SourceFabric: Consistent and Scalable Security Policies for Git Repositories

Aditya Sirish A Yelgundhalli
Bloomberg and New York University
New York, NY, USA
aditya.sirish@nyu.edu

Patrick Zielinski
New York University
New York, NY, USA
patrick.z@nyu.edu

Marcela S. Melara
Intel Corporation
Hillsboro, OR, USA
marcela.melara@intel.com

Dennis Roellke
Bloomberg
New York, NY, USA
droellke@bloomberg.net

Reza Curtmola
New Jersey Institute of Technology
Jersey City, NJ, USA
reza.curtmola@njit.edu

Justin Cappos
New York University
New York, NY, USA
jcappos@nyu.edu

***Abstract*—A company's software is often developed across many repositories, with large companies utilizing tens or hundreds of thousands of Git repositories. The current approach to enforcing security policies across these repositories (such as multi-factor authentication, secret scanning, and branch protection) is to use grouping constructs provided by forges like GitHub, GitLab, and Bitbucket. However, these solutions impose a policy management burden, resulting in misconfigured, inconsistent, and outdated security policies across different repositories and groups.**

We introduce SourceFabric, an open source system that enables consistent and updated policies across a company's myriad repositories through scalable and secure cross-repository policy reuse. SourceFabric modularizes security policies, allowing companies to define flexible security policies that can be reused by appropriate company repositories. In doing so, SourceFabric meets specific design and security requirements. First, SourceFabric ensures that a repository's effective security policy is complete and sound, *i.e.*, all required policies are applied and no extraneous policies are applied. This is necessary to prevent a repository's policy from being overly restrictive or overly permissive. Second, SourceFabric ensures that the parties responsible for managing the reused security policies can update them as needed, without being constrained by factors such as the number of repositories reusing the policy or other policies reused alongside the updated policy. SourceFabric also automatically updates a repository's reused policies to ensure they do not grow stale. Finally, SourceFabric enables verifiability and auditability of reused policies, preventing attacks such as policy equivocation and untrusted policy issuers.

SourceFabric was created by a team of academic and industry practitioners to address an acute, real-world policy management need. SourceFabric has been upstreamed into production within the popular OpenSSF project, gittuf, and is being piloted at Bloomberg.

1. Introduction

Modern software is not developed in a vacuum. Companies typically separate the software they produce into many thousands of distinct source repositories [1]–[9] (*e.g.*, to delegate development of individual components of a software system to distinct teams [10]), often using Git [11], today's most popular version control system [12]. To mitigate software supply chain attacks that target source code development [13]–[19], companies apply security policies to their Git repositories, typically using the security mechanisms provided by centralized forge software (*e.g.*, GitHub [20], GitLab [21], and Bitbucket [22]), or using distributed repository security systems like gittuf [23], [24]. These security policies, which protect the integrity of a repository and enforce development procedures, are based on regulations [25], [26], standards [27], industry best practices [28], [29], and internal company requirements.

Companies often have guidance that should apply to their repositories (*e.g.*, the use of multifactor authentication), but enabling and tracking these requirements separately across thousands of individual repositories is burdensome and error-prone. To reduce the burden of policy management, forges implement constructs for grouping repositories that share the same owners (*e.g.*, GitHub organizations [30], GitLab groups [31], Bitbucket workspaces [32]), which we call *repository groups*. The owners of each distinct repository group are trusted to configure the expected policy for all contained repositories and update the policy when necessary. The forge enforces the policy for all repositories within the group.

Although this approach leads to a linear decrease in complexity, it is not a true solution to the problem. Large companies may still have thousands of groups and those repositories may need to apply a wide array of security policies. For example, a project written by the payments processing team may need to comply with the Payment Card Industry Data Security Standard (PCI DSS) [25], while a project written by the infrastructure team may need to comply with the National Institute of Standards and Technology (NIST) Secure Software Development Framework (SSDF) [27]. If these projects are used by residents of the EU, they also need to comply with the new Cyber Resilience Act (CRA) [26]. And this is all without considering any policy differences due to different programming languages, teams of developers, maturity models, security policy differences from acquired companies, etc., which are common in large companies. This

means that a large company may have many thousands of repository groups, many of which are meant to be overlapping subsets of each other's policies. Configuring these policies correctly and consistently across all repositories in a company becomes a Herculean task, leading to frequent errors [33]–[36]. These errors have, in some cases, resulted in data breaches and financial penalties [37], [38].

To address these issues, we present **SourceFabric**, a system that enables consistent, scalable, and secure cross-repository policy enforcement and reuse. SourceFabric treats a repository's effective security policy as a composition of one or more independently managed security policies. This makes Git repository security policies modular as each set of security requirements (*e.g.*, baseline company policy, regulatory compliance, open source standards, etc.) can be represented by a distinct policy that can be reused by any number of repositories. SourceFabric is agnostic of the underlying policies it orchestrates, instead supporting the types of policies (*e.g.*, fine-grained or high-level) that the underlying repository security system supports (in this paper, we use gittuf).

Notably, SourceFabric does not restrict the reused security policies in a repository to coming from only one upstream policy issuer (*i.e.*, analogous to limiting a repository belonging to a single repository group, as forges do today). Instead, a repository can reuse security policies from an arbitrary number of sources, even transitively; indeed, to our knowledge, SourceFabric *is the only system in existence today* with this capability. Thus, in practice, each repository reuses existing security policies rather than defining its own complete security policy from scratch, making Git repository security policy management tractable even at enterprise scale. SourceFabric shifts the responsibility of repository security policy management away from the many administrators of individual repositories (or repository groups) to the few entities that define the security requirements in the first place, requiring only a one-time setup procedure per repository to enable and correctly configure SourceFabric.

SourceFabric fulfills three requirements for *securely applying reusable Git repository security policies at enterprise scale*. First, in composing a repository's effective security policy from multiple other policies, SourceFabric ensures *completeness and soundness*. These properties ensure that all security policies that must be applied to a repository are applied, and no extraneous policies are applied. In turn, this ensures that the repository's effective security policy is neither overly permissive nor overly restrictive.

Second, SourceFabric ensures *agility and freshness* in how security policies are updated to ensure a repository's effective security policy does not become stale. Security requirements, and thus policies, are regularly updated to account for emerging threats. SourceFabric's agility empowers the entities that create security policies to update them as needed, without being constrained by factors such as the number of repositories where the policies are applied and conflicts with other policies applied alongside them. Freshness ensures that any security policy applied in a repository is the latest version, and that updates are applied in a reasonable time frame. To do so, SourceFabric tracks the source of each reused policy for a repository and automatically updates the policy when it is modified by its issuer.

Third, SourceFabric ensures *verifiability and auditability* to prevent tampering with a repository's effective security policy. SourceFabric establishes trust in each reused policy using cryptographic signatures, ensuring a policy is only reused when it is issued by a trusted party. SourceFabric also allows any party to inspect the evolution of each component policy.

To summarize, our contributions are as follows:

– A characterization of the challenges involved in applying security policies at scale across thousands of Git repositories, based on our collaboration with a large enterprise that manages hundreds of thousands of repositories.

– SourceFabric, a novel system that enables enforcement of repository security policies at scale via flexible and secure cross-repository policy reuse. *SourceFabric modularizes repository security policies* and treats a repository's effective security policy as a composition of one or more other, independently managed security policies. SourceFabric is the only system that allows a repository to reuse security policies from an arbitrary number of sources.

– A set of system security requirements for SourceFabric, *completeness and soundness*, *agility and freshness*, *verifiability and auditability*. We describe the impact of not meeting these requirements, and show how SourceFabric meets these requirements.

– An implementation of SourceFabric that has been upstreamed into gittuf, the open source Git repository security system that is an OpenSSF [39] project, and is now part of a pilot at Bloomberg. SourceFabric incurs negligible overhead, adding only ∼42KB of data and a few milliseconds of verification time on top of gittuf.

Roadmap. In the rest of this paper, we describe the aspects of Git that are relevant to SourceFabric, source code contribution policies, the existing mechanisms provided by forges to apply policies across repositories at large scale, as well as their limitations (Section 2). We then discuss the problem statement SourceFabric is meant to address, our assumptions, SourceFabric's participants, and the system security requirements we considered for SourceFabric (Section 3). We proceed to SourceFabric's design (Section 4), followed by an analysis of SourceFabric's properties (Section 5) and the performance evaluation of SourceFabric's implementation (Section 6). We end with a discussion of SourceFabric and its limitations (Section 7), related work (Section 8), and our conclusions (Section 9).

2. Background

Before we describe SourceFabric, we provide some background about Git and source code contribution security policies. We also briefly describe current solutions for applying Git repository security policies at scale, and the gittuf security system, which this work utilizes.

