ported patches. Therefore, retrieving accurate patch propagation times directly from the commit history of a fork is not possible in Git. This clearly prevents any reliable study of patch-propagation timing for cloned code.

3 MEASURING PATCH PROPAGATION TIMES IN GIT

In this section, we study the problem of analyzing the propagation times of patches in Git across forked projects (i.e., the time it takes a patch to be ported from the main project to the forked project). We begin with formalizing the various Git operations that are relevant to software patches, then we discuss in detail the effect of `rebase` operations on commit timestamps. Finally, we propose three heuristics that can be used to estimate the patch-propagation time for GitHub-hosted forked projects.

3.1 Git Operations

Commit. This command allows tracking changes made to one or more files in the repository. We define a commit as a pair $C = (M, D)$

of *metadata* and *data*, containing context information about the commit and the actual changes made by that commit, respectively. Below, we highlight the metadata fields that are most relevant to describe our methodology in the next section:

$$h : \text{commit hash (or commit ID)}, \tag{1}$$
$$p : \text{parent commit}, \tag{2}$$
$$a : \text{author}, \tag{3}$$
$$c : \text{committer}, \tag{4}$$
$$t_a : \text{author timestamp}, \tag{5}$$
$$t_c : \text{committer timestamp}, \tag{6}$$

where h is a cryptographic hash over the changes D along with the remaining metadata, i.e.:

$$h = H(p, a, c, t_a, t_c, D), \tag{7}$$

and it uniquely identifies a given commit; the parent p is a reference to the previous commit; the author a denotes the author of the changes introduced in the commit, while the committer c denotes the user who made the latest changes to that commit; the author timestamp t_a indicates when (date and time) the original commit operation was performed by the author, and the committer timestamp t_c records when the latest change was made (to that commit) by the committer. Whenever the commit is amended, the committer is replaced with the user who modified the commit and similarly the commit timestamp is updated to the current time. Metadata information is essential for examining the *history* of a repository.

Git allows associating *tags* to commit operations (e.g., to mark released versions of software). Given a commit C with commit hash h, a tag referencing C is defined by a tuple $\tau = (\lambda, t, h)$ where λ is a human-readable label (e.g., the version number of the release) and t is the timestamp of the tag.

Push. A user can apply its local changes to a remote repository by invoking `push`. This operation applies all (new) local commits to the remote repository. Each such commit corresponds to an updated snapshot of the repository's content. A "batch" of commits pushed by a given user forms a sequence $(C_1, \ldots, C_m)$ defined implicitly by the references to parent commits. Formally, if u denotes the user who created and pushed the commits, for $i = 1, \ldots, m$, we have:

$$C_i.a = C_i.c = u \quad \wedge \quad C_i.t_a = C_i.t_c, \tag{8}$$

i.e., author and committer coincide with the user who pushes the commit, and so do the author timestamp and committer timestamps. In addition, for $i = 2, \ldots, m$, it holds:

$$C_i.p = C_{i-1}.h \quad \wedge \quad C_i.t_c > C_{i-1}.t_c, \tag{9}$$

i.e., every commit, except for the first one in the sequence, has the previous local commit as parent, and the commit timestamps in the sequence progressively increase (i.e., the commit order reflects the chronological order of execution).

Pushing a sequence of commits triggers the addition of new snapshots, typically one per commit C_i, to the history of the repository. Let $\mathcal{CH}$ denote the commit history of the repository, i.e., the collection of commits that have been pushed so far. Then, pushing $(C_1, \ldots, C_m)$ has the effect of appending the commit sequence to the commit history:

$$\mathcal{CH} \xleftarrow{\text{push}} \mathcal{CH} \parallel \{C_1, \ldots, C_m\}. \tag{10}$$

Notice that the commit history is not always a sequence, due to commits pushed concurrently by different users. For instance, merged commits have the same parent commit, thus forming a directed acyclic graph (rather than a sequence). In the rest of the paper, to simplify the notation we write the commit history as a sequence.

Fork. A fork (a.k.a. *branch*) of an existing repository R is a repository R^{χ} that shares a common history with R—the latter is called the *main branch*. The latest commit that R and R^{χ} have in common is called *base commit*. Let $\mathcal{CH}$ and $\mathcal{CH}^{\chi}$ denote the commit histories of R and R^{χ} respectively. Then there exist $m, s, r \in \mathbb{N}$, $r > 0$, such that:

$$\mathcal{CH} = (C_0, \ldots, C_m, \ldots, C_{m+s}) \tag{11}$$
$$\mathcal{CH}^{\chi} = (C_0, \ldots, C_m, C_1^{\chi}, \ldots, C_r^{\chi}) \tag{12}$$

where C_m denotes the base commit and $C_1^{\chi}, \ldots, C_r^{\chi}$ are the commits diverging from the main branch. While forks typically proceed independently of their main branch, the developers of a forked project might still need to monitor the evolution of the main branch, e.g., to discover bugs and port patches.

