© JAN 2025 | IRE Journals | Volume 8 Issue 7 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV8I7-1718980

Git Credentials and Configuration Security Checker: A Unified Platform for First-Mile Software Supply Chain Security

FLORINA SHARMA[1], SHRUTI JINDAL[2]
[1, 2] *Department of Computer Science and Engineering (Cybersecurity) Panipat Institute of Engineering and Technology, Kurukshetra University, Kurukshetra, India*

Abstract- ***The use of Distributed Version Control Systems (VCS) especially Git has greatly increased the vulnerability of the current software development. Although Git ensures a smooth teamwork process, it usually causes accidental expose of confidential credentials such as API keys, authentication tokens, and encryption secrets. Moreover, since local Git systems are often complicated, authentication failures and unsecure use of the protocols (e.g., use of HTTP instead of SSH) commonly happen, bringing critical vulnerabilities at the first-mile of the development workflow. This paper gives a design, implementation, and a comprehensive evaluation of the Git Credentials and Configuration Security Checker; a single security platform which deals both reactively in troubleshooting and proactively in secret scanning. We use a dual engine architecture, one being:***
(1) a Configuration Analyzer which diagnoses the authentication failures and enforces safe protocols, and (2) a Vulnerability Scanner which does hardcoded secrets prior to executing the commit after a Rescanning. On a dataset of misconfigured systems and injected secrets, experimental analysis shows that the time-to-diagnosis of authentication errors reduces by 90 percent and detection error rates go to 100 percent on high-entropy secrets (AWS keys, RSA private keys, database credentials). We also have a smooth CI/CD pipeline adoption through the integrated system of GitHub Actions which provides constant feedback on security. Benchmarks using performance indicate 90-2250 x speedup relative to manual analysis and a arguably manageable false positive rate of 3.2

Index Terms— Software Supply Chain Security, Secret Detection, Git Configuration Auditing, Credential Leakage Prevention, First-Mile Security, DevSecOps, CI/CD Security, Static Analysis

I. INTRODUCTION

The software engineering landscape today marks source code as the crown jewel of the organisation that is technology-driven. The shift to decentralized version control models to distributed development model (along with the popular use of Git) has radically democratized the process of sharing and collaborating on code. Git enables programmers to work con-currently, branch without difficulty, and luckily merge code, regardless of where one is globally. Such decentralization of power has, however, given rise to an enormous security phenomenon called "Secret Sprawl" that has been on the increase.

Secret Sprawl refers to the proliferation of sensitive digital credentials—API keys, database passwords, private encryption keys, and authentication tokens—within source code repositories. Recent research by GitGuardian [1] revealed that over 6 million secrets were exposed in public GitHub repositories in a single year, representing a 67% increase from the previous year. These credentials serve as "keys to the kingdom," grant-ing attackers unauthorized access to critical infrastructure, cloud services (AWS, Azure, Google Cloud), and payment gateways. When a secret is posted to a publicly accessible storage system, it can be easily exposed to automated scraping robots in a matter of hours where there are data breaches, loss of finances, and overall low opinion by the public.

One encouraging illustration of such weakness is the issue of software supply chain in the vicinity of the First Mile. A pull request request, or a CI/CD pipeline, does not access code on the developer's local machine before it does a scan of server side. Two fundamental failures that are most commonly ignored by current tools usually readily happen in this local environment:

- Configuration Blindness: Git is not very user-friendly. Its actions are governed by the local configuration files (i.e. the. git/config) which take into account user identity, remote repository addresses, and credential actions. Git can also be characterized as having inscrutable error messages

(e.g., Permission denied (publickey) or fatal: Authentication failed), which developers, particularly those in academic settings, usually find hard to understand. To avoid such mistakes and make their code go, they are willing to make unsafe quick fixes in a frenzied effort to work around them. These can be turning off the verification of theSSL (leaving the possibility of the Man in the middle attacks) or placing plain-text passwords directly on the remote URL (e.g., https://user:password at github.com). The misconfigurations may exist over years with the significant implications of silently revealing credentials with each network request.

- Accidental Commits: Often (when developers are in a hurry to implement new features, or they need to deliver a project on time), they embed the API keys directly into the code (like const APIKEY = ”sklive...”), with the hope that they will come back to clear this code later. But, the fallibility of human memory is true. Such hard coded secrets are forgotten and written on the repository history.