2.1. Git

Git [11] is a distributed version control system that tracks the evolution of a project's source in a "repository". The contents of a repository are tracked in a content-addressed store [40] using SHA-1 or SHA-256 identifiers,

depending on the configuration of the repository. A Git revision such as a commit or a tag represents the state of the repository at some point in time. In addition to revisions, Git supports named pointers called references. A reference has a human-friendly name and points to a Git object (such as a commit).

Most commonly, Git references are used as *branches*, which are used to track different lines of development. At any given point in time, the commit that a branch points to is known as the "tip of the branch" and represents the latest state of development in that branch. When a change is made to a branch, a new commit is created that describes the change and contains the representation of the repository's files with the change applied. Then, the branch is updated to point to the new commit.

Git also supports *custom references*, which are frequently used by Git-based systems [23], [41]–[45] to store additional information within a Git repository, thus leveraging the change tracking and versioning features of the Git object store.

Although Git is a distributed version control system, collaborative software development requires a centralized location where developers can synchronize their changes. Thus, the typical development workflow has a primary or canonical copy of the repository that acts as the synchronization point. This can be a simple, standalone Git repository or a forge.

2.2. Securing Source Code Contributions

Security policies for source code contribution are rules designed to protect the integrity of the software development process. These rules can take many forms. For example, companies often use access controls to limit who can modify a project and set requirements for code reviews. Many best practice guides [28], [29] and regulatory standards [25], [27] recommend that every change be approved by at least two trusted developers.

Similarly, many companies use workflow control rules to enforce consistent development practices. These policies ensure that developers follow established procedures. For example, some companies require a repository's development history to remain *continuous*, also a recommendation in best practice guides [28], [29]. This conflicts with Git features that allow rewriting history. As a result, it is common to prohibit actions such as Git rebase and force pushes that would break this continuity. Some other common workflow controls require code quality checks [46], like linters and tests, to pass before changes are accepted.

2.3. Applying Security Policies at Scale

Forges like GitHub, GitLab, and Bitbucket provide the ability to group repositories with shared ownership [30]–[32]. Group owners can configure a single security policy that applies uniformly to all repositories within the group, simplifying policy management for repository owners and reducing the risk of inconsistencies between repositories in the same group. Unfortunately, forge grouping constructs do not solve the problem of inconsistent policies across a company, resulting in several issues.

First, manual policy configuration in every group can lead to inconsistent but compliant policies across different groups. This complicates verifying a repository's security posture as the verifier, like a company's security team, must reason about the correctness of each distinct policy.

Even worse, manual configuration of group policy can lead to misconfigurations due to human error, a frequent problem encountered in security systems [33]–[36]. The people responsible for configuring a group's security policy—the administrators—are different from those who define the company's security requirements, *i.e.*, the security team and compliance officers. Thus, group owners may misinterpret requirements communicated to them by the security team and apply incorrect policies. Such misconfigurations can undermine the security of the affected repositories, putting them at risk of software supply chain attacks.

Finally, manual configuration at every group can lead to the irregular application of policy updates, where some groups are overlooked and left underprotected. This is another instance of misconfiguration due to human error. However, in this case, it is a consequence of the *company's scale* rather than misinterpretation of security requirements.

2.4. gittuf

gittuf [23], [24] is a forge-agnostic, decentralized repository security system. It stores security policy as signed metadata (configured by the repository's administrators) in a custom Git reference in the repository. gittuf tracks repository activity in an append-only "reference state log" (RSL) [47], also stored in the repository in a custom Git reference. Each entry in the RSL is signed by the party making the repository change.

The repository administrators and other developers trusted to set security policy do so by invoking a gittuf client on the local copy of the repository. After the client is installed and configured on a repository, it automatically records repository activity in the RSL. Subsequently, all parties working on the repository enforce the policy using their gittuf client operating over their local copy of the repository. Specifically, the gittuf client enforces the security policy stored in the repository over events recorded in the RSL. If the client identifies a policy violation, it can correct the affected Git reference and files. Thus, as every developer independently enforces the repository's security policy, a single honest developer is sufficient to detect policy violations.

Crucially, however, gittuf operates within the boundaries of a single repository. Every repository's policy must be configured independently. This means that gittuf does not solve the problem of misconfigured, stale, or inconsistent policies across a large number of repositories. It is a formidable challenge to extend the guarantees provided by gittuf in a single repository to a large collection of repositories, while fulfilling an additional rich set of security requirements (completeness, soundness, agility, freshness, verifiability, auditability) and also dealing with the complexities of scale and network issues, such as delays and outages.

3. System Overview

In this section, we describe the problem SourceFabric addresses, as well as the assumptions, concepts, and participants necessary to describe SourceFabric's design. In addition, we outline the system requirements that guided SourceFabric's design.

3.1. Problem Statement

Our setting is any large company that has a sizeable surface area for policy management. First, such a company may have tens or hundreds of thousands of Git repositories for which security policies must be defined and kept up-to-date. Second, a repository in such a setting is subject to a myriad of security requirements. For instance, all repositories may be subject to the company's baseline security policies as well as open source best practices standards such as SLSA [28] or the OpenSSF Security Baseline [48]. In addition, a subset of repositories may be subject to additional requirements due to the product they provide (*e.g.*, payments processing repositories subject to PCI DSS [25], infrastructure code subject to NIST SSDF [27]), the way they are developed (*e.g.*, programming language specific policies), or jurisdictional reasons (*e.g.*, software intended for the European Union must abide by the CRA [26]). Third, such a company frequently has legally independent organizations (*e.g.*, subsidiaries, acquisitions), and this can both further complicate how policies are managed across these boundaries, and add more security requirements.

In such a large-scale operational setting, the current options provided by forges for policy management, discussed in Section 2.3, have shortcomings that impact the security posture of the company's repositories. By default, a comprehensive security policy that handles all applicable security requirements must be defined for every repository. When security requirements change, the policy for every repository must be independently updated. Intuitively, this approach to policy management leads to security policies that are misconfigured, stale, and inconsistent across repositories. To mitigate this, forges provide grouping constructs. Grouping constructs reduce the number of places policies must be managed, but only linearly; in the setting we consider, there continue to be thousands of places (*i.e.*, groups) where policies must be independently managed, meaning the aforementioned threats remain.

A misconfigured or stale policy can lead to a repository having an effective security policy that is either **overly permissive** (*i.e.*, a change that should have been blocked is allowed) or **overly restrictive** (*i.e.*, a change that should have been allowed is blocked). Inconsistencies across repositories, especially when one depends on another, can enable a malicious actor to target the repository with the weaker security policy (*i.e.*, the "weakest link"). Such an attacker can corrupt the final product that the repositories are for, including those components from other repositories with appropriate policies.

In summary, the problem SourceFabric addresses is that of managing and applying Git repository security policies in a setting with a complex organizational structure, a large number of repositories, and varied security requirements that must be met. In addressing this gap, SourceFabric aims to mitigate the threats posed by misconfigured, stale, or inconsistent security policies being applied across the company's repositories.

3.2. Assumptions & Participants

SourceFabric addresses the problem of managing and applying Git repository security policies described in Section 3.1 by providing a framework for flexible, secure, cross-repository policy reuse. SourceFabric treats a repository's effective security policy as a composition of one or more other security policies that are managed independently. This makes policy management tractable, as it enables each repository to reuse existing security policies authored and maintained by other parties, rather than requiring the repository to define and continually maintain its own security policy from scratch. We now outline the assumptions that underpin SourceFabric.

– Git stores data and ensures data integrity.

– The open source, forge-agnostic Git repository security system gittuf [23], [24] enforces a repository's security policies, handles key distribution, rotation, revocation, tying of keys to different actors, and prevents "metadata manipulation attacks" [47].

– Cryptographic algorithms are secure, using standard cryptographic assumptions.

As SourceFabric is intended to enable retrieval and application of multiple gittuf policies, we define the following concepts in SourceFabric.

– A *local policy* is a gittuf security policy (Section 2.4) that is configured for a repository. This represents a repository's policy in a gittuf-enabled repository.

– A *reusable policy* is a gittuf security policy that is distributed for use by other parties (not unlike open source software). Reusable policies may be intended both only for internal use within a company, or by any party.

In addition, SourceFabric considers the following participants.

Security Framework Publishers are trusted entities that author publicly-available specifications and frameworks that describe requirements for reducing particular software supply chain risks. The individuals authoring these security frameworks are subject-matter experts with a firm understanding of source code and repository management risks.

In SourceFabric, security framework publishers can represent governmental organizations (*e.g.*, NIST [49]), standards bodies (*e.g.*, IETF [50]) or cross-industry organizations such as open source foundations (*e.g.*, OpenSSF [39], CNCF [51], OWASP [52]). For instance, the NIST Secure Software Development Framework [27] specifies practices for protecting source code from unauthorized access and tampering. Depending on a security framework publisher's scope, their specifications may form part of regulatory requirements.