Rebase. This operation allows integrating changes from the main branch R (e.g., Bitcoin) to a fork R^{χ} (e.g., an altcoin) by re-applying all commits pushed to R^{χ} starting from a new base commit in R—hence the term *rebase*. This operation is most often adopted to fetch the latest version of the original repository. Invoking `rebase` effectively "re-builds" the changes made in the fork on top of the new base commit, thereby modifying the commit history of the fork. Namely, suppose the commit histories of the two repositories are as follows:

$$\mathcal{CH} = (C_0, \ldots, C_m, \ldots, C_{m+s}), \tag{13}$$
$$\mathcal{CH}^{\chi} = (C_0, \ldots, C_m, C_1^{\chi}, \ldots, C_r^{\chi}). \tag{14}$$

Then, invoking `rebase` on R^{χ} with new base commit C_{m+k}, for $0 < k \leq s$, has the following effect on the commit history of R^{χ}:

$$\mathcal{CH}^{\chi} \xleftarrow{\text{rebase}} (C_0, \ldots, C_{m+k}, C_1'^{\chi} \ldots, C_r'^{\chi}), \tag{15}$$

where each commit $C_i'^{\chi}$, for $i = 1, \ldots, r$, is an updated version of the original commit C_i^{χ} adapting the metadata to the new base commit. Concretely, the committed changes (i.e., the data D) remain the same, i.e., for all $i = 1, \ldots, r$, it holds:

$$C_i'^{\chi}.D = C_i^{\chi}.D, \tag{16}$$

and the metadata is updated to reflect the replacement of the base commit C_m with C_{m+k}. This update modifies the first parent commit as follows:

$$C_1'^{\chi}.p \leftarrow C_{m+k}.h \tag{17}$$

which, in turn, triggers a chain reaction and modifies all subsequent parent commits, i.e., for $i = 2, \ldots, r$, we have:

$$C_i'^{\chi}.p \leftarrow C_{i-1}'^{\chi}.h \tag{18}$$

At a lower level, rebasing does not technically replace every commit C_i^{χ} with $C_i'^{\chi}$, it creates a new sequence $C_1'^{\chi}, \ldots, C_r'^{\chi}$ of commits, each with its own fresh commit ID. However, after rebasing the

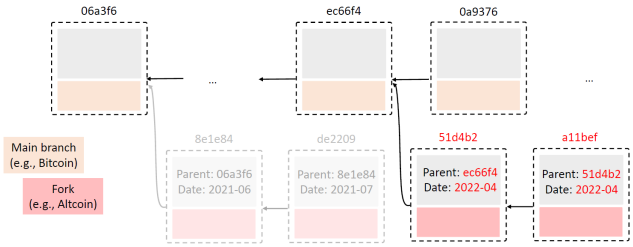


Figure 1: Effect of `rebase` on commit metadata: parent commit, committer date, and commit id are modified. Each dotted box represent a commit/snapshot, with metadata (top, gray) and data (bottom, colored), while arrows represent pointers to parent commits.

original commits are no longer referenced, hence they become "dangling" commits.

Finally, rebasing also has the crucial effect of updating the committer timestamp with the current time, while the author timestamp is preserved:

$$C_i'^{\chi}.t_c \leftarrow \text{'current time'} \quad \wedge \quad C_i'^{\chi}.t_a = C_i^{\chi}.t_a. \tag{19}$$

Rebasing makes timestamps unreliable. Although recording the rebasing time in the committer timestamp appears as a natural approach to maintain information about relevant events in $\mathcal{CH}^{\chi}$, it can cause the loss of relevant timing information in the case of multiple `rebase` operations being performed on the same repository. Indeed, every new `rebase` invocation preserves the author timestamp t_a of the original commits, however, it re-sets all committer timestamps t_c in the commit history to the current time—thereby overwriting all timestamps of previous `rebase` operations. This behavior is illustrated in Figure 1.

After a rebase, the old commits $C_1^{\chi}, \ldots, C_r^{\chi}$ become unreferenced and are called "dangling" commits. In order to save up space, dangling commits are automatically pruned by Git. However, when a rebase transforms commit C_i into C_i', the two commits are factually different due to their differences in the metadata and are therefore initially both accessible via their respective commit ID. Assuming no pruning, this observation provides us with a strategy to retrieve the timestamp of rebases: by listing all the different versions of a commit C_i and their respective committer timestamp. Our methodology described in Section 4 is based on this intuition to estimate the timing of rebases, yet it is compatible with the pruning of dangling commits.

3.2 Extracting timing information from GitHub

GitHub generates events for all operations on public repositories that can be subscribed to. To extract meaningful information about patch propagation time—even when the patch is applied via rebasing—we rely on two main resources: GitHub's event API and GH archive. GH archive [13] is an openly accessible service that provides the history of all events of GitHub since 2011. We observe that while rebases create dangling commits that are not retrieved when cloning, these commits can still be queried through GitHub's event API by requesting the corresponding hash.

GitHub events. Events are associated to one or more commits—e.g., a `push` operation triggers a corresponding event associated to the list of commits $(C_1, \ldots, C_m)$ pushed by the author. We define an event e as a tuple $e = (\boldsymbol{C_e}, t_e)$, where $\boldsymbol{C_e} = (C_1, \ldots, C_m)$ is the list of commits associated to the event, and t_e is the event timestamp recorded in GitHub. We denote by ϕ the mapping from commits to events, i.e., for an event $e = (\boldsymbol{C_e}, t_e)$, we define $\phi(C_i) := e$ for all $i = 1, \ldots, m$. With a slight abuse of notation, we write $C \in e$ to indicate that commit C belongs to the sequence of commits $\boldsymbol{C_e}$ associated to event e. Using this notation, we have $C \in e \iff \phi(C) = e$.

Using GH archive to retrieve all the hashes pertaining to commits of a given GitHub project, we are able to reconstruct the full graph of commits by fetching the metadata of those commits through GitHub's API—including dangling commits that are pruned following rebase operations. We effectively use GH archive to find the commit hashes and then query the metadata of those hashes through GitHub's API. By inspecting the full tree of Git commits along with their timestamps provided by the GitHub event API, we are able to estimate the time when a patch was introduced in each of the analyzed altcoins using different heuristics. We then compare these estimates to the original timestamp of the Bitcoin patch, so that we can derive (an estimate for) the patch propagation time from Bitcoin to the altcoin.

