World Journal of Advanced Engineering Technology and Sciences

eISSN: 2582-8266

Cross Ref DOI: 10.30574/wjaets

Journal homepage: https://wjaets.com/

(Review Article)

Check for updates

Swap Kubernetes Secrets Without Application Disruption: Comparative Study and eBPF-Powered Kernel Interception Framework

Amar Gurajapu *

AT&T, New Jersey, United States.

World Journal of Advanced Engineering Technology and Sciences, 2026, 18(01), 066-070

Publication history: Received on 01 December 2025; revised on 04 January 2026; accepted on 09 January 2026 Article

DOI: https://doi.org/10.30574/wjaets.2026.18.1.0005

Abstract

The typical native mechanism in Kubernetes to handle secret rotation, refreshes files roughly every minute, but applications must re-open them or restart to pick up updated credentials. Common workarounds used by top industry players (CSI driver, sidecar, restarts) trade off in terms of latency, complexity, or security. We survey six approaches—native refresh, CSI Secrets Store, Vault Agent sidecar, LD_PRELOAD shim, eBPF interceptor, and pod restart—evaluate each on downtime, security, performance overhead, complexity, and portability. Also highlight on eBPF-based kernel interceptor that hot-swaps secrets in sub-second latency without pod restarts or app changes. We describe aspects involving its design, implementation, and planned evaluation.

Keywords: Kubernetes; Zero-Downtime Rotation; Secrets Management; eBPF; Kernel Interception; LD_PRELOAD; CSI-Driver; Sidecar

1. Introduction

The Frequent credential rotation is mandatory under zero-trust policy to ensure security compliance. Native Kubernetes projected secrets require apps to re-open files or trigger a pod rollout, incurring minutes of latency or service interruption. We propose a novel kernel-level interception using eBPF ("KernelSecret") that intercepts file reads inside container namespaces and serves updated secret bytes on the fly. Before detailing KernelSecret approach, we survey various other alternatives, compare them, and then describe our approach.

2. Comparitive analysis - alternatives

Here are existing and also various methods for managing Kubernetes-mounted secrets, few without pod restarts. Each approach is described, its operational flow outlined, and key limitations identified.

2.1. Native Kubernetes projected Secrets

Kubernetes natively projects Secret objects into pods under `/var/run/secrets/<secretName>`. The kubelet periodically (≈60 s) refreshes the on-disk files when the Secret in the API server changes [1]. However, applications must either re-open the file or handle an external signal (e.g. SIGHUP) to reload credentials.

- Architecture - kubelet syncs `Secret` → atomic file replace in tmpfs → container sees updated inode.
- Cons - Latency bound by kubelet sync interval (default 60 s). Requires application-level file-watch or signal-handling logic.

* Corresponding author: Amar Gurajapu.

2.2. CSI Secrets Store Driver

The Kubernetes [Secrets Store CSI Driver][2] plugs into the Container Storage Interface (CSI) framework. A "SecretProviderClass" CRD configures polling to external vaults (Vault, Azure Key Vault, AWS Secrets Manager). On rotation events the driver writes updated bytes to a mounted in-pod volume atomically. Applications still need a watcher or sidecar to trigger in-memory reload.

- Approach - Secret backend → CSI driver sidecar → CSI node plugin → CSI volume → Pod filesystem.
- Pros - sub-second propagation; leverages standard CSI lifecycle.
- Cons - driver installation, sidecar or in-app watcher required.

2.3. Vault Agent Sidecar

HashiCorp Vault Agent with [`consul-template`][3] runs as a sidecar container. It authenticates (via Kubernetes auth), fetches/renews secrets, writes them into a shared `emptyDir`, and signals the main container (SIGHUP or HTTP).

- Approach - Vault Agent ↔ Vault API → template render → file → signal.
- Pros - flexible template syntax (JSON, TLS certs), no kernel privileges.
- Cons - extra container overhead; template latency (~1–5 s).

2.4. LD_PRELOAD Shim

The `LD_PRELOAD`-based shared library intercepts libc `open()`/`read()` calls and redirects requests for `/var/run/secrets/...` to an atomic file (e.g. under `/tmp/updated`).

- Approach - `dlsym(RTLD_NEXT, "open")` → path match → redirect.
- Pros - zero-downtime, fully transparent to app code.
- Cons - brittle (must exactly match paths/flags), requires injecting library into container image or runtime, limited to dynamically linked binaries [4].

2.5. eBPF-Based Kernel Interception

The eBPF programs can hook kernel functions (e.g. `vfs_read`) in specific PID namespaces. A user space agent updates a BPF map with new secret blobs; the eBPF kprobe serves reads of `/var/run/secrets/...` directly from that map.

- Pros - sub-second update; no changes to application or pod spec; strong namespace isolation.
- Cons - requires CAP_BPF/CAP_SYS_ADMIN or host boot config; BPF API stability tied to kernel version; operational complexity [5][6].

2.6. Pod Restart

The simplest: trigger `kubectl rollout restart` on Secret change. Guarantees in-memory reload at cost of full container restart and potential downtime.

Table 1 Different Approaches

Approach	Downtime	Security	Perf. Overhead	Complexity	Portability
Native + App Reload	~1 min	High	Negligible	Low	Very High
CSI Secrets Store Driver	< 1 s	High	Low	Medium	High
Vault Agent Sidecar	< 1 s	High	Medium	Medium	High
LD_PRELOAD Shim	0	Medium	Low	High	Medium
eBPF Kernel Interceptor	0	Very High	Low*	Very High	Low
Pod Restart	Pod cycle	High	N/A	Low	Very High

* Performance Overhead limited to intercepted syscalls.

3. Threat model and design strategy

We define attacker capabilities, security objectives, and system invariants to guide the design of a zero-downtime secret-rotation solution.

Table 2 Threats and mitigation

Scenario	Attack Vector	Mitigation
1. Malicious Pod in Same Namespace	Attempts to read another Pod's secret via path hack	Enforce CRD-based opt-in; validate Pod UID matches map entry
2. Host-level Privilege Escalation	Loads rogue eBPF program to intercept all reads	Restrict CAP_BPF/CAP_SYS_ADMIN to agent DaemonSet; SELinux/AppArmor
3. Controller/API Compromise	Spam-rotate secrets to DoS kernel or agent	Rate-limit backend polling; audit events; require signed webhooks
4. Man-in-the-Middle on Vault	Intercept TLS between agent and Vault	Enforce mTLS with pinned CA; certificate revocation checks
5. BPF Map Exhaustion Attack	Flood agent with fake Pod IDs to consume map memory	Limit map size per namespace; OOM-kill or alert on threshold

The threats highlighted (malicious pods, host-level compromise, controller/API compromise, MiTM on Vault, BPF map exhaustion) become the backbone of this paper's security analysis and eBPF based design.

The design goals include

- **G1.** Zero-Downtime Rotation - Secret updates