Companies are trusted entities that develop and maintain software across thousands of Git repositories hosted on forges. In SourceFabric, companies include commercial or governmental entities as well as open source organizations,[1] which commonly operate like enterprises in the

1. We use "companies" instead of "organizations" to avoid confusion with the GitHub organization [30] grouping construct.

way they apply security policies during software development. We assume each company has reputational and economic incentives to adhere to the published security frameworks that address supply chain security risks applicable to the company's processes and resources.

Security experts within the company are typically responsible for determining the specific regulations, standards and best practices that enable the company to meet its compliance requirements. Based on the selected security frameworks, these experts may then choose security policies for source code contributions to apply to the company's repositories, *i.e.*, the set of security-related configurations that software repositories must implement to be in compliance.

Policy Issuers are entities that author and maintain reusable gittuf policies in Git repositories they control. We assume that security framework publishers, software companies, forges, and other independent parties may act as policy issuers. Further, we assume the company can out-of-band establish trust in a policy issuer. This may be because the issuer is another stakeholder within the company (*e.g.*, the security team) or based on reputational or contractual reasons (*e.g.*, regulators, security foundations).

In the case of companies, the security experts may determine that they cannot reuse an existing, publicly available policy, and therefore issue a reusable policy themselves. Any policies they issue may reveal proprietary or privacy-sensitive information about a company's software development processes or individual employees. Thus, companies may opt to keep the reusable policies they maintain internal and not make them available for reuse outside the company.

Repository Administrators oversee development in the repository and are authorized to configure the repository's local and reused policies. A repository's administrators are determined by the company. Thus, we expect that repository administrators faithfully implement the company's compliance requirements.

Developers are individuals that contribute changes to a repository. They are neither trusted to configure a repository's security policy nor the policies to reuse.

3.3. System Security Requirements

As SourceFabric is reponsible for applying the configured security policies to a repository, we derive the following system requirements that SourceFabric must meet to ensure a repository's security posture meets the company's expectations and that *SourceFabric itself does not become an attack vector that can weaken a repository's security posture*. We explain in each instance why the requirement is vital to SourceFabric's purpose, and the undesirable consequences on the repository's security posture if the requirement is not met.

Completeness and Soundness. Completeness necessitates that all policies that are required to be applied to a repository are indeed applied by SourceFabric, and soundness requires that no extraneous policies are applied to the repository. Without completeness and soundness, a repository will have an effective security policy that is overly permissive, overly restrictive, or both. In other words, a change that would have been blocked by a policy that was missed (*i.e.*, no completeness) may be allowed, and a change that would have been allowed may be prevented (*i.e.*, no soundness). Crucially, the lack of these properties will misrepresent the repository's security posture to its stakeholders, misleading them into believing that the repository is compliant with certain security requirements.

Agility and Freshness. Agility ensures that a policy issuer is able to rapidly update the security policy to protect repositories from emerging threats or to meet new security requirements. The policy issuer must not be prevented from updating the security policy by factors such as the number of repositories that use the policy in question, or any other security policies reused by those repositories. Without agility, a policy issuer will be unable to update the security policy as needed, subjecting a repository to stale security policies that may be overly permissive or overly restrictive.

Freshness requires that the applied policies are the latest versions of those policies. As a counterpart to agility, freshness also requires that policy updates made by a policy issuer are reflected in the repository's effective security policy within a reasonable time frame. Without freshness, a repository may have an outdated effective security policy that is overly permissive or overly restrictive.

Note, freshness may not be possible to achieve in all real-world cases. For example, if a party retrieves policies from multiple sources, there could be an update to a policy after it is retrieved, but before it is applied[2]. In this instance, freshness will be violated. The goal of SourceFabric is to maximize freshness within reasonable tolerances for a real-world system. We expect SourceFabric to meet this goal in the vast majority of cases.

Verifiability and Auditability. Verifiability requires that a participant is able to establish trust in the policy issuer that maintains a reusable policy. Without verifiability, an attacker can trivially pose as a legitimate policy issuer and impact the effective security policy of the repository to be overly permissive or restrictive.

Auditability requires that the effective security policy applied in a repository is open to scrutiny for any point in the repository's history. An attacker who is able to act as a policy issuer (*e.g.*, by compromising the actual issuers) can equivocate the reusable policy they make available (*i.e.*, temporarily issuing malicious policies that are not presented again later). Without auditability in the repository that reuses policies, it will not be possible to detect such equivocation.

4. SourceFabric Design

At a high-level, SourceFabric policy issuers make the policies they author publicly available in a Git repository associated with the publisher (§4.1). As SourceFabric enhances the gittuf security system, this reusable policy metadata is stored as an extension to the repository's local policy metadata ("Policy Issuers" in Figure 1). To enable reuse of these policies, SourceFabric provides two core mechanisms that are built into clients running on each repository administrators' and developers' machines.

First, for each trusted issuer, the administrator declares specific reusable gittuf policies (on behalf of the company)

2. This is akin to a Time-of-check Time-of-use (TOCTOU) situation.

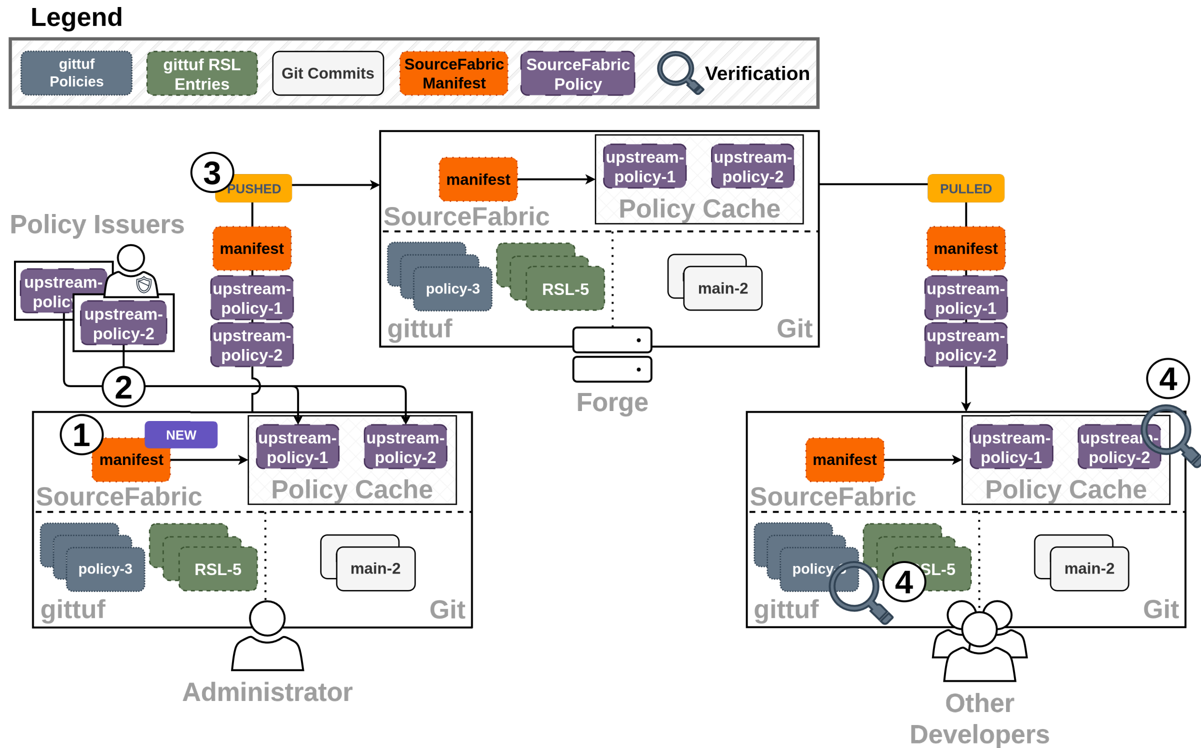


Figure 1: An overview of SourceFabric and how it interacts with Git and gittuf. In gittuf, the RSL is updated anytime when both gittuf policy changes and Git commits occur. In SourceFabric, a repository administrator first defines a SourceFabric manifest (①), which is used by SourceFabric to fetch the reused policies (②). The reused policies are validated, after which they are stored in the repository's policy cache, pushed to the forge (③), and retrieved by other developers. Each developer's gittuf client enforces the repository's local policy and all cached reused policies (④).

that apply to the repository using a *SourceFabric manifest* (§4.2, ① in Figure 1). Second, the administrator's SourceFabric client retrieves all reused policies and authenticates each policy issuer via their digital signature. The client then stores each validated reused policy in the repository's policy cache (§4.3, ② in Figure 1). SourceFabric enhances the gittuf policy verification workflow to enforce the cached reused policies in addition to the local policy.

The client pushes the cached policies to the forge hosting the canonical copy of the repository to make them available to all users (③ in Figure 1). Thereafter, each user's gittuf client is responsible for enforcing all of the repository's policies (§4.4, ④ in Figure 1). All SourceFabric clients operating on a repository check the policy issuers' repositories for updates. When an updated policy is found, the policy cache in the repository is updated accordingly.

Figure 2 shows an example of SourceFabric in action. Box 1 shows two reusable policies (`regulatory-policy` and `baseline-policy`) issued in distinct repositories. The administrators of the `fast` repository add a SourceFabric manifest to reuse the two policies (box 2). Box 3 of Figure 2 shows an instance of the `baseline-policy` being updated, which is retrieved and applied by a developer working on the repository (box 4).

4.1. Issuing Reusable Security Policies

Policy issuers author reusable gittuf policy as metadata files, which are stored in a repository they control. SourceFabric follows gittuf conventions, storing the reusable policy's metadata in a well-known custom Git reference in the repository. A SourceFabric client inspects this Git reference to identify the policies issued in the repository. The policy issuers configure a local policy (§3.2) using gittuf to protect the reusable policy.

To provide authenticity for reusable policies, SourceFabric policy issuers cryptographically sign the reusable policy's metadata files. The gittuf root of trust in a policy issuer's repository is used to manage the associated keys (including key rotation and revocation). When an issuer modifies a reusable policy, they add a new signature, and this action is recorded in the issuer repository's RSL. Thus, reusable policies are issued and updated in a single, canonical location, enabling consistent and compliant policy definition across repositories.

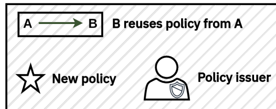
4.2. Declaring Security Policies to Reuse

Companies use a *SourceFabric manifest* to declare the policies that must be reused in a particular repository. This manifest is cryptographically signed by the repository's administrators and is stored alongside the local gittuf policy (§3.2) of each repository, meaning there is a one-time setup procedure per repository to setup SourceFabric.

The SourceFabric manifest contains *policy reuse entries*. Each entry contains the following information:

– **Location.** The URL of the policy issuer's Git repository where the canonical version of the reusable policy is located. We note that a standard Git client must be able to use the location to retrieve the repository.

Legend



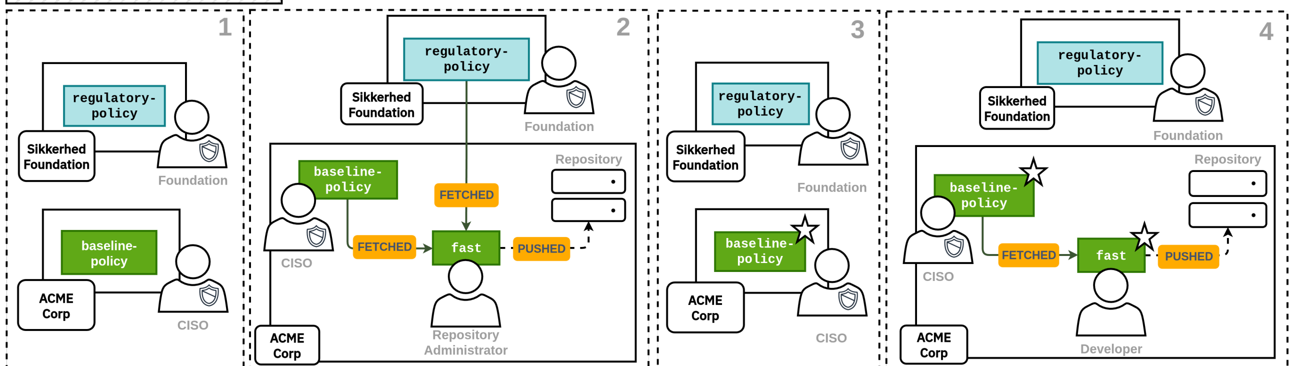


Figure 2: Two policy issuers create reusable policies in their respective repositories. The Sikkerhed Foundation issues `regulatory-policy` and the CISO division of ACME Corp issues the company's `baseline-policy` (box 1). The repository administrator for the `fast` repository reuses both policies using a SourceFabric manifest, and fetches the metadata for both reusable policies (box 2). The administrator then pushes the retrieved policy metadata to the canonical copy of the `fast` repository (also box 2). The CISO division issues an update to the company's baseline policy in the repository they control (box 3). A developer working on the `fast` repository fetches and applies this update, and pushes the change to the canonical copy of the repository (box 4).

– **Root of trust.** The set of public keys that represent trusted policy issuers, used to verify the cryptographic signatures on the reusable policy. The set of root keys must be determined by the repository administrators using either an out-of-band mechanism or via trust-on-first-use (TOFU).

– **Policy selectors.** The parts of a reusable gittuf policy that must be reused. A gittuf policy can contain different parts such as workflow control rules, access control rules, identity declaration, etc. For example, a repository may only reuse `workflow-controls` declared in one reused gittuf policy and the `trusted-identities` part declared in another reused policy. Only the selected parts of the reused policy are enforced in the repository.

Listing 1: The SourceFabric manifest for ACME Corp's `fast` repository. The repository reuses the policy of ACME Corp's `baseline-policy` issued by the company's CISO division, and `fast` reuses a `regulatory-policy` provided by the external Sikkerhed Foundation.

```
- location: https://acme.corp/ciso/baseline-policy
  rootOfTrust:
      - publicKey: ciso.pub
      - publicKey: soc-lead.pub
      - publicKey: integrity-lead.pub
  selectors:
      - workflow-controls
- location: https://sikkerhed.fdn/regulatory-policy
  rootOfTrust:
      - publicKey: teresa.pub
  selectors:
      - workflow-controls
```

Listing 1 depicts an example manifest for company ACME Corp's SourceFabric manifest for the `fast` repository depicted in Figure 2. This manifest has two policy reuse entries, one for the company-issued policy stored in the `baseline-policy` repository, and the other for the Sikkerhed Foundation's `regulatory-policy` repository. In both cases, the repository selects the `workflow-controls` aspect of the reusable policy.

4.3. Caching Reused Security Policies

SourceFabric has a secure "policy caching" mechanism to retrieve and cache all reused policies in the repository. Any parties contributing to a repository, like the administrators and developers, invoke the policy caching mechanism when they interact with the repository to ensure the repository has all intended policies applied, and that each reused policy is not stale beyond an administrator configured bound. Since different users perform this action, the set of policies quickly converges to the latest version. And, a bot can be used to automatically apply policy updates periodically.

The policy caching mechanism is said to be successful when the client is able to resolve all intended reusable policies, authenticate each policy issuer, and store the reused policies in the repository's policy cache. And, as the reused policies are cached in the repository, SourceFabric ensures that a recent version of each reused policy is enforced, even if a policy issuer's repository is unreachable (*e.g.*, due to network outages or malicious actors).

4.3.1. Resolving the Set of Reused Policies. SourceFabric ensures that the transitive set of all intended reusable policies are retrieved and applied to the repository. Starting with the policy reuse entries in the manifest, the SourceFabric client resolves and downloads any outdated or missing policies and (if available) the issuer's SourceFabric manifest from the location in each entry. The client then examines each issuer's SourceFabric manifest for additional policy reuse entries that apply to the reused policies, and recursively downloads any outdated or missing versions of these indirectly reused policies and manifests. In doing so, SourceFabric prunes those transitively reused

policies that do not match the policy selectors specified in the policy reuse entry.

4.3.2. Storing Resolved Policies. SourceFabric stores a copy of each reused policy in the repository, similar to the practice of "vendoring" [53] software dependencies. Thus, SourceFabric ensures that every reused policy can be enforced in the repository even if the policy issuers' repositories become unavailable, such as due to an outage or a denial of service attack.

Each reused policy is stored in the policy cache, a unique location alongside the repository's local policy. When recording this change in the repository's gittuf RSL, the client also records the ID of the RSL entry in the issuer's repository for the custom Git reference from which the reused policy was retrieved. This is used to mitigate attempts by the policy issuer to equivocate the specific policy they made available for reuse.

4.3.3. Verifying Integrity of Reused Policies. For each issuer repository the client interacts with, it identifies the latest set of policy metadata as well as the SourceFabric manifest using the upstream repository's RSL. The RSL is the source of truth for what is current, and prevents rollback attacks and "metadata manipulation attacks" [47] targeting the issuer repository's contents. The client verifies the signatures of each retrieved reusable policy and the issuer's SourceFabric manifest using the root of trust information in the policy reuse entry. Any failure in verification aborts the policy caching mechanism with an error, ensuring only verified reusable policies from trusted issuers are applied to the repository.

The client also checks that the issuer repository's RSL has not been tampered with, another indication of a metadata manipulation attack or an attempt to present different reusable policies at different times (*i.e.*, an equivocation attack). The client checks that the latest RSL entry it is presented with is a descendant of the RSL entry ID recorded previously.[3]

Listing 2: The SourceFabric manifest for ACME Corp's `fast` repository after it is updated by the repository's administrators. Now, `fast` also reuses policy from `go-guidelines`.

```
- location: https://acme.corp/ciso/baseline-policy
  rootOfTrust:
      - publicKey: ciso.pub
      - publicKey: soc-lead.pub
      - publicKey: integrity-lead.pub
  selectors:
      - workflow-controls
- location: https://sikkerhed.fdn/regulatory-policy
  rootOfTrust:
      - publicKey: teresa.pub
  selectors:
      - workflow-controls
- location: https://acme.corp/devx/go-guidelines
  rootOfTrust:
      - publicKey: devx-lead.pub
      - publicKey: devx-go-lead.pub
  selectors:
      - workflow-controls
```

3. This check is omitted during the very first invocation of the policy caching mechanism as there is no previously recorded RSL entry ID to compare the entry presented by the upstream repository against.