Graph Generation. Our methodology to obtain the timestamp of a patch is based on the investigation of dangling commits. To this end, for each altcoin χ we build a graph of all commits, including dangling commits. Formally, graph $\mathcal{G}_{\chi} = (V, E)$ contains all commits C of altcoin χ as vertices, while the edges represent the parent to child relationship:

$$C \in \mathcal{CH}^{\chi} \implies C \in V \wedge (C.p, C) \in E. \tag{20}$$

Given $\mathcal{G}_{\chi}$, we consider the following three heuristics to estimate the patch propagation times.

Patch-commit finder (PCF). The first heuristic relies on the concept of "non-Bitcoin child commits" for a given fork χ. Intuitively, given $\mathcal{G}$, we locate a non-Bitcoin commit C_j^* that contains the patch in its history, and use the committer date of C_j^* as the patching date. The patch propagation time is then estimated to be the time between the committer timestamp of the located commit and the timestamp of the original commit.

Formally, given a Bitcoin commit C_i, i.e., $C_i \in \mathcal{CH}^{BC}$, we define the set of non-Bitcoin child commits of C_i as follows:

$$\begin{aligned} C_j \in \text{nbcc}_{\chi}(C_i) \iff C_j \notin \mathcal{CH}^{BC} \ \wedge \ C_j \in \mathcal{CH}^{\chi} \ \wedge \\ \exists (E_1, .., E_n) \in \mathcal{G}_{\chi} \ : \ E_1.\texttt{from} = C_i \wedge E_n.\texttt{to} = C_j \ \wedge \\ \forall i \in [1, n-1], \ E_i.\texttt{to} = E_{i+1}.\texttt{from}. \end{aligned} \tag{21}$$

In order to estimate the propagation delay of a given patch commit $C_i \in \mathcal{CH}^{BC}$, from Bitcoin to an altcoin χ, PCF locates a[1] commit $C_j^* \in \mathcal{CH}^{\chi}$ such that:

$$C_j^* \in \text{nbcc}_{\chi}(C_i) \ \wedge \ C_j^*.t_c \leq C_j.t_c \ \forall C_j \in \text{nbcc}_{\chi}. \tag{22}$$

Finally, we define the estimated patch propagation delay Δ^{PCF} as:

$$\Delta^{\text{PCF}} \leftarrow C_j^*.t_c - C_i.t_a. \tag{23}$$

[1] A commit C_j^* with this property is not necessarily unique, as multiple commits can have the same committer timestamp.

Patch-event finder (PEF). Our second heuristic relies, in addition to the graph of commits $\mathcal{G}_\chi$, also on the mapping ϕ between commits and the events they belong to. Similarly to PCF, we look for the earliest non-Bitcoin commit C_j^* that contains the patch C_i in its history, the only difference being the meaning of the term "earliest": here, we measure elapsed time with respect to event timestamps, rather than commit timestamps. According to PEF, the patch propagation time is defined as the time span between the creation of the patch commit C_i and the oldest event that references a commit C_j that has C_i in its history.

Formally, let $\mathcal{E}_\chi$ denote the set of events pertaining to the altcoin χ and recorded in the GH archive:

$$\mathcal{E}_\chi := \{e \mid \exists C \in \mathcal{CH}^\chi : \phi(C) = e\}. \tag{24}$$

According to PEF, a relevant commit C_j^* meets the following conditions:

$$\begin{gathered} C_j^* \in \text{nbcc}_\chi(C_i) \cap \mathcal{E}_\chi \wedge \\ \phi(C_j^*).t \leq \phi(C_j).t \; \forall C_j \in \text{nbcc}_\chi(C_i) \cap \mathcal{E}_\chi. \end{gathered} \tag{25}$$

We define the estimated patch propagation time Δ^{PEF} as follows:

$$\Delta^{\text{PEF}} \leftarrow \phi(C_j^*).t - C_i.t_a. \tag{26}$$

Patch-tag finder (PTF). The third heuristic we propose is based on timestamps recorded for relevant tags. Intuitively, we estimate the patch propagation time as the time between the creation of the Bitcoin patch C_i and the creation of the first non-Bitcoin tag that links to a commit in χ that has the patch C_i in its history.

Formally, we define the concept of "non-Bitcoin child tag" analogously to that of non-Bitcoin child commit:

$$\begin{gathered} \tau \in \text{nbct}_\chi(C_i) \iff \tau.h \in \text{nbcc}_\chi(C_i) \wedge \\ \tau \in Tags_\chi \wedge \tau \notin Tags_{BC}. \end{gathered} \tag{27}$$

According to PTF, the tag τ^* relevant to the patch C_i is defined as follows:

$$\tau^* \in \text{nbct}_\chi(C_i) \;\wedge\; \tau^*.t \leq \tau.t \; \forall \tau \in \text{nbct}_\chi(C_i). \tag{28}$$

Finally, PTF estimates the patch propagation delay as follows:

$$\Delta^{\text{PTF}} \leftarrow \tau^*.t - C_i.t_a. \tag{29}$$

Comparison between heuristics. Assuming successful retrieval of all dangling commits, the patch-event finder (PCF) can identify every relevant commit C^* even when multiple rebase operations occurred, by listing all the different versions of C^* (one per rebase), thus allowing us to select the most accurate commit timestamp. With other words, PCF overcomes the problem of retrieving reliable timestamps in the presence of rebasing (c.f. Section 3.1). The major limitation of PCF is that it could under-approximate the patching time, say by Δ, in case a developer creates the commit locally (or on a dev branch) but waits some time Δ before pushing the patch to the repository main branch (e.g., for testing the patched code locally). It further relies on the local clock of the developer which could be skewed or maliciously altered.