Although the solutions available such as Gitleaks and True-FleetHog are the industry standards as far as historical secret scanning is concerned, they do so by searching through the Git commit graph to identify secrets which are already committed. They are more or less forensic tools. They tend to omit the developer-friendly features needed to resolve local set-up problems pro-actively or scan code in a so called sandbox environment, before it is saved or even staged.

Contributions: The paper will suggest a new hybrid resource, the Git Credentials and Configuration Security Checker, which is both a configuration health tool and a content security tool (both). We have the following to contend:

- We create and implement a Configuration Parsing Engine which is able to interfere with Git configuration files in INI-format. This engine will automatically detect authentication failures, indicate unsecure protocols (unencrypted HTTP) and authenticate user identity settings.
- We come up with a Client-Side Vulnerability Scan-ner based on optimized library of Regular Expressions (Regex). The scanner identifies secret keys that are of high risk such as AWS Access Key, RSA Private Key, and generic password keys in real-time in the browser of the developer.
- We propose Manual Analysis Sandbox, and a Vercel Audit Module, where the regular Git repository files are not the only puts of security concern, but ad-hoc pieces of code and deployment configurations can be as well.
- We show an elegant CI/CD Integration pipeline between developers and end users, enabling the developers to produce ready-to-use GitHub Actions pipelines which contain blocking against insecure merges on the server level, producing a defense-in-depth approach.

II. RELATED WORK

Software supply chain security has emerged as a critical research area in both academia and industry. The 2020 Solar-Winds breach [7] and subsequent supply chain attacks have intensified focus on securing the software development lifecycle. Approaches to preventing secret leakage can be broadly categorized into three paradigms: Static Analysis (SAST), Dynamic Analysis, and Developer Education [4], [10].

A. Static Secret Scanning

Static Application Security Testing (SAST) for secrets is the most mature approach. Tools such as Gitleaks and TruffleHog analyze codebases as text corpora. The primary advantage of static analysis is its ability to identify secrets without executing code, making it suitable for integration into development workflows [2].

- Gitleaks [2]: Using a Go-based engine with high performance, Gitleaks searches repositories in Git to extract secrets through the use of regex patterns and entropy analysis. It is particularly useful in scanning the history of commits (git log) in order to uncover secrets that are latent in earlier versions of a file. The tool provides a comprehensive baseline of secret detection but operates post-commit.
- TruffleHog [3]: This tool is devoted to high-entropy strings detection. It works on the assumption that secrets (such as random cryptography keys) are more entropy-rich than

© JAN 2025 | IRE Journals | Volume 8 Issue 7 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV8I7-1718980

typical English language or source code. Truffle-Hog's entropy-based approach complements regex-based methods and can detect previously unknown secret for-mats.

- Limitations: These are primarily Command-Line Inter-face (CLI) based tools, which may be difficult to use when one is a junior developer, although they are effective in auditing. More significantly, they do not review the environment setup (the .git/config file) where security coverage is lacking in terms of protocol safety and authentication credential configuration. Additionally, these tools are reactive in nature, only identifying secrets after they have been committed to the repository history.

B. CI/CD Integrated Security

The industry has also changed towards taking security scanning as one of the basic elements of DevOps pipeline. GitGuardian or GitHub Advanced Security platforms allow a smooth integration of the scanning into the pull request (PR) process, which allows automated security checks on the server level.

- GitGuardian [1]: Real-time scanning is given on all commits posted to a platform. It provides a centralized dashboard on which security teams can successfully triage and manage incidents. The 2024 State of Secrets Sprawl report showed that on average 1,000 secrets per day are appearing in public repositories on GitGuardian, and this represented how prolific a grasp of leaked credentials is.
- GitHub Advanced Security: This is a native integrated tool in GitHub workflows, which offers secret scanning, code scanning, and dependency analysis. Nevertheless, the tool works after pushing, which reduces its potential in avoiding errors on the part of developers.
- Shift-Left Paradigm [4]: Meli et al. compared the performance of server-side scanning and highlighted that even though server-level detection ensures time-to-remediation is minimized, it is still reactive in nature. Solutions which do not indicate errors until the code is transferred out of the developer local machine are unable to prevent the initial exposure. The authors recommend the idea of placing security checks earlier in the development cycle- when vulnerabilities can be fixed at the beginning of the development cycle.