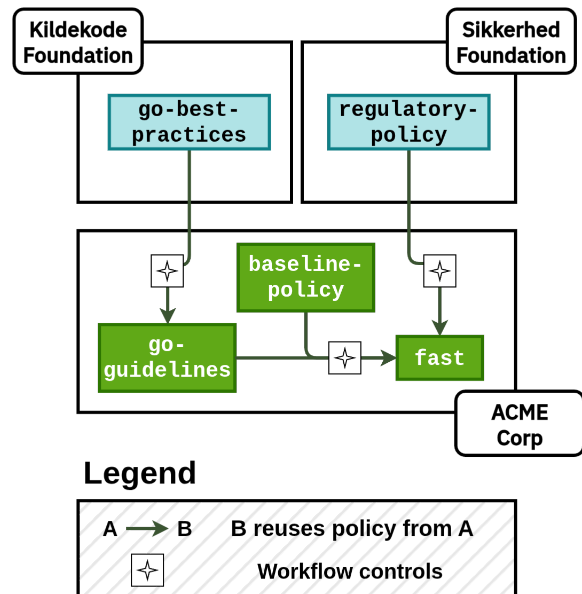


Figure 3: An illustration of ACME Corp's `fast` repository which explicitly reuses three policies: `baseline-policy`, `regulatory-policy`, and `go-guidelines`. The `go-guidelines` policy reuses `go-best-practices` maintained by the external Kildekode Foundation. Thus, `fast` transitively reuses the `go-best-practices` policy.

4.3.4. Policy Caching in Action. Figure 3 shows ACME Corp's `fast` repository. The repository administrators explicitly reuse three policies:

- `baseline-policy`
- `regulatory-policy`
- `go-guidelines`

The SourceFabric manifest for the repository is shown in Listing 2. The `go-guidelines` policy itself reuses the policy of `go-best-practices` issued by the external Kildekode Foundation.

On invoking the caching mechanism in `fast`, the SourceFabric client determines the following set of issuer repositories from where reused policies must be retrieved:

- `baseline-policy`
- `regulatory-policy`
- `go-guidelines`
- `go-best-practices` (reused transitively, via `go-guidelines`)

4.4. Enforcing Reused Security Policies

At the end of the policy caching mechanism, the policy cache in the repository contains a copy of each reused policy as well as its local policy. The vanilla gittuf verification mechanism introduced by Yelgundhalli et al. [24] enforces only the repository's local policy. SourceFabric extends this mechanism to enforce the cached reused policies in addition to the local policy.

SourceFabric's updated verification mechanism serially enforces each policy stored in the repository, *i.e.*, the local policy as well as every reused policy. If a repository change violates any of the policies, the enforcement mechanism terminates and outputs a record of the offending change and the policy it violated. For example,

in the `fast` repository depicted in Figure 3, a change is considered valid only if it passes verification for the policy configured directly in `fast`, as well as the policies reused from `baseline-policy`, `regulatory-policy`, `go-guidelines`, and `go-best-practices`. Note, the enforcement order of these policies does not matter because a change is considered valid only if it passes all policies. So, if a change violates even one of the reused policies, say `go-guidelines`, then the verification mechanism does not complete successfully.

It is possible for two policies to conflict with each other, *i.e.*, a repository modification cannot satisfy both. In such cases, SourceFabric's support for policy selectors can be used to reuse two policies that conflict by selecting the aspects that do not conflict. Beyond that, SourceFabric does not address conflicts (*e.g.*, by prioritizing one reused policy over another), as policy conflicts indicate a mismatch in expectations about development practices in the repository between stakeholders (*i.e.*, administrators, security team, compliance, etc.). SourceFabric *surfaces such issues* for the company to reconcile and ensures there cannot be an unsafe resolution by requiring administrators to adjudicate these changes. Conflicts in security policies, while rare, can be complex to resolve and may require input from legal and security experts. Once the experts determine how to address the conflict, the modifications they make to the policies are applied by SourceFabric to repositories in the organization. In this manner, SourceFabric balances two practical concerns: it ensures that the composition of policies is safe and meets the organization's wishes, while also keeping the maintenance burden to a minimum as the organization only needs to resolve a conflict once (rather than once per repository).

5. System Analysis

5.1. System Security Analysis

First, we examine how SourceFabric meets the system requirements defined in Section 3.3. We consider the impact of malicious or misbehaving SourceFabric clients on SourceFabric's ability to meet these requirements, showing that SourceFabric's behavior degrades gracefully in the face of attacks.

5.1.1. Completeness and Soundness. SourceFabric's policy caching mechanism is responsible for identifying the set of reused policies to apply in a repository. By definition, the complete set of policies to be applied is the tree of policies resolved by the policy caching mechanism. Any policy not part of this tree is extraneous.

When all parties are honest, it is self-evident how this requirement is met. When there are malicious or malfunctioning clients invoking the policy caching mechanism, such a client can omit required policies (violating completeness), introduce extraneous policies (violating soundness), or both. In such a scenario, the effective policy for the repository is corrected to be complete and sound the next time an honest client invokes the policy caching mechanism. Such a client will notice that the applied set of policies is incomplete or unsound, fix the issue, and alert their user of the issue. The misbehaving or malicious client itself must then be handled out-of-band.

5.1.2. Agility and Freshness. SourceFabric enables policy issuers to make changes to the reusable policy they manage at will. The issuers do not have to take into account any other factors such as how many repositories reuse the policy or conflicts with other reusable policies. Thus, policy issuers can rapidly update their policies to provide protections against emerging threats and to meet evolving security requirements.

In any one repository, SourceFabric ensures that any reused policy applied is fresh. When the policy caching mechanism is invoked, it uses the RSL of the repository where the reusable policy is stored to identify the latest version. A "metadata manipulation attack" [47] cannot be used to serve stale policies. And, as all users in the repository invoke the policy caching mechanism when interacting with the repository, the reused policies cannot grow stale beyond an administrator configured bound. This property holds even in the face of a subset of clients becoming malicious or malfunctioning; even if such a client applies a stale version of a policy or falsely indicates a policy update is not available, the next honest client applies the update.

5.1.3. Verifiability and Auditability. Policy issuers are required to sign the policy metadata they issue. SourceFabric verifies these signatures using the root of trust information provided as part of the reuse entry in the SourceFabric manifest. This ensures that any reused policy is only applied when the SourceFabric client verifies that the policy was issued by the trusted party.

SourceFabric stores a copy of the reused policy in the repository where it is applied, along with the issuer repository's RSL entry information corresponding to that version of the policy. The downstream repository's RSL is also updated when an updated version of a reused policy is stored. These are key to ensuring that SourceFabric enables auditability of the set of reused policies applied in the repository at any point in its development history. And, by storing the upstream repository's RSL entry information, SourceFabric ensures that any misbehavior by the issuer (*e.g.*, equivocation attacks where a malicious policy is issued temporarily) is caught during inspection of the downstream repository's policy history.

5.2. Considering the Effect of Delay and Outages

Next, we examine the impact of delays or outages in the network on SourceFabric. Crucially, as each repository stores a copy of the reused policies from the last successful invocation of the policy caching mechanism, SourceFabric is not prevented altogether from applying security policies in the repository. That said, any network delay or outage prevents a SourceFabric client's ability to successfully execute the caching mechanism, violating the *agility and freshness* requirement. This means that a client could use outdated policies for a short time window.

The impact of using an outdated policy will vary depending on how the outdated policy differs from the latest policy. The outdated policy may fail to reuse policies that are now supposed to be applied or reuse policies that should no longer be applied (violating *completeness and soundness*). In turn, this means that a change that should be blocked is allowed, or a change that should

be permitted is blocked. In fact, as a policy change may impact different developer actions in different ways, it is possible that multiple of these behaviors occur at the same time for a single outdated policy.

As described in Section 4.3, every developer working on the repository attempts to check for and apply policy updates using their SourceFabric client. And so, the amount of time the outdated policy is used has an upper bound based on the frequency of developers contributing to the repository. The issue will be naturally remediated after any SourceFabric client retrieves and provides the updated policy file to the repository. Finally, SourceFabric prevents rollback attacks and attempts to present different versions of the policy at different times (§4.3.3). This helps SourceFabric recover gracefully in the face of network issues or network based attackers.

In summary, network delays or outages degrade the *completeness and soundness* requirement as well as the *agility and freshness* requirement of SourceFabric. The *verifiability and auditability* requirement is not affected, and the last set of policies successfully applied by SourceFabric will continue to be enforced.