Due to its similarity to the graph-based heuristic, PEF suffers from the same limitation: it could provide a too pessimistic timestamp in case the relevant event has been missed by the event API. On the other hand, it has the advantage over PCF that the timestamp cannot be faked or wronged due to a skewed clock.

We expect the tag-based method PTF to output the most accurate estimate. Indeed, most users do not compile the latest modifications based on the current version of the main branch (which may be unstable); they are more likely to use released versions of the code, which are marked with tags.

3.3 Putting all together—`GitWatch`

We leverage the aforementioned analysis in devising our tool, `GitWatch`. To measure the propagation time of patches for a project χ, `GitWatch` first has to build the graph $\mathcal{G}_\chi$. Here, `GitWatch` crawls GH archive for all events pertaining to χ in order to retrieve all the commits from GitHub's API. Given a patch commit C_i within $\mathcal{G}_\chi$, `GitWatch` leverages PCF, PEF and PTF to determine respectively $\Delta_P CF$, $\Delta_P EF$ and $\Delta_P TF$. The reliance on all three heuristics helps in eliminating possible false positives that may arise due to missing events in the GH archive. Whenever we obtain different results from the heuristics, `GitWatch` returns the smallest timeframe by default, regardless of which method produced it. This is the best case for the developers, as some heuristic can output too optimistic values.

To validate the soundness of the proposed heuristics, we use PTF as a pessimistic estimate of the propagation time; indeed, a fix may have been introduced before the information provided by the tag, but could not have been produced afterwards. We use this information for cross-validation with the other two approaches, and ensure that the results of the other approaches were consistent with the information constructed from the tag-based analysis.

Since all three heuristics rely on inspecting Git events, a malfunctioning of the GH archive could harm the accuracy of our tool. More specifically, if the GH archive misses a relevant event (and its corresponding commit), as a consequence our heuristics could over-estimate the patching time. Another limitation of our heuristics is that they exclusively inspect commits that are either rebased or merged with the same code base: patches introduced with a different code base may not be identified by our tool. For instance, if an altcoin relies on a "self-made" patch , and then rebases a few months/years later to the latest version of Bitcoin, our heuristics will point to the time of the rebase.

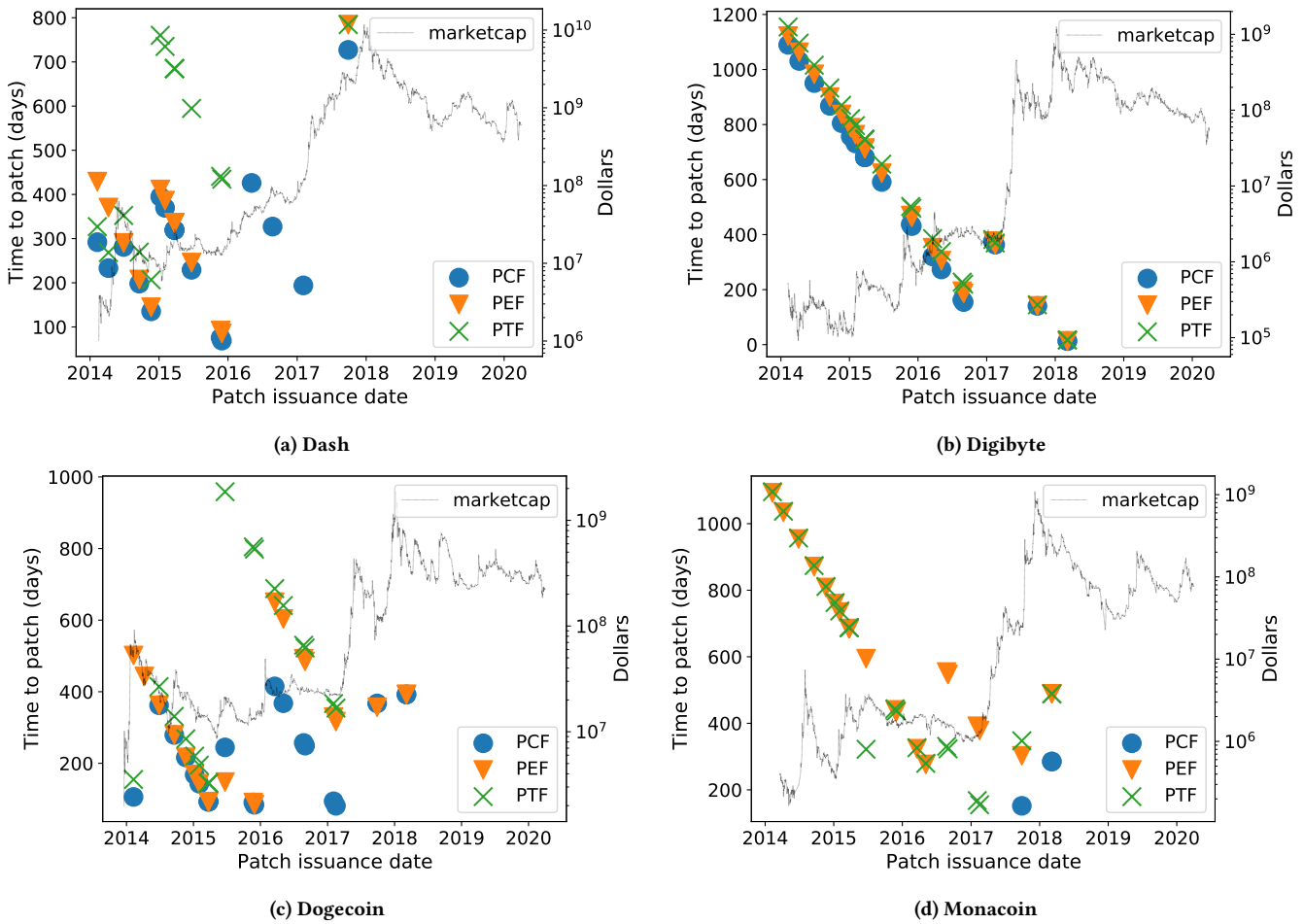
4 METHODOLOGY & EVALUATION

In this section, we validate our heuristics by comparing their effectiveness using ground truth data. We then use our heuristics to estimate the time it took to address a large number of patches in Dash, Digibyte, Monacoin, Litecoin, and Dogecoin.