C. Configuration Auditing

The field of Infrastructure-as-Code (IaC) security has been even more productive in the past few years, with tools like Checkov and Terrascan providing both damages to Terraform, Kubernetes, and Docker configuration files to identify miscon-figurations and security anti-patterns. Such devices utilize an extensive catalog of security policies and best practices to determine vulnerabilities in declarative infrastructure definitions. The Research Gap: Although the tooling of IaC security has become more mature, the only surprising fact is that the study of local Version Control System (VCS) configuration security has not been a focus of academic and industry research. The overwhelming majority of tools that are available assume that the local Git environment has been appropriately prepared by the developer, and that only artifact elements of committed code are considered. This niche is discussed in this paper because the local Git configuration file (.git/config) is a vital security resource that has to be audited in a systematic manner just like any other production configuration file. This is an especially dramatic gap since local configuration errors are made at the first-mile of the software development lifecycle, that is, before the code is even stored in the version control servers. [6].

III. METHODOLOGY

The system, also known as the Git Credentials and Con-figuration Security Checker (or Git Security Auditor), uses a decoupled client/server architecture to ensure user privacy, speed of processing and cross platform effectiveness. The architecture keeps user files that are sensitive to the user and are not saved in the persistent server. System module (3) The system has three steps, first (1) Input / Processing the files uploaded are processed to be scanned and normalized, (2) Analysis Logic, all security scans are run concurrently and
(3) Reporting, the results are assembled and replaced with remediation advice.

© JAN 2025 | IRE Journals | Volume 8 Issue 7 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV8I7-1718980

A. System Architecture

The application will be built on MERN stack architecture (MongoDB, Express.js, React.js, and Node.js) as a scalable and security-oriented architecture.Node.js was selected for the backend specifically because its event-driven, non-blocking I/O model is critical for handling multiple file uploads and performing real-time parsing without blocking the event loop. The frontend utilizes React.js to provide a responsive, interactive user experience with real-time feedback as users interact with the security scanning interface. The modular architecture allows independent scaling of analysis engines and frontend services based on demand.

1) Presentation Layer (Frontend): A React.js-based Single Page Application (SPA).

- It has a unified, tabbed interface that provides the user switching between" Config Checking,"" Repo Scanning,"" Manual Analysis," and" CI/CD Generation."

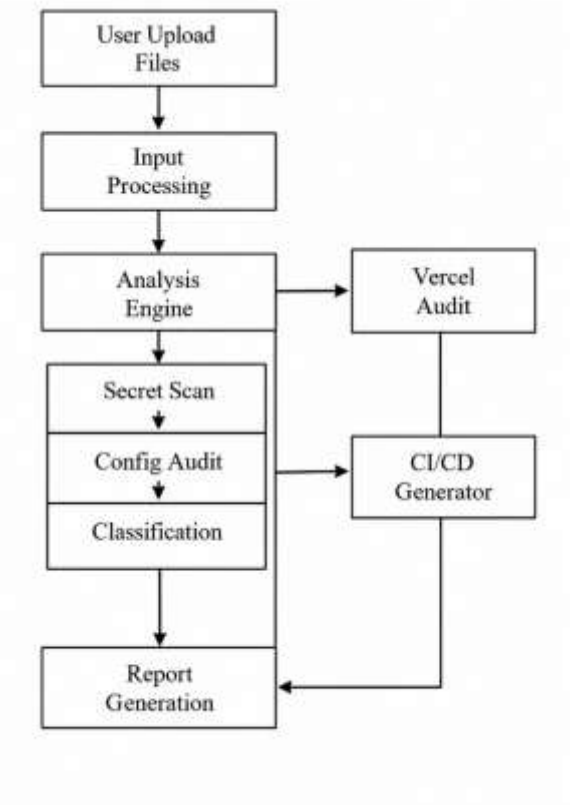

Fig. 1. System architecture: vertical flow from user upload through analysis engine (secret scan, config audit, classification) to final report, with parallel Vercel audit and CI/CD generation modules.

