Rethinking Trust in Forge-Based Git Security

Aditya Sirish A Yelgundhalli*, Patrick Zielinski*, Reza Curtmola†, and Justin Cappos*
*New York University
{aditya.sirish, patrick.z, jcappos}@nyu.edu
†New Jersey Institute of Technology
reza.curtmola@njit.edu

***Abstract*—Git is the most popular version control system today, with Git forges such as GitHub, GitLab, and Bitbucket used to add functionality. Significantly, these forges are used to enforce security controls. However, due to the lack of an open protocol for ensuring a repository's integrity, forges cannot prove themselves to be trustworthy, and have to carry the responsibility of being non-verifiable trusted third parties in modern software supply chains.**

In this paper, we present gittuf, a system that decentralizes Git security and enables every user to contribute to collectively enforcing the repository's security. First, gittuf enables distributing of policy declaration and management responsibilities among more parties such that no single user is trusted entirely or unilaterally. Second, gittuf decentralizes the tracking of repository activity, ensuring that a single entity cannot manipulate repository events. Third, gittuf decentralizes policy enforcement by enabling all developers to independently verify the policy, eliminating the single point of trust placed in the forge as the only arbiter for whether a change in the repository is authorized. Thus, gittuf can provide strong security guarantees in the event of a compromise of the centralized forge, the underlying infrastructure, or a subset of privileged developers trusted to set policy. gittuf also implements policy features that can protect against unauthorized changes to branches and tags (*i.e.*, pushes) as well as files/folders (*i.e.*, commits). Our analysis of gittuf shows that its properties and policy features provide protections against previously seen version control system attacks. In addition, our evaluation of gittuf shows it is viable even for large repositories with a high volume of activity such as those of Git and Kubernetes (less than 4% storage overhead and under 0.59s of time to verify each push).

Currently, gittuf is an OpenSSF sandbox project hosted by the Linux Foundation. gittuf is being used in projects hosted by the OpenSSF and the CNCF, and an enterprise pilot at Bloomberg is underway.

I. Introduction

In recent years, there has been a significant surge in software supply chain attacks, with some being carried out by suspected nation-state actors [1], [2]. This trend continues to grow year over year as highlighted in numerous surveys [3]–[5]. While efforts have been made to bolster software supply chain security with increased transparency and verifiability, these solutions have primarily concentrated on other aspects like software builds and distribution [6]–[10], neglecting the crucial but often attacked area of Version Control Systems (VCS) [11]–[17], where source code resides. Of these systems, Git [18] stands out as the most widely used VCS, as indicated by 93.87% of surveyed developers [19].

While Git is a distributed VCS, developers use a *synchronization point* to submit and receive changes, with centralized *forges*, or *social coding platforms*, filling this role and providing additional features such as issue trackers, wikis, continuous integration (CI), etc. Today, forges such as GitHub [20], GitLab [21], or Bitbucket [22] are used to *enforce security controls* that protect the integrity of a repository's contents. As these security controls are not part of the Git protocol, users are reliant on their forge's opinionated implementations of these features.

We identify three key features forges provide that are imperative for ensuring the integrity of a repository. First, forges have mechanisms for *policy declaration* that can enumerate a repository's trusted users and rules about which user can write to which portion of a repository. Forge policy declaration capabilities do not apply the principles of least privilege and separation of duties. Instead, these capabilities are concentrated in a small set of privileged users who can each act unilaterally, exposing the repository to the compromise of a single privileged user [11], [16].

Second, the forge is responsible for *activity tracking* by maintaining an audit log of all actions. Such a log is used to monitor and inspect actions performed in a repository. Third, when a user makes a change, the forge performs *policy enforcement* to ensure the change is valid. Unfortunately, users cannot verify the integrity of the repository's audit log themselves, nor can they validate the forge's policy enforcement and integrity checks, meaning that a compromised forge can subvert the enforcement of repository policies [12]–[16]. At the same time, the forge cannot prove itself trustworthy, and has to carry the responsibility of being a *non-verifiable trusted third party*. This has been highlighted in software supply chain security threat models in academia [23], the industry [24] by the Supply-chain Levels for Software Artifacts (SLSA) [8] project, and in NIST recommendations [25], [26].

In this paper, we introduce **gittuf**, a forge-agnostic Git security system. Like forges, gittuf implements policy semantics that can protect the contents and metadata of a Git repository. Specifically, gittuf can be used to ensure pushes to a repository's branches and tags as well as modifications to its files/folders (especially important in monorepos[1]) are

Network and Distributed System Security (NDSS) Symposium 2025
24–28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241008
www.ndss-symposium.org

[1] In a monorepo setup, a single repository is used to house multiple projects, with each project stored in a different folder. This is used by several enterprises (*e.g.*, Google [27], Uber [28], [29], and X [30], formerly known as Twitter).

performed by authorized users. However, unlike forges, gittuf **empowers every developer to enforce the repository's security**. gittuf achieves this by decentralizing the aforementioned three aspects of repository security: policy declaration, activity tracking, and policy enforcement.

First, gittuf distributes the responsibility of declaring policy among multiple privileged users and ensures consensus. In other words, gittuf policy can be configured to ensure that no user can unilaterally make policy changes. Further, gittuf allows for granular authentication: a developer who is also a repository owner can authenticate separately for each role, minimizing the chances of exposing the higher privileged credentials. Similarly, gittuf makes policy declaration more granular: a privileged user can be allowed to *extend* (and not override) existing policy, and only for specified parts of the repository (branches, tags, files/folders). Implemented using *namespaced delegations*, this feature enables distributing policy declaration responsibilities amongst more users without overprivileging them. In this manner, gittuf allows more users to take an active role in policy declaration, enabling *shared responsibility*. Simultaneously, gittuf limits the impact of compromising any one privileged user trusted to declare repository policy, mitigating the causes of some prior incidents [11], [16].

Second, gittuf decentralizes the tracking of repository activity. To do this, gittuf implements a "reference state log" (RSL) that tracks pushes to Git branches, tags, changes to gittuf policy, and code review approvals. The RSL is an authenticated, ordered, append-only log maintained collectively by all users of the repository. Every time a user pushes to the synchronization point, they add a cryptographically signed entry to this log. As it is designed to be append-only, *gittuf ensures that any attempts to drop, reorder, or otherwise tamper with the entries in the log are detected by other users*. Thus, no single entity is entirely trusted to maintain the log of repository activity. The RSL was originally proposed to address the related class of "metadata manipulation attacks" [31]. gittuf extends its design with improved developer authentication management as well as support for common Git workflows such as force pushes.

Third, gittuf decentralizes the enforcement of the repository's security policies by enabling every developer to independently verify and enforce its security policies. Rather than have the forge be the *sole* arbiter of whether a repository change is authorized, gittuf makes policy enforcement a collective undertaking and ensures that the compromise of any one point is not sufficient to subvert the repository's security [12]–[16]. Note that gittuf enables all collaborating parties to partake in enforcing a repository's security controls, *including the forge* itself as one of the parties. With gittuf, the forge can prove itself trustworthy by adding transparency and verifiability to its existing policy mechanisms.

gittuf has been designed and developed in collaboration with the industry as an open source security project. As such, all significant feature additions to gittuf were authored and reviewed by participants from both academia and industry. gittuf has a growing open source community spanning multiple academic institutions, companies, and independent individual contributors. Seeing potential in gittuf, the Open Source Security Foundation (OpenSSF) [32] Technical Advisory Council (composed of representatives of GitHub, Google, IBM, Intel, Snyk, and Kusari) accepted the project into the OpenSSF sandbox, part of the Linux Foundation. gittuf is being used in projects hosted by the OpenSSF and the Cloud Native Computing Foundation (CNCF) [33]. Finally, gittuf is being piloted for enterprise use at Bloomberg [34].

To summarize our contributions, we present in this paper:

- our analysis of the current forge security model used by Git repositories
- the shortcomings in forge features for policy management, activity tracking, and policy enforcement
- the architecture of gittuf, a novel system that decentralizes policy declaration, activity tracking, and policy enforcement
- our analysis of gittuf's properties and its ability to mitigate prior attacks
- our evaluation of gittuf's performance in terms of storage and runtime overhead introduced

In the rest of this paper, we present a more detailed description of Git as well as the centralized security model operated by forges (Section II). We then describe gittuf's threat model and system goals (Section III), and gittuf's design and implementation (Section IV). This is followed by our security analysis of gittuf including its ability to handle prior incidents(Section V), its performance evaluation and current deployments (Section VI), and a discussion of various gittuf properties and present limitations (Section VII). We close with an analysis of related work (Section VIII) and our conclusions (Section IX).

II. Background

Before describing gittuf, we review Git internals, Git's security features and collaboration model, and features provided by forges, **including their limitations**. We also define general terms used in the paper.

A *developer* is any user performing commits or other actions in a Git repository. A privileged set of *repository owners* serve as the root of trust for gittuf. Client devices, repositories, and other services like CI/CD systems can act as *verifiers*, which means they perform verification over repository actions. In the most common use case, a single repository will serve as the *synchronization point* where developers discover each others' changes. All these parties, whether developers, repository owners, or authorized bots, are identified by their signing keys or identities when they make changes to the repository. Finally, a *forge* is a centralized social coding platform that often serves as a repository's synchronization point.

A. Git

Git implements a content-addressable store [35], henceforth known as the Git object store, for each repository. Every Git object is addressed using its SHA-1 hash, which serves as the object's ID. The store records several types of Git objects that collectively represent the software tracked in the repository.

An essential Git object is a *commit* which is a self-contained representation of the whole repository at some

point in time, and can more generally be referred to as a *revision*. Each commit contains a pointer to a Git *tree* object that represents the root folder of the repository using one or more entries. Each entry points to a file or another tree object to represent subdirectories using their respective SHA-1 identifiers.

Git supports named *references* that serve as pointers to Git objects. Each reference is a file that contains the ID (*i.e.*, SHA-1 hash) of a Git object. The most common type of Git reference is a *branch*, which represents an ongoing line of development. A branch points to a commit that is at the tip of that line of development. When a change is made to the branch, a new commit is created and set as the branch's tip. Another type of Git reference is a *tag*, which simply points to a specific commit object and is meant to serve as a static point in the project's history. Optionally, a tag reference may point to a tag object, which in turn identifies the commit object using its SHA-1 identifier. Thus, a tag object can also serve as an identifier for a repository revision.

Note that while Git repositories typically use branch and tag references, Git supports other *custom references* as well. These references exist outside the typical branch and tag namespaces, and are used by forges like GitHub [36], GitLab, and Bitbucket (for pull or merge requests), Git Notes [37] (to attach additional information to a Git commit), Gerrit [38] (to submit changes for code review), Git-appraise [39] (to store code review results), or Git-bug [40] (to track issues in the repository). *gittuf leverages this Git feature to store additional metadata within the repository*.

Finally, Git implements custom protocols to communicate with remote repositories (*e.g.*, the synchronization point). Git supports modifying these workflows using a feature called a remote helper [41]. gittuf makes use of this feature to synchronize additional metadata.

B. Git Security Features and Collaboration Model

Git provides a basic defense layer using several security features. First, each commit object protects the integrity of the corresponding revision by including the ID of the tree object for that revision, which acts as a Merkle hash tree over the repository data in that revision. Second, each commit object contains IDs of its parent commits. This creates a hash dependency between commits and ensures that the history of commits cannot be altered arbitrarily without being detected. Third, Git provides users with the option to cryptographically sign commit and tag objects, which allows an auditor to unequivocally identify the user who committed code and prevents users from repudiating their commits. It also ensures the integrity of the signed object. Git supports signing with GPG, SSH, and X.509 (which in turn has been leveraged for identity based signing [7], [42]), but does not provide mechanisms to distribute and manage which keys must be trusted for signatures.