4.1 Patch selection

In our evaluation, we restrict our analysis to bugs and patches related to Bitcoin, in particular, how they are propagated through altcoins that are based on the same code base. Since we are interested in patches that are not specific to Bitcoin but relevant to most altcoins, we mainly focus on reported bugs on the peer-to-peer layer as this layer is generally inherited by altcoins (including those that introduce non-negligible modifications to the code base).

We focused on analyzing five altcoins which we selected among existing open-source forks of Bitcoin, namely Dash [11], initially known for its early adoption in darknet markets, currently worth



(a) Dash

(b) Digibyte

(c) Dogecoin

(d) Monacoin

Figure 2: Time for a patch issued by Bitcoin-core to be included to the different altcoins. The blue circle, the orange triangle, and the green cross represent respectively the output values given by the graph, the event, and the tag approach. The market capitalisation over time of each coin is plotted as a black dotted line against the values on the right y-axis.

2.65 Billion USD; Digibyte [12], a cryptocurrency advertised for its improved functionality and security, currently worth 1.12 Billion USD; Monacoin [4], a cryptocurrency aimed to become a national payment system in Japan, currently worth 0.11 Billion USD; Litecoin [6] and Dogecoin [7], which emerge among the most popular first-generation derivatives of Bitcoin with a market capitalization of 14.82 Billion USD and 40.07 Billion USD respectively.

We then selected a list of 47 Bitcoin commits, 11 representing patches suggested by academic papers [20, 21, 26], 23 representing patches of CVEs, 3 representing Bitcoin improvement proposals, 3 representing CVEs in libraries used by Bitcoin and the remaining 7 representing bugs found on the GitHub repository with tags related to the peer-to-peer network. These patches include the majority of network and peer-to-peer vulnerabilities that were reported in the last decade.

Notice that as the altcoins' software is becoming more mature with each passing year, we have only few datapoints for the last couple of years. This is due to the small number of reported vulnerabilities and CVEs targeting Bitcoin-core recently (most recent CVEs target peripheral software, such as the Lightning Network).

4.2 Validation of our heuristics

To validate the effectiveness of our heuristics, we manually searched for publication dates of relevant patches (by investigating corresponding release notes), and we compared these dates with the output of our heuristics for the same vulnerability. Our results, shown in Table 1, confirm that for all the ground-truth data point we found, the actual patching time falls within the interval reported by our heuristics (i.e., between the minimum and maximum estimated propagation time). As expected, PTF emerges as the most precise heuristic, especially since release notes are usually part of a new release to which a tag is assigned— while PEF is the least reliable as a missed event in GJ archive could cause an over-estimation of the patch propagation time of several months.

Table 1: Ground-truth data to validate the effectiveness of our heuristics.

Vulnerability	Altcoin	Time	PCF	PEF	PTF
BIP 65	Litecoin	179 days[2]	159	160	181
BIP 65	Dogecoin	958 days [3]	244	147	958
BIP 66	Dogecoin	194 days[4]	142	142	194
CVE-2013-4627	Litecoin	33 days[5]	17	45	18
CVE-2013-4165	Litecoin	28 day[5]	10	529	13

4.3 Analysis

We now analyse the time it took to patch each of the 47 selected vulnerabilities for the five studied altcoins.

As shown in Figures 2 and 3, our three heuristics provide similar timing estimates, which converge in most cases, thereby supporting the soundness of this approach. Dash (Figure 2a) appears to port patches more quickly, compared to the other blockchains, most of the times with a delay between 200 and 400 days. Dogecoin and Litecoin instead (Figures 2c and 3) show more variable patching delays, ranging between 50-600 days, respectively, and 100-500 days on average. Digibyte and Monacoin (Figures 2b and 2d) exhibit an apparent linearly decreasing delay—partly visible also for the other analyzed blockchains. This peculiar behavior suggests that `rebase` operations to import the Bitcoin's patches are executed on a regular pace, in a manner that appears to be decoupled from the actual patch release. This would explain the downward lines in the plots, indicating that groups of patches are actually ported on the corresponding fork at the same time. To summarize, all five analyzed altcoins apply patches with a delay between several months to a few years. Based on these results, we conclude that disclosed vulnerabilities remain in the software for several months, as we discuss in detail in Section 5.

We include the detailed results of our study in Table 3. Our results show that Bitcoin issues patches to most critical vulnerabilities and CVEs in a prompt manner, often before the publication of vulnerability (i.e., in compliance with the vulnerability disclosure process). That said, there were some cases where Bitcoin took some months to issue a less critical patch reported from the academic community (e.g., forwarding of double spending attempts).

5 CASE STUDIES

We now look more closely at two specific vulnerabilities found in Bitcoin: the first one, disclosed in an academic paper, allows adversaries to tamper with the block delivery of honest users [20]; the second one, CVE-2017-18350 [5], is a reported vulnerability of the Bitcoin software. We selected these vulnerabilities because they are prominent and relatively recent (disclosure in 2015 and 2017 respectively). Although both vulnerabilities were patched and released by the Bitcoin-core team, some altcoins—still worth several billions of dollars—applied patches only months or even years after disclosure.