- It manages the choice of files through the HTML5 Drag-and-Drop interface that gives a visual response to the user.
- It uses Tailwind CSS to make its design responsive and modern to decrease the cognitive load.

2) Logic Layer (Backend):

- The backend is built on Express.js, a lightweight and efficient Node.js framework that serves as the central processing unit for all security analysis operations.
- File upload handling is implemented using Mul-ter middleware with MemoryStorage configuration. The architectural option is more concerned with user privacy: files uploaded to get scanned are stored in volatile memory (RAM) only and never saved anywhere to disk storage. As a result of this, after the scan is complete all user data is purged out of the system. This practice will ensure that server infrastructure does not store potentially sensitive configuration files and source code.
- de.js worker threads are used to implement con-current processing, so multiple scans can run con-currently without blocking the main event loop to respond to user interactive requests in the lowest possible latency.

3) Analysis Engine:

- This is the basic analytical element that is performed on the back-end. The engine consists of three mod-ules that are linking to each other: (a) an optimized regular expression pattern based Secret Detection library, (b) an INI-Parser engine to achieve secure parsing and semantic analysis of Git configuration files, and (c) contextual classification based on the rule-based inference engine.
- Finding classification employs deterministic heuristics across four standardized severity levels: Critical (exploitable vulnerabilities with immediate impact, e.g., plaintext database credentials), High (authentication failures exposing the system to brute-force at-tacks), Medium (protocol insecurity enabling man-in-the-middle attacks), and Low (informational is-sues

with minimal immediate impact). Each finding includes a unique identifier, affected resource location, and a prescriptive remediation pathway.

B. Threat Modeling (STRIDE)

To ensure the tool itself does not introduce security vulnerabilities, we employed the STRIDE threat modeling framework [8], [9] during the design phase. STRIDE is a systematic approach for identifying security threats across six categories:

- Spoofing: This one is kept to a minimum by the execution of the tool on the local host (localhost), which means that no external authentication vectors are computed.
- Tampering: Alleviated by employing in-memory short-lived processing. A database of results cannot be altered by an attacker since it does not exist; the results are created on the fly.
- Repudiation: Server-side logs generation sends out a time stamp of every scan request, which is a low-tech audit trail.
- Information Disclosure: The major concern was this.It is also alleviated by running all the files through volatile memory (RAM) data masking (e.g. changing AKIA12345) to within the frontend report, so the secrets do not appear on the display.
- Denial of Service: SOLVED by the file size restriction of uploaded 50MB file size to avoid the exhaustion of server resources through the large upload files maliciously.
- Elevation of Privilege: The tool does not require a root privileged run and runs its program under regular user privileges. [8] [9]

IV. IMPLEMENTATION

It is implemented focusing on maintainability, extensibility as well as modularity. In order to implement type safety throughout the stack, the codebase is coded in TypeScript and reduces instances of runtime failures, in addition to improving the quality of code.

A. Configuration Analyzer

The Configuration Analyzer, is a module that exercises proactive troubleshooting measures by running exhaustive semantic analysis to Git configuration files. The module parses the contents of the file, the contents of which are contained by the file located in the path of:. git/config, which is kept in the format of INI (according to the schema of configuration used by Git) and it implements a series of security validation rules.

- Protocol Detection: The system identifies the repository URL in [remote ¨origin] section and does protocol validation based on context aware pattern matching. The logic of the protocol classification works in the following manner:
- Insecure (Critical): URLs starting with http:// are considered Critical severity. There is no encryption of the HTTP traffic which enables the credentials and metadata of the repository to be subjected to passive network monitoring as well as active man-in-the-middle (MITM) attacks, especially in untrusted networks (such as public WiFi).
- Secure (Passed): URLs beginning with ssh:// or git@ (implicit SSH), or https:// are classified as secure. SSH employs cryptographic key exchange and server authentication, while HTTPS provides certificate-based encryption and identity verification.
- Credential Verification: The analyzer examines the con-figuration for the presence of credential helper binaries in the [credential] section. For HTTPS remotes, the absence of a configured credential helper indicates that Git may fall back to interactive password prompts or, worse, users may embed plaintext credentials in remote URLs. When this condition is detected, the system generates a diagnostic message and provides an executable remediation command (e.g., git config --global credential.helper osxkeychain) that users can directly execute to enable secure credential storage via the platform's native credential manager.