5.3. Participant Capabilities and Impacts

Finally, we analyze how an attacker can use SourceFabric to influence the repository's effective security policy, summarizing our findings in Table 1. The attacker's ability to circumvent SourceFabric depends on the participant role they act under, which in turn determines their access to the policies. As such, we consider the damage that repository administrators, policy issuers, network attackers, developers and outsiders, can cause. An attacker with access to multiple roles, can cause the damage of the most privileged participant. In order to understand the impact of an attack, we consider a policy's effect on repository changes. That is, since a policy either allows or denies changes, an attacker can aim to either allow changes that should have been blocked or block changes that should have been allowed. We now discuss individual roles and their attack capabilities in more detail.

Repository administrators. A repository's administrators can arbitrarily tamper with the repository's cached security policies. As administrators effectively serve as the root of trust for the repository's policy configuration, they can arbitrarily change the repository's local policy.

Note that this is no different from the capabilities of repository administrators without SourceFabric, *i.e.*, with only gittuf. They can similarly impact the security posture of the repository by arbitrarily modifying the repository's local policy. Indeed, this is the case with any repository security system, whether gittuf or one offered by a forge such as GitHub, GitLab, and Bitbucket.

Policy issuers. As we mention in Section 3, we assume the repository administrators or the company have out-of-band reasons to trust the policy issuers. This may be because the policy issuer is a stakeholder within the company (*e.g.*, the security team) or based on reputational or contractual reasons (*e.g.*, regulators, security foundations). But, for completeness, we analyze the impact of a malicious issuer.

A malicious policy issuer can weaken their own policy or omit a reused policy to weaken the repository's security posture. However, this does not enable a malicious policy issuer to allow all developer proposed changes through since other policy issuers will still have their policies applied. This also includes any rules or restrictions created by the repository's local policy.

On the other hand, the policy issuer can arbitrarily block changes that should have been allowed by overly strengthening their policy or reusing conflicting policies. SourceFabric does not prevent these actions. However, such actions are detected by the developers the next time an action that should be allowed is blocked, and SourceFabric enables developers to reason about the policy changes the blocked the action.

The policy issuer may also create an infinite policy chain by generating reuse entries and issuer repositories unendingly that the client must resolve. This essentially results in a denial-of-service attack which will become evident in the SourceFabric client when the policy caching mechanism times out, prompting an investigation.

Network attackers. A network attacker is any party which can globally and arbitrarily control (*e.g.*, block, read, etc.) network traffic. Note that since SourceFabric policies are locally cached in the repository, a network attacker cannot tamper with the policies already retrieved and stored in the repository. However, a network attacker can prevent a policy from being updated by blocking retrievals of the updated policy from the policy issuer's repository. If the update strengthens a policy, the attacker can prevent the update from reaching the repository, thus allowing changes to occur that should have been blocked. If the update weakens a policy, the attacker can block changes that are now supposed to be allowed.

If the attacker makes a policy issuer's repository entirely unavailable, this is detected by any SourceFabric client invoking the caching mechanism. On the other hand, the attacker may perform a freeze attack [54] to prevent a client from applying an update available in the policy issuer's repository by hiding the existence of the update. SourceFabric does not prevent this attack.[4] But, this attack is not trivial. The attacker must present precisely the same version of the policy that has already been retrieved during a prior execution of the caching mechanism. If the attacker serves an older version, the SourceFabric client will detect the issue as it has a newer version of the policy than expected. Additionally, the SourceFabric client does not advertise the version of the policy that was previously retrieved, meaning the attacker must be able to independently determine the policy version to serve, such as by fetching the repository from the canonical copy and inspecting its contents, *i.e.*, the specific version of the reusable policy currently stored in the repository.

The network attacker has no way to serve arbitrary policies. This is because a network attacker does not possess the policy issuers' keys and thus cannot create a legitimately signed policy.

Developers and outsiders. Outsiders have no ability to impact the repository's security posture. While it would seem that a developer that is not permitted to change the repository's policy may have more ability to tamper with

4. A freeze attack can be performed for any contents of a Git repository and is not specific to reusable policy. We do not address this in SourceFabric as a general-purpose solution is necessary in the repository security system, *i.e.*, gittuf.

TABLE 1: An analysis of whether SourceFabric participants can impact a repository's security policy to either allow changes that should not be allowed or block changes that should be allowed.

	Allow too much	Block too much
Repository Administrator	Not Prevented	Not Prevented
Policy Issuer	Limited in Scope	Not Prevented
Network Attacker	Limited in Scope	Limited in Scope
Developer or Outsider	Prevented	Prevented

policies, this is not the case. The reason is that SourceFabric uses gittuf to apply access control permissions to prevent two major concerns here.

First, a developer must be prevented from tampering with the repository's manifest, either by adding or removing policies. This is prevented by gittuf which only allows repository administrators to change the repository's manifest. If a developer pushes a change here, other clients will observe a policy violation when they pull the copy down and will automatically revert it and push a fixed version back up to the repository. Unauthorized changes to the local policy are prevented in a similar manner.

Second, a developer must be prevented from tampering with the stored cache of policies from other policy issuers. Note that in this case, the developer does have access to write to these policies. However, SourceFabric ensures that changes to these policies are correctly signed by the policy issuer and that all versions of the policies include non-decreasing version numbers. Thus, a malicious developer can do no more than fail to update a policy (which would be performed by the next honest developer to check that issuer). Thus, a developer cannot tamper with the policies in any meaningful way.

6. Implementation and Performance

Our implementation of SourceFabric is an extension of the open source gittuf project, which is at the incubating stage at the OpenSSF [39]. This requires (among other things) diversity in the set of maintainer affiliations, open source governance policies, and numerous adoptions. We also present our findings from evaluating the storage and runtime overhead imposed by SourceFabric in comparison with using only a local policy.

Implementation. We added SourceFabric to gittuf's Go implementation, with approximately 1,400 lines of code changed (excluding tests). gittuf stores policy and a reference state log (RSL) that tracks repository activity as additional metadata in custom Git references in the repository. We implemented the SourceFabric manifest as a part of the existing gittuf root of trust metadata and added support to the RSL to track upstream repository revisions. Additionally, we modified gittuf's synchronization workflows to include the SourceFabric policy caching mechanism. Finally, we extended gittuf's verification workflow to enforce all policy metadata, including those retrieved from upstream repositories.

Our changes to gittuf have been adopted by the project's maintainers. Our implementation can be found online at https://github.com/gittuf/gittuf and this implementation of SourceFabric is currently being piloted by Bloomberg.

Evaluation setup. Our experiments were conducted on a system with an Intel Xeon Gold 5416S CPU and 128 GB of RAM, running Fedora 42. We used Git v2.43.0 and gittuf v0.11.0, which includes our implementation of SourceFabric. To analyze how SourceFabric performs in the smallest to even the largest repositories, we chose four repositories of varying sizes. For the smallest repository, we chose the gittuf repository, with 1,864 commits (taking 26.340 MiB) at version v0.11.0 [55]. Our medium-sized repository was the curl library, with 35,238 commits (taking 132.635 MiB) at version v8.14.1 [56], and our large-sized repository, the Git repository in which Git itself is developed, containing 77,303 commits (taking 324.243 MiB) at version v2.50 [57]. The largest repository is the Kubernetes repository with 129,464 commits (taking 1,418.88 MiB) at version v1.33.3 [58].

Throughout, we use two copies of each repository: one which only uses gittuf, and the other which relies on SourceFabric to reuse gittuf policies. In both repositories, we applied varying numbers of policies to protect the `master` branch of the repository. In the gittuf-only repository, these policies were specified directly as one policy containing multiple rules. In the repository with SourceFabric, we created multiple upstream repositories that policies were reused from, one repository per policy to reuse. For experiments that involve measuring the time of SourceFabric operations, we ran each experiment ten times, and report the mean over the ten runs. Experiments that involved obtaining storage overheads were performed by taking one measurement of the storage overhead imposed by SourceFabric.

Storage overhead. To quantify the storage overhead introduced by SourceFabric, we measured the additional disk space required in the repository when individual policies are reused via SourceFabric, instead of batched into a single local gittuf policy. We analyzed three variations of this setup to account for varying numbers of rules and reused policies. Table 2 contains our findings.

In each case, we either directly configured the corresponding number of rules in the local gittuf policy or declared the same number of reusable policies in the SourceFabric manifest. We find that the storage overhead increases linearly with the number of reused policies. However, the overall storage occupied by the metadata remains negligible compared to each repository's total size, as the contents tracked in each repository dominate the overall storage usage. In the smallest repository we evaluated, the contents alone make up 26.3 MiB. With the addition of the metadata of 10 reused policies via SourceFabric, *i.e.*, the maximum tested configuration, the total size is 26.4 MiB, representing an increase of **0.15%**.