[2]https://github.com/litecoin-project/litecoin/blob/v0.10.4.0/doc/release-notes-litecoin.md
[3]https://github.com/dogecoin/dogecoin/releases/tag/v1.14-alpha-1
[4]https://github.com/dogecoin/dogecoin/releases?q=BIP66&expanded=true
[5]https://litecoinmirror.wordpress.com/2013/09/04/litecoin-0-8-4-1-release-notes/amp/

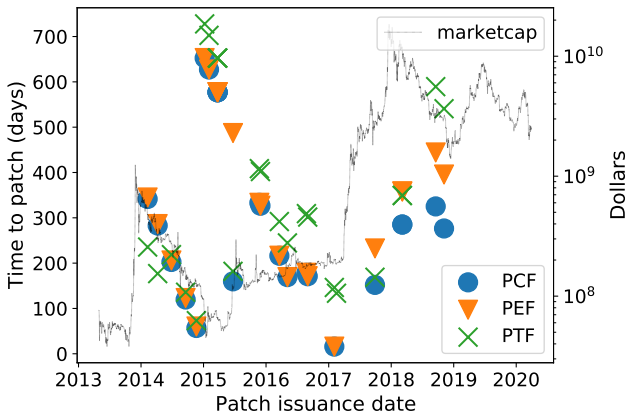


Figure 3: Time for Litecoin to include a patch issued on Bitcoin-core, similar to Figure 2.

5.1 Case Study 1: Tampering with the Delivery of Information in Bitcoin [20]

Summary of the vulnerability. In order to sustain higher throughput and scalability, Bitcoin implemented a number of optimizations and scalability measures. Gervais *et al.* [20] show that some of those measures come at odds with the security of the system—effectively allowing adversaries to tamper with the delivery of blocks/transactions to nodes. As a direct outcome of this vulnerability, a resource-constrained adversary would be able to mount a large-scale Denial-of-Service attack on Bitcoin—effectively halting the delivery of all blocks and transactions in the system.

The authors suggested various improvements that resulted in multiple patches, particularly the following three:

- Patch 1 - f59d…1392 [1], penalizing nodes that do not respond to block requests after advertising it.
- Patch 2 - 5029…1bff [2], enforcing to accept only one INV message per IP address in order to prevent adversaries from filling up the INV log of nodes.
- Patch 3 - 5026…ca1c [3], replacing the invitation message with the full block header, allowing nodes to verify that the received message actually depicts a correctly mined block.

Patching time. As shown in Table 2, Dash and Dogecoin took roughly 3 months to port these patches from the Bitcoin repository. On the other hand, Monacoin, Litecoin and Digibyte required between 7 months and 3 years.

	Patch 1 [1]	Patch 2 [2]	Patch 3 [3]
Bitcoin	**2014-02-08**	**2015-11-23**	**2015-11-29**
Dogecoin	106 days	90 days	84 days
Monacoin	1092	440 days	433 days
Litecoin	235 days	333 days	326 days
Digibyte	1089 days	437 days	431 days
Dash	291 days	75 days	69 days

Table 2: Time (from the original Bitcoin patch) to apply the mitigation suggested by Gervais *et al.* [20].

5.2 Case Study 2: CVE-2017-18350

Summary of the vulnerability. CVE-2017-18350 [5] is a buffer-overflow vulnerability of the Bitcoin-core software. The vulnerability was located in the proxy support and would enable a malicious proxy server to overwrite the program stack, allowing it to perform remote code execution. However, to be vulnerable, the wallet software needs to be configured to use a malicious proxy, therefore reducing the general risk on the users. Since remote code execution could allow any third party full access to the machine running the node, we deemed that this CVE is of particular interest due to its potential drastic impact.

Patching time. This vulnerability was discovered on September 21st, 2017 and was patched two days later, on September 23rd. Moreover, the patch was merged with the main branch of the Bitcoin-core repository four days later on September 27th, 2017, and was included in the subsequent releases.

To give enough time to the users for applying the patch, the CVE itself was published only on the June 22nd, 2019. While this patch was applied directly to most of the different altcoins based on the Bitcoin-core software, Dash [11] only patched it several months after it was published, on November 19th, 2019. While no attack performed through this vulnerability was reported as far as we are aware, Dash users were seemingly using a vulnerable software with no available update for several months after the disclosure of the vulnerability.

Dogecoin, Digibyte, Monacoin and Litecoin took respectively 91 days, 140 days, 151 days and 151 days to patch this vulnerability after it was discovered. While they all patched it before the vulnerability was disclosed, the software still remained unpatched for several months.

6 RELATED WORK

The Bitcoin protocol and many of its descendants have been found vulnerable to a wide variety of attacks. Gervais *et al.* [19] propose a quantitative framework to analyze the security and performance of proof-of-work blockchains with respect to network propagation, block sizes, block-generation intervals, information-propagation mechanism, and the impact of eclipse attacks. An analysis of Bitcoin's vulnerabilities from a network-security perspective is proposed by Apostolaki *et al.* [16], demonstrating partitioning- and delay attacks via hijacking a small number of prefixes. Security vulnerabilities have been found also for cryptocurrencies other than proof-of-work blockchains. Kanjalkar *et al.* [25] present a resource exhaustion attack affecting various PoS cryptocurrencies whose code base was forked from a variant of Bitcoin, leveraging that the affected cryptocurrencies did not properly validate the proof of stake before allocating resources to peers. Heilman *et al.* [22] demonstrated a collision-finding attack on IOTA's cryptographic hash function Curl-P-27, leading to forging signatures and multi-signatures of valid spending transactions.