While Git is a decentralized VCS, a common model for Git based development workflows is to use a centralized synchronization point where all developers submit their changes to and from which they receive updates. Some popular options for synchronization points are Git forges like GitHub [20], GitLab [21], and Bitbucket [22], that add a variety of collaboration features for repositories such as an issue tracker, wiki, and CI features. In addition to their public instances, these forges, and other open source options [43]–[45], can also be self-hosted by enterprises.

As Git does not provide semantics for enforcing access control rules such as which parties can write to different portions of the repository. As a result, the forge is frequently tasked with applying security policies. We examine the security features provided by forges and their limitations next.

C. Forge Security Features and Limitations

Forge access control policies allow repository owners to declare trusted developers, where each developer is authenticated using their forge credentials. Developers can be grouped into teams, and each team or developer can be assigned a custom set of permissions, enabling for the creation of a hierarchical set of roles with different permissions. However, the policy declaration capabilities offered by forges have shortcomings. First, any developer trusted to make a change to a repository's security policies can do so at will, unilaterally. This means that **compromising a single privileged user can bypass the forge's protections while evading detection**: disable a rule for a brief period of time, make changes to the repository that would violate the disabled rule, and then re-enable the rule. Second, forge authentication systems are not granular: a developer who may be the owner of a repository cannot choose to authenticate with the forge without those privileges for a session. Any exposure of these credentials will lead to the attacker gaining all of the privileges of the user. Third, policy declaration semantics are not sufficiently granular. If a user is trusted to declare policy for a repository, they are trusted for all parts (branches, tags, files/folders) of the repository. Additionally, it is not possible to allow them to *extend* rather than override existing policies. By allowing a user to extend existing policy granularly for a part of the repository, they cannot take actions such as deprivileging other trusted users or removing rules for other parts of the repository. The result is repositories end up with overprivileged users, especially in large repositories such as monorepos, which can be disastrous when we also consider that each user is trusted to act unilaterally. These shortcomings which concentrate policy declaration capabilities have been leveraged in incidents in the past [11], [16].

Further, forges maintain an authenticated activity log that can be used to audit changes in the repository. While this is a valuable feature, the forge is a single point of trust for this information. A malicious or compromised forge can **arbitrarily drop, reorder, or tamper with events in the log**. This is also especially of concern in enterprise contexts that have to contend with insider threats among the administrators of the forge software.

Finally, forges enforce the policy established by a repository's owners and maintainers. When a developer attempts to push to the forge, the forge authenticates the user and verifies they are allowed to push to the namespace in question. Alternatively, if a contribution workflow offered by the forge is used (*e.g.*, GitHub's pull request functionality [36]), the forge enforces policies on that workflow, such as requiring a minimum number of code review approvals. In this model, the forge

is the sole entity that can verify and enforce the policy, and vulnerabilities in the past [13], [46]–[48] have shown that these can be bypassed. Users are unable to perform any verification of their own. In part, this is because forges use non-standard authentication protocols for user management [49], [50] which cannot be independently verified, in contrast to cryptographic signatures. Forges also lack policy transparency [51]: security policies (such as the different teams and their permissions) are managed outside of the repository, and only visible to users with significant privileges. In fact, forges require the same privileges to both unambiguously view and set repository policy. As a result, any attempts at independent verification depends on the forge to provide the correct set of policies applicable to some change, with it being impossible to verify the right set of policies were in fact provided. Thus, while forges help ensure the integrity of a repository, developers must **entirely trust the forge software, the infrastructure used to host it, and its administrators (whether those of a public instance or enterprise employees for a self-hosted instance) to behave correctly**.

III. Threat Model and Security Goals

We consider the standard scenario where a forge is used to manage a Git repository on a centralized synchronization point. This forge can be a publicly hosted solution (*e.g.*, the github.com service), or self-hosted on premises by an enterprise. Either option exposes the forge instance to both external attackers and insider threats. External attackers may circumvent security measures and compromise the version control system, manifesting themselves as advanced persistent threats (APT) and making unnoticed changes to the system. Similarly, insider threats may be posed by rogue employees with escalated privileges who abuse their authority to make unnoticed changes.

To protect the integrity of the repository's contents, the maintainers of the repository define security controls such as which contributors can write to different parts of the repository. gittuf is meant to protect against scenarios where any party, individual developers, bots that make changes, or the forge itself, may be compromised and act in an arbitrarily malicious way as seen in prior incidents [11]–[17]. This includes scenarios such as:

- T1: Modifying configured repository security policies, such as to weaken them
- T2: Tampering with the contents of the repository's activity log, such as by reordering, dropping, or otherwise manipulating log entries
- T3: Subverting the enforcement of security policies, such as by accepting invalid changes instead of rejecting them

Note that we consider out of scope a freeze attack [9], where the forge serves stale data, as development workflows involve a substantial amount of out-of-band communication which prevents such attacks from going unnoticed. We similarly consider weaknesses in cryptographic algorithms as out of scope for this paper. Based on this threat model, we aim to decentralize three key aspects of Git security, deriving the following system goals for gittuf.

Decentralize policy declaration. gittuf must enable distributing the responsibility of setting a repository's policy amongst multiple trusted developers, with the ability to configure the number of developers who must approve to achieve consensus. A privileged developer must not be able to unilaterally make policy changes, offering protections against a malicious insider threat and scenarios where an attacker manages to compromise a single user. Thus, gittuf must enable requiring *threshold approvals for policy changes*, following the principle of separation of duties.

Further, gittuf must allow for authenticating granularly, to minimize the chances of exposing high privilege credentials that must only be used for policy declaration. This applies the principle of least privilege for the credentials used for regular repository operations. Additionally, it must be possible to enable a privileged party to extend the trust they have (or a subset) to a new party by specifying *granular delegations*. In other words, a privileged party must be allowed to declare and extend the existing policy for some portion of a repository without having to trust them to set policy for other portions of the repository.

Decentralize activity tracking. The log of activity in a repository is vital for auditing and verifying actions performed by developers. Audit logs are often used to triage specific incidents, which highlights the importance of the trustworthiness of the log's contents. gittuf must ensure the log is maintained in a *decentralized* manner, so that a single trusted entity cannot arbitrarily tamper with the contents of the sole copy of the log.

Decentralize policy enforcement. gittuf must enable any party that can read the contents of a repository to *independently verify* that policy was followed, without trusting any other party to do so on their behalf. This means that when multiple developers are trusted to contribute to the repository, they can each independently verify policy, detect a violation, and act appropriately to correct it. In such a distributed model with multiple independent verifiers, a single honest verifier is sufficient to *detect* a policy violation.

IV. Design and Implementation

A. gittuf Overview

gittuf allows developers to define policies that restrict who can write to portions of the repository, such as branches, tags, and files/folders. Significantly, gittuf allows a user to be granted policy declaration privileges in a granular manner, limiting what they are allowed to set policy for. gittuf also enables requiring consensus among a set of privileged users for policy changes, and allows for granular authentication to avoid exposing high privilege credentials for tasks that do not need the elevated privileges. As achieving consensus from all privileged users can impose significant delays in development workflows, gittuf allows for configuring the number of privileged users who must approve to achieve consensus.

Further, gittuf decentralizes the tracking of repository activity by implementing a *Reference State Log (RSL)*, an authenticated, append-only, synchronized log of repository actions. gittuf uses the RSL to maintain a global state of the repository in a distributed fashion as a custom Git reference

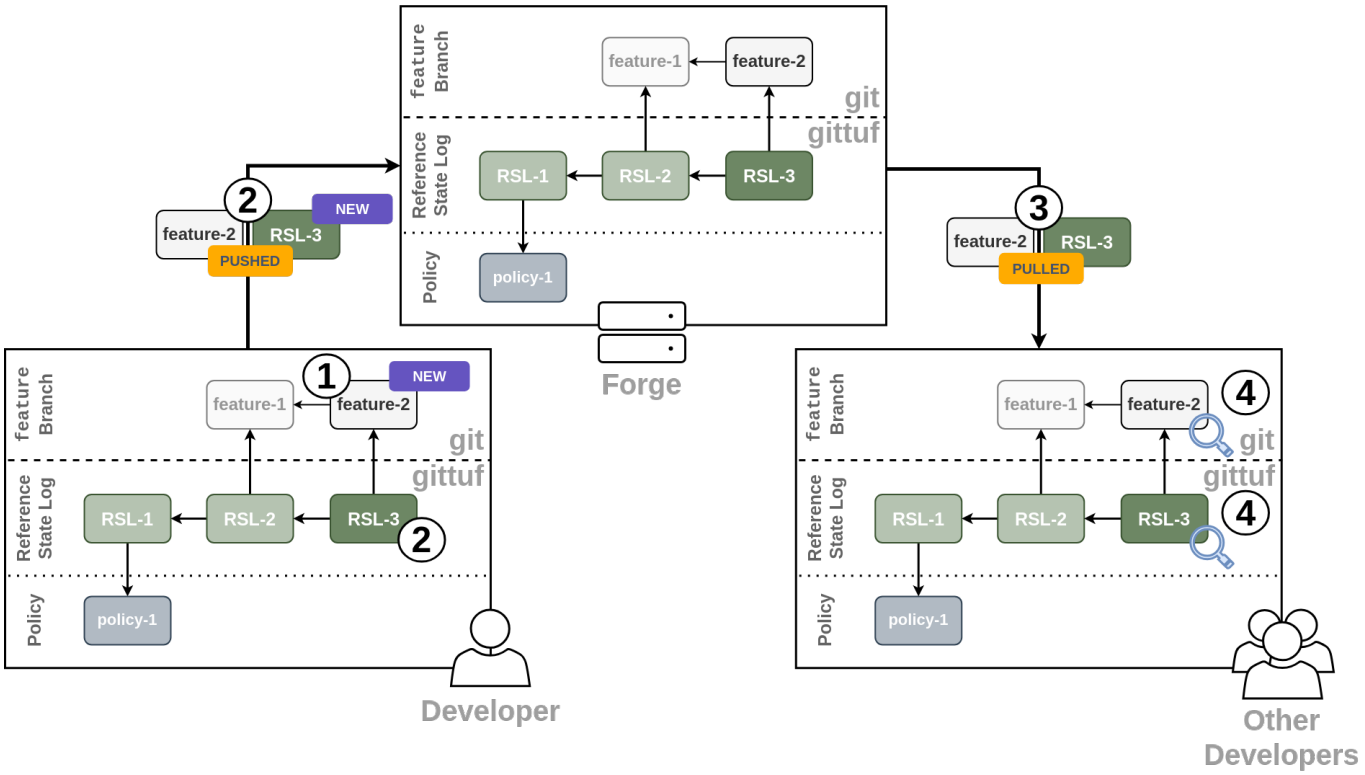

Fig. 1: A developer commits to the `feature` branch (step ①). When the developer pushes the new commit to the forge, she also creates and pushes a corresponding RSL entry (step ②). Other developers fetch the change to the `feature` branch and the RSL entry (step ③). Each developer independently verifies the consistency of the RSL and that the change to the `feature` branch is authorized (step ④).

in the repository. Every gittuf client, including developers, CI/CD bots, etc., adds a signed entry to this log for each push operation to the remote repository; this entry not only records the user's changes in the RSL and their intent to make it available to others, but it also is a validation that the user's view of the repository matches the view recorded by other users in the RSL, preventing metadata manipulation attacks [31]. The RSL also records other repository actions using *authorization attestations* stored as additional metadata. These records can be customized to meet the requirements of the specific action.

