VERIGOV: A CRYPTO-ECONOMIC PROTOCOL FOR DECENTRALIZED SOFTWARE SUPPLY CHAIN SECURITY

Nikhil Ranjan
nikhil@polkassembly.io

ABSTRACT

The proliferation of open-source software has created a complex and vulnerable software supply chain, where centralized trust in repositories and developers has become a significant liability. Recent attacks demonstrate that current security measures, such as checksums and basic code signing, are insufficient to prevent the distribution of malicious or tampered software. These mechanisms fail to provide two critical guarantees: *provenance integrity* (a verifiable link from source code to binary) and *release authorization integrity* (assurance that a release was approved by a legitimate, collective consensus). This paper introduces VeriGov, a crypto-economic protocol that addresses these gaps by establishing a Decentralized Autonomous Organization (DAO) to govern the software release lifecycle. VeriGov leverages a public, permissionless blockchain to create an immutable, auditable ledger for release management. Inspired by Polkadot's on-chain governance, it establishes a multi-stakeholder system where release approvals are subject to a configurable m-of-n vote, weighted by staked holdings of the native utility token, `VGOV`. Furthermore, VeriGov incorporates cryptographic attestations from reproducible build processes to create a non-repudiable link between a software binary and its corresponding source code. This paper details the protocol's architecture, its DAO-based governance for release and stakeholder management, and its tokenomic model, which uses staking and slashing mechanisms—including separately-adjudicated negligence and malice slashing—to create economic incentives for security and trust.

1 Introduction

1.1 Open Source as Critical Infrastructure

The modern digital infrastructure is built upon a foundation of open-source software (OSS). The collaborative and transparent nature of OSS development has accelerated innovation at an unprecedented scale, but the same openness has also turned the software supply chain into one of the most attractive attack surfaces in computing. Modern applications are not monolithic entities but are composed of vast and intricate dependency trees, often pulling in hundreds or even thousands of third-party components. [1] A single vulnerability or malicious component introduced anywhere in this complex web can propagate downstream, creating a cascading failure with devastating consequences. [2] The widespread deployment of this software on millions of devices, coupled with the prevalence of automatic update mechanisms, transforms this interconnectedness into a vast and attractive attack surface for malicious actors. [3]

1.2 The Centralized Trust Crisis

Despite the decentralized ethos of the open-source community, the infrastructure that supports it remains critically centralized. Package registries like npm and PyPI, and source code repositories such as GitHub, act as central points of trust and control. The authority to publish or modify a package is often concentrated in the hands of a single repository owner or a small, insular group of maintainers. [4] This centralization represents a fundamental architectural vulnerability. The infamous SolarWinds attack, where a compromise of the build system led to the distribution of malicious updates to over 18,000 organizations, serves as a stark reminder of the risks inherent in placing trust in a single vendor or a single point in the supply chain. [5] When the security of a global ecosystem hinges on the security practices and integrity of a few individuals or systems, the model itself becomes the primary risk.

1.3 Illustrative Catastrophes: The Evolving Threat

The theoretical risks of this centralized trust model have been realized in numerous high-profile attacks, which demonstrate an evolving threat landscape that targets different stages of the software lifecycle.

In one model, an external adversary compromises a weak link in the chain. The May 2025 attack on the `rand-user-agent` npm package is a canonical example. An attacker utilized a stolen, outdated, and insufficiently secured automation token to publish malicious versions of the package directly to the npm registry. Crucially, these malicious versions had no corresponding source code commits in the project's official GitHub repository. [4] This incident starkly illustrates the dangerous disconnect between the perceived source of truth (the code repository) and the actual distribution channel (the package registry). The checksums provided by npm would correctly verify the integrity of the downloaded malicious package, but they offer no assurance that the package was derived from the trusted source code.

A more insidious threat model involves the trusted insider. In March 2022, the legitimate maintainer of node-ipc, a package with over a million weekly downloads, intentionally introduced malicious code into new versions. This "protestware" was designed to target users based on their geolocation, specifically wiping files on systems in Russia and Belarus with a heart emoji to protest the war in Ukraine. [6] This attack vector is particularly pernicious because it bypasses all traditional authentication mechanisms. The maintainer's identity was correct, their credentials were not compromised, and they had the legitimate authority to publish the package. The failure was not one of authentication but of trust and intent. The 2024 `xz-utils` backdoor (CVE-2024-3094) further escalated this threat model: a multi-year social-engineering campaign cultivated a maintainer position in a foundational compression library and inserted an SSH backdoor smuggled inside binary test fixtures, undetectable by source code review of the upstream repository. [7, 8] These incidents prove that security models based solely on authenticating the identity of a single developer are fundamentally flawed in the face of a malicious, coerced, or long-game adversarial insider.

1.4 The Fundamental Trust Gaps

These incidents expose two critical gaps in the security of the modern software supply chain, which this paper aims to address:

1. **The Provenance Gap:** There is no standard, cryptographically verifiable mechanism for an end-user to confirm that a binary artifact they have downloaded was produced by the specific, publicly visible source code it claims to represent. A checksum verifies that the bits downloaded are the same as the bits uploaded to the server, but it provides no guarantee about the origin or the process that created those bits. [9]
2. **The Authorization Gap:** There is no mechanism to verify that a release was the result of a legitimate, collective decision-making process. The approval of a release is an opaque, off-chain process. Consequently, a single point of compromise—whether it is a stolen developer credential, a compromised build server, or a malicious insider—is sufficient to authorize and distribute a malicious update to the entire user base. [4]

The evolution of threats from simple data corruption to sophisticated attacks targeting developer intent necessitates a paradigm shift. Initial security measures focused on preventing unauthorized changes (e.g., checksums). This evolved to authenticating the source of the change (e.g., GPG signing). Attacks like `rand-user-agent` revealed that even an "authentic" credential could be misused, highlighting a process failure between development and distribution. The `node-ipc` attack represents a further maturation of the threat, where the identity is correct and the credentials are secure, but the intent of the trusted actor is malicious. This demonstrates that trusting any single actor, no matter how well-authenticated, is an insufficient security model. The security paradigm must therefore evolve from authenticating a single actor's identity to verifying a collective, multi-stakeholder decision. This requires a move away from centralized trust models towards a trustless, permissionless protocol governed by a DAO and secured by crypto-economic incentives. [10]

1.5 Thesis and Contribution

This paper proposes VeriGov, a decentralized crypto-economic protocol designed to close the provenance and authorization gaps in the software supply chain. VeriGov integrates concepts from distributed ledger technology, on-chain governance, and verifiable build systems to create a resilient and transparent ecosystem for software distribution and updates.

The primary contributions of this work are:

1. A decentralized, on-chain governance protocol, inspired by Polkadot's OpenGov, that requires multi-stakeholder approval for software releases via a configurable cryptographic voting mechanism, weighted by staked tokens.

2. A protocol for generating and verifying a cryptographic attestation that creates a non-repudiable, verifiable link between a specific version of the source code and its compiled binary artifact.
3. A comprehensive protocol architecture that combines these elements to provide end-to-end security, from source code to end-user device.
4. A novel tokenomic model, centered around the `VGOV` token, that aligns incentives and secures the protocol through staking and slashing mechanisms.

This paper will proceed by first providing a formal analysis of the threat landscape, followed by a critical review of existing solutions. It will then detail the VeriGov architecture and its core protocols, and conclude with a rigorous security analysis against the identified threats.

2 The Software Supply Chain Threat Model

A comprehensive defense requires a systematic understanding of the adversary's methods. The software supply chain is not a single entity but a complex graph of dependencies, tools, and human actors, each presenting a potential attack vector. Recent surveys provide systematic taxonomies of attacks on open-source supply chains and demonstrate the structural fragility of large package ecosystems. [2, 1, 11, 12] This section synthesizes these results with recent real-world incidents to motivate the security properties VeriGov is designed to provide.

2.1 A Systematic Review of Attack Vectors

The attack vectors can be broadly categorized based on the point of compromise within the supply chain. These categories are not mutually exclusive and are often combined by attackers to form a sophisticated kill chain.

2.1.1 Package Registry Attacks

These attacks target the public repositories that developers and build systems trust to fetch dependencies.

- **Typosquatting:** This technique relies on simple human error. Attackers publish malicious packages with names that are common misspellings or typographical variations of popular, legitimate packages. For instance, a developer intending to install `request` might accidentally type `reqeust`. Examples include the malicious `express-cookie-parser` imitating the popular `cookie-parser` library, and `xlsx-to-json-lh` impersonating `xlsx-to-json-lc`. [1] This vector has proven remarkably effective due to its simplicity and the high volume of package installations in typical development workflows. [11]
- **Dependency Confusion:** A more advanced attack, dependency confusion targets organizations that use a mix of private, internal package repositories and public ones like npm or PyPI. An attacker first discovers the names of internal packages (which may leak through public code snippets or configuration files). They then publish a malicious package with the same name to a public repository, but assign it a much higher version number (e.g., `99.9.9`). When a developer's package manager or an automated build system is configured to resolve dependencies, it may see the higher-versioned public package and "confuse" it for a legitimate update to the internal one, thereby pulling and executing the malicious code. [1] This attack vector exploits the default behaviors of many package managers, which prioritize the newest available version of a package regardless of its source. [1]

2.1.2 Upstream Compromise

These attacks target the assets and infrastructure of the open-source project itself, further up the supply chain.

- **Maintainer Account Takeover:** Attackers actively target the credentials of legitimate package maintainers through phishing, social engineering, or by exploiting weak authentication practices. Once an attacker gains control of a maintainer's account or an associated automation token, they can publish malicious updates to trusted packages that are then automatically consumed by thousands of downstream projects. [1] The `rand-user-agent` incident, where an attacker leveraged a compromised and outdated npm automation token that was not protected by two-factor authentication, is a perfect illustration of this threat. [4]
- **Source Code Injection:** This involves directly compromising the project's source code repository (e.g., GitHub) or, more commonly, an individual developer's workstation. By injecting malicious code into the source, the attacker ensures that the malicious logic becomes a permanent part of the project's history. Securing developer workstations and the development platform itself is critical to mitigating this threat, as these

environments are often less controlled and can be a gateway for attackers to alter code, introduce backdoors, and steal intellectual property. [2]

- **Build/CI/CD Pipeline Compromise:** This is one of the most dangerous attack vectors because it can be invisible to developers reviewing the source code. An attacker who compromises the build server or the continuous integration/continuous deployment (CI/CD) pipeline can inject malicious code directly into the binary artifacts during the compilation or packaging process. The source code in the repository remains clean, but the distributed software is compromised. The SolarWinds attack is the most prominent example of this technique, where attackers compromised the Orion Platform's build system to insert a backdoor into signed, legitimate-looking software updates. [5]

2.1.3 Malicious Maintainer and "Protestware"

This category of threat challenges the very notion of a "trusted insider" and demonstrates that authentication alone is not security.

- **Intentional Sabotage:** A legitimate, authenticated maintainer with full authority over a project can choose to intentionally sabotage it. The `node-ipc` incident is the canonical case, where the maintainer embedded destructive malware into the package as a form of political protest. [6] A similar, though less destructive, incident involved the maintainer of `colors.js` and `faker.js` who intentionally pushed broken versions, causing widespread application failures. [1]
- **"Protestware":** A growing trend involves developers using their software as a platform for activism. This can range from the destructive (`node-ipc`) to the benign. For example, new versions of the popular `styled-components` and `es5-ext` packages were modified to detect if they were being installed on Russian systems and, if so, print anti-war messages to the console. [1] While the intent may not be traditionally malicious, this practice fundamentally undermines the trust contract between software producers and consumers. Users expect software to perform its stated function, not to serve as a vehicle for the developer's political views. [2] This vector proves that even a perfectly authenticated identity cannot guarantee benign intent.

The combination of these vectors can form a sophisticated kill chain. An attacker could begin by using dependency confusion to gain an initial foothold, exfiltrating credentials from a compromised build environment. [13] With these credentials, they could then access and modify the build pipeline, injecting a remote access trojan (RAT) into the final binary artifact, all without ever touching the public source code. [5] The legitimate maintainer, unaware of the compromise, would then sign and publish this malicious binary, leveraging the project's established trust to distribute the malware. This scenario highlights that securing isolated components is insufficient; the entire lifecycle from dependency resolution to build and publication must be verifiable.

2.2 Formal Problem Definition

To rigorously address these threats, we formally define the security properties that are currently lacking in the software supply chain. Let S be the source code of a project at a specific commit c, denoted S_c. Let B be the resulting binary artifact. Let H be a secure cryptographic hash function (e.g., SHA-256).

- **Provenance Integrity** (P_I): A system is said to provide provenance integrity if, for a given binary B and source commit c, there exists a verifiable cryptographic attestation, A_{build}, that non-repudiably binds the hash of the source code, $H(S_c)$, to the hash of the binary, $H(B)$. This attestation must be generated as part of a trusted and verifiable build process.

  $$P_I \iff \exists A_{build} : \text{verify}(A_{build}, H(S_c), H(B)) = \text{true}$$

  Current systems largely fail to provide this. A user typically receives B and a checksum $H(B)$, but this provides no cryptographic link back to the source code S_c that supposedly produced it.
- **Release Authorization Integrity** (R_A): Let $\{V_1, V_2, ..., V_n\}$ be the set of public keys corresponding to the authorized stakeholders of a project. Let a release policy be defined by a threshold m where $1 \leq m \leq n$. A system provides release authorization integrity if a release of binary B is considered valid if and only if it is accompanied by a set of at least m valid digital signatures, $\{\sigma_1, \sigma_2, ..., \sigma_m\}$, where each σ_i is a signature over $H(B)$ from a distinct stakeholder V_i.

  $$R_A \iff \exists\{\sigma_{i_1}, ..., \sigma_{i_k}\} \text{ s.t. } k \geq m \wedge \forall j \in \{1, ..., k\}, \text{verify\_sig}(V_{i_j}, H(B), \sigma_{i_j}) = \text{true}$$

  Current systems fail this property, as a release can often be authorized by a single signature or a single compromised account ($m = 1$), creating a single point of failure. [4]

3 Related Work: A Critical Assessment of Existing Solutions

Several frameworks and technologies have been developed to address aspects of software supply chain security. However, as this section will demonstrate, they are incomplete solutions that fail to address the fundamental gaps of provenance and authorization integrity in a holistic manner.

3.1 The Update Framework (TUF)

The Update Framework (TUF) is a CNCF-graduated project designed to secure software update systems. [14] Its primary contribution is providing compromise-resilience for the distribution channel.

- **Mechanism:** TUF achieves its security goals by separating signing responsibilities into distinct roles, each with its own set of keys. The four main roles are:

- **Root:** The root of trust for the entire repository. It signs the keys used by all other top-level roles.
- **Targets:** Signs the actual software artifacts (or their hashes), providing integrity guarantees. This role can be delegated to specific developers or teams for specific parts of a repository.
- **Snapshot:** Signs a versioned list of all files in the repository, preventing mix-and-match attacks where an attacker tries to present a client with a set of files that never existed together on the repository at the same time.
- **Timestamp:** Provides freshness guarantees. It is a frequently updated, short-lived signature over the latest snapshot file, protecting against freeze attacks where an attacker indefinitely serves old, but still valid, metadata. [14] TUF's design explicitly accounts for key compromise by allowing for threshold signatures (e.g., m-of-n keys) for the most sensitive roles, particularly the Root role, and by keeping the Root keys offline to protect them from online server compromises. [15]

- **Critique:** While TUF is exceptionally effective at securing the *delivery* of software, it does not address the governance of the software *release process* itself. The trust model is ultimately centralized in the Root role. The individuals or entities who control the Root keys have ultimate authority over the repository's contents. The process of defining who these individuals are, how they make decisions, and how a release is approved is an *out-of-band, human-centric process* that is not captured or enforced by the framework. [16] TUF ensures that what the Root role approves is delivered securely, but it does not provide a transparent, auditable, or decentralized mechanism for the *approval* decision itself. It solves the problem of a compromised mirror or a freeze attack, but it does not solve the problem of a compromised or malicious Root authority. Crucially, it lacks any crypto-economic mechanism to incentivize honest behavior or penalize malicious actions by the root authorities.

3.2 Sigstore

Sigstore is another CNCF-graduated project that aims to make cryptographic signing easier and more accessible for developers. [17]

- **Mechanism:** Sigstore's innovation is its "keyless" signing model, which lowers the barrier to entry for code signing. Instead of requiring developers to manage long-lived GPG keys, Sigstore uses OpenID Connect (OIDC) to bind a signature to a developer's existing identity (e.g., their GitHub or Google account). The workflow involves three main components:

- **Fulcio:** A root certificate authority (CA) that issues short-lived code-signing certificates based on a validated OIDC token.
- **Rekor:** A public, immutable, and tamper-resistant transparency log that records all signing events.
- **Cosign:** A client tool used for signing and verifying container images and other artifacts. [17] When a developer signs an artifact, Cosign generates an ephemeral key pair, gets a short-lived certificate from Fulcio binding the public key to their OIDC identity, signs the artifact, and then uploads the signature, certificate, and artifact hash to Rekor. The private key is then discarded. [17]

- **Critique:** Sigstore provides excellent guarantees about the *identity* of the signer and makes the act of signing transparent and publicly auditable. It effectively solves the problem of "who signed this?" However, it does not natively enforce a formal *governance process* for authorizing a release. A project using Sigstore still operates on an implicit, off-chain policy, such as "a release is valid if it is signed by a maintainer with the

OIDC identity maintainer@project.com." It does not provide a mechanism to enforce a policy like "a release requires signatures from 3 out of 5 core developers." The governance remains social and external to the tooling. Furthermore, the public good instance of Sigstore, while highly available and managed by a trusted community, represents a centralized trust anchor for identity verification and logging. [17] It does not possess an intrinsic economic model to disincentivize a malicious insider who has the correct, verifiable identity.

3.3 Reproducible Builds and Verifiable Compilation

These are foundational technologies that aim to solve the provenance gap by creating a verifiable link between source code and its binary output.

- **Mechanism**:
  - **Reproducible Builds** (also known as deterministic builds) are a set of software development practices that ensure compiling the same source code within the same, specified build environment will always produce a bit-for-bit identical binary artifact. [9] This allows any third party to independently compile the source code and verify that their resulting binary matches the one distributed by the project, thereby confirming that no unauthorized changes were introduced during the official build process. [9]
  - **Verifiable Compilation** takes this a step further by using formal methods to create a mathematically proven-correct compiler. A verified compiler, such as CompCert, comes with a machine-checked proof that it preserves the semantics of the source program. [18] This eliminates the compiler itself as a potential vector for a "Trusting Trust" style attack, where a malicious compiler injects backdoors into the programs it compiles.

- **Critique:** These are powerful and essential *technologies*, but they are not complete security *frameworks*. They provide a strong solution to the provenance gap, but they do not address the authorization gap at all. A project can have a perfectly reproducible and verifiably compiled build process, but a malicious maintainer can still use that process to build and sign a malicious version of the source code. The decision of *what* source code to build and release is a governance question that lies outside the scope of these technologies. Moreover, achieving reproducible builds in practice is notoriously difficult and complex, with numerous sources of non-determinism such as timestamps, build paths, file ordering, locale settings, and toolchain variations that must be meticulously controlled. [9] Verified compilers like CompCert are academically significant but are not in widespread use and typically support only subsets of complex languages like C. [18]

3.4 in-toto and CHAINIAC

Two prior systems combine multiple primitives in directions adjacent to VeriGov and warrant explicit comparison. **in-toto** [19] provides cryptographic, multi-step, multi-party attestations across the entire supply chain: project owners define a layout enumerating each step (clone, build, test, package), the principals authorized to perform it, and the artifacts that flow between steps. End users verify that every step was performed by an authorized actor in the specified order. in-toto's contribution is largely orthogonal to VeriGov's: its layout is itself an off-chain, signed policy that depends on a static set of authorized public keys without any economic stake or on-chain governance for evolving the layout. We view VeriGov's release-authorization layer as composable with in-toto's step-attestations, with A_{build} corresponding directly to an in-toto link metadata block.

CHAINIAC [20] is the closest existing work in spirit: it proposes a decentralized, collectively-signed software-update transparency system using skipchains, threshold signatures, and reproducible builds. CHAINIAC demonstrates the feasibility of decentralized release approval and was deployed in a Debian update prototype. VeriGov differs from CHAINIAC in three substantive ways: (i) CHAINIAC's approver set is fixed at deployment with no on-chain meta-governance for adding or removing approvers; (ii) it uses cryptographic incentives only (signing keys), not economic incentives, so a malicious approver suffers no financial loss; and (iii) it does not use a public, smart-contract-capable ledger, so policy enforcement (e.g., "m-of-n across roles") is implicit in the verification client rather than codified as on-chain rules. The crypto-economic layer (staking, rewards, slashing) is the principal axis on which VeriGov extends CHAINIAC.

The state of the art presents a fragmented landscape of powerful but incomplete solutions. When a user downloads a binary, they are faced with a series of trust questions that existing tools only partially answer. TUF can assure the user that the download channel is secure and the metadata is valid according to a predefined root of trust, but it cannot answer the question of who controls that root and how they made their decision. Sigstore can assure the user that a person with a specific, known identity signed the binary, but it cannot answer what should happen if that single person is compromised or acts maliciously. Reproducible builds can provide a method to verify that the binary corresponds to

a specific version of the source code, but they cannot answer who decided that this particular version of the source code was the one that should have been released. in-toto and CHAINIAC begin to integrate these guarantees but rely on a static, off-chain root of trust without economic incentives.

There is a clear and critical gap for a framework that integrates these guarantees. A truly secure release must be (1) authorized by a legitimate, collective governance process, (2) verifiably built from specific, public source code, (3) signed by an accountable identity, and (4) delivered through a secure channel. VeriGov is designed to be this integrating framework, using decentralized governance to solve the authorization problem, cryptographic attestations to solve the provenance problem, and a secure client to ensure safe delivery.

4 The VeriGov Framework: A Decentralized Architecture

To address the identified gaps, we propose VeriGov, a framework designed to provide end-to-end security for the software supply chain. VeriGov is founded on a set of core principles that shift trust from centralized authorities and individual actors to a transparent, distributed, and cryptographically enforced process.

4.1 Core Principles

1. **Decentralized Authority:** Release authorization is not vested in a single individual or entity. Instead, it is a collective responsibility distributed among a defined set of stakeholders who must reach a consensus, recorded on an immutable ledger.
2. **Verifiable Provenance:** Every software release must be accompanied by a cryptographic attestation that provides a non-repudiable link between the binary artifact and the exact source code and build environment that produced it.
3. **Transparent Auditability:** All governance decisions, including release approvals, votes, and changes to the set of stakeholders, are public transactions on a ledger. This creates a permanent, auditable history that allows anyone to verify the integrity of the process.
4. **Crypto-Economic Security:** The protocol is secured by economic incentives. Honest participation is rewarded, and malicious behavior is penalized through the native `VGOV` token, aligning the incentives of all participants with the long-term health and security of the ecosystem. [21]

4.2 Architectural Components

The VeriGov framework is composed of three primary components that work in concert to manage the software lifecycle from development to deployment.

- **VeriGov Client:** This is a minimal, lightweight binary that serves as the root of trust on the end-user's system. Its initial installation is the only point where the user must place trust in an external distribution channel (e.g., the project's official website). The client's sole responsibilities are to:
  - Connect to the VeriGov Ledger.
  - Fetch and cryptographically verify release metadata and governance decisions.
  - Download the appropriate software binary from the Distributed Artifact Store.
  - Verify the binary's integrity and provenance against the on-chain data.
  - Execute the installation or update.

  This client is the user's secure gateway into the VeriGov ecosystem and manages all subsequent updates in a fully verified manner.
- **The VeriGov DAO and Ledger:** This is the governance and metadata backbone of the framework. It is implemented as a Decentralized Autonomous Organization (DAO) on a public, permissionless blockchain, which offers a balance between decentralized verification and controlled participation. [22] For concreteness we assume the ledger is implemented on *Substrate* [23], the same framework underlying Polkadot, which provides built-in primitives for staking, on-chain governance, and forkless runtime upgrades; the protocol is, however, agnostic to the choice of public, smart-contract-capable ledger. The key data structures stored on the ledger include:
  - **The Stakeholder Registry:** A list mapping stakeholder roles (e.g., `Core Developer`, `Security Auditor`) to their respective public keys. This registry defines the set of trusted participants in the governance process.

  - **Release Proposals:** Records of proposed new releases. Each proposal is a transaction containing metadata such as the proposed version number, a cryptographic hash of the source code commit, and a pointer (hash) to the corresponding build attestation.
  - **Votes:** The digital signatures cast by stakeholders for or against a given proposal. Each vote is a transaction that is permanently recorded, providing a transparent and non-repudiable audit trail of the decision-making process.
  - **The Official Release Log:** A canonical, append-only list of binary hashes that have successfully passed a governance vote and are deemed "trusted" and ready for distribution. The VeriGov Client will only ever accept binaries whose hashes appear on this list.
  - **The `VGOV` Token Contract:** A smart contract that defines the protocol's native utility and governance token, managing its supply, transfers, and staking logic. [24]

  The ledger does not store the large binary software artifacts themselves, as this would be inefficient and costly. [25] Instead, it stores small, critical pieces of metadata as transactions in a distributed, immutable log [26], while all governance rules and actions are encoded and executed via smart contracts, removing the need for trusted intermediaries. [24]

- **Distributed Artifact Store:** A content-addressable storage system, such as the InterPlanetary File System (IPFS), used for hosting the large data files: the software binaries themselves and their corresponding build attestations. [25] Artifacts are stored and retrieved based on their cryptographic hash. This ensures data integrity (the content cannot be changed without changing its address) and decouples storage from the governance logic on the ledger. The ledger simply stores the "blessed" hashes, while the artifact store handles the efficient distribution of the data.

4.3 Cryptographic Attestation of Source-to-Binary Provenance

A cornerstone of VeriGov is the ability to cryptographically prove that a binary was created from a specific version of the source code. This is achieved through a verifiable build process and a resulting build attestation.

- **The Verifiable Build Process:** We define a build as verifiable if it is both *reproducible* and *attested*. To achieve this, the build must be performed in a *hermetic* environment, meaning it is isolated from the network and has no access to external resources beyond a precisely defined set of dependencies. [9] This is typically accomplished using containerization technologies like Docker. The build environment itself (the container image) must be versioned and its hash recorded to ensure the build can be reproduced by anyone.
- **The Build Attestation (A_{build}):** This is the data structure that captures the proof of provenance. It is a JSON object that is digitally signed by the build service that performed the compilation. [27] This attestation is then stored in the Distributed Artifact Store, and its hash is included in the release proposal on the VeriGov Ledger. The structure of A_{build} is critical and contains the following fields:
  - `source_commit_hash`: The full Git commit hash of the source code version that was checked out for the build.
  - `source_tree_hash`: The cryptographic hash of the entire source code tree at that commit. This protects against any modifications to the source after the commit was made.
  - `build_environment_hash`: The digest (hash) of the container image or a hash of the complete build environment configuration. This ensures that the same toolchain, system libraries, and dependencies are used for verification builds. [9]
  - `output_binary_hash`: The cryptographic hash (e.g., sha256) of the final binary artifact, $H(B)$.
  - `builder_signature`: The entire JSON object containing the above fields is signed by the build service's private key.

This attestation creates a verifiable chain of trust. Any stakeholder or end-user can verify the provenance of a binary by fetching the source code specified by `source_commit_hash`, running the build process inside the container specified by `build_environment_hash`, and confirming that the hash of their resulting binary matches the `output_binary_hash` in the attestation. This process leverages established concepts from the reproducible builds community and the formalisms of cryptographic attestation. [9]

4.4 Bootstrapping and Cross-Project Trust

Two architectural questions deserve explicit treatment before we evaluate VeriGov against existing frameworks: how a project's governance is initiated, and how multiple projects share the protocol.

Genesis state and project registration. A project joins the VeriGov ecosystem by posting a `ProjectGenesis` transaction that contains: (i) a unique project identifier; (ii) a list of initial stakeholder public keys with their roles; (iii) the initial role policy (quorums $\{\tau_r\}$ and required-role set $\mathcal{R}_{\text{req}}$); (iv) the slashing parameters $(\sigma_{\text{neg}}, \sigma_{\text{mal}}, \Delta_{\max})$; and (v) an initial stake escrow per stakeholder. The genesis transaction is itself signed by the founding maintainer set and is the project's irreducible root of trust—analogous in spirit to TUF's offline Root key, but recorded immutably on-chain so that the entire history of stakeholder evolution is publicly auditable from the project's first moment. From genesis onwards, every change to the stakeholder set or policy must pass a meta-governance proposal (Section 5).

Cross-project trust and shared security. Each project maintains its own stakeholder set, release log, and policy: there is no global "maintainer of all software." However, VeriGov uses a *single, global `VGOV` token* across all projects rather than per-project tokens. This choice is deliberate. A unified token means that (i) the same liquid market price p secures every project's economic argument (Equation 2); (ii) a stakeholder active in multiple projects cannot evade slashing in one by hoarding tokens of another; and (iii) the protocol's treasury, audit-funding bounties, and governance-of-the-protocol-itself remain coherent. The trade-off is that small projects inherit the price volatility of the broader ecosystem; we view this as a feature, not a bug, since it concentrates economic security where the bulk of the token holders live.

End-user root-of-trust selection. The end user still faces a one-time choice of which project's release log to trust—installing software requires having a project identifier in mind. This is no worse than today, where users implicitly trust npm/PyPI/GitHub; VeriGov's contribution is that, once a project has been chosen, every subsequent update is verified against an on-chain, multi-stakeholder, economically-bonded approval rather than a single signature.

4.5 Table 1: Comparison of Software Update Security Frameworks

To clearly position VeriGov's contributions relative to existing solutions, the following table provides a comparative analysis based on key security features.

Table 1: Security Feature Comparison: Checksum/GPG, TUF, Sigstore, and VeriGov

Feature	Checksum/GPG	TUF	Sigstore	VeriGov
Protection Against Transit Errors	Yes (Checksum)	Yes	Yes	Yes
Protection Against Rollback/Freeze Attacks	No	Yes (Timestamp/Snapshot roles)	Partial (via Rekor log)	Yes (via immutable ledger)
Source-to-Binary Provenance	No	No	No	**Yes (Cryptographic Build Attestation)**
Authenticates Signer Identity	Yes (GPG)	Yes (Key-based roles)	**Yes (OIDC-based identity)**	Yes (Public key identity)
Decentralized Release Authorization	No (Single signer)	No (Centralized Root role)	No (Implicit single-signer policy)	**Yes (m-of-n on-chain voting)**
Transparent Governance/Auditability	No	No (Governance is off-band)	Partial (Signing acts are public)	**Yes (All governance actions are on-chain)**
Resilience to Single Maintainer Compromise/Protestware	No	Partial (if Root role is multi-sig)	No	**Yes (Collective approval required)**
Dynamic Stakeholder Management	No (Manual key management)	No (Root role update is off-band)	No	**Yes (On-chain meta-governance)**
Crypto-Economic Security (Incentives/Penalties)	No	No	No	**Yes (Staking, Rewards, Slashing)**

5 The VeriGov DAO: Governance and Tokenomics

The heart of the VeriGov protocol is its on-chain governance framework, which formalizes and enforces decision-making processes as a Decentralized Autonomous Organization (DAO). [22] This approach transforms software release management from an opaque, centralized process into a transparent, community-driven system governed by token holders and executed by immutable smart contracts. [22] This protocol draws direct inspiration from the sophisticated on-chain governance system of Polkadot, known as OpenGov, which is designed to enable forkless, community-driven network upgrades. [23] We adapt its core concepts to the specific domain of software supply chain management.

5.1 Stakeholders and Roles

In the VeriGov ecosystem, every participant with governance rights is a stakeholder, represented by an on-chain identity which is cryptographically secured by a public/private key pair. To allow for flexible and secure policies, the framework implements a role-based system. While the specific roles can be customized by each project, a typical configuration might include:

- **Core Developer:** These are the primary contributors to the project. Stakeholders with this role are granted the permission to submit New Release proposals to the ledger.
- **Security Auditor:** This role is assigned to individuals, trusted security firms, or automated analysis services responsible for vetting the code for vulnerabilities. Their vote may be a mandatory requirement for a release to be approved. In principle, automated agents (e.g., static analysis or fuzzing services) could also hold this role; we discuss the open problems this raises—automated false-positives, prompt injection on source code, and the oracle problem of trusting an automated verdict—in Section 7.
- **Community Trustee:** These are stakeholders elected by the broader user community to represent their interests in the governance process. This role ensures that decisions are not made solely by the development team and provides a mechanism for user oversight.
- **Build Service:** This is a non-human, automated stakeholder. Its role is to perform the verifiable build, generate the `A_build` attestation, and publish it to the Distributed Artifact Store. Its signature on the attestation provides the crucial link in the provenance chain.

5.2 The Release Lifecycle (On-Chain Process)

A new software release progresses through a transparent and verifiable lifecycle, with each step recorded as a transaction on the VeriGov Ledger.

1. **Proposal Submission:** A stakeholder with the `Core Developer` role initiates the process by submitting a `New Release` proposal transaction. This transaction does not contain the binary itself but rather the essential metadata required for verification and voting. The payload includes the proposed version number, the `source_commit_hash` from the project's Git repository, and the content hash of the corresponding `A_build` attestation (which has already been generated by a `Build Service` and stored in the artifact store).
2. **Verification Phase:** Upon submission, the proposal enters a predefined voting period. This phase triggers an off-chain verification process by all voting stakeholders. They are automatically notified of the new proposal and perform their designated duties:
   - Security auditors review the source code identified by the `source_commit_hash` for security vulnerabilities.
   - Other developers perform code reviews for quality and correctness.
   - Crucially, any stakeholder can (and auditors are expected to) independently perform the verifiable build. They fetch the source code, use the specified build environment, and verify that the hash of their locally produced binary matches the `output_binary_hash` contained within the `A_build` attestation. This step independently confirms the integrity of the build process.
3. **Voting:** Stakeholders cast a vote by signing a message $(\text{proposal\_id}, \text{output\_binary\_hash}, \text{yes/no})$ and submitting it as a `Vote` transaction. We now state the acceptance rule precisely. Let $\mathcal{R}$ be the set of roles defined for the project, $\mathcal{R}_{\text{req}} \subseteq \mathcal{R}$ the subset of roles whose approval is required, S_r the set of stakeholders holding role r, $\text{stake}(i)$ the amount of `VGOV` staked by stakeholder i at the proposal's enactment block, and $\tau_r \in [0, 1]$ the project-defined quorum for role r. A proposal is accepted if and only if the stake-weighted approval within

every required role meets its quorum:

$$\forall r \in \mathcal{R}_{\text{req}} : \quad \frac{\sum_{i \in S_r,\, v_i = \text{yes}} \text{stake}(i)}{\sum_{i \in S_r} \text{stake}(i)} \;\geq\; \tau_r. \tag{1}$$

Equation 1 unifies the two governance regimes the protocol blends: the role partition encodes the multi-stakeholder structure (m-of-n across roles), while stake weighting within each role aligns voting influence with economic commitment. [28] Setting $\text{stake}(i)$ to a constant within a role recovers pure m-of-n voting; setting $|\mathcal{R}_{\text{req}}| = 1$ and pooling all stakeholders into a single role recovers pure stake-weighted voting.

4. **Tallying and Enactment:** A smart contract on the VeriGov Ledger serves as the automated and impartial tallying mechanism, continuously evaluating Equation 1 for each active proposal against the project's pre-configured release policy. A typical policy might require, for example, $\tau_{\text{Core Dev}} = 3/5$ and $\tau_{\text{Sec Auditor}} = 1/2$ (with equal stake within each role). Once the policy is satisfied, the contract records the `output_binary_hash` from the successful proposal into the canonical, append-only Official Release Log. From this moment on, VeriGov Clients worldwide will recognize this binary hash as a trusted and valid update.

5.3 Meta-Governance: Evolving the Trust Fabric

A static set of stakeholders is brittle and cannot adapt to the changing dynamics of a project. People join and leave projects, and trust can be gained or lost. Therefore, the VeriGov protocol includes mechanisms for "meta-governance"—the ability for the stakeholders to govern themselves.

- **Add Stakeholder Proposal:** Introducing a new voting member into the trust fabric is a highly sensitive operation. An Add Stakeholder proposal, which nominates a new public key for a specific role, must be subject to a significantly higher voting threshold, such as a super-majority (e.g., two-thirds) of all existing stakeholders. This makes it difficult for a small faction to collude and inject a malicious or controlled voter into the system.
- **Remove Stakeholder Proposal:** Conversely, the ability to remove a stakeholder who has been compromised, has lost their keys, or has acted maliciously must be swift. A Remove Stakeholder proposal might therefore have a lower threshold (e.g., a simple majority) to enable the community to react quickly to threats and maintain the integrity of the governance body. [10]

This on-chain management of the stakeholder set is a critical feature that distinguishes VeriGov from frameworks like TUF, where changes to the root of trust are manual, opaque, and off-band.

5.4 Tokenomics and Economic Incentives

The security and sustainability of the VeriGov protocol are underpinned by its native utility token, `VGOV`. The tokenomics are designed to create a robust crypto-economic system that aligns the incentives of all participants with the long-term security of the software supply chain. [29]

5.4.1 Token Utility

The `VGOV` token serves several critical functions within the ecosystem:

- **Governance:** `VGOV` tokens are required to participate in the DAO's governance. The weight of a stakeholder's vote is proportional to the amount of `VGOV` they have staked, ensuring that those with the most significant economic commitment have a corresponding level of influence over the protocol's direction. [28]
- **Staking and Security:** To participate in governance (i.e., to propose or vote on releases), stakeholders must lock up, or "stake," a certain amount of `VGOV` tokens. This stake acts as a security deposit or economic bond, which can be forfeited if the stakeholder acts maliciously. [21]
- **Incentive Alignment and Rewards:** The protocol rewards honest and beneficial behavior. A portion of the protocol's revenue (e.g., from transaction fees or a pre-defined inflation schedule) is distributed as rewards to stakeholders who participate correctly in the governance process, such as proposing and voting for releases that are successfully verified and widely adopted. [21]
- **Bounties and Funding:** The VeriGov DAO can maintain a treasury funded by protocol fees. `VGOV` token holders can vote to allocate these funds to initiatives that benefit the ecosystem, such as funding security audits, paying for development of new features, or offering bounties for the discovery of vulnerabilities in proposed software. [22]

5.4.2 Slashing: A Crypto-Economic Disincentive

The most potent security feature of the VeriGov tokenomic model is **slashing**: programmatically destroying a portion of a stakeholder's staked tokens in response to provably-bad behavior. [21] VeriGov adapts this primitive to release governance, but distinguishes two qualitatively different conditions that trigger it.

Negligence slashing. A negligence offense is one that is *cryptographically self-evident* and adjudicated by the chain itself, without external evidence. The canonical example is a `Build Service` signing a A_{build} attestation whose `output_binary_hash` does not match the result of an independent reproducible rebuild submitted by another stakeholder. Any party may submit such a counter-attestation as a slashing transaction; if the smart contract verifies the discrepancy, the offending build service's stake is automatically slashed. Negligence slashing also covers double-signing (signing two conflicting votes for the same proposal), signing votes after one's role has been revoked, and missing a sufficiently large fraction of votes (a liveness fault).

Malice slashing. A malice offense is one that requires *off-chain evidence*—the discovery, days or months after release, that a binary on the Official Release Log is malicious. Because the predicate "binary B is malicious" is not a cryptographic relation, malice slashing cannot be fully on-chain. Instead, VeriGov defines a separate *Dispute proposal* track with the following properties: (i) the challenger posts a substantial bond b in `VGOV`, forfeit on a frivolous challenge; (ii) the dispute is decided by a high-quorum vote of stakeholders *who did not vote on the original release proposal* (avoiding the obvious circularity of having the accused vote on their own guilt); (iii) the dispute window has a configurable statute of limitations $\Delta_{\max}$; (iv) successful disputes slash the original yes-voters in proportion to their stake and reward the challenger from the slashed pool; (v) unsuccessful disputes burn the challenger's bond. A diligent voter who reviewed in good faith and was nonetheless fooled by a sufficiently subtle backdoor (xz-style) is still slashed, but at a reduced rate $\sigma_{\text{neg}} < \sigma_{\text{mal}}$ if they participated in the verifiable rebuild and signed the attestation accordingly. This preserves the incentive to participate while still penalizing approval of malware.

5.4.3 Economic Security: A Cost-of-Attack Lower Bound

The crypto-economic argument can be quantified. Consider an adversary attempting to push a malicious release. By Equation 1, they must obtain "yes" votes whose stake-weighted fraction in every required role meets the quorum. Let $\mathcal{R}_{\text{req}} = \{r_1, \ldots, r_k\}$ and let $T_r = \tau_r \cdot \sum_{i \in S_r} \text{stake}(i)$ be the minimum stake mass needed in role r. For each compromised stakeholder, the adversary either purchases their cooperation (a bribe β_i) or compromises their key (an offensive-security cost κ_i). At the malice slashing rate $\sigma_{\text{mal}} \in (0, 1]$, every cooperating stakeholder forfeits $\sigma_{\text{mal}} \cdot \text{stake}(i) \cdot p$, where p is the `VGOV` unit price at slashing time. A risk-neutral rational adversary therefore faces a cost lower-bounded by

$$C_{\text{attack}} \;\geq\; \sum_{r \in \mathcal{R}_{\text{req}}} \sum_{i \in C_r} \big(\sigma_{\text{mal}} \cdot \text{stake}(i) \cdot p \;+\; \min(\beta_i, \kappa_i)\big), \tag{2}$$

where $C_r \subseteq S_r$ is the minimum cooperating set in role r such that $\sum_{i \in C_r} \text{stake}(i) \geq T_r$. Equation 2 makes three security-relevant facts explicit. (i) Security scales with the *product* of stake-per-stakeholder, slashing fraction, and token price; halving any of these halves the security margin. (ii) Concentration of stake in few large holders reduces $|C_r|$ and therefore the bribe budget, an argument for either reputation-weighted or quadratic-voting refinements (Section 7). (iii) The model assumes p is stable on the attack timescale; an adversary who can short `VGOV` before executing the attack effectively reduces p in their cost calculation, motivating long unbonding delays so that slashing is enforced at the post-attack price.

5.5 Worked Example: Releasing `acme-utils` v1.2.3

To make the abstract architecture concrete, we walk through one end-to-end release. Project `acme-utils` has five Core Developers (Alice, Bob, Carol, Dan, Eve), two Security Auditors (the firm Foo Audit and an automated fuzzing service `fuzz.io`), and a single Build Service. Its policy requires $\tau_{\text{Core Dev}} = 3/5$ stake-weighted approval and $\tau_{\text{Sec Auditor}} = 1/2$.

Step 1: Build (off-chain). Alice tags commit `c0ff33d` on `main`. The Build Service checks out `c0ff33d`, runs the build inside container image `deadbeef...` (recorded as `build_environment_hash`), produces binary B with $H(B) =$ `a1b2...`, signs the resulting A_{build}, and pins it to IPFS at CID `Qm...42`.

Step 2: Proposal (on-chain). Alice submits a transaction

$$\texttt{NewRelease}(\text{project=acme-utils},\ \text{version=1.2.3},\ \text{src=c0ff33d},\ \text{attestation=Qm}\ldots42)$$

to the VeriGov Ledger; the runtime auto-stakes 1,000 `VGOV` from Alice as proposal bond.

Step 3: Verification (off-chain, parallel). Bob, Carol, and Foo Audit each independently rebuild from `c0ff33d` inside the specified container; all three obtain $H(B') =$ `a1b2...`, matching the attestation. `fuzz.io` runs its automated test suite against the source tree and finds no regressions.

Step 4: Voting (on-chain). Bob, Carol, and Dan submit `Vote(yes)`; Eve abstains; Foo Audit submits `Vote(yes)`; `fuzz.io` submits `Vote(yes)`. Stake-weighted approval is computed at the enactment block.

Step 5: Tally and enactment. The smart contract evaluates Equation 1: Core Dev approval is $4/5$ stake-weighted (since stakes are equal here) $\geq 3/5 = \tau_{\text{Core Dev}}$; Sec Auditor approval is $2/2 \geq 1/2 = \tau_{\text{Sec Auditor}}$. The proposal is accepted. The contract appends $H(B) =$ `a1b2...` to `acme-utils`'s Official Release Log after a 24-hour enactment delay.

Step 6: Client install. A user's VeriGov Client polls the Release Log, observes the new entry, fetches B from IPFS by CID, verifies $H(B) =$ `a1b2...`, and installs.

Failure case. Suppose a compromised Build Service had silently injected a malicious payload at Step 1. Bob's independent rebuild at Step 3 would yield $H(B') \neq$ `a1b2...`. Bob submits a counter-attestation; the negligence-slashing contract verifies the discrepancy and slashes the Build Service's stake without further governance action (Section 5.4.3).

5.6 Table 2: VeriGov Governance Parameters and Actions

The flexibility of the VeriGov protocol is demonstrated by its ability to define different parameters for different types of governance actions. This allows a project to balance the need for speed with the need for security. This concept of distinct proposal "tracks" is inspired by Polkadot's governance design, which includes mechanisms for handling routine proposals, treasury spending, and emergencies with different rules. [30]

Table 2: Governance Proposal Lifecycle and Voting Policies

Proposal Type	**Initiator Role(s)**	**Voting Policy (m-of-n)**	**Stake**	**Slash**	**Period**	**Delay**
New Release	Core Developer	3 of 5 Core Dev **AND** 1 of 2 Sec Auditor	Yes	High	7 days	24 hrs
Emergency Patch	Core Dev, Sec Auditor	2 of 5 Core Dev **OR** 2 of 2 Sec Auditor	Yes	High	24 hrs	1 hr
Add Stakeholder	Any stakeholder	66% of all stakeholders	Yes	N/A	14 days	7 days
Remove Stakeholder	Any stakeholder	51% of all stakeholders	Yes	N/A	7 days	24 hrs
Update Params	Any stakeholder	75% of all stakeholders	Yes	N/A	30 days	14 days

This table illustrates how a project can establish a standard, deliberate process for regular releases while also defining an expedited track for critical security patches. It also shows how changes to the governance framework itself—adding or removing members, or changing the voting rules—are subject to the highest levels of scrutiny, requiring broad consensus from the existing stakeholders. This ensures that the system is not only secure in its operation but also resilient in its evolution.

6 Security Analysis

A security framework is only as strong as its ability to withstand known and anticipated threats. This section analyzes VeriGov's resilience against the attack vectors identified in Section 2 and explicitly states the trust assumptions upon which the framework is built.

6.1 Threat Mitigation Analysis

We revisit the primary threats to the software supply chain and evaluate how the VeriGov architecture and protocol provide mitigation.

- **Compromised Maintainer Account:** In a traditional system, an attacker who steals a maintainer's credentials can unilaterally publish a malicious package. [4] In VeriGov, this is not possible. A single compromised account can, at worst, *propose* a malicious release. However, this proposal will fail to be enacted because it

will not receive the required m-of-n signatures from the other, uncompromised stakeholders during the voting phase. The crypto-economic model adds another layer of defense: even if an attacker compromises multiple accounts, they would need to convince the owners to risk their staked `VGOV` tokens, which would be slashed if the malicious activity is discovered. [21]

- **Malicious "Protestware":** The case of a legitimate maintainer intentionally creating malicious software like `node-ipc` [6] is a direct assault on the trust placed in developers. VeriGov mitigates this insider threat through mandatory, collective review and economic consequence. The malicious maintainer can author and propose the destructive code, but it cannot be released without the explicit, staked approval of other stakeholders. The `Security Auditor` role is specifically designed for this purpose. Upon discovering the malicious payload, auditors would vote against the release, and the on-chain record of the malicious proposal would serve as grounds for a `Remove Stakeholder` vote, permanently revoking the malicious actor's governance rights. The threat of both reputational damage and financial loss via slashing makes such an act prohibitively costly. [21]
- **Build Server Compromise (SolarWinds-style):** This attack vector bypasses source code reviews by injecting malware during compilation. VeriGov directly counters this through the verifiable build and attestation mechanism. If an attacker compromises the official `Build Service` and causes it to inject malware, the `output_binary_hash` in its generated `A_build` will be for the malicious binary. However, when other stakeholders (e.g., security auditors) perform their own independent, reproducible build from the clean source code, their locally generated binary hash will not match the one in the attestation. This cryptographic discrepancy is undeniable proof of tampering. The proposal would be rejected, and an investigation into the build service's integrity would be triggered.
- **Dependency Confusion:** This attack vector tricks a build system into fetching a malicious public dependency instead of a private one. [13] VeriGov mitigates this at a fundamental level through the requirement for a hermetic and reproducible build environment. The configuration of the build environment—including the specific versions and sources (e.g., pointing exclusively to a private registry) of all dependencies—is captured in the `build_environment_hash`. [9] Any attempt to use a different dependency would result in a different environment hash, which would invalidate the build attestation and cause the verification to fail.

6.2 Trust Assumptions

No security system is absolute. It is crucial to be explicit about the assumptions that underpin VeriGov's security model.

- **Cryptography:** We assume the underlying cryptographic primitives—specifically, the hash function (e.g., SHA-256) and the digital signature algorithm (e.g., ECDSA)—are computationally secure and have not been broken.
- **Client Security:** The initial installation of the VeriGov Client on the end-user's machine is a critical bootstrapping step. We assume this initial binary is authentic and has not been tampered with. Securing this initial download remains an out-of-band challenge, common to all update systems.
- **Stakeholder Non-Collusion & Economic Rationality:** The core trust assumption of VeriGov is shifted from a single entity to a collective. We assume that a malicious actor cannot compromise or coerce the required threshold (m) of stakeholders to collude and approve a malicious release. This assumption is strengthened by the economic model; we assume stakeholders are rational economic actors who will not vote for a malicious proposal if the expected financial loss from being slashed is greater than the potential gain from collusion. [21]
- **Liveness of Stakeholders:** The framework requires that a sufficient number of stakeholders remain active and participate in voting for the governance process to function. A project could become deadlocked if too many stakeholders become inactive or lose their keys.

6.3 Limitations and Future Threats

- **Governance Attacks:** A sufficiently motivated and well-resourced adversary could attempt to subvert the governance model itself. This could involve social engineering, bribery, or simultaneously compromising the systems of multiple stakeholders to gain control of enough keys to meet the voting threshold. The defense against this is a robust and diverse stakeholder set and strong operational security for all participants.
- **Release Latency and Throughput:** A 7-day default voting window for a New Release proposal (Table 2) is incompatible with continuous-deployment projects that ship hourly. Three mitigations apply: (i) *batched releases*, where a single proposal authorizes a range of micro-releases that share a build attestation pattern; (ii) *delegated pre-approval*, where Core Developers stake-delegate to a smaller signing committee whose decisions are auto-ratified absent challenge within a short window; and (iii) *adversarial-resistance tracks* that

trade voting period for stake bond size. Independent of latency, the underlying ledger's transaction throughput and economic-security properties (e.g., resistance to MEV-style attacks [31]) bound the protocol's practical capacity and must be considered in any implementation.

- **The "Trusting Trust" Compiler Attack:** The current framework relies on reproducible builds, which assumes the compiler itself is benign. A highly sophisticated attacker could execute a "Reflections on Trusting Trust" attack by providing a compromised compiler to all stakeholders. [32] This malicious compiler could appear to function correctly but subtly inject a backdoor into the compiled binary. Since all stakeholders would be using the same malicious compiler, their reproducible builds would all produce the same malicious binary, and the attack would go undetected. Mitigating this threat requires advancing the state of the art to either use multiple, independently developed compilers for diverse double-compilation [33] or, ideally, a formally verified compiler like CompCert that is mathematically proven to be correct. [18]

7 Conclusion and Future Work

7.1 Summary of Contributions

The modern software supply chain is plagued by vulnerabilities stemming from centralized trust models that fail to provide adequate guarantees for either provenance or authorization. Existing solutions like TUF and Sigstore address important but isolated parts of this problem, leaving critical gaps that have been exploited in numerous high-profile attacks. This paper has introduced VeriGov, a holistic, crypto-economic protocol that provides a robust, end-to-end security solution for software distribution and updates.

VeriGov's principal contribution is the unique synthesis of three key security paradigms:

1. **Decentralized, On-Chain Governance:** By establishing a DAO and moving release authorization to a public blockchain, VeriGov eliminates the single point of failure inherent in current systems. It provides transparent, auditable, and cryptographically enforced governance, making it resilient to both external attacks on single accounts and internal threats from malicious maintainers. [10]
2. **Verifiable Source-to-Binary Provenance:** Through the use of cryptographic build attestations generated from hermetic and reproducible build environments, VeriGov creates an undeniable link between a software binary and the precise source code that created it. This closes the provenance gap and allows any party to independently verify that a binary has not been tampered with during the build process. [27]
3. **Crypto-Economic Security:** The introduction of the `VGOV` token, with its integrated staking and slashing mechanisms, fundamentally changes the security model. It aligns the financial incentives of all participants with the security of the network, making malicious behavior economically irrational and prohibitively expensive. [21]

By combining these pillars, VeriGov establishes a new standard for trust in the software supply chain, shifting reliance from fallible human actors and opaque processes to a system of collective responsibility and cryptographic and economic certainty.

7.2 Future Work

The conceptual framework presented in this paper lays the groundwork for several promising avenues of future research and development.

- **Prototype Implementation:** The immediate next step is to build a proof-of-concept prototype of the VeriGov framework. This could be implemented using an existing blockchain framework like Hyperledger Fabric or Substrate for the VeriGov Ledger, coupled with IPFS for the Distributed Artifact Store. A simple VeriGov Client could be developed to demonstrate the end-to-end workflow of proposing, voting on, and securely installing an update.
- **Advanced Tokenomic Models:** The current model uses token-weighted voting. Future work could explore more advanced governance mechanisms to further decentralize power and mitigate the risk of "whales" (large token holders) dominating votes. This includes exploring **reputation-based governance**, where voting power is influenced by past positive contributions, and **quadratic voting**, where the cost of each additional vote increases, giving smaller stakeholders a more proportional voice. [28]
- **Formal Verification of the Protocol:** To achieve the highest level of assurance, the VeriGov smart contracts and governance protocols themselves should be formally verified. Using proof assistants like Coq or Isabelle,

it would be possible to create a mathematical proof that the protocol correctly implements its specified security properties (e.g., that a release cannot be enacted without meeting the m-of-n threshold). [18]

- **AI-Powered Stakeholders.** A forward-looking research direction is the use of automated agents as Security Auditor stakeholders—running static analysis, fuzzing, or formal verification on a proposal's source tree and casting a vote based on the result. Several non-trivial open problems must be addressed before this is deployable: false-positive rates that would block legitimate releases, prompt-injection attacks where adversarial source code subverts an LLM-based reviewer, the oracle problem of whether a passing test result is itself trustworthy, and the question of how much stake an automated agent should carry given that it cannot be "slashed for negligence" in the same human-accountable sense. We see these as a research agenda rather than a near-term feature.

References

[1] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Proceedings of the 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2020, pp. 23–43.

[2] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK: Taxonomy of attacks on open-source software supply chains," in *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, 2023, pp. 1509–1526.

[3] National Highway Traffic Safety Administration (NHTSA), "Cybersecurity of firmware updates," 2020. [Online]. Available: https://www.nhtsa.gov/sites/nhtsa.gov/files/documents/cybersecurity_of_firmware_updates_oct2020.pdf

[4] Aikido Security, "Supply chain attack on the rand-user-agent npm package," 2025. [Online]. Available: https://www.aikido.dev/blog/rand-user-agent-supply-chain-attack

[5] Cybersecurity and Infrastructure Security Agency (CISA), "Advanced persistent threat compromise of government agencies, critical infrastructure, and private sector organizations," Alert AA20-352A, 2020. [Online]. Available: https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-352a

[6] NIST National Vulnerability Database, "CVE-2022-23812: Malicious code in node-ipc," 2022. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-23812

[7] A. Freund, "backdoor in upstream xz/liblzma leading to ssh server compromise," oss-security mailing list, 2024. [Online]. Available: https://www.openwall.com/lists/oss-security/2024/03/29/4

[8] NIST National Vulnerability Database, "CVE-2024-3094: Malicious code in xz liblzma," 2024. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-3094

[9] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2022.

[10] A. Wright and P. De Filippi, "Decentralized blockchain technology and the rise of Lex Cryptographia," *SSRN Electronic Journal*, 2015. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2580664

[11] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 995–1010.

[12] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[13] A. Birsan, "Dependency confusion: How I hacked into Apple, Microsoft and dozens of other companies," Medium, 2021. [Online]. Available: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

[14] 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 (CCS)*, 2010, pp. 61–72.

[15] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, J. Capelis, and J. Cappos, "Diplomat: Using delegations to protect community repositories," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016, pp. 567–581.

[16] The Update Framework, "The update framework specification," 2025. [Online]. Available: https://theupdateframework.github.io/specification/latest/

[17] 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 (CCS)*, 2022, pp. 2353–2367.

[18] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[19] 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 *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1393–1410.

[20] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds," in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 1271–1287.

[21] V. Buterin and V. Griffith, "Casper the friendly finality gadget," arXiv preprint arXiv:1710.09437, 2017. [Online]. Available: https://arxiv.org/abs/1710.09437

[22] Q. DuPont, "Experiments in algorithmic governance: A history and ethnography of "The DAO," a failed decentralized autonomous organization," in *Bitcoin and Beyond: Cryptocurrencies, Blockchains, and Global Governance*. Routledge, 2017, pp. 157–177.

[23] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," Polkadot White Paper, Draft 1, 2016. [Online]. Available: https://polkadot.network/PolkaDotPaper.pdf

[24] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," Ethereum White Paper, 2014. [Online]. Available: https://ethereum.org/en/whitepaper/

[25] J. Benet, "IPFS: Content addressed, versioned, P2P file system," arXiv preprint arXiv:1407.3561, 2014. [Online]. Available: https://arxiv.org/abs/1407.3561

[26] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[27] OpenSSF, "SLSA: Supply-chain levels for software artifacts," Specification v1.0, 2023. [Online]. Available: https://slsa.dev/spec/v1.0/

[28] S. P. Lalley and E. G. Weyl, "Quadratic voting: How mechanism design can radicalize democracy," *AEA Papers and Proceedings*, vol. 108, pp. 33–37, 2018.

[29] F. Saleh, "Blockchain without waste: Proof-of-stake," *The Review of Financial Studies*, vol. 34, no. 3, pp. 1156–1190, 2021.

[30] Web3 Foundation, "Polkadot OpenGov," 2024. [Online]. Available: https://wiki.polkadot.network/learn/learn-polkadot-opengov/

[31] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges," *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, pp. 910–927, 2020.

[32] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984.

[33] D. A. Wheeler, "Countering trusting trust through diverse double-compiling," in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 2005, pp. 33–48.