Jia *et al.* [24] compare various altcoins based on code similarities and study the correlation between code innovations and market capitalization. Despite focusing on different aspects than security, the authors suggest that code similarity might indicate inherited vulnerabilities. In [23], Hum *et al.* propose a code evolution technique and a clone detection technique to indicate which cryptocurrencies are vulnerable once a vulnerability has been discovered. However, similar to all GitHub parsers, these techniques cannot infer when a given patch has been ported onto an altcoin in case of rebase operations since such timestamps are overwritten by `rebase`.

7 CONCLUSION

In this paper, we showed that various altcoins exhibit weaker stability compared to Bitcoin Core. Beyond confirming the folklore result that patch propagation is slow for some altcoin, we introduced a new technique to estimate patch-propagation times (which is non-trivial for GitHub forks), we shed light on the time for altcoins to propagate security patches, and which altcoins are faster in adopting a patch. For instance, CVE-2017-18350 was patched by Dash 5 months after the public release of the CVE. Moreover, among the five altcoins we analyzed (some of which are worth several billions) Litecoin is the only one to have consistently ported patches within 1 year of their release.

As a by-product, another important purpose of our work is to motivate the need for a proper responsible disclosure of vulnerabilities to all forked chains prior to any publication of the vulnerability. `GitWatch` offers an effective means for developers to check whether a given patch has been included in relevant forks—before publicly releasing the CVE.

ACKNOWLEDGMENTS

This work was partially funded by the Deutsche Forschungs-gemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and the European Union (INCODE, Grant Agreement No 101093069). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] 2014. Bitcoin patch propagation 1. https://github.com/bitcoin/bitcoin/commit/f59d8f0b644d49324cabd19c58cf2262d49e1392.
[2] 2015. Bitcoin patch propagation 2. https://github.com/bitcoin/bitcoin/commit/5029698186445bf3cd69d0e720f019c472661bff.
[3] 2015. Bitcoin patch propagation 3. https://github.com/bitcoin/bitcoin/commit/50262d89531692473ff557c1061aee22aa4cca1c.
[4] 2018. Monacoin. https://monacoin.org/.
[5] 2019. CVE-2017-18350. https://medium.com/@lukedashjr/cve-2017-18350-disclosure-fe6d695f45d5.
[6] 2020. Litecoin. https://litecoin.com/en/.
[7] 2021. Dogecoin. https://dogecoin.com/.
[8] 2022. Bootstrap. https://github.com/twbs/bootstrap.
[9] 2022. CoinTracker. https://www.cointracker.io/.
[10] 2022. Cryptocurrency Prices, Charts And Market Capitalizations. https://coinmarketcap.com/.
[11] 2022. Dash. https://www.dash.org/.
[12] 2022. Digibyte. https://digibyte.org/en-us/.
[13] 2022. GH Archive. https://www.gharchive.org/.
[14] 2022. Git Documentation. https://git-scm.com/docs.
[15] 2022. Linux Kernel. https://github.com/torvalds/linux.
[16] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. 2017. Hijacking Bitcoin: Routing Attacks on Cryptocurrencies. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 375–392.
[17] John Businge, Openja Moses, Sarah Nadi, and Thorsten Berger. 2021. Reuse and Maintenance Practices among Divergent Forks in Three Software Ecosystems. In *Empirical Software Engineering*.
[18] Bitcoin Core developing team. 2022. Bitcoin Core. https://bitcoincore.org/.
[19] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. 2016. On the Security and Performance of Proof of Work Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on*

Table 3: Summary of the time to patch the vulnerabilities studied in the paper. Here, we use the best (i.e. earliest) result out of the three heuristics. We use a dash (-) when our heuristics could not find the patch in the altcoin, while "NA" refers to the case where the vulnerability was discovered and patched prior to the release of the altcoin. All the values are in number of days