To decentralize policy enforcement and thus mitigate the impact of a compromised centralized forge, gittuf enables all entities that support gittuf (*i.e.*, gittuf clients, gittuf-enabled synchronization point, other services that can act as gittuf verifiers) to *independently verify* security policies. Every verifier must have access to the policy [51] and authorization attestations, and thus, as with the RSL, they are also maintained in custom Git references in the repository. As with pushes to regular branches, the RSL also tracks pushes to the policy and authorization attestations Git references: whenever the policy is updated or an authorization attestations is added, the act is recorded with an entry in the RSL.

During verification, gittuf applies the policy for entries in the RSL, with each entry representing a push action in the repository. To independently authenticate the entity making a change, gittuf uses cryptographic signatures. Entities writing to the repository, whether a human developer or a bot, are identified by their signing key (for GPG or SSH keys) or identity (for Gitsign [7], [42]). gittuf includes features to securely distribute, rotate, and revoke the trust of these signing keys and identities.

Figure 1 shows an example of gittuf's use by the developers of a repository. A developer commits changes to the `feature` branch and pushes it to the forge. In the process, the developer creates an entry in the RSL to record the push action, which is also sent to the forge. Other developers fetch the new change as well as the new entry in the RSL. Each developer validates the existing RSL has not been tampered with, and that the only change is the addition of the new entry. Each developer also verifies that the change to the `feature` branch is authorized.

In the rest of this section, we first describe gittuf's policy declaration features, followed by how gittuf decentralizes activity tracking using the RSL and authorization attestations. We close with gittuf's verification and recovery workflows, used by all developers in enforcing gittuf policy, and a brief discussion of our implementation of gittuf.

B. Policy Declaration in gittuf

gittuf implements **access control policies** to protect the integrity of the contents of a Git repository. To implement the access control policy, gittuf adds a metadata layer that is tracked using a custom Git reference in the repository. If a Git branch, tag, or file/folder in the repository has a policy rule associated with it, we refer to it as a *protected namespace*. The metadata consists of a root of trust and one or more rule files.

Root of trust. A gittuf client must have some initial basis for knowing why it can trust the gittuf policy metadata. To address this, gittuf policy stores in a fixed location *root of trust metadata*. The root of trust metadata is cryptographically signed by a *threshold* of repository owners. The threshold is a configurable, numeric value that indicates the minimum number of signatures required for the metadata to be considered valid. The threshold for the root of trust metadata and the repository owners' keys are declared in the root of trust metadata. When a new version of the root of trust metadata is created, it must be signed by a threshold of the root keys declared in the previous root of trust metadata. The threshold feature can be used to ensure no repository owner acts unilaterally. It is important to note that the initial set of root signing keys must be established for the repository either using an out-of-band mechanism or in a trust-on-first-use (TOFU) manner when contacting a repository (likely over HTTPS).

All trusted developers in gittuf policy directly or indirectly gain their trust from the root of trust. This ensures that the owners of a repository can ultimately revoke the trust of any developers that become malicious (and submit malevolent changes to the repository or gittuf policy) as long as a threshold of root keys are not compromised as well.

Rule files. To declare protected namespaces and which developers are trusted to modify them, gittuf employs *rule files*. A rule file is a cryptographically signed piece of metadata that consists of a *name*, a list of *trusted keys or signing identities* representing developers, and a list of *rules*. A gittuf rule is the key semantic used to declare protected repository namespaces (such as branches, tags, or files/folders) and the list of users that are trusted to modify the namespace. As such, each rule has a *name*, a set of *patterns* that identify the namespaces protected by the rule, and the set of *trusted keys or signing identities* authorized by the rule to sign changes for those namespaces. As with the root of trust, the rule can specify a numeric *threshold* that indicates the minimum number of signatures from the rule's trusted keys/identities required to meet the rule. This enables declaring rules that require multiple developers to authorize changes to protected namespaces. Like the threshold for policy changes, this ensures that a compromised developer cannot unilaterally make arbitrary changes to protected repository namespaces.

To enable the distribution of policy declaration responsibilities, gittuf implements semantics to allow some party to *extend* the existing policy for a specific namespace only. To support this, gittuf uses **delegations**, as in other security systems [6], [9], [10]. Any rule can be converted into a delegation by creating a rule file with the same name as the rule. This allows a subset of trust to be *extended* to another party, with this process continuing transitively, as desired. Importantly, the threshold semantic also applies to a delegation, verified by the signatures on the corresponding rule file. As before, the threshold feature ensures that even delegated rule files can be protected from a single malicious or compromised developer.

gittuf policy has a *primary rule file*, for which the signing keys and threshold are declared by the repository owners in the root of trust metadata. Stored in a fixed location, the primary rule file is the starting point for determining the rules that apply to a protected namespace. All delegated rule files are reached directly or indirectly from the primary rule file (and therefore from the root of trust metadata).

gittuf's implementation of the threshold semantic for all policy metadata provides protections against a subset of trusted developers becoming malicious and making ill-intentioned changes to the gittuf policy. With gittuf's use of cryptographic signatures, a developer can also choose to divide different privileges they hold among separate signing keys, distributing trust among different credentials. Additionally, gittuf's support for namespaced delegations allows for extending policy declaration capabilities granularly. Thus, gittuf policy can be used to *safely* distribute policy declaration responsibilities among multiple parties.

To better understand gittuf's policy metadata, consider Listing 1 which contains root of trust metadata and three rule files. The root of trust declares three keys for the subsequent root of trust and three keys for the primary rule file, with a threshold of two for both pieces of metadata. Two of the four rules in the `primary` rule file have been converted into delegations with corresponding rule files that add additional trusted developers and grant them permissions to sign for the protected namespaces. The `protect-ios-app` rule file, signed by Alice, grants Dana and George permission to write to the `ios` folder. While Alice is trusted to grant other developers permission to write to the `ios` folder, she is not trusted to do so for files in the `android` folder. She can only extend the trust (or a subset) she has been granted, and cannot override the rules she inherits or grant herself more trust.

Listing 1: Example of gittuf policy state with its root of trust and three distinct rule files connected using delegations. The root of trust declares the trusted signers for the next version of the root of trust as well as the primary rule file. Signatures are omitted.

```
rootOfTrust:
keys: {R1, R2, R3, P1, P2, P3}
signers:
  rootOfTrust: (2, {R1, R2, R3})
  primary: (2, {P1, P2, P3})

ruleFile: primary
keys: {Alice, Bob, Carol, Helen, Ilda}
rules:
  protect-main-prod: {git:refs/heads/main,
                      git:refs/heads/prod}
    -> (2, {Alice, Bob, Carol})
  protect-ios-app: {file:ios/*}
    -> (1, {Alice})
  protect-android-app: {file:android/*}
    -> (1, {Bob})
  protect-core-libraries: {file:src/*}
    -> (2, {Carol, Helen, Ilda})

ruleFile: protect-ios-app
keys: {Dana, George}
rules:
  authorize-ios-team: {file:ios/*}
    -> (1, {Dana, George})

ruleFile: protect-android-app
keys: {Eric, Frank}
rules:
  authorize-android-team: {file:android/*}
    -> (1, {Eric, Frank})
```

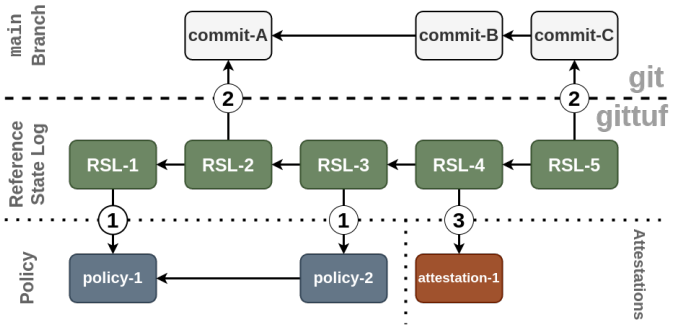

Fig. 2: An illustration of gittuf's Reference State Log. Entries may describe gittuf policy (①), Git commits (②), or gittuf Attestations (③). During verification of an RSL reference entry, the contemporaneous policy and set of attestations are used. Commits, RSL entries, and policy entries may contain hashes of other items, indicated by the arrows.

C. Activity Tracking in gittuf

To record changes in a repository, gittuf leverages a *Reference State Log (RSL)* [31]. The RSL is an append-only, linear hash chain that serves as a log of activity in the repository. Thus, the RSL indicates the order in which repository actions occur. It is maintained collectively by gittuf clients in a custom Git reference in the repository. For any activity that modifies the state of the repository, an entry is added to the RSL via a new commit to the RSL's Git reference. For some actions, the entry points to metadata, known as an authorization attestation, that captures additional context.

When a developer fetches updates from the synchronization point (including new entries in the remote RSL), they use the RSL to ensure that their perceived state of the repository is consistent with that of the other users. Effectively, their gittuf client ensures that the RSL only grows with the addition of new entries, with no changes made to historic entries or their order.

Figure 2 shows a repository in which the RSL (in green) starts with the policy described in `policy-1`. A commit is added to the `main` branch (`commit-A`). After this, the policy is changed to `policy-2`, and two new commits are submitted together (`commit-B` and `commit-C`) to the `main` branch. The actions making these changes are all recorded in the RSL.

gittuf's RSL design is inspired by the RSL introduced by Torres-Arias et al. [31], but differs in some important aspects. First, gittuf's RSL does not natively support distributing trusted keys. Instead, gittuf's RSL leaves that responsibility to the policy metadata introduced in Section IV-B. Second, gittuf's RSL supports the ability to revoke a prior entry. This allows for marking an entry in the RSL as invalid without losing the RSL's append-only property. We discuss this in greater detail further in this section.

Regular pushes. When a gittuf user performs a push to a remote repository, the user creates an *RSL reference entry* that binds the reference being pushed to a particular revision. Specifically, the entry contains the reference name indicating the updated branch or tag, and a commit or tag ID. The RSL reference entry is cryptographically signed by the user who created it, and is then appended to the RSL via a new Git commit. While an RSL reference entry directly records changes to branches and tags, changes to files/folders are recorded indirectly. This is because in Git, the state of the repository's files/folders depends on the current revision's Git tree. For example, the latest RSL entry for the `main` branch specifies the commit at the tip of the branch, and this commit's Git tree represents the state of the files/folders for the `main` branch at that commit.

Force pushes and push revocations. As the RSL is append-only, a reference entry cannot be amended or removed. However, it is sometimes necessary to add new information to an existing entry or to indicate an entry must not be considered valid any longer (*e.g.*, a push to a branch is overwritten with a force push, or a push violates gittuf policy). To support such cases while also preserving the RSL's append-only property, gittuf extends the RSL's design to add the *RSL annotation entry*. An RSL annotation entry lists one or more prior RSL reference entries it applies to, a message, and a boolean parameter indicating whether the reference entries listed must be revoked. Like the RSL reference entry, each RSL annotation entry is also cryptographically signed.

Policy changes. In the distributed policy verification and enforcement model, it is necessary to unambiguously identify the policy applicable to a particular change in the Git repository. gittuf tracks all versions of policy as it evolves in the repository's RSL. Each version of the policy is known as a **policy state**. A policy state, like the example in Listing 1, contains the full set of gittuf policy metadata, such as the root of trust metadata and all rule files applicable at that point. When any of the policy metadata, such as rule files or the root of trust, is changed, a new policy state with the result of these changes is written to the repository and recorded in the RSL. For example, the repository in Figure 2 has two policy states. The `RSL-1` entry records the original policy state and the `RSL-3` entry records an updated policy state.

Importantly, gittuf's tracking of all policy versions allows an auditor to verify changes in the repository at some historical point in time by rewinding to a prior policy state that was governing the repository. In addition, gittuf's policy states allows for determining the validity of a trusted key/identity or rule. When a key is revoked and a new policy state is created, the key is still trusted for signatures created before the key's revocation using the policy states applicable at the time. Thus, revoking a key does not invalidate all signatures ever issued by it [52].

Other repository activity. gittuf uses **authorization attestations**, stored as additional cryptographically signed metadata, to record other types of repository activity. Like the policy metadata, gittuf stores authorization attestations in a custom Git reference that is tracked by the RSL. When an attestation is added, this event is recorded as a new RSL entry. Such RSL entries are used to record the authorization attestations available at that point in time, and are used when verifying subsequent RSL entries. We describe one type of authorization attestation here, but note that other types can be defined to support recording more repository actions.

In some repositories, a rule may require multiple develop-

ers to authorize a change, such as two-party code review for merges into the default branch. To enforce such a policy, gittuf uses an authorization attestation to record a developer's approval of a forthcoming change, known as an approval attestation. In the example of Listing 1, the `protect-main-prod` rule requires two developers to sign off on a change to the `main` branch. To meet this requirement, when a change must be made to the `main` branch, one of the trusted developers—Alice, Bob, or Carol—must first sign an approval attestation approving the change to the branch. Then, one of the two who did not sign the approval attestation may update the state of the `main` branch and record an RSL reference entry for the change. gittuf does not define the specific method by which a developer is prompted to approve a change; this is typically determined by the project's development workflow, and may be via a GitHub Pull Requests [36] or a GitLab Merge Requests [53]. Additionally, an approval attestation can include more than one signature, meaning the mechanism can be used to meet gittuf rules that require thresholds greater than two.

An approval attestation contains three pieces of information to identify the change being approved. First, it contains the name of the *Git reference* that will be updated. In the example above, the change is to the `main` branch, and this will be recorded in the accompanying approval attestation. Second, the approval attestation contains the *from* Git revision, which is the state of the Git reference prior to the change. This ensures that an old approval cannot be used to perform a rewind attack where the reference is reverted to an old state from a more recent state. Finally, the approval attestation contains the hash of the *target Git tree*, identifying the state of the Git reference after the change is made[2]. This target Git tree must match the Git tree of the revision recorded in the RSL reference entry once the change is approved and made.

In Figure 2, a developer approving updating `main` from `commit-A` to `commit-C` creates an approval attestation in `attestation-1`, which records `main` as the reference name, `commit-A` as the *from* Git revision, and the expected Git tree, matching the tree object of `commit-C`, as the *target tree*. The actual change to the `main` branch is recorded in the RSL using the `RSL-5` entry, which comes after `RSL-4` entry corresponding to `attestation-1`.

D. Policy Enforcement in gittuf

In this section, we describe how gittuf enforces policies. First, we present how gittuf can be used to *independently* verify repository activity against policy. Then, we describe how gittuf recovers from policy violations detected during the verification process.

1) Verifying Policy Compliance: Every gittuf user performs the same verification steps on each protected namespace they are tracking whenever communicating with the synchronization point. This means that each party tracking a protected namespace independently verifies that all RSL entries and policy changes are valid for all times in the repository lifecycle.

[2]We use the ID of the tree object because it can be computed ahead of time using `git merge-tree` [54], as opposed to using the ID of the commit/tag object which, although deterministic, cannot be determined ahead of time (as it includes information such as creation time).

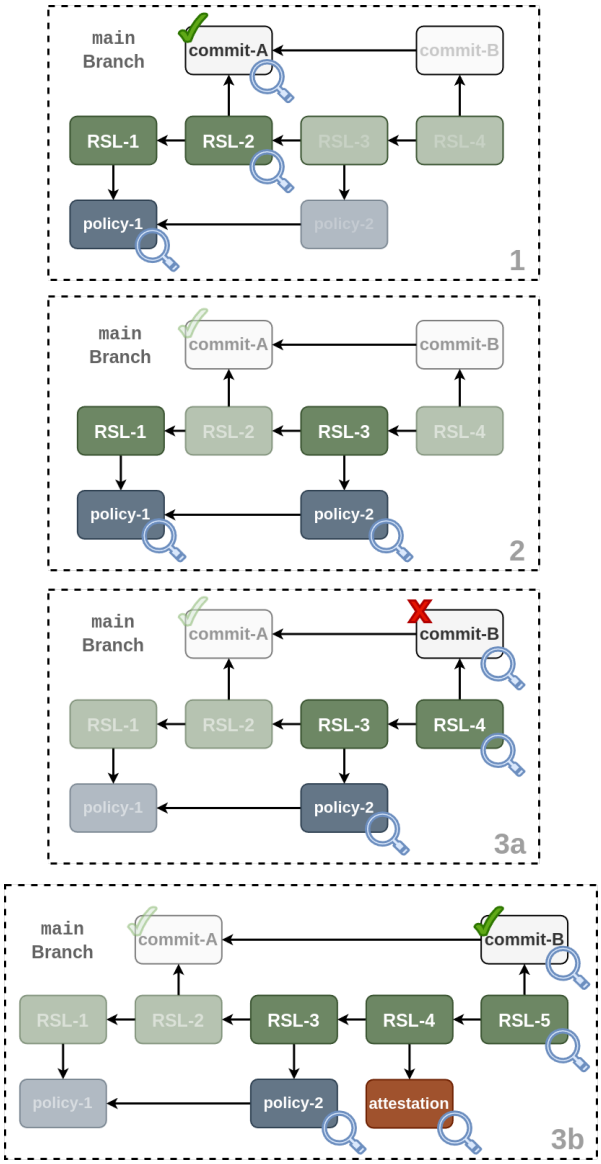

Fig. 3: An overview of gittuf verification. In box 1, gittuf uses `policy-1` to verify `RSL-2` and `commit-A`. Next, the change in policy to `policy-2` is verified using the existing policy `policy-1` in box 2. Box 3a shows the push of `commit-B`, recorded as `RSL-4`, violating `policy-2`, which now requires additional approval. A successful push of the same commit is shown in box 3b, where the push (`RSL-5`) occurs after an approval attestation is recorded (`RSL-4`).

The gittuf verification workflow is triggered when a gittuf user either fetches new changes from or submits changes to the synchronization point. The input to the verification workflow is an RSL reference entry, and the workflow identifies the specific policy state applicable to the entry by finding the latest entry for the policy Git reference in the RSL. Similarly, the workflow also identifies the set of authorization attestations available to verify the entry.

Figure 3 illustrates gittuf's verification workflow. In box 1, gittuf verifies `RSL-2` and `commit-A` using `policy-1`, as that is the applicable policy. Next, in box 2, gittuf moves on to

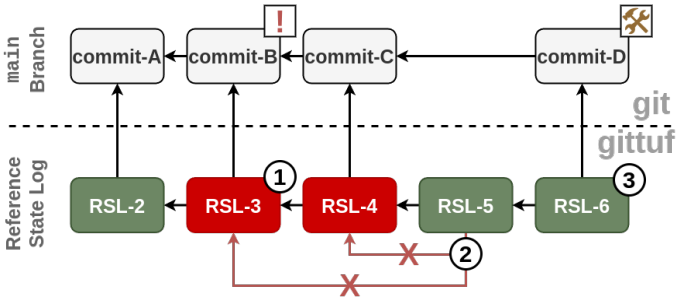

Fig. 4: An illustration of gittuf's recovery workflow. The recovery workflow uses an RSL annotation entry (②) to revoke invalid entries (①), following which a fix RSL reference entry (③) is applied that restores the affected Git reference to the last good state.

the next RSL entry, which introduces `policy-2`. gittuf uses `policy-1` to verify the root of trust metadata in `policy-2`. The new policy adds a rule requiring an additional approval for changes. Thus, in box 3a, gittuf verification detects a violation with the push introducing `commit-B`, `RSL-4`. While the violation must be fixed using gittuf's recovery workflow, we demonstrate the correct way to introduce that commit in box 3b. Here, `commit-B` is pushed (now represented using `RSL-5`) after it is approved using an approval attestation, recorded using `RSL-4`.

To verify file protection rules, gittuf identifies the files modified by a commit. For example, in Figure 3, `commit-B` is inspected against `commit-A` to find modified files. If the commit is the first commit in the history, `commit-A`, all of the files in that commit's Git tree[3].

2) Recovering from Violations: As noted in Section III, gittuf aims to decentralize the enforcement of security policies. When an honest gittuf client (with push access to the synchronization point) detects a policy violation (*i.e.*, the verification workflow fails for some RSL entry because it is signed by an unauthorized developer or it lacks sufficient approvals), it must be able to issue a correction to fix the violation. gittuf defines a recovery procedure that fixes the violation and indicates to other gittuf clients that a fix was issued.

The aim of the recovery procedure is to return the affected reference to its last valid state. For this, gittuf identifies the last RSL reference entry for the affected reference that passes policy verification and determines the Git tree for its commit. This tree is the recovery workflow's desired "fixed state" for the affected reference.

In addition to the violating RSL reference entry, other RSL reference entries that have been added after the violating entry are also considered to be violations so long as they are for the same reference. This is because, even if these changes to a reference are individually valid, they build on changes that violate policy and thus could contain maliciously misleading changes. After the recovery procedure is complete, authorized developers for the affected reference can individually reapply changes to the reference using standard Git tooling. Also note that a gittuf client that adds a "valid" RSL reference entry without addressing the policy violation is likely buggy or malicious, and must be investigated further.

To *revoke* all identified invalid entries, gittuf creates an RSL annotation entry, described in Section IV-C, that lists the invalid entries. In addition, to restore the reference to a good state, gittuf creates a *fix* RSL reference entry for the Git reference. This entry comes *after* the violating entry and the revoking annotation entries, and has a Git revision with the same Git tree identified earlier as being the last valid state. This is demonstrated in the repository in Figure 4. The RSL annotation entry `RSL-5` revokes the original invalid entry `RSL-3` and the subsequent entry `RSL-4`, even though `RSL-4` did not violate policy itself. After the annotation entry, a fix reference entry `RSL-6` is added that sets the `main` branch to a valid state.

The recovery workflow purposely applies a new commit (`commit-D`) as the fix rather than reusing the commit from the last valid entry (`commit-A`). This "revert" based mechanism, as opposed to a "reset" mechanism where the same revision as the last valid state is used, ensures that clients have an update to fetch in case of a recovery. This mitigates bugs that could lead to a client on an invalid commit incorrectly assuming it is ahead of the forge.

E. Implementation

Our implementation of gittuf has been developed in partnership with industry and open source stakeholders to ensure that it is optimized for common user workflows with sane defaults. This reduces the chances of an inexperienced user misconfiguring gittuf policy and gaining a false sense of security. The implementation consists of the features described in this section thus far: the RSL, policy metadata, and authorization attestations, with each stored in a custom Git reference under the `refs/gittuf/` namespace. All of the additional metadata is stored using the Git object store. By leveraging these Git semantics, the implementation of gittuf is compatible with tools in the broader ecosystem such as popular forges.

The policy includes support for multiple signing mechanisms, including GPG and SSH keys, and Sigstore Gitsign [7], [42] identities. These can be extended to support additional signing mechanisms that Git may add. Further, authorization attestations are implemented using the in-toto attestation framework [55]. in-toto attestations are widely used in the realm of software supply chain security [7], [8], [56]–[58] as an authenticated, structured mechanism to record supply chain actions and outcomes. Additionally, the implementation includes the verification and recovery workflows.

Developers invoke the implementation in two ways. First, the `gittuf` command provides gittuf-specific functions to manage the policy metadata. The repository owners and developers use the command to sign the root of trust, create rule files, add rules, etc. Second, gittuf implements a Git remote helper [41], described in Section II-A, which performs additional gittuf actions invisibly during interactions with the synchronization point. After a one-time setup and configuration on a developer's local copy of a repository, the remote helper synchronizes the additional gittuf metadata during the pushes and pulls invoked by the developer, and validates the

[3]The files in the initial commit's Git tree are enumerated recursively to include all files, even those in all subfolders.

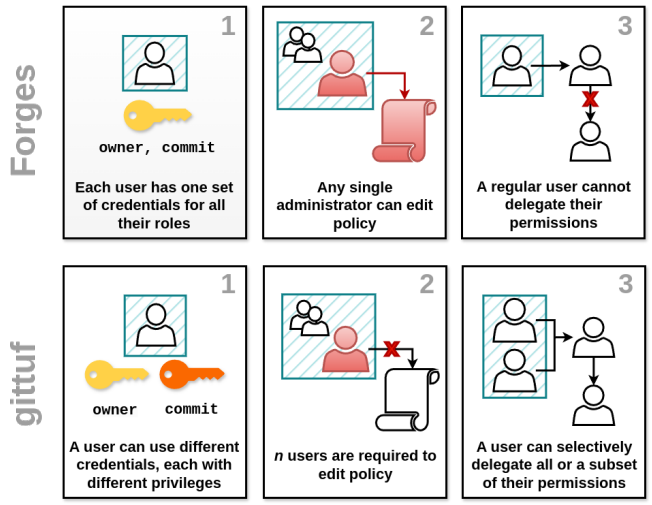

Fig. 5: Comparison of a forge's centralized policy declaration with gittuf's decentralized policy declaration.

consistency of the RSL. The remote helper also automatically creates one or more RSL reference entries for each reference pushed by the developer to the synchronization point.

V. Security Analysis

Our security analysis of gittuf consists of two parts. We first evaluate and discuss gittuf's ability to meet the security goals defined in Section III. We then analyze gittuf's efficacy in protecting against previously seen attacks.

A. Achieving System Goals and Mitigating Threats

gittuf's policy semantics protect the integrity of a repository's branches, tags, and files/folders. We analyze here how gittuf meets its system goals, mitigating the threats in our threat model.

Decentralize policy declaration. gittuf achieves this goal with three properties. First, gittuf's support for thresholds for changes to policy metadata ensures that no single trusted party can unilaterally update the repository's policy. Instead, the trust required for such a policy change is distributed among multiple parties, a threshold of whom must approve. In contrast, a single privileged user can unilaterally modify the policies enabled on a forge. Significantly, gittuf provides the means for requiring approvals for policy changes: the repository's privileged users can determine the approval threshold for individual rule files, allowing them to pick thresholds that balance security and practicality.

Second, gittuf allows for separation of credentials by role. For example, a developer who is also a repository's owner can define and use separate signing keys for each role. This separation of privileges ensures minimizes the chances of exposing the high privilege key during everyday operations.

Third, gittuf's support for namespaced granular delegations enables more users to extend a repository's policy while limiting what portions of the repository they are trusted for. This decentralizes the ability to set policy, rather than concentrating it in a small set of privileged users. Using this feature, it is possible to limit the use (and therefore exposure) of credentials with significant privileges (such as that of the repository owners). In enterprise contexts, where developers frequently change teams and responsibilities, delegations allow for policy updates to happen locally (*e.g.*, adding a developer to a particular package in a monorepo by its immediate manager) but safely (*e.g.*, the manager of said package in the monorepo is not trusted for other packages).

We illustrate the differences between gittuf's policy declaration and that of forges in Figure 5. By meeting this goal, gittuf provides protections against threat T1. A threshold of privileged developers, due to malicious insiders or key compromises, may still update gittuf policy to reduce its protections. In such a scenario, gittuf ensures that the change in policy is visible to all users, and allows for recovery after detection, as long as a threshold of root of trust keys are not also compromised.

Decentralize activity tracking. The RSL implemented by gittuf tracks all activity in the repository in an ordered, authenticated, and *decentralized* manner. All developers working on the repository maintain a copy of the log, and thus ensure the log is not tampered with. Any attempt to reorder, drop, or manipulate the log's contents is detected the next time a gittuf client fetches the RSL, thus providing protections against threat T2. Consider a scenario where a developer pushes to the forge with an accompanying entry in the RSL. The forge may carry out a fork attack to drop this push from the RSL. Other developers would continue adding entries to the RSL, unaware of the dropped entry, but the original developer whose RSL entry was dropped would detect the divergence in their RSL and everyone else's. The forge may also try to maintain two divergent RSLs, but for this to go undetected, the forge must also control developer signing keys. This is because the forge would have to recreate the entries from one group of developers, *signed by their keys*, to present to another group of developers to evade detection. Thus, the RSL being tracked in the repository in a distributed manner mitigates attacks where a malicious forge presents divergent states to different verifiers [59], [60].

This protection extends to all activity tracked by the RSL, and therefore applies to the repository's authorization attestations. In contrast, a malicious forge without gittuf may falsely claim a developer approved a change, telling each developer there is an approval from someone other than themselves. This type of behavior is mitigated when using gittuf's RSL.

Note that a malicious forge serving as the synchronization point could still misbehave in a Byzantine way by executing freeze attacks [9] which replay stale state. However, the common workflow for Git repositories involves a substantial amount of both in-band (*i.e.*, via the Git repository) and out-of-band (*e.g.*, email, instant messaging, etc.) developer communication. Hence, a freeze attack will be detected almost immediately.

Decentralize policy enforcement. Every gittuf-enabled party that receives the RSL and policy information performs verification for actions pertaining to the references it tracks. Thus, any party can independently perform gittuf verification for the references they are aware of using the workflow described in

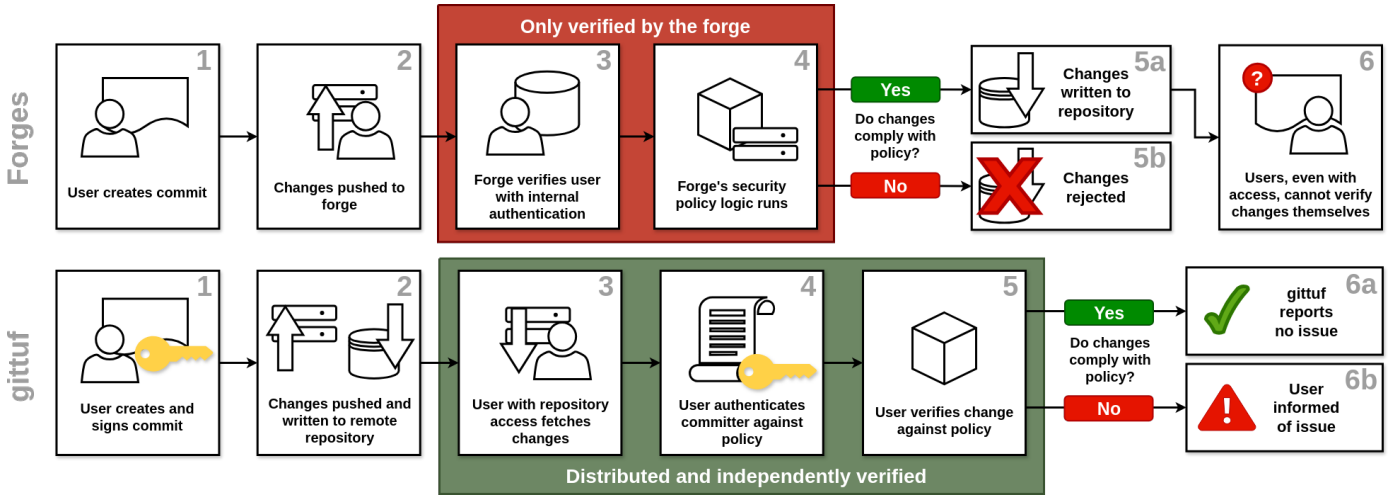

Fig. 6: Comparison of a forge's centralized policy enforcement model with gittuf's decentralized policy enforcement model.

Section IV-D1.

In turn, this decentralizes policy enforcement in the repository. As any and all developers can use execute the verification workflow for a Git reference, they can also enforce the policy by issuing a fix using the recovery workflow described in Section IV-D2 when they detect a violation. Thus, by enabling the independent verification of security policies, gittuf makes it possible to decentralize the enforcement of these policies, in turn mitigating threat T3.

We contrast gittuf's distributed policy enforcement with a forge's centralized policy enforcement in Figure 6. Significantly, a single honest gittuf verifier with push access is enough to detect and correct policy violations. By empowering all developers to verify policies, gittuf improves the chance that there is at least one honest verifier. Even if there is an overwhelming number of malicious verifiers, a single honest verifier is sufficient to detect the violation. As before, if a threshold of trusted users update gittuf policy to reduce its protections (*e.g.*, as a result of malicious insiders or key compromises), this is not considered a policy violation. Here, recovery is still possible after detection as long as a threshold of gittuf's root of trust keys are not also compromised.

B. Prior Incidents

To see how gittuf performs in real-world incidents, we selected attacks from the CNCF [33] catalog of software supply chain attacks [61] that were representative of SLSA [8] source threats [24]. The attacks involved the unauthorized modification of source code, typically by bypassing a synchronization point's security controls or authentication. After filtering out code exfiltration attacks, which fall beyond our threat model, we identified the seven relevant attacks [11]–[17]. We evaluate whether gittuf's features can protect against similar attacks. Our analysis, described here, shows that gittuf can *protect against all of these attacks*.

Unknown Juniper Code [17], [62]. The networking company Juniper disclosed in a security advisory that it found unauthorized code in its firewall operating system. From the advisory, it is clear that an unknown attacker managed to submit unauthorized changes to the source code by accessing the server where it was hosted. Thus, the attackers were able to bypass the security controls of the synchronization point. With gittuf, it would not be sufficient to bypass the server's authentication and security enforcement. Instead, they would have had to compromise all developers performing policy enforcement. In addition, the attackers would have to record their change in an authenticated manner in gittuf's RSL, making it more difficult still to evade detection.

FSF Website Defacement [14]. The Free Software Foundation had its website defaced by an unknown attacker in 2010. Through a SQL injection attack, the attacker was able to gain access to the CVS repository containing the source code of the website. The website used a CI/CD workflow to rebuild the website when its source changed and deploy it. Such CI/CD workflows are common today, but these automated workflows implicitly trust the contents of the source repository as they have no way to independently perform policy verification. However, such an attack performed on a Git repository using gittuf would ensure the unauthorized changes would not be deployed, as the automated workflows would independently verify the changes and stop the deployment.

Malicious PHP Commits [15]. The PHP project had two malicious commits appear in its source repository, claiming to be authored by two of its maintainers. The attacker compromised the repository platform used to host the repository, thus bypassing its authentication and security enforcement. The commits were only flagged because other contributors did not understand the changes made, and could have gone undetected in a repository with fewer reviewers. With gittuf in place, the attacker would have had to compromise either the developers in question or compromise all developers performing verification of policies. Compromising the centralized synchronization point would not suffice.

Kernel Backdoor Attempt [12]. Before Git, the Linux Kernel was maintained in a BitKeeper repository, with a mirror set up on CVS for those who did not wish to use BitKeeper. An attacker pushed malicious code directly to the CVS mirror, bypassing the synchronization point's authentication and

security controls. The attack was luckily discovered because a contributor noticed a change in the mirror that was not in the primary copy of the repository. While this attack predates the Linux Kernel's use of Git for source control, this attack could have occurred had the underlying VCS been Git for the compromised repository. gittuf's decentralized policy enforcement would have detected such an attack, as the attacker only managed to compromise the centralized copy of the repository.

Additionally, for Git repositories where automated mirroring (as was seen in this incident) is necessary, the secondary repository mirroring the first one can independently verify the first's policies, only updating the mirror when verification passes. Further, the secondary repository can have its own policies that ensure pushes to the mirror were from the authorized bot.