Runtime overhead. SourceFabric impacts runtime by introducing cost to retrieve and validate policies, manipulating the policy cache, and the time needed to enforce the policies during verification. Table 3 shows the runtime overhead of SourceFabric in comparison with gittuf which does not have any of these costs.

There are three main scenarios in which overhead occurs. First, when a manifest is newly added and the policy cache is completely empty. This is the "Cache Cold Start" case, where all information must be retrieved from the policy issuer repositories, and then populated into the cache. This is typically only done once, most likely by

TABLE 2: The storage overhead imposed by the addition of reused policies, across the four repositories.

Number of Reused Policies	gittuf-only	SourceFabric-enabled
gittuf		
1	3,370 B	7,136 B
5	3,886 B	22,464 B
10	4,526 B	41,646 B
curl		
1	3,366 B	7,124 B
5	3,886 B	22,460 B
10	4,534 B	41,650 B
Git		
1	3,366 B	7,124 B
5	3,874 B	22,468 B
10	4,530 B	41,650 B
k8s		
1	3,366 B	7,124 B
5	3,874 B	22,460 B
10	4,530 B	41,642 B

TABLE 3: The overhead imposed by reusing policies and various SourceFabric policy cache states instead of a single local gittuf policy during verification.

Reused Policies	gittuf Time	SourceFabric Verification Time		
		Cache Cold Start	Pre-Populated Cache	Cache Warm Start
gittuf				
1	0.121s	0.889s	0.197s	0.111s
5	0.118s	2.745s	0.757s	0.120s
10	0.122s	5.733s	1.541s	0.127s
curl				
1	0.123s	0.883s	0.183s	0.124s
5	0.112s	2.795s	0.577s	0.117s
10	0.119s	4.773s	1.156s	0.122s
Git				
1	0.121s	0.894s	0.205s	0.131s
5	0.121s	2.657s	0.695s	0.125s
10	0.122s	5.046s	1.397s	0.132s
k8s				
1	0.116s	0.867s	0.188s	0.123s
5	0.112s	2.590s	0.554s	0.120s
10	0.108s	5.388s	1.652s	0.112s

the repository administrator, when they first set up the repository with a manifest. This cost is quite high when compared to gittuf, due to all of the data that must be downloaded, and can result in more than 5 seconds of overhead. However, since this is only during the initial setup, it is not a significant cost in practice.

The second scenario is when a developer first retrieves a copy of the cache, which we call the "Pre-Populated Cache" case. In this case, the cache is already populated with the policies that are in the manifest, but the developer's SourceFabric client has not locally verified the policies. In order to protect against attacks where a malicious developer tries to poison the cache, the SourceFabric client will reverify all of the policies in the cache. This causes a small amount of overhead, but is not overly noticeable given the other overhead of doing git operations. This is also quite rare in practice, with the worst case usually only occurring when a new SourceFabric client is installed.

The third scenario is when the cache is already populated, and the client has already verified the policies in the cache. In this case, the SourceFabric client only needs to enforce the policies, which has minimal overhead. This is the "Cache Warm Start" case, and is the most common case where the client has validated all policies in the cache and none of them need to be checked for expiry. This case incurs a delay of a few milliseconds, which is negligible compared to the time to run a git command.

There is one final scenario that can occur, which is when some of the cached policy information has been prevalidated and some is not. This can occur when another developer has updated the cache with a new policy file, when the local client's cache of information has an expired policy, when the manifest has been updated, or any combination of these. In this case, the cost will be a combination of the above three scenarios. If the manifest is completely changed and all policies are new, then the cost will be similar to the "Cache Cold Start" scenario. If the manifest is unchanged, but some policies have been updated, then the cost will be similar to the "Pre-Populated Cache" case, with the number of policies that are new impacting the cost. However, these costs are only applied for the policies that are new or updated, and so the overall cost is still quite low in practice.

7. Discussion and Limitations

Can SourceFabric manage the policy formats from existing forges and other repository security systems? Despite gittuf being forge agnostic, SourceFabric would need a few changes to support the distribution of policies from forges like GitHub, GitLab, etc. Some forges implement REST APIs that can read and write policies, but these APIs are not yet fully adequate for SourceFabric since their scope is limited to individual repositories [59] or only available in certain product tiers [60]. In general, forges have not opted for interoperability between their security systems, which explains this shortcoming. Hence, SourceFabric's choice of gittuf as the underlying security system is a pragmatic one, as gittuf can be used across forges. Note also that SourceFabric can still be used on repositories stored on a forge; an issuer's repository can be hosted on any system without any restrictions.

How does a company ensure all repositories reuse the company's policies? SourceFabric simplifies policy management and helps prevent policy misconfigurations and staleness. However, a repository's security posture is dependent on its administrators *reusing the right policies*. While we assume that repository administrators faithfully adhere to the company's security requirements, the administrators might unintentionally omit a policy they must reuse, leaving their repository exposed. But, SourceFabric applies policy transparency [61] to the manifest metadata via the repository's gittuf RSL. This allows any stakeholder (*e.g.*, the company's compliance team) to monitor the manifest to ensure the repository reuses the expected set of policies.

How does SourceFabric handle policy conflicts? SourceFabric policies are applied one after another, effectively creating a series of filters that are applied consecutively. An action must pass through all filters to be allowed. This means that two policies may conflict, such that an action cannot meet both policies. So, when a valid action is blocked, then repository administrators or the appropriate policy issuers must resolve the conflict. This is because each reused policy is chosen by these parties to represent the company's intent. So, policy conflicts

indicate a mismatch in expectations that must be resolved out-of-band. These types of mismatches do happen in the real world, and the remediation is typically exactly what we prescribe—manual clarification by the parties involved. Most often, the security team at the company may be asked to decide which policy to prioritize or to identify alternative policies that do not conflict.

Can you retrieve upstream policies from intermediate repositories? One limitation of SourceFabric is that while SourceFabric applies policy reuse transitively, clients currently always retrieve content from the upstream repository. But, for a variety of reasons, an upstream repository may not be *available* to the repository. For example, if `foo` reuses policy maintained in `bar`, and `bar` reuses policy from `baz`, `foo` must also reuse policy from `baz`. However, `baz` may only be available to `bar` and not `foo`. To address this, we plan to explore using the copy of the transitively reused repository's policy metadata from the nearest *available* reused repository, `bar` in this example.

Can SourceFabric be used to manage arbitrary vendored repository contents? As noted in Section 4, SourceFabric stores a copy of each reused policy in the repository, similar to the practice of "vendoring". Developers frequently have to store in their repository the contents of another repository, which raises the question of whether SourceFabric can be employed for contents that are not security policies. While Git supports vendoring an upstream repository's contents natively with the submodules [62] feature, SourceFabric's approach to vendoring contents has a number of advantages. First, SourceFabric supports automatic updates for vendored contents. A Git submodule, though, must be manually updated by a developer. Second, SourceFabric leverages gittuf's RSL to identify the revision to use from the upstream repository, which prevents "metadata manipulation attacks" [47]. On the other hand, Git retains the history of the revision tracked as a submodule, which is omitted by SourceFabric. So, while SourceFabric's approach to vendoring shows promise, a more thorough examination is necessary to determine SourceFabric's use as a general purpose solution to securely vendoring in a repository another repository's contents. We leave this to future work.

8. Related Work

Source control security. Prior work in the security of source control systems describe novel attacks that can target source code development and defenses. Chen et al. [63] studied how to secure repositories in delta-encoding based version control systems hosted on untrusted servers. Vaidya et al. [64] studied how centralized version control systems can be extended to add support for cryptographic signatures, required for verifying authenticity of changes in a repository. Wheeler [65] describes security requirements that must be considered in the design of version control systems.

Git security. Given Git's prominence, a number of systems have been proposed to secure repositories. Ranging from adding authentication to enforcing authorization for repository contents, these solutions all operate over a single repository and do not address the problems of policy management at the scale of a large company. Courtès [66] presents a system adopted by the Guix package manager to authenticate all commits made to the repository. Every commit is expected to be signed by a key declared in an authorization file available its parent commit's Git tree. Torres-Arias et al. [47] present Git's weaknesses against "metadata manipulation attacks" and describes an append-only "reference state log" that tracks repository activity as a solution to these attacks.

Other researchers have presented Git repository security systems. Xu et al. [67] propose Gringotts, a repository security system that prevents unauthorized directory reads and writes using a shadow encrypted repository. Yelgundhalli et al. [24] present gittuf, a system that decentralizes Git repository policy enforcement by making repository policy independently verifiable. Xu et al. [68] present Disac, which provides access control for a repository in a fully decentralized manner by leveraging attribute-based encryption and signatures, with policies stored using Ethereum smart contracts. Like before, all of these systems operate within the confines of a single Git repository.

Policy reuse. At its core, SourceFabric implements the notion of policy reuse, where a security policy is defined in a single location and inherited at various other points. While SourceFabric is the first system to explore policy reuse for securing source code at the scale of a large company, the concept has been studied in a variety of other contexts. Bonatti et al. [69] present how access control policies that are specified independently can be combined. Decat et al. [70] highlight the value of modularization and reuse for reducing the management complexity of security rules, and present STAPL, a policy language that adds templating over the standardized XACML language used to specify access control policies. Schreuders et al. [71] demonstrate the value of functionality based access control (FBAC), where end users assign policies crafted by experts to different applications. SourceFabric applies a similar pattern, where a repository's owners can choose which policies to reuse without crafting those policies themselves. This was shown by Schreuders et al. [72] to be a practical solution to enabling end users to protect themselves. The authors studied the practicality of a Linux Security Module implementation of FBAC for constraining applications against SELinux and AppArmor.

Software supply chain security. A sharp increase in attacks in recent years has led to the development of a number of software supply chain security standards [28], [73]. A number of researchers have studied this space as well. Okafor et al. [74] emphasize the fundamental patterns of software supply chain security. Numerous systems [54], [75]–[80] have all been proposed for improving the transparency and security of software development and distribution.

SourceFabric is designed as a way to strengthen company-wide security postures to prevent software supply chain attacks that target the source code development process, but it is only one part of securing the software supply chain. SourceFabric builds on gittuf [24], which itself integrates with the aforementioned supply chain systems through its use of in-toto attestations [75], [81]. Further, while existing systems include semantics for defining software supply chain policies, they do not consider the practicalities of policy management in a distributed setting such as source code development at large companies. The fundamental concept of policy composition and reuse for

software supply chain security is, therefore, complementary to these systems, and would likely be a valuable extension.

9. Conclusion

In this paper, we introduced SourceFabric, a system for applying security policies to thousands of Git repositories consistently and robustly. SourceFabric enables *policy reuse* across repositories, mitigating the threat of misconfigured, inconsistent, and stale policies across these repositories. In doing so, SourceFabric removes the burden placed on repository administrators to create and update policies, providing a way for each repository to have a consistent, up-to-date, and compliant security posture. We show SourceFabric meets key security requirements of completeness and soundness, agility and freshness, and verifiability and auditability.

Our implementation of SourceFabric in the open source gittuf repository security system was developed in collaboration with industry stakeholders, and is now part of a pilot at Bloomberg. It introduces minimal storage overhead (an increase of 0.15% in the worst case scenario we tested) and adds similarly acceptable runtime overhead to gittuf's verification workflow.

Acknowledgments

We would like to thank the reviewers for their feedback, as well as Billy Lynch, Santiago Torres-Arias, and Yongjae Chung for their help in making SourceFabric and this paper a reality.

This material is based upon work supported by the United States Department of Education (ED) under grant No. P200A210096. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the ED.

References

[1] Palantir, "How Palantir Secures Source Control," https://blog.palantir.com/how-palantir-secures-source-control-105c49079eae.

[2] M. McGarr and D. Marsh, "Towards true continuous integration: distributed repositories and dependencies," https://netflixtechblog.com/towards-true-continuous-integration-distributed-repositories-and-dependencies-2a2e3108c051.

[3] N. Brousse, "The issue of monorepo and polyrepo in large enterprises," in *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*, ser. Programming '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3328433.3328435

[4] "Intel," https://github.com/intel.

[5] "Microsoft," https://github.com/microsoft.

[6] "Google," https://github.com/google.

[7] "International business machines," https://github.com/ibm.

[8] "Datadog inc." https://github.com/datadog.

[9] "Adobe, inc." https://github.com/adobe.

[10] D. Slater, "Powering Innovation and Speed with Amazon's Two-Pizza Teams," https://aws.amazon.com/executive-insights/content/amazon-two-pizza-team/.

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

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

[13] 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.

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

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

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

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

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

[19] 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

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

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

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

[23] "gittuf," https://gittuf.dev.

[24] A. S. A. Yelgundhalli, P. Zielinski, R. Curtmola, and J. Cappos, "Rethinking Trust in Forge-Based Git Security," in *Network and Distributed System Security (NDSS) Symposium*, 2025.

[25] "PCI Data Security Standard (DSS)," https://www.pcisecuritystandards.org/standards/pci-dss/.

[26] "EU Cyber Resilience Act," https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32024R2847, 2024.

[27] 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

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

[29] Open Source Security Foundation (OpenSSF) Best Practices Working Group, "Source Code Management Platform Configuration Best Practices," https://best.openssf.org/SCM-BestPractices/, 2023.

[30] GitHub, "About organizations," https://docs.github.com/en/organizations/collaborating-with-groups-in-organizations/about-organizations.

[31] GitLab, "Groups," https://docs.gitlab.com/user/group/.

[32] Bitbucket, "What is a workspace?" https://support.atlassian.com/bitbucket-cloud/docs/what-is-a-workspace/.

[33] C. Dietrich, K. Krombholz, K. Borgolte, and T. Fiebig, "Investigating system operators' perspective on security misconfigurations," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1272–1289.

[34] C. C. Wood and W. W. Banks, "Human error: an overlooked but significant information security problem," *Computers & Security*, vol. 12, no. 1, pp. 51–60, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/016740489390012T

[35] R. May, C. Biermann, J. Krüger, and T. Leich, "Asking security practitioners: Did you find the vulnerable (mis) configuration?" in *Proceedings of the 19th International Working Conference on Variability Modelling of Software-Intensive Systems*, 2025, pp. 30–39.

[36] "A05:2021 – Security Misconfiguration," https://owasp.org/Top10/A05_2021-Security_Misconfiguration/, 2021.

[37] "Equifax to Pay $575 Million as Part of Settlement with FTC, CFPB, and States Related to 2017 Data Breach," https://www.ftc.gov/news-events/news/press-releases/2019/07/equifax-pay-575-million-part-settlement-ftc-cfpb-states-related-2017-data-breach.

[38] "Settlement reached with Target following major consumer data breach," https://www.attorneygeneral.gov/taking-action/settlement-reached-with-target-following-major-consumer-data-breach/.

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

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

[41] 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.

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

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

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

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

[46] OpenSSF, "Best Practices Badge Program," https://www.bestpractices.dev/en.

[47] 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

[48] OpenSSF, "Open Source Project Security Baseline," https://baseline.openssf.org/versions/2026-02-19.

[49] "National Institute of Standards and Technology (NIST)," https://www.nist.gov/.

[50] "Internet Engineering Task Force (IETF)," https://www.ietf.org/.

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

[52] "Open Worldwide Application Security Project (OWASP)," https://www.owasp.org/.

[53] C. Gross, "Vendoring," https://htmx.org/essays/vendoring/, 2025.

[54] 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

[55] "gittuf source code - v0.11.0," https://github.com/gittuf/gittuf/tree/v0.11.0.

[56] "curl source code - v8.14.1," https://github.com/curl/curl/tree/curl-8_14_1.

[57] "Git Source Code Mirror - v2.50.0," https://github.com/git/git/tree/v2.50.0.

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

[59] GitHub, "Rest api endpoints for rules," https://docs.github.com/en/rest/repos/rules?apiVersion=2022-11-28.

[60] GitLab, "Project push rules api," https://docs.gitlab.com/api/project_push_rules/.

[61] 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

[62] "Git tools - submodules," https://git-scm.com/book/en/v2/Git-Tools-Submodules.

[63] B. Chen and R. Curtmola, "Auditable version control systems." in *Network and Distributed System Security (NDSS) Symposium*, 2014.

[64] 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.

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

[66] 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/

[67] 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.

[68] X. Xu, Z. Yang, Q. Cai, J. Lin, L. Ren, B. Chen, and Y. Huang, "Enforcing cryptographic distributed-vcs access control with no trust on servers," *Journal of Information Security and Applications*, vol. 93, p. 104103, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212625001401

[69] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati, "An algebra for composing access control policies," *ACM Trans. Inf. Syst. Secur.*, vol. 5, no. 1, p. 1–35, Feb. 2002. [Online]. Available: https://doi-org.proxy.library.nyu.edu/10.1145/504909.504910

[70] M. Decat, J. Moeys, B. Lagaisse, and W. Joosen, "Improving reuse of attribute-based access control policies using policy templates," in *Engineering Secure Software and Systems*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:17650753

[71] Z. C. Schreuders and C. Payne, "Reusability of functionality-based application confinement policy abstractions," in *Information and Communications Security*, L. Chen, M. D. Ryan, and G. Wang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 206–221.

[72] Z. C. Schreuders, T. McGill, and C. Payne, "Empowering end users to confine their own applications: The results of a usability study comparing selinux, apparmor, and fbac-lsm," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 2, Sep. 2011. [Online]. Available: https://doi-org.proxy.library.nyu.edu/10.1145/2019599.2019604

[73] "Supply Chain Integrity, Transparency, and Trust (SCITT)," https://datatracker.ietf.org/wg/scitt/about/.

[74] 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

[75] 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

[76] 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

[77] 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

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

[79] 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

[80] 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.

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