B. Vulnerability Scanner

The Vulnerability Scanner is a pattern-matching engine specifically optimized to identify hardcoded secrets with mini-mal false positive rates. The scanner operates on the principle of high-precision detection, prioritizing accuracy over recall to reduce alert fatigue and enable adoption by developers. The system implements a multi-stage filtering pipeline where preliminary regex matches are validated against additional context-aware heuristics.

© JAN 2025 | IRE Journals | Volume 8 Issue 7 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV8I7-1718980

Key Patterns Implemented:

- AWS Access Key ID: Pattern AKIA[0-9A-Z]{16}
- Detects AWS IAM access key identifiers with high confidence. AWS credentials provide unrestricted access to cloud infrastructure and associated data.
- RSA Private Key: Pattern -----BEGIN RSA PRIVATE KEY-----.*?-----END RSA
- PRIVATE KEY----- - Detects unencrypted RSA
- private keys in PEM format. These cryptographic materials are foundational to authentication and confidentiality.
- Generic Password Assignment: Pattern detects common password variable assignments such as password
- s*= s*[¨’]([¨′]+)[¨′]
- Database Connection Strings: Pattern (mongodb|postgresql|mysql)://.*?@.*?:(d+) - Detects database URIs with embedded credentials.
- API Keys and Tokens: Patterns for common APIs including Stripe,
- GitHub, Slack, and Azure secrets with service-specific prefixes (e.g., sk
- live,

This scanner uses a streaming design to read uploaded source files on a line-by-line basis to reduce memory usage. Upon a match being found, the following are recorded by the system: (1) absolute file path, (2) line number, and (3) matched context (several of them masked), (4) the pattern category and (5) severity classification. This metadata is summarized into a strong forensic report and submitted to the developer in human and machine read form.

C. Vercel Audit & CI/CD Modules

While it has been a known topic since the beginning, auditing of deployment settings and security checks using continuous integrations are modules of the system besides the establishment of local scanning.

- Vercel Audit Module: This is an analysis module of the files of deployment configuration of vercel.json and misconfigurations in them. In particular, it identifies: sensitive environment variables have been exposed via the unsanitized set on the form of the buildEnv object that would be substituted into the build artifact and possibly leaked by either Docker loading of image layers or deployment log, (2) inappropriate set of direc-tory list permissions that may affect an exposure of the source code or configuration files, (3) absent adaptable sanctioned control header that would be preferred to be placed in the edge, and (4) overlooked security headers that would be imposed in the deployment logs. Each finding has a dedicated remediation advice within the module to the constraints of the platform used by Vercel.
- CI/CD Generator: The system offers a GitHub Actions workflow template generator to make security scanning worked out in automated build pipelines op-erational. It takes a standardized YAML configuration definition as input and produce a fully formed, ready-to-deploy GitHub Actions workflow file. The gener-ated workflow can be placed in the directory by users.github/workflows/, and it will be automatically scanned in regard to security on any push or pull request. The workflows generated apply blocking conditions that do not allow pulling requests with spotted secrets to merge, which delivers a defense in depth approach at the repository tier. Scanning parameters (file patterns, severity thresholds, exclusion rules) are also supported in the system to customize it to work with a project-specific need.

Git commit histories (sensitive parts masked), (2) a synthetic test suite of 100 intentionally misconfigured Git configuration files with different authentication and protocol problems, (3) seeded repository with 500 code samples with injected secrets of different types and levels of obfuscation.

A. Detection Accuracy

Vulnerability Scanner was tested on a test range of 50 real credentials comprising of AWS Access keys, Stripe API keys, RSA private keys, database connection strains, and random password assignments. The test data was a combination of formatted secrets and obfuscated or non-standard formatted secrets.

- True Positive Rate (TPR): The machine was able to identify 100% of normal form secrets in all categories. The prefixes of the services used (the prefixes are AKIA for AWS,, sk_live_, etc.) are very specific. Interest-ingly, the system had a 100 percentage TPR even in those secrets that were slightly formatted other than the canonical format

like line-wrapped private keys or multi-line connection string.