Name	Published Date	Description	Bitcoin	Litecoin	Dash	Dogecoin	Digibyte	Monacoin
Paper [21]	2015-08-14	deterministic random eviction	-143	19	15	92	681	684
Paper [21]	2015-08-14	random selection sha1	-143	19	15	92	681	684
Paper [21]	2015-08-14	random selection sha2	-143	19	15	92	681	684
Paper [21]	2015-08-14	test before evict	935	285	-	392	13	285
Paper [21]	2015-08-14	feeler connections	375	58	327	256	162	165
Paper [21]	2015-08-14	more buckets	-143	19	15	92	681	684
Paper [21]	2015-08-14	more outgoing connections	1482	-	-	-	-	-
Paper [20]	2015-10-16	no inv messages	44	326	69	84	430	433
Paper [20]	2015-10-16	filtering inv by ip address	38	333	75	90	437	440
Paper [20]	2015-10-16	penalizing non-responding nodes	-615	235	291	106	1089	1092
Paper [26]	2012-10-18	forward double spending attempts	617	202	281	362	950	953
Vulnerability	-	limit the number of IPs addrman learns from each DNS seeder	0	103	-	392	13	103
Vulnerability	-	ensure tried table collisions eventually get resolved	0	281	-	-	-	-
GitHub bug	-	fixes fee estimate and peers files only when initialized	0	119	198	279	867	870
GitHub bug	-	check block header when accepting headers from peers	0	56	135	216	804	808
GitHub bug	-	introduce block download timeout	0	8	87	168	756	759
GitHub bug	-	fix de-serialization bug where AddrMan is left corrupted	0	169	426	367	273	276
GitHub bug	-	dont deserialize nVersion into CNode	0	15	194	94	376	167
CVE-2010-5137	2010-07-28	DoS: OP_LSHIFT crash	18	NA	NA	NA	NA	NA
CVE-2010-5141	2010-07-28	Theft: OP_RETURN could be used to spend any output.	3	NA	NA	NA	NA	NA
CVE-2010-5138	2010-07-29	DoS: Unlimited SigOp DoS	40	NA	NA	NA	NA	NA
CVE-2010-5139	2010-08-15	Inflation: Combined output overflow	1	NA	NA	NA	NA	NA
CVE-2010-5140	2010-09-29	DoS: Never confirming transactions	1	NA	NA	NA	NA	NA
CVE-2011-4447	2011-11-11	Exposure: Wallet non-encryption	4	210	NA	NA	NA	NA
CVE-2012-1909	2012-03-07	Netsplit: Transaction overwriting	-4	101	NA	NA	NA	NA
CVE-2012-1910	2012-03-17	Non-thread safe MingW exceptions	-1	88	NA	NA	NA	NA
CVE-2012-2459	2012-05-14	Netsplit: Block hash collision (via merkle root)	-14	43	NA	NA	NA	NA
CVE-2012-3789	2012-06-20	DoS: (Lack of) orphan txn resource limits	-33	25	NA	NA	NA	NA
CVE-2012-4684	2012-08-24	DoS: Network-wide DoS using malleable signatures in alerts	2	263	NA	NA	NA	NA
CVE-2013-2272	2013-01-11	Exposure: Remote discovery of node's wallet addresses	15	110	NA	NA	NA	NA
CVE-2013-2273	2013-01-30	Exposure: Predictable change output	0	106	NA	NA	NA	NA
CVE-2013-3219	2013-03-11	FakeConf: Unenforced block protocol rule	7	60	NA	NA	NA	NA
CVE-2013-3220	2013-03-11	Netsplit: Inconsistent BDB lock limit interactions	7	60	NA	NA	NA	NA
BIP 0034	2013-03-25	FakeConf: block protocol update	-217	269	NA	NA	NA	NA
CVE-2013-4627	2013-06-01	DoS: Memory exhaustion with excess tx message data	62	17	NA	NA	NA	NA
CVE-2013-4165	2013-07-20	Theft: Timing leak in RPC authentication	19	10	NA	NA	NA	NA
CVE-2013-5700	2013-09-04	DoS: Remote p2p crash via bloom filters	-15	0	NA	NA	NA	NA
CVE-2014-0160	2014-04-07	Remote memory leak via payment protocol	1	176	233	0	1030	1034
BIP 66	2015-02-13	FakeConf: Strict DER signatures	-12	4	61	142	731	734
BIP 65	2015-11-12	FakeConf: OP_CHECKLOCKTIMEVERIFY	-143	159	229	147	591	322
CVE-2016-10724	2018-07-02	DoS: Alert memory exhaustion	-836	216	-	414	320	323
CVE-2018-17144	2018-09-17	Inflation: Missing check for duplicate inputs	0	1	1	-	155	1
CVE-2017-18350	2019-06-22	Buffer overflow from SOCKS proxy	-632	151	727	91	140	151
CVE-2018-20586	2019-06-22	Deception: Debug log injection via unauthenticated RPC	-229	41	380	-	106	244
CVE-2014-0224	2014-06-05	OpenSSL CVE	0	118	174	0	972	975
CVE-2018-12356	2018-06-14	Regex bug	1	184	-	291	50	184
CVE-2019-6250	2019-01-13	Vulnerability in the ZeroMQ libzmq library	5	31	-	-	-	31
		Average	7.53	114.85	188.0	185.17	519.55	503.3
		Number of fixes	47/47	41/42	21/28	23/28	25/28	26/28

Computer and Communications Security, Vienna, Austria, October 24-28, 2016, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 3–16. https://doi.org/10.1145/2976749.2978341

[20] Arthur Gervais, Hubert Ritzdorf, Ghassan O. Karame, and Srdjan Capkun. 2015. Tampering with the Delivery of Blocks and Transactions in Bitcoin. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 692–705. https://doi.org/10.1145/2810103.2813655

[21] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. 2015. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 129–144. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman

[22] Ethan Heilman, Neha Narula, Garrett Tanzer, James Lovejoy, Michael Colavita, Madars Virza, and Tadge Dryja. 2020. Cryptanalysis of Curl-P and Other Attacks on the IOTA Cryptocurrency. *IACR Trans. Symmetric Cryptol.* (2020), 367–391.

[23] Qingze Hum, Wei Jin Tan, Shi Ying Tey, Latasha Lenus, Ivan Homoliak, Yun Lin, and Jun Sun. 2020. CoinWatch: A Clone-Based Approach For Detecting Vulnerabilities in Cryptocurrencies. In *IEEE International Conference on Blockchain, Blockchain 2020, Rhodes, Greece, November 2-6, 2020*. IEEE, 17–25. https://doi.org/10.1109/Blockchain50366.2020.00011

[24] Ang Jia, Ming Fan, Xi Xu, Di Cui, Wenying Wei, Zijiang Yang, Kai Ye, and Ting Liu. 2020. From Innovations to Prospects: What Is Hidden Behind Cryptocurrencies?. In *MSR*. ACM, 288–299.

[25] Sanket Kanjalkar, Joseph Kuo, Yunqi Li, and Andrew Miller. 2019. Short Paper: I Can't Believe It's Not Stake! Resource Exhaustion Attacks on PoS. In *Financial Cryptography (Lecture Notes in Computer Science, Vol. 11598)*. Springer, 62–69.

[26] Ghassan Karame, Elli Androulaki, and Srdjan Capkun. 2012. Double-spending fast payments in bitcoin. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM, 906–917. https://doi.org/10.1145/2382196.2382292

[27] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf.