UNIVERSITY OF SURREY

School of Computer Science and Electronic Engineering

MSc Cybersecurity

Academic Year 2025-2026

COMM047
Secure Systems and Applications
Coursework 2

Title: BEYOND BLIND TRUST: END-TO-END ENCRYPTION AND VERIFIABLE PROVENANCE FOR SECURE GIT FORGES.

Declaration of originality of work:

I confirm that the submitted work is my own work. No element has been previously submitted for assessment; ow where it has, it has been correctly referenced. I have clearly identified and fully acknowledged all material that should be attributed to others (whether published or unpublished, including any GenAI tool and have also included their source references wherever relevant, using the referencing system required by my course in this specific assignment.
I agree that the university may submit my work to means of checking this, such as the plagiarism detection service Turnitin UK and the Turnitin Authorship Investigate service. I confirm that I understand that assessed work that has been shown to have been plagiarised will be penalised.

Sanya Arora URN: 6944081

ABSTRACT

Modern software development relies heavily on centralised Git forges such as GitHub and GitLab. While these platforms enable efficient collaboration, they concentrate trust in a single provider. This creates a systemic risk: if a forge is compromised, attackers can steal confidential source code and manipulate project history to insert supply-chain backdoors [4] [8]. Existing protections, such as TLS and server-side access controls, reduce exposure, but they cannot prevent forged operators or attackers with forge-level access from reading repositories or rewriting references.

This essay argues that secure collaboration should shift from relying on **institutional trust** to relying on **cryptographic verification**. It proposes a layered model combining **end-to-end encryption (E2EE)** [1] **with decentralised, client-verifiable provenance** [2]. Neither approach alone is sufficient: encryption protects confidentiality but not history integrity, while verification ensures integrity but not confidentiality. Together, they provide defence-in-depth against forged compromise, at the cost of increased usability challenges, workflow friction, and reduced server-side functionality.

INTRODUCTION

Secure systems must balance four core features: **security, privacy, trust, and usability**. In Git forge–based development, this balance becomes high-stakes because organisations outsource storage and collaboration for their most valuable intellectual assets—source code, configuration, and release history—to third-party platforms. In the conventional model, the forge is treated as a trusted authority: it stores plaintext repositories, enforces access control, and is implicitly relied upon to preserve the integrity and authenticity of project history.

However, forging compromise breaks these assumptions [4]. If an attacker gains forge-level control (or a malicious insider exists), they can

(1) **read or leak confidential code**, and

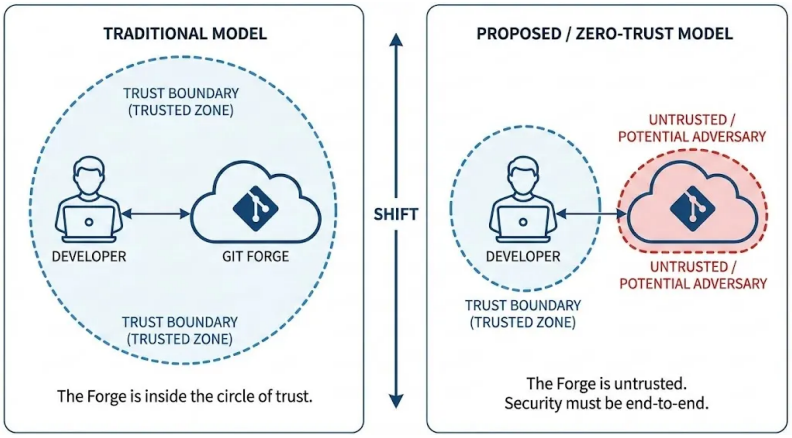
(2) **rewrite or selectively present history** to hide malicious changes.

These are distinct failure modes: confidentiality failures expose intellectual property and secrets; integrity/provenance failures enable supply-chain attacks where downstream users build and ship compromised code.

This essay's core claim is that secure collaboration should treat the forge as **untrusted infrastructure** [2] and replace "trust the platform" with "verify cryptographically." It analyses two complementary mechanisms:

- **End-to-End Encryption (E2EE)** [1] to ensure repository contents are encrypted on developer devices before upload, limiting the forge to ciphertext storage and reducing disclosure risk.
- **Decentralised trust verification** [2] (specifically *gittuf*) to provide client-verifiable provenance and policy enforcement, so that reference updates and releases remain auditable and tamper-evident even if the forge lies.

The essay concludes that a layered approach is necessary for resilience, but it creates practical trade-offs: encryption reduces server-side functionality (search, rendering, CI), while verification increases cognitive and operational load (keys, policies, warnings, and verification costs).



[9]

SECTION 1 : THREAT MODEL FOR GIT FORGES

Adversaries In a forge-based development environment, the primary adversary is the **platform (forge) itself**—whether due to a malicious insider or an external attacker hijacking the infrastructure. Unlike traditional models where the server is trusted, we assume the forge is an adversary capable of reading all stored code, injecting malicious commits, or rewriting history without detection. Secondary threats include external network attackers (mitigated mostly by TLS) and compromised developer credentials.

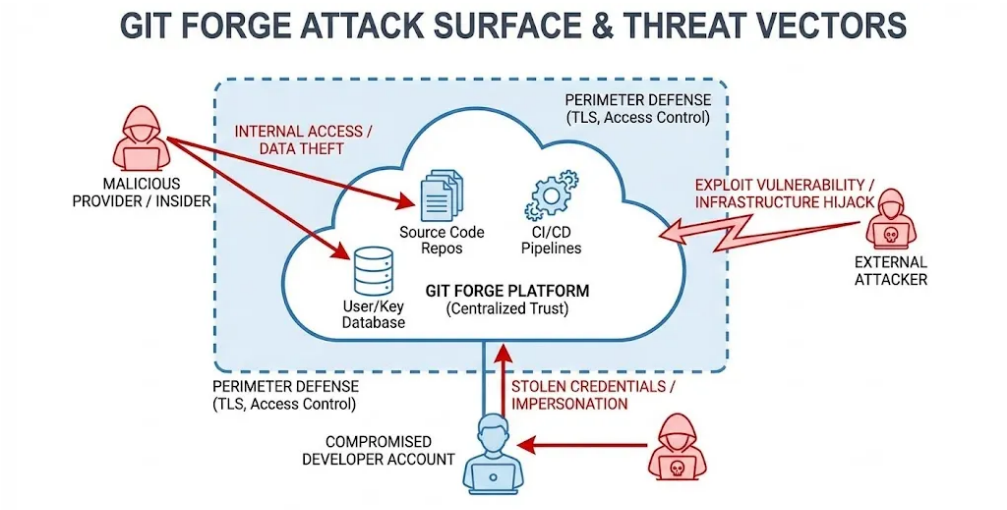
Assets We aim to protect three core categories of assets:

- **Confidentiality:** Source code, intellectual property, and sensitive configuration secrets (e.g., API keys) often stored in repositories.
- **Integrity:** The immutable history of the project. The sequence of commits, tags, and releases must remain exactly as authored.
- **Identity & Reputation:** The developer's digital identity. If an attacker can spoof commits, they destroy the reputation of the contributor and the trust in the project.

We assume a "Zero Trust" architecture regarding the server: the forge cannot be trusted with plaintext data or unsupervised history management. Conversely, we assume developers hold and secure their own cryptographic keys. However, we acknowledge two critical areas where theoretical security conflicts with reality:

1. **Functionality Friction:** Modern features like server-side CI/CD, search, and web-based editing typically require the forge to access code in plaintext, creating a direct conflict with end-to-end encryption.

2. **Human Factors:** While we assume strong cryptography, we recognize that developers may mishandle keys or fall victim to phishing. Therefore, a robust system must ensure that no single point of failure (whether a server compromise or a single stolen key) can catastrophically compromise the entire repository.



Multiple adversaries exploit centralized trust: insiders, external breaches, and compromised user keys [10]

SECTION 2: SECURITY & PRIVACY VIA END-TO-END ENCRYPTED GIT SERVICES

The Challenge: Confidentiality and Insider Threats The first fundamental question for secure development is: *"Can the platform read or leak my code?"* Traditional Git hosts require blind trust; if the server is compromised or the provider is malicious, they have full access to intellectual property. To solve this, we must shift the trust boundary so that the forge becomes merely an untrusted storage medium, incapable of deciphering the data it holds.

The Solution: Encrypted Git Services: Li et al. (2025) propose a formal framework for end-to-end encrypted Git services [1]. In this model, the forge sees only encrypted blobs, while decryption keys are held exclusively by the developers. This ensures **confidentiality**: even a total breach of the server yields no intelligible code to the attacker, effectively neutralising insider threats [1]. Crucially, the researchers implemented this using a "dumb storage" approach that remains backwards-compatible with standard platforms like GitHub and GitLab. This allows teams to adopt encryption unilaterally without requiring the provider to upgrade their infrastructure.

Performance & Efficiency: Historically, encrypting version control systems failed due to unacceptable performance overhead. Li et al. overcome this using **incremental encryption**. Instead of re-encrypting an entire file for every commit, the system leverages Git's delta-compression model to encrypt only the changes (e.g., character-level diffs rather than full files). This design keeps the computational cost proportional to the size of the edit rather than the repository size, making the system practical for real-world collaborative workflows [1].

The Usability Trade-offs: While this approach secures confidentiality, it creates significant friction in the developer workflow:

- **Feature Loss:** Server-side features that require plaintext access—such as global code search, web-based editing, and syntax highlighting—break immediately because the server can no longer parse the content.

- **Key Management Burden:** Security now depends entirely on developers managing their keys. If a private key is lost, the code is irretrievable; if leaked, the system is compromised. This shifts the complexity from the provider to the end-user [1].

- **Integrity Limitations:** Encryption prevents the server from *reading* code, but it does not fully prevent the server from *lying* about history (e.g., omitting recent commits).

Summary Encryption solves the privacy problem effectively but creates a "usability gap" and leaves integrity only partially protected. To fully secure the system, we must address how we verify the forge's honesty regarding the repository's state.

SECTION 3: TRUST & INTEGRITY VIA FORGE-BASED TRUST REDESIGN

The Challenge: The Centralized Trust Problem The second layer of defense addresses the question: "Can the platform lie or be compromised without detection?" In the standard model, developers implicitly trust the forge to enforce policies (e.g., branch protection) and maintain history. If the forge is subverted—as seen in the 2021 PHP repository breach—it can inject malicious code or rewrite history, and clients have no independent mechanism to verify the truth.

The Solution: Decentralised Verification (gittuf) Yelgundhalli et al. (2025) propose "Rethinking Trust in Forge-Based Git Security" (implemented as **gittuf**), which removes the forge as a single point of failure [2]. The system shifts security from "server-enforced" to "client-verified" using three key mechanisms:

1. **Policy Externalisation:** Security rules are defined in a signed policy file stored *within* the repository, rather than in the forge's proprietary database. This ensures policies move with the code and are cryptographically enforceable.

2. **The Reference State Log (RSL):** This is the cornerstone of the system - an authenticated, append-only log of every repository state. Similar to Certificate Transparency logs, the RSL makes the history tamper-evident [2]. If a malicious forger attempts to delete a commit or rewrite history, the cryptographic chain breaks, and clients immediately detect the discrepancy.

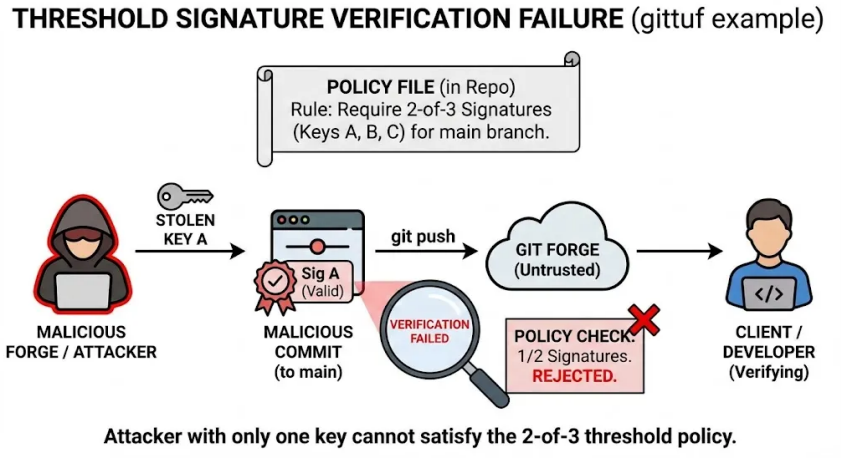
3. **Key Management & Threshold Signatures:** Unlike standard Git, gittuf handles key distribution and revocation within the repository metadata. It supports **threshold policies** (e.g., requiring 2-out-of-3 maintainers to sign a release). This provides fault tolerance: compromising a single developer's key is no longer sufficient to hijack the project [3].

Security Guarantees: This architecture fundamentally changes the threat landscape. As Cappos et al. note, an attacker would need to compromise **multiple independent keys** to bypass detection [3]. A single honest developer with a copy of the RSL can prove that the forge has lied, ensuring accountability and non-repudiation. The forge is demoted from a "trusted authority" to a verifiable service provider.

Trade-offs and Deployment While gittuf is designed to be backwards-compatible (running as a client-side layer on top of GitHub), it introduces new friction:

- **Cognitive Load:** Developers must manage signing keys and understand complex policy requirements.
- **Performance:** Verifying the entire RSL during a clone operation adds latency.
- **The "Human Gap":** The system provides *evidence* of tampering, but it relies on humans (or CI pipelines) to check that evidence. If developers ignore verification warnings or are socially engineered into signing malicious updates, the cryptographic protections are bypassed.

Summary: This approach solves the **Integrity** problem by effectively "decentralising" trust. However, it assumes that at least some maintainers remain uncompromised and vigilant.



[11]

SECTION 4: BALANCING CONFIDENTIALITY & TRUST IN SECURE GIT COLLABORATION

The Challenge: Centralised Trust as a Single Point of Failure
Relying on a centralised Git forge as a trusted intermediary introduces systemic risk. A compromised or malicious platform can both exfiltrate confidential source code and silently tamper with repository history. In the traditional model, developers implicitly trust the forge to store code correctly, enforce policies honestly, and preserve history faithfully.

However, encryption and trust verification each address only part of this problem. Encryption alone protects confidentiality but does not prevent history manipulation. Trust verification alone enforces integrity but leaves code fully exposed [2]. A secure Git-based collaboration, therefore requires a layered design that combines both approaches, explicitly rejecting blind trust in the forge.

The Layered Solution: Encryption + Trust Verification

End-to-end encryption mitigates information disclosure by ensuring that repository contents remain opaque to the hosting platform and any attacker who compromises it. This directly undermines supply-chain attacks aimed at intellectual property theft or secret leakage.

Trust verification mechanisms - such as signed commits, append-only logs, and policy enforcement - address a different adversary goal: undetected modification of code or history. These mechanisms constrain the forge's ability to rewrite history, inject backdoors, or suppress evidence of tampering.

When combined, the system eliminates single-point dependence:

- Confidentiality no longer depends on the forge's honesty.
- Integrity no longer depends on the forge's correctness.

The forge is no longer trusted - it is **verifiable**.

Security Effect: Raising Attacker Cost

The layered approach does not claim absolute prevention. Instead, it raises attacker cost and complexity. An attacker must compromise multiple independent assets:

- Developer signing keys
- Decryption keys
- Policy thresholds

Rather than exploiting a single server breach, the attacker now faces distributed barriers [2][3]. Even partial compromise yields limited benefit: encryption prevents profitable data theft, while integrity verification exposes unauthorized modifications.

The interaction between layers further strengthens detection. Abnormal signing behaviour can signal key misuse even when cryptographic verification technically succeeds.

The Trade-offs: Usability and Transparency

This design introduces explicit trade-offs:

- **Operational load:** Developers and CI systems must manage encryption & signing keys.
- **Configuration risk:** Poor key hygiene or misconfigured policies can undermine security.
- **Transparency tension:** Encryption restricts content visibility, while trust verification relies on auditable metadata.

This tension is mitigated - but not eliminated - by encrypting repository content while preserving commit graphs, hashes, and signatures. Provenance remains verifiable even when data is private.

System-Level Limitation: Insider Compromise

From a full threat-model perspective, security must be evaluated across the entire workflow: creation, encryption, signing, logging, storage, verification, and decryption.

This reveals a fundamental limitation. The system fails under **multi-party insider compromise**. If several trusted developers' keys are stolen or coerced, cryptographic policies may still be satisfied, allowing malicious commits to propagate. This mirrors real-world incidents such as SolarWinds and demonstrates that cryptography alone cannot fully defend against coordinated insider threats [8].

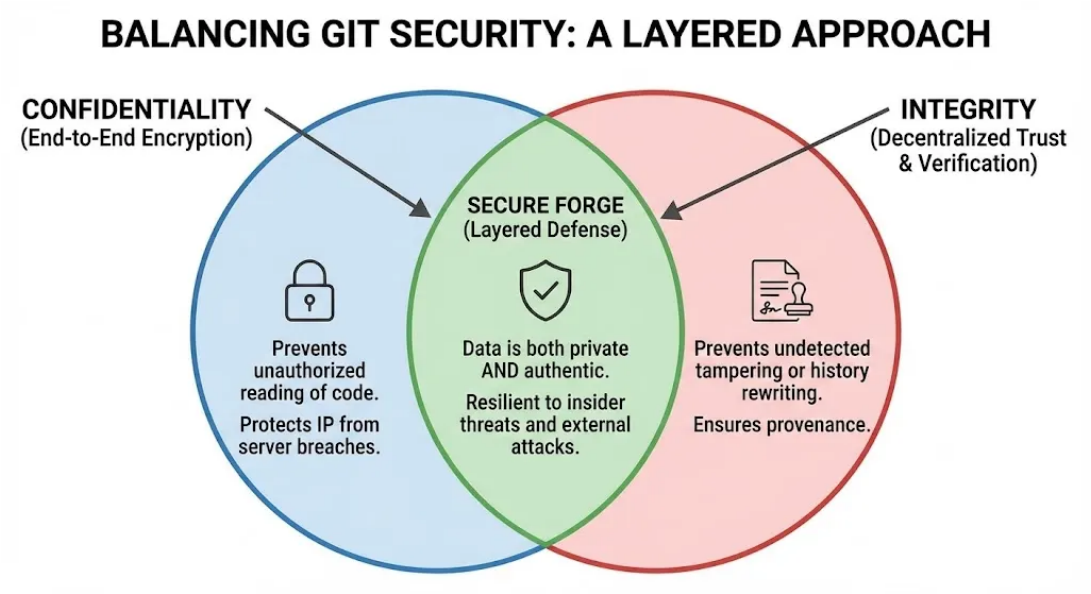
Therefore, cryptographic controls must be complemented by organizational safeguards, anomaly detection, and social verification.

Summary: Combining encrypted Git with decentralized trust verification significantly reduces supply-chain risk by constraining attacker capabilities and eliminating implicit platform trust. While the approach introduces usability and operational costs, these trade-offs are justified in environments where confidentiality and integrity failures have severe consequences.

The system does not eliminate attacks, but it:

- Reduces attacker scalability
- Increases detection probability
- Shifts security from institutional trust to verifiable guarantees

This transition is not optional — it is necessary for modern software ecosystems.



[12]

SECTION 5 — TRADE-OFFS AND CHALLENGES IN SECURE GIT COLLABORATION

Implementing layered cryptographic security in Git forges improves confidentiality and integrity but introduces unavoidable trade-offs in performance, usability, and organisation. These costs must be balanced against the security benefits.

End-to-end encryption conflicts with CI/CD pipelines that assume plaintext access. Mitigations such as secure enclaves or client-side builds restore functionality but add complexity and new trust assumptions [7]. At scale, cryptographic overhead, key management, and growing verification logs threaten long-term performance.

Usability remains a major barrier. Key loss risks permanent repository inaccessibility, forcing recovery mechanisms that partially reintroduce centralisation [1]. Migration requires coordinated cut-overs, though gradual adoption is feasible. Without seamless IDE integration, cryptographic workflows increase cognitive load and developer resistance.

Human behaviour further limits security: phishing, alert fatigue, and authority bias can bypass cryptographic guarantees. Secure systems must therefore make correct behaviour the default.

Organisational and economic factors compound these challenges. Adoption requires training, tooling migration, and a shift of responsibility from platforms to developers. Meanwhile, encryption reduces platform visibility, weakening provider incentives to deploy strong protections.

Even with layered cryptography, insider compromise, legal coercion, and endpoint attacks remain possible. Thus, cryptography reduces risk but cannot eliminate systemic threats.

Overall, the model exposes a security–usability–trust trilemma. Effective system design cannot remove these trade-offs, but must manage them in line with organisational risk tolerance.

SECTION 6 — FUTURE DIRECTIONS FOR SECURE GIT COLLABORATION

Automation and Usability: Encrypted Git and forge verification significantly improve security, but usability remains the main adoption barrier. Future work must prioritize automated key management through methods such as device-based key derivation and social recovery. Forge interfaces should also expose trust visually, for example through a “Verified” badge indicating gittuf compliance, making security intuitive for non-expert users.

End-to-End Supply Chain Trust: Repository security alone is insufficient. By integrating Git provenance with Sigstore and TUF, an unbroken trust chain can be established from source commit to final binary [5][6]. Transparency logs enable users to verify that distributed software matches a policy-compliant commit, closing critical supply-chain attack vectors.

High-Assurance and Privacy: Secure enclaves (e.g., Intel SGX) can enable CI pipelines to operate on encrypted data with remote attestation [7]. Zero-knowledge proofs further allow policy compliance to be verified without revealing developer identities, supporting accountable anonymity.

Integration Challenges: Existing solutions address confidentiality, provenance, and automation separately but lack unified integration. Secure Git collaboration is therefore primarily a system-integration challenge rather than a cryptographic one.

Cross-Domain Trust Evolution: This shift parallels AI security, where evaluation frameworks replace blind trust in models. In both domains, institutional trust is being replaced by verifiable evidence.

Summary: Although still largely experimental, these approaches indicate a clear direction: future collaboration platforms must be automated, transparent, privacy-preserving, and verifiable by design.

Conclusion

The Crossroads of Trust: Modern software development relies heavily on centralised Git forges, enabling global collaboration while simultaneously creating high-value single points of failure. True security, therefore, cannot be treated as a feature of isolated components, but as a property of the entire development ecosystem. The combination of end-to-end encryption and decentralised verification represents a fundamental shift—from implicit platform trust to explicit cryptographic verification.

The Cost of Security: Security inevitably introduces complexity, overhead, and workflow change. A system remains only as secure as its weakest element, which is often human behaviour. If key management is unusable or users are socially engineered, cryptography alone cannot protect the system. Future designs must therefore embed human-centred security, guiding users toward correct decisions rather than assuming perfect discipline.

Resilience and Adoption: Layered protections provide structural resilience: no single forge, developer, or key can fully compromise the system. While not eliminating risk, this model sharply limits its impact. High-risk sectors justify immediate adoption, while modular architectures allow gradual deployment in lower-risk environments. Performance scalability, however, remains a critical challenge.

Final Verdict: This work advocates a transition from **institutional trust to cryptographic truth**. Trust is no longer placed in platforms, but in verifiable protocols [1][2]. By combining encrypted storage with decentralised verification, the core guarantees become clear: *The platform cannot read your code & the platform cannot falsify your history.*

As supply-chain attacks escalate, such verifiable systems are likely to redefine Git forges as resilient foundations for secure global collaboration [8].

REFERENCES

[1] J. Li, Y. Zhang, X. Wang, and K. Ren, “Towards End-to-End Encrypted Git Services,” in *Proc. ACM Conf. on Computer and Communications Security (CCS)*, 2025.

[2] Y. Yelgundhalli, J. Cappos, and J. Samuel, “Rethinking Trust in Forge-Based Git Security,” in *Proc. IEEE Symposium on Security and Privacy*, 2025.

[3] J. Cappos, J. Samuel, A. Dadgar, and S. Baker, “Survivable Key Compromise in Software Update Systems,” *IEEE Security & Privacy*, vol. 10, no. 5, pp. 16–25, 2012.

[4] PHP Group, “PHP Git Server Compromise Incident Report,” 2021.

[5] J. Samuel, S. Baker, J. Cappos, and A. Dadgar, “The Update Framework (TUF): Secure Software Updates,” in *Proc. USENIX Security Symposium*, 2010.

[6] The Linux Foundation, “Sigstore: Software Signing for Everyone,” 2022.

[7] Intel Corporation, “Intel Software Guard Extensions (SGX) Developer Guide,” 2023.

[8] M. Barlow et al., “The SolarWinds Cyberattack: Lessons for Supply Chain Security,” *IEEE Computer*, vol. 54, no. 6, pp. 12–21, 2021.

[9] Google, “AI-assisted illustration generated using Gemini Pro from prompt: ‘<Trust Boundary shift>’,” Gemini Pro, 2025. [Online]. Available: https://ai.google

[10] Google, “AI-assisted illustration generated using Gemini Pro from prompt: ‘<Git Forge Attack Surface>’,” Gemini Pro, 2025. [Online]. Available: https://ai.google

[11] Google, “AI-assisted illustration generated using Gemini Pro from prompt: ‘<**Threshold Signatures"** (requiring 2-out-of-3 keys) and **Policy Verification**>’,” Gemini Pro, 2025. [Online]. Available: https://ai.google

[12] Google, “AI-assisted illustration generated using Gemini Pro from prompt: ‘<A Venn Diagram showing "Confidentiality" (Encryption) and "Integrity" (Trust), with the intersection labeled "Secure Forge.>’,” Gemini Pro, 2025. [Online]. Available: https://ai.google