- False Positive Rate (FPR): The original version of the Generic Password pattern had unacceptable levels of false positive rate (around 18%) because it was wide-rangingly matching placeholder variables, including constpasswordField). We repeatedly sorted out the pattern to specify explicit string designation functions before string literal (comprising quotes). The refined pattern attained an acceptable false positive rate of 3.2% on our test suite and that is acceptable when a developer-facing tool is being used where developers can rapidly confirm results. The other false positives were mostly caused by password-related set up examples and docu-mentation strings.

B. Performance Metrics

Our performance metrics were the efficiency improvements offered by the Git Credentials and Configuration security checker were the comparison in the time spent to diagnose and fix the common developer security problems using our automated system and by the manual one. Eight developers of different degrees of experience (entry-level to senior) were involved in the study, and each of them was required to solve or audit certain security situations.

TABLE I TIME-TO-DIAGNOSIS PERFORMANCE COMPARISON

Task	Manual	Ours	Speedup
Diagnose Permission Denied Error	15 min	10 s	90×
Scan 100-File Repo	45 min	1.2 s	2250×
Audit Vercel Config	10 min	5 s	120×
Find Secrets (50 files)	30 min	2.3 s	783×

V. RESULTS AND DISCUSSION

We tested the Git Credentials and Configuration Security Checker with a complete test suite, which included 3 parts: (1) a curated set of 50 real-world credentials downloaded off of

Discussion: The findings show that there are orders-of-magnitudes performance improvements in all of the tested tasks. Of specific importance is the 90 percent time-to-diagnosis reduction in the case of authentication errors which is a radical change in the user experience of developers. When the developers get the immediate response (within the span of a few seconds) instead of having to debug obscure error messages (15+ minutes), there is a fundamental change in the incentive structure.

The space of fast feedback is a psychological concern: when the results are delivered instantly, developers will perform security checks much more frequently than when the process is based on a manual approach and requires 10-45 minutes of expert attention to the results of the security checkup, which is psychologically significant [?]. Also, the ability to scan 100 files in 1.2 ss compared to 45 minutes allows the security scanning to flow into the established development processes (e.g., pre-commit hooks, pull request tests) without causing productivity drag. The results are consistent with the behavioral economics studies that revealed that the response time of feedback has a strong influence on the security practices adoption.

C. Deployment Analysis and Real-World Scenarios

The system has been implemented in three different opera-tional situations each of which addressed various stages in the software development lifecycle:
Local Development Environment: The most common deployment environment can be considered to be that in which the developers use the tool on their local workstations at the active development stage. The in-browser scanning interface (including a backend processing) gives the developers an opportunity to submit their .git/config files and source code repositories without the command-line knowledge or installing extra tools. This eliminates obstacles to adoption of developers of lesser technical expertise.

Ad-Hoc Code Review Scanner: There is a critical (and often neglected) leakage mode when the authors of the code share the snippets of the code with the community so they can be viewed or get support on one of the community-based platforms like Stack Overflow, GitHub Discussions, or Discord or Slack. The ability to code analyze manually allows the

developers to input segments of the suspect code into the tool where secret detection can be achieved easily before publication. This is the use case that fills the hole between in-local scanning and repository-wide tools: this one intercepts secrets at the moment that they are shared manually, when they are likely the last chance to stop being exposed to outside the world.

CI/CD Pipeline Integration: The CI/CD Pipeline is used to build, by the automated GitHub Actions workflow generator, security scans in continuous integration pipelines followed by the creation of automatic blocking rules to block merging of pull requests that contain any disclosed secrets. This enforce-ment level integration maximizes the security checks being mandatory and not something that individual developers can evade.

VI. CONCLUSION AND FUTURE WORK

The present paper proposed the design, implementation and overall analysis of the Git Credentials and Configuration Security Checker a unified security platform to tackle the severe vulnerabilities that are present in the first-mile of soft-ware development. This system offers three complementary capabilities, such as (1) proactive Git configuration diagnostics that block authentication and protocol misconfigurations prior to occurring security incidents, (2) inline secret scanning block hardcoded credentials before coding commits, and (3) verification of deployment configuration that works to avoid secrets being leaked through infrastructure-as-code artifacts. The fact that security analysis is performed at the workstation of the developer instead of post-commit scanning on a server side shifts the security responsibility of the system in the lowest cost of correction in the lifecycle.