Top.gg GitHub Compromise [11]. Top.gg is a bot discovery site for Discord, the popular instant messaging platform. Unidentified attackers compromised the GitHub account of a privileged contributor of one of the repositories, suspected to be via stolen cookies. The attackers then used GitHub's web interface to add a malicious commit to the repository. This attack highlights the benefit of separating the authentication to the forge from the authentication of the author of a commit. The attack vector used to authenticate with GitHub would not work with gittuf. Significantly, the developer's SSH keys do not seem to have been compromised as the attackers opted to create the malicious commit via the web interface.

Gentoo GitHub Compromise [16]. An attacker compromised the GitHub organization for the Gentoo project and removed every developer's access. In addition, the attacker pushed malicious changes to repositories. The developers believe that an attacker compromised an administrator's GitHub account via their password, leading to compromising the repository platform due to administrative privileges, and highlighting the benefit of separating credentials based on privileges. More generally, gittuf can be configured so that the compromise of a single administrator can be contained, by setting higher thresholds for significant metadata such as the root of trust and important rule files. Such multi-authorization requirements cannot be set for forge policy and security controls today.

Ruby on Rails on GitHub [13]. A security researcher was able to leverage a GitHub vulnerability to push an unauthorized commit to the Ruby on Rails repository. This was a compromise of the repository platform rather than that of the maintainers of the Ruby on Rails repository. Vulnerabilities that allow for bypassing forge authentication and policy enforcement capabilities are not uncommon [46]–[48], with some of these cases potentially allowing an attacker to hide their traces by manipulating the audit log. gittuf's decentralization of activity tracking and policy enforcement in particular provide protections against such attacks.

VI. Performance Evaluation and Deployments

In this section, we describe our findings from evaluating the storage and runtime overhead of our implementation of gittuf described in Section IV-E. We also briefly describe ongoing real-world deployments of gittuf.

Repo	Git-only	gittuf-enabled	Increase
Packed			
Git	273.04MiB	282.91MiB	3.62%
K8s	1171.03MiB	1188.66MiB	1.51%
Unpacked			
Git	2844.45MiB	2864.22MiB	0.70%
K8s	3992.27MiB	4065.93MiB	1.85%

TABLE I: Storage overhead of gittuf in the Git and Kubernetes (shown as K8s) repositories. The packed size indicates the size transmitted over the network.

# Pushes	Git	Kubernetes	
		Branch only	Branch + File
10	0.693s	0.952s	2.864s
20	0.698s	1.191s	11.769s
50	0.708s	1.861s	26.675s
100	0.711s	2.668s	47.148s
500	1.078s	11.030s	183.332s
1000	1.491s	22.110s	412.732s

TABLE II: The runtime overhead of performing gittuf verification in the Git and Kubernetes repositories. For the Git repository, only reference protection rules are employed, while for the Kubernetes repository, we measured the overhead without and with file protection rules.

Choice of repositories and experimental setup. We selected two Git repositories for our evaluation: the repository for Git [63] itself and the repository for Kubernetes [64]. Both of these repositories have a large volume of activity, and thus serve as candidates to confirm gittuf can scale to large repository setups. We also specifically chose the Kubernetes repository because it is developed as a monorepo. In addition to the core Kubernetes source code, 30 other projects related to Kubernetes are developed in the same repository [65].

In the repository for Git, we added two Git reference rules, to protect the default `master` branch and tagged releases between v1.8.0 to v2.43.0. We created RSL reference entries for pushes to / merges into the respective references. In the Kubernetes repository, we added a Git reference protection rule for the default `master` branch and 30 file/folder rules for the separate projects, with each rule applying to all of the files for the corresponding project. Collectively, the rules apply to just under 12,000 files at the most recent commit used for this evaluation, and contain over 2,400 keys (representing contributors). As with the repository for Git, we created RSL reference entries for merges into the `master`. We conducted our experiments on a machine with an AMD Ryzen 7640U CPU and 32 GB of RAM running Ubuntu 24.04, and Git v2.43.0.

Storage overhead. We measured the storage overhead added by gittuf and present the results in Table I. We find that the storage increase ranges from **0.70% to 3.62%**. The storage used depends primarily on the repository's configuration, *i.e.*, whether the repository is *packed* [35] or not, and how well the other contents in the repository can be compressed by Git when packed. With gittuf introducing under 4% storage overhead across all configurations, we conclude that the storage overhead imposed by the additional metadata recorded by gittuf is practical for real-world repositories.

Runtime overhead. In addition to the storage overhead, we measured the runtime overhead imposed by gittuf verification. We measured this overhead based on the number of *pushes* that need to be fetched by a verifier. Note that each push can encompass multiple commits. In the Git case, we find that verifying branch protection adds an average of **0.0015s** to **0.0693s** per push action, as shown in Table II. When verifying only branch protection rules in the Kubernetes repository, we find that it takes an average of **0.022s** to **0.095s** per push action. When verifying both branch and file protection rules, this number rises to an average of **0.29s** to **0.59s** per push action, indicating that verifying file policies is relatively more expensive. The Kubernetes runtime overhead is shown in Table II. Given that these verification workflows are performed after changes are fetched from the synchronization point, which includes expensive network communication, we find the runtime overhead of gittuf to be practical.

Community and deployments. gittuf is released under the open source Apache 2.0 license. Part of the OpenSSF sandbox, gittuf has a community composed of members from two academic institutions and a number of enterprises contributing to it. The majority of features were either added by or reviewed and approved by industry participants in conjunction with academic participants. To date, gittuf has had five releases.

Today, implementation is being used in Git repositories of CNCF and OpenSSF projects, including that of gittuf itself. gittuf is also being piloted at Bloomberg. The company uses an on-premise deployment of a popular forge, with gittuf being piloted to provide verifiability for the features the forge is trusted for: policy declaration, activity tracking, and policy enforcement.

VII. Discussion and Limitations

In this section, we discuss other security guarantees provided by gittuf and its adoptability. We also present gittuf's limitations and a discussion of how we plan to address them in future work.

Secure distribution and management of a repository's trusted keys and signing identities. gittuf policy can be used to distribute the trusted keys and signing identities for the repository. Further, gittuf policy states can be used to rotate and revoke keys. If a developer leaves or experiences a key compromise, gittuf can be used to issue a new policy state without that developer's key or identity. Notice that gittuf allows for successfully verifying signatures issued using the old key up until the rotation or revocation event is recorded via a new policy state in the RSL. The updated policy state is used to verify signatures issued after this event. In contrast, existing key management solutions flag as untrusted the signatures issued before a rotation or revocation event [52].

Backwards compatible. gittuf can be initialized with existing Git repositories, with the caveat that gittuf's protections only apply from that point onwards. Changes in the repository prior to gittuf's initialization are assumed to be verified. Further, as gittuf uses Git semantics like the object store, references, and support for cryptographic signing of commits and tags, it can be used in conjunction with repositories hosted on popular forges like GitHub, GitLab, and Bitbucket with no impact to gittuf's security properties.

Extensible. gittuf's design is extensible in a number of ways. Fundamentally, gittuf stores additional metadata in dedicated portions of the repository, and this can be extended to support additional security features. For example, gittuf's support for authorization attestations can be used for validating a particular change underwent the right tests before it was merged into a protected branch. Similarly, gittuf's design can be used to support Git-only users contributing to a repository. Git-only users do not create RSL reference entries, which are required for ordering and security in gittuf. Nonetheless, whenever a gittuf user who is working on the same Git references pushes to the repository, they could create RSL reference entries on behalf of the Git-only users for those changes while capturing additional authentication details as an authorization attestation. We leave the detailed design of this feature and the analysis of its security guarantees to future work. Finally, gittuf's policy semantics can be extended to support other attributes beyond the ability to write to a namespace and define policies for it.

Limitations and future work. gittuf still has several limitations that we plan to address in future work. Specifically, we plan to add support for cross-repository policies, where multiple Git repositories can share a gittuf root of trust and primary rule file. This feature simplifies the management of gittuf root of trust, and is necessary for the ongoing enterprise pilot of gittuf. We defer detailed discussion of this feature and other lessons from the enterprise pilot of gittuf to future work. In addition, we aim to extend namespaced access control rules with the ability to also set constraints on individual users, simplifying policy declarations.

Further, we want to leverage prior work such as le-git-imate [66] to make it easier for gittuf users to use gittuf with forges like GitHub and GitLab. Also, gittuf still relies on the baseline access control security of synchronization points, that of the ability to push to the server. We aim to remove this dependence for gittuf-enabled synchronization points that can use gittuf policy to determine who can push to the server.

When only a subset of a repository's developers use gittuf, the overall security of the repository is still improved due to protections against threats T1 and T3. However, RSL entries will not be created for pushes from those developers who do not use gittuf. This leaves the repository susceptible to metadata manipulation attacks [31], until a gittuf user pushes to the same branch and updates the RSL. Thus, to achieve all of its security goals, gittuf requires adoption by all developers contributing to a repository. While this is similar to security controls offered elsewhere (a forge cannot enforce security controls on changes made by a user who does not use the forge), we aim to extend gittuf's implementation of authorization attestations to support additional mechanisms for authenticating a Git-only developer (*e.g.*, a signed push certificate [67], a signed email patch, etc.).

Finally, we plan to build on gittuf's metadata layer to enable read access control rules, hash algorithm agility to transition Git away from SHA-1, and integrations with other Git tools [20], [21], [39], [40], [68] and industry security efforts such as SLSA [69].

VIII. Related Work

The security of Git has been studied before. Torres-Arias et al. [31] describe a class of attacks against Git repositories, known as "metadata manipulation attacks", and propose a "reference state log" as a solution. In essence, either the synchronization point or a party with commit access can maliciously move any branch pointer to any commit in a repository, even if all commits and tags are signed. gittuf builds on the reference state log concept to address a variety of other security concerns. Specifically, gittuf improves the authentication mechanisms proposed by Torres-Arias et al. to be resistant to a single key compromise, adds mechanisms to revoke RSL contents, and builds support for distributed access control policy declaration and enforcement. In doing so, it provides fork consistency for a repository's activity log, similar to SUNDR [59] and SPORC [60] in networked file systems and cloud contexts. Courtès [70] describes the solution adopted by the Guix package manager. This system embeds an authorization file with a list of all trusted keys at the commit. Each commit's signature is verified against the list of trusted keys from the parent commit. This system is focused on enabling independent authentication of the author of a commit. gittuf also leverages signatures as a mechanism to authenticate Git commits, but implements a hierarchical key management system to protect against a single key compromise. The Guix system also does not support *authorization* policies for repository namespaces, and does not track repository activity.

Xu et al. propose Gringotts [71], a system that maintains an encrypted Git repository for each plaintext Git repository to protect against unauthorized directory reads (which gittuf lacks) and writes to branches. Gringotts is capable of enforcing these access control policies but it does so by relying on centralized key management performed by repository administrators. It also relies on centralized parties to declare all repository access control rules and does not decentralize the repository's activity tracking. On the other hand, gittuf primarily focuses on write permissions, for branches, tags, and files/folders in the repository, and enables distributed, *shared* responsibility for policy declaration. To better integrate Git signing into forges, Afzali et al. [66] describe le-git-imate, a tool to digitally sign Git objects created using the web UI of Git forges like GitHub. Such signatures can be verified using gittuf, making le-git-imate complementary to gittuf. Afzali et al. [72] also describe a mechanism to ensure the trustworthiness of forge code review features. This system protects the integrity of the code review process, rather than repository's contents, lacking the ability to declare access control rules in a shared trust manner.

In the broader area of source control system security, Chen et al. [73] studied weaknesses with version control systems that use delta-encoding, where a repository stores differences between revisions. Git includes in-built protections with its content addressed object store against such attacks. Vaidya et al. [74] added support for cryptographic signatures to centralized version control systems like Apache Subversion [75] that currently lack this feature. Git already includes these features, which are leveraged by gittuf. The Invisible Internet Project (I2P) implemented [76] some access control features for Monotone [77], a predecessor to Git. In addition to key management, the system also included namespace based rules, but relied on server side hooks for enforcement. Wheeler authored an essay [78] that describes security properties for source control systems.

More generally, the area of software supply chain security has seen renewed interest due to high profile attacks [1] with the release of standards [25], [26], practical guidelines for securing software supply chains [79], [80], and academic research. Okafor et al. [81] systematize software supply chain security patterns, showing that three properties (transparency, validity, and separation) are vital for a secured supply chain. gittuf achieves all three properties in its design. The in-toto [6], [55] framework allows for enforcing policies on the process used to build software, but its verification aspects do not extend to the cyclical aspect of source code development, where changes build on prior changes. Similar to other software supply chain security efforts [8], [82], gittuf leverages in-toto as a mechanism to record verifiable information, but is otherwise tailored to the nuances of developing source code using Git.

Software supply chain defense mechanisms rely on cryptographic signing for authenticating trusted entities. Sigstore [7] proposes a way for creating cryptographic signatures using identities via OIDC, addressing issues with managing long-lived keys, with extensions like Speranza [83] and DiVerify [84] adding privacy preserving semantics and removing identity providers as single points of trust. Sigstore is complementary to gittuf, and indeed, gittuf policy supports Sigstore identities for Git signatures. Of note, signing mechanisms have been leveraged for signing in packaging ecosystems, with prior research [85] measuring the factors that drive the adoption of signing factors. In the context of securing package registries, The Update Framework [9], [10] and Uptane [86], [87] secure the delivery of software via update systems using features like delegations, which gittuf builds on, but like in-toto, it does not address the nuances of Git-based source code development. Finally, Ferraiuolo et al. [51] introduced the idea of policy transparency, where policies are inserted into a transparency log. Policy transparency is key to gittuf's aim of decentralized Git repository policy enforcement, with the collectively maintained append-only RSL acting as the transparency log.

IX. Conclusion

In this paper, we introduced gittuf, a forge-agnostic Git security system. gittuf empowers all parties working on a repository, developers and the forge alike, to collectively ensure a repository's security by decentralizing policy declaration, activity tracking, and policy enforcement. In addition to providing strong security guarantees even in the event of a forge compromise, gittuf provides forges a way to prove that they are trustworthy, rather than carry the burden of being non-verifiable trusted third parties.

We validated gittuf's design by evaluating it against seven previously seen version control system attacks and found that gittuf's design would protect against all of them. We also found that gittuf is feasible even with a high volume of repository activity using the repositories for Git and Kubernetes. Therefore, given its practicality and its ability to defend against historic attacks, we conclude that gittuf significantly improves the security of source code development, and thus, the software

supply chain as a whole. This is borne out by gittuf's adoption in OpenSSF and CNCF projects and the ongoing enterprise pilot at Bloomberg.

Acknowledgments

We would like to thank the NDSS Symposium reviewers for their feedback on this paper. We would also like to thank Billy Lynch, Dennis Roellke, Marcela Melara, Santiago Torres-Arias, and Trishank Karthik Kuppusamy for their parts in making gittuf and this paper a reality.

This material is based upon work supported by the United States National Science Foundation (NSF) under Grants No. CNS 2247829, CNS 2054692, and DGE 2043104. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the NSF.

References

[1] FireEye, "Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor," https://www.mandiant.com/resources/blog/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor, 2020.

[2] S. Ramakrishna, "New Findings From Our Investigation of SUNBURST," https://orangematter.solarwinds.com/2021/01/11/new-findings-from-our-investigation-of-sunburst/, 2021.

[3] Sonatype, "8th Annual State of the Software Supply Chain," https://www.sonatype.com/resources/2022-software-supply-chain-report, 2023.

[4] Anchore, "Software Supply Chain Security Report," https://anchore.com/software-supply-chain-security-report-2022/, 2022.

[5] "SLSA++: A Survey of Software Supply Chain Security Practices and Beliefs," https://www.chainguard.dev/unchained/new-slsa-survey-reveals-real-world-developer-approaches-to-software-supply-chain-security, 2023.

[6] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "in-toto: Providing farm-to-table guarantees for bits and bytes," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1393–1410. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias

[7] Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: Software signing for everybody," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2353–2367. [Online]. Available: https://doi.org/10.1145/3548606.3560596

[8] The Linux Foundation, "SLSA: Suppy-chain levels for software artifacts," https://slsa.dev.

[9] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable key compromise in software update systems," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 61–72. [Online]. Available: https://doi.org/10.1145/1866307.1866315

[10] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, "Diplomat: Using delegations to protect community repositories," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016, pp. 567–581. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/kuppusamy

[11] Checkmarx Security Research Team, "Over 170k users affected by attack using fake python infrastructure," https://checkmarx.com/blog/over-170k-users-affected-by-attack-using-fake-python-infrastructure/, 2024.

[12] J. Corbet, "An attempt to backdoor the kernel," http://lwn.net/Articles/57135/, 2003.

[13] E. Homakov, "Hacking rails/rails repo," https://homakov.blogspot.com/2012/03/how-to.html, 2012.

[14] Free Software Foundation, "Savannah and www.gnu.org downtime," https://www.fsf.org/blogs/sysadmin/savannah-and-www.gnu.org-downtime, 2010.

[15] N. Popov, "php.internals: Changes to Git commit workflow," https://news-web.php.net/php.internals/113838.

[16] The Gentoo Developers, "Project:Infrastructure/Incident reports/2018-06-28 Github ," https://wiki.gentoo.org/wiki/Github/2018-06-28.

[17] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R.-P. Weinmann, E. Rescorla, and H. Shacham, "A systematic analysis of the juniper dual ec incident," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 468–479. [Online]. Available: https://doi.org/10.1145/2976749.2978395

[18] "Git," https://git-scm.com.

[19] Stack Overflow, "2022 Developer Survey - Version Control," https://survey.stackoverflow.co/2022/#version-control-version-control-system, 2022.

[20] "GitHub," https://github.com.

[21] "GitLab," https://about.gitlab.com.

[22] "Bitbucket," https://bitbucket.org/product/.

[23] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1509–1526.

[24] The Linux Foundation, "SLSA: Threats & mitigations," https://slsa.dev/spec/v1.0/threats, 2023.

[25] M. Souppaya, K. Scarfone, and D. Dodson, *Secure Software Development Framework (SSDF) version 1.1*, Feb. 2022. [Online]. Available: http://dx.doi.org/10.6028/NIST.SP.800-218

[26] R. Chandramouli, F. Kautz, and S. Torres-Arias, "Strategies for the integration of software supply chain security in DevSecOps CICD pipelines," Tech. Rep., 2 2024. [Online]. Available: https://doi.org/10.6028/nist.sp.800-204d

[27] R. Potvin and J. Levenberg, "Why Google stores billions of lines of code in a single repository," *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[28] A. Zeino, "Faster Together: Uber Engineering's iOS Monorepo," https://www.uber.com/blog/ios-monorepo/, 2017.

[29] A. Lucido, "The Journey To Android Monorepo: The History Of Uber Engineering's Android Codebase Organization," https://www.uber.com/blog/android-engineering-code-monorepo/, 2017.

[30] D. Ordogh, "Typelevel Boston Summit 2018: Pants and Monorepos," https://www.youtube.com/watch?v=IL6LBWNi3fE, April 2018.

[31] S. Torres-Arias, A. K. Ammula, R. Curtmola, and J. Cappos, "On omitting commits and committing omissions: Preventing git metadata tampering that (re)introduces software vulnerabilities," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 379–395. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/torres-arias

[32] "Open Source Security Foundation (OpenSSF)," https://openssf.org.

[33] "Cloud Native Computing Foundation (CNCF)," https://cncf.io.

[34] "Bloomberg," https://www.bloomberg.com/company/.

[35] S. Chacon and B. Straub, *Pro Git*. Apress, 2014.

[36] GitHub, "Checking out pull requests locally," https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/reviewing-changes-in-pull-requests/checking-out-pull-requests-locally.

[37] "Git Notes," https://git-scm.com/docs/git-notes.

[38] Gerrit, "The refs/for namespace," https://gerrit-review.googlesource.com/Documentation/concept-refs-for-namespace.html.

[39] Google, "git-appraise: Distributed code review for Git," https://github.com/google/git-appraise.

[40] M. Muré, "git-bug: Distributed, offline-first bug tracker embedded in Git, with bridges," https://github.com/MichaelMure/git-bug.

[41] "gitremote-helpers: Helper programs to interact with remote repositories," https://git-scm.com/docs/gitremote-helpers.

[42] "gitsign: Keyless Git signing using Sigstore," https://github.com/sigstore/gitsign.

[43] "Gitea," https://about.gitea.com/.

[44] "Forgejo: Beyond coding. We forge." https://forgejo.org/.

[45] "Gogs: A painless self-hosted Git service," https://gogs.io/.

[46] "CVE-2023-7028," https://www.cve.org/CVERecord?id=CVE-2023-7028.

[47] "CVE-2022-2992," https://www.cve.org/CVERecord?id=CVE-2022-2992.

[48] "CVE-2024-39930," https://www.cve.org/CVERecord?id=CVE-2024-39930.

[49] GitHub, "About authentication to GitHub," https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/about-authentication-to-github.

[50] GitLab, "User account," https://docs.gitlab.com/ee/user/profile/index.html.

[51] A. Ferraiuolo, R. Behjati, T. Santoro, and B. Laurie, "Policy transparency: Authorization logic meets general transparency to prove software supply chain integrity," in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED'22. New York, NY, USA: Association for Computing Machinery, 2022, p. 3–13. [Online]. Available: https://doi.org/10.1145/3560835.3564549

[52] A. K. Kornel, "Welp, there go my Git signatures," https://karl.kornel.us/2017/10/welp-there-go-my-git-signatures/, 2017.

[53] GitLab, "Merge Requests," https://docs.gitlab.com/ee/user/project/merge_requests/.

[54] "git-merge-tree: Perform merge without touching index or working tree," https://git-scm.com/docs/git-merge-tree.

[55] "in-toto Attestation Framework," https://github.com/in-toto/attestation.

[56] B. DeHamer and P. Harrison, "Introducing npm package provenance," https://github.blog/2023-04-19-introducing-npm-package-provenance/, 2023.

[57] Docker, "Build attestations," https://docs.docker.com/build/attestations/.

[58] Google Cloud, "View build provenance," https://cloud.google.com/build/docs/securing-builds/view-build-provenance.

[59] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (sundr)," in *Proc. of the 6th USENIX Symposium on Operating Systems Design & Implementation (OSDI '04)*, 2004.

[60] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "Sporc: Group collaboration using untrusted cloud resources," in *Proc. of the 9th USENIX Symposium on Operating Systems Design & Implementation (OSDI '10)*, 2010.

[61] "Catalog of Supply Chain Compromises," https://github.com/cncf/tag-security/tree/main/supply-chain-security/compromises.

[62] J. Robertson, "Juniper Breach Mystery Starts to Clear With New Details on Hackers and U.S. Role," https://www.bloomberg.com/news/features/2021-09-02/juniper-mystery-attacks-traced-to-pentagon-role-and-chinese-hackers, 2021.

[63] "Git Source Code Mirror," https://github.com/git/git.

[64] The Kubernetes Authors, "Kubernetes Source Code Repository," https://github.com/kubernetes/kubernetes.

[65] The Kubernetes Authors, "External Repository Staging Area," https://github.com/kubernetes/kubernetes/blob/3f7a50f38688eb332e2a1b013678c6435d539ae6/staging/README.md.

[66] H. Afzali, S. Torres-Arias, R. Curtmola, and J. Cappos, "le-git-imate: Towards verifiable web-based git repositories," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 469–482. [Online]. Available: https://doi.org/10.1145/3196494.3196523

[67] "git-push: –signed," https://git-scm.com/docs/git-push#Documentation/git-push.txt---signedtruefalseif-asked.

[68] "Gerrit code review," https://www.gerritcodereview.com/.

[69] M. Lieberman, "The Breadth and Depth of SLSA," https://slsa.dev/blog/2023/04/the-breadth-and-depth-of-slsa, 2023.

[70] L. Courtès, "Building a secure software supply chain with GNU Guix," *The Art, Science, and Engineering of Programming*, vol. 7, no. 1, June 2022. [Online]. Available: https://programming-journal.org/2023/7/1/

[71] W. Xu, H. Ma, Z. Song, J. Li, and R. Zhang, "Gringotts: An encrypted version control system with less trust on servers," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–18, 2023.

[72] H. Afzali, S. Torres-Arias, R. Curtmola, and J. Cappos, "Towards verifiable web-based code review systems," *Journal of Computer Security*, vol. 31, no. 2, pp. 153–184, 2023.

[73] B. Chen and R. Curtmola, "Auditable version control systems." in *NDSS*, 2014.

[74] S. Vaidya, S. Torres-Arias, R. Curtmola, and J. Cappos, "Commit signatures for centralized version control systems," in *Proc. of the 34th International Conference on ICT Systems Security and Privacy Protection (IFIP SEC '19)*. Springer International Publishing, 2019, pp. 359–373.

[75] "Apache subversion," https://subversion.apache.org/.

[76] "I2P: Monotone Guide," https://geti2p.net/en/get-involved/guides/monotone.

[77] "Monotone," https://www.monotone.ca/.

[78] D. A. Wheeler, "Software Configuration Management (SCM) Security," https://dwheeler.com/essays/scm-security.html, 2015.

[79] A. Vega, E. Fox, F. Razzak, A. F. Marshall, C. Kennedy, M. Swift, J. Meadows, A. S. A. Yelgundhalli, N. Kumar, J. Lock, A. Martin, M. Moore, V. Anandan, M. Logan, R. Julian, B. Lum, M. Lieberman, and G. Ing, "Software Supply Chain Best Practices," Tech. Rep., 5 2021. [Online]. Available: https://github.com/cncf/tag-security/blob/main/community/working-groups/supply-chain-security/supply-chain-security-paper/CNCF_SSCP_v1.pdf

[80] A. S. A. Yelgundhalli, A. F. Marshall, A. Vega, A. Block, A. Chetal, A. Simon, B. Lum, B. Mitchell, C. Kennedy, D. Papandrea, G. Aguiar, J. Hall, J. Kjell, M. Moore, M. Moore, M. Lieberman, P. Patel, P. Wadhwa, and S. Nadgowda, "The Secure Software Factory," Tech. Rep., 5 2022. [Online]. Available: https://github.com/cncf/tag-security/blob/main/community/working-groups/supply-chain-security/secure-software-factory/Secure_Software_Factory_Whitepaper.pdf

[81] C. Okafor, T. R. Schorlemmer, S. Torres-Arias, and J. C. Davis, "Sok: Analysis of software supply chain security by establishing secure design properties," in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED'22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 15–24. [Online]. Available: https://doi.org/10.1145/3560835.3564556

[82] M. Moore, A. S. A. Yelgundhalli, and J. Cappos, "Securing automotive software supply chains," in *Symposium on Vehicles Security and Privacy (VehicleSec)*, 2024.

[83] K. Merrill, Z. Newman, S. Torres-Arias, and K. R. Sollins, "Speranza: Usable, privacy-friendly software signing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 3388–3402. [Online]. Available: https://doi.org/10.1145/3576915.3623200

[84] C. L. Okafor, J. C. Davis, and S. Torres-Arias, "Diverify: Diversifying identity verification in next-generation software signing," 2024. [Online]. Available: https://arxiv.org/abs/2406.15596

[85] T. R. Schorlemmer, K. G. Kalu, L. Chigges, K. M. Ko, E. A. Ishgair, S. Bagchi, S. Torres-Arias, and J. C. Davis, "Signing in four public software package registries: Quantity, quality, and influencing factors," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 1160–1178.

[86] T. K. Kuppusamy, A. Brown, S. Awwad, D. McCoy, R. Bielawski, C. Mott, S. Lauzon, A. Weimerskirch, and J. Cappos, "Uptane: Securing software updates for automobiles," in *International Conference on Embedded Security in Car*, 2016, pp. 1–11.

[87] T. K. Kuppusamy, L. A. DeLong, and J. Cappos, "Uptane: Security and customizability of software updates for vehicles," *IEEE vehicular technology magazine*, vol. 13, no. 1, pp. 66–73, 2018.

APPENDIX A
ARTIFACT APPENDIX

This appendix contains information on the artifact evaluation for gittuf.

A. Description & Requirements

1) How to access: The artifact consists of two parts: the source code of gittuf itself and the set of experiments for evaluating gittuf. You can obtain these files in two ways.

Option A - Zenodo

The permanently available copy of both parts is available on Zenodo with the DOI https://doi.org/10.5281/zenodo.14252266. The upload contains two `.zip` files, one for `gittuf` itself and the other, `gittuf-ndss-eval`, for the evaluation scripts.

Option B - GitHub

The GitHub-hosted copy of gittuf itself is available at https://github.com/gittuf/gittuf, and the set of experiments for evaluating gittuf are available at https://github.com/adityasaky/gittuf-ndss-eval. As suggested below in the manual installation instructions, we suggest using gittuf version `v0.7.0`, available at https://github.com/gittuf/gittuf/releases/tag/v0.7.0.

2) Hardware dependencies: None.

3) Software dependencies: We provide two ways to run our experiments. The first, and recommended option, is to use the included Dockerfile. For this option, only a container runtime such as `docker` or `podman` is necessary. The second option, the manual installation, requires the following dependencies to be installed:

- A recent version of Python 3
- GNU Make
- Go 1.22.8 or newer
- Git 2.43 or newer

4) Benchmarks: None.

B. Artifact Installation & Configuration

There are two options for installation and configuration.

Option A - Containerized

1) Download the `gittuf-ndss-eval` repository linked above.
2) Follow the instructions inside on building and running the Docker container.

Option B - Manual Installation

1) Ensure that all prerequisites listed above are installed on your system.
2) Download the gittuf source code, available above in the `gittuf/gittuf` repository. Make sure to use the `v0.7.0` release. Once downloaded, build gittuf by running `make install` in the gittuf directory.
3) Download the artifact evaluation repository (`gittuf-ndss-eval`). The downloaded repository will contain the scripts and sample repository data to allow for evaluation of gittuf.

C. Experiment Workflow

Each experiment is contained in its own Python file. The experiments are structured such that the script walks through each command that is being run and displays the results to the screen.

D. Major Claims

gittuf's major claims are as follows:

- (C1): gittuf must enable distributing the responsibility of setting a repository's policy amongst multiple trusted developers, with the ability to configure the number of developers who must approve to achieve consensus. This is demonstrated by experiments (E1) and (E2).
- (C2): gittuf must decentralize how the log of repository activity is maintained to ensure a single trusted entity cannot arbitrarily tamper with the log's contents. This is demonstrated by experiment (E3).
- (C3): gittuf must enable any party that can read the contents of a repository to independently verify that policy was followed, without trusting any other party to do so on their behalf. This is demonstrated by experiment (E4).

E. Evaluation

1) Experiment (E1): [Unilateral Policy Modification] [5 human-minutes]: This experiment simulates a scenario where a single developer is prevented from editing a policy that was configured to require two developers to sign off.

[Preparation]

Ensure all prerequisite software is installed.

[Execution]

Run `experiment1.py` with Python.

[Results]

The script simulates the following scenario: First, a repository owner (with key `root`) creates a gittuf-enabled repository and delegates trust in the policy to two developers (`developer1` and `developer2`, respectively). To prevent a single developer from making changes themselves, the threshold for policy metadata signatures is set to two. That is, both developers need to authorize changes to the policy by signing the rule file.

`developer1` and `developer2` initialize and set a rule to protect the main branch in the policy, with both developers signing off on the change.

`developer1` then attempts to add another rule without `developer2`'s agreement, which causes verification to fail.

Successful completion of all the steps run by the script indicates a successful simulation.

2) Experiment (E2): [Delegations] [5 human-minutes]: This experiment simulates utilization of gittuf's granular delegations feature that allows for distributing policy declaration responsibilities amongst multiple developers without overprivileging them.

[Preparation]

Ensure all prerequisite software is installed.

[Execution]

Run `experiment2.py` with Python.

[Results]

The script simulates the following scenario: First, a repository owner defines a policy and delegates authority to make changes to the `main` branch to `developer1` and the `feature` branch to `developer2`. `developer1` then delegates trust to `developer3` for the `feature` branch, *despite `developer1` not being trusted for that branch*.

When `developer3` attempts to verify their change to the `feature` branch, gittuf alerts them that they are not trusted for the branch. In summary, this highlights that gittuf's delegations can be used to enable developers to extend the policy in limited ways: `developer1` is trusted to delegate trust only in the `main` branch.

Successful completion of all the steps run by the script indicates a successful simulation.

3) Experiment (E3): [RSL Divergence] [5 human-minutes]: This experiment simulates a scenario focusing on how gittuf's Reference State Log (RSL) propagates across repository copies, enabling detection of tampering.

[Preparation]

Ensure all prerequisite software is installed.

[Execution]

Run `experiment3.py` with Python.

[Results]

The script simulates the following scenario: First, a repository owner creates a gittuf-enabled repository and makes a commit. `developer1` then clones the repository and makes a change (authorized by the policy), and then pushes their changes. The upstream repository maliciously drops these changes.

`developer2` then clones the repository, unaware as to what has happened. The user makes a commit and pushes it to the remote repository. `developer1` then attempts to pull the latest changes, but is warned that their RSL has diverged from what is on the remote repository, alerting them of the server's misbehavior.

Successful completion of all the steps run by the script indicates a successful simulation.

4) Experiment (E4): [Write Rule Violations] [5 human-minutes]: This experiment simulates a scenario where a user writes to a part of a repository they are not allowed to; another user then pulls the latest copy of the repository and then attempts to verify the changes. After a violation is discovered, gittuf's recovery procedure is invoked.

[Preparation]

Ensure all prerequisite software is installed.

[Execution]

Run `experiment4.py` with Python.

[Results]

The script simulates the following scenario: A repository owner creates a gittuf-enabled repository and sets policy authorizing `developer1` to make changes to the main branch. Another developer, `developer2`, who is only allowed to edit the `feature` branch, submits a commit that affects the `main` branch.

Another developer then clones the repository onto their machine and attempts to verify the changes, but gittuf raises an alert that an unauthorized signature is on a commit (against the policy).

`developer2` then reverts the unauthorized commit and revokes the RSL entry for the commit, successfully restoring the repository. Note that even though `developer2` is only authorized to make changes to the 'main' branch, they were still able to *fix* the violation. This is because any user who is allowed to write to the repository at all is allowed to fix it to bring it back into compliance with policy.

Successful completion of all the steps run by the script indicates a successful simulation.

F. Customization

For convenience, each script can be configured using two (optional) options:

1) `--automatic`, which skips waiting for input before each step is run.
2) `--repository-directory`, which controls the directory used for the script's working repository. Note that this option will not automatically delete the folder once the script has completed.