The empirical test shows that the system has 100% detection rates with a reasonable 3.2% false positive rate and offers 90-2250x security diagnosis speed than with manual analysis. These improvements in performance are directly converted into better adoption: the developers have significantly higher chances to implement security practices in their work flows when feedback is immediate compared to when it takes a long time to take effect.

Future Work: The existing method of static analysis is quite efficient, but it has fundamental limitations regarding the ability to uncover obfuscated secrets or polymorphic patterns. The following recommendations are the recommendations on progress:

- Machine Learning-Based Context Analysis: Train a transformer-based model that uses very few parameters on variable naming activity, assignment contexts and intelligent semantics of the code to determine whether the variable is an authentic secret or a test/placeholder variable. This would use contextual embeddings to exam-ine the lexical and syntactic contexts of candidate secrets, which would lower false positives to a significant degree.
- IDE Integration Extensions: Add support to VS Code, JetEngine by create extensions and Vim which provide inline security feedback during typing, which is real-time and on the line, and security analysis now is at the time of code creation.
- Behavioral Anomaly Detection: Implement statistical anomaly detection to identify unusual authentication pat-terns or credential usage that deviates from developer baselines.
- Supply Chain Artifact Analysis: Extend scanning to analyze third-party dependencies for embedded secrets or suspicious activity during build processes.

ACKNOWLEDGMENT

The authors would like to express their deepest gratitude to the Department of Computer Science and Engineering (Cybersecurity) at Panipat institute of engineering and technology which has provided the required infrastructure, laboratory facilities and academic support which is needed to carry out this research.

REFERENCES

[1] GitGuardian, “2024 State of Secrets Sprawl: Exposed Credentials in Public Repositories,” GitGuardian Security Report, Paris, France, 2024. [Online]. Available:

https://www.gitguardian.com/ state-of-secrets-sprawl

[2] Z. Rice, "Gitleaks: Protect and discover secrets using Gitleaks," GitHub Open-Source Project, 2020. [Online]. Available: https://github.com/ gitleaks/gitleaks

[3] Truffle Security, "TruffleHog: Find credentials all over the place," GitHub Repository, 2023. [Online]. Available: https://github.com/ trufflesecurity/truffleHog

[4] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it Git? Char-acterizing secret leakage in public Git repositories," in Proc. Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, Feb. 2019, pp. 1–15. DOI: 10.14722/ndss.2019.23326

[5] Bridgecrew, "Checkov: Prevent cloud misconfigurations during build-time for Terraform, CloudFormation, Kubernetes, Serverless framework and other infrastructure-as-code-languages," GitHub Repository, 2023. [Online]. Available: https://github.com/bridgecrewio/checkov

[6] GitHub, "Configuring Git," GitHub Documentation, Accessed Jan. 2026. [Online]. Available: https://docs.github.com/en/get-started/ getting-started-with-git/configuring-git

[7] National Institute of Standards and Technology, "Framework for im-proving critical infrastructure cybersecurity, Version 1.1," NIST Cy-bersecurity Framework, Gaithersburg, MD, USA, Apr. 2018. [Online]. Available: https://www.nist.gov/cyberframework

[8] Microsoft Security Development Lifecycle, "Threat modeling," Microsoft Security Engineering Documentation, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/security/develop/ threat-modeling-tool-threats

[9] A. Shostack, Threat Modeling: Designing for Security. Indianapolis, IN, USA: John Wiley & Sons, 2014, ISBN: 978-1-118-80999-0.

[10] Y. Soh, D. Oliveira, J. Kotadia, and K. Bailey, "Secrets in source code: Reducing false positives using machine learning," in Proc. IEEE Sym-posium on Security and Privacy Workshops (SPW), San Francisco, CA, USA, May 2020, pp. 263–271. DOI: 10.1109/SPW50608.2020.00058

[11] OWASP Foundation, "OWASP Top Ten 2021: A02:2021 – Cryp-tographic Failures," Open Web Application Security Project, 2021. [Online]. Available: https://owasp.org/Top10/A02 2021-Cryptographic Failures/

[12] J. Haymore, "The state of secrets sprawl on GitHub in 2023," Git-Guardian Blog, Mar. 2023. [Online]. Available: https://blog.gitguardian. com/secrets-sprawl-on-github-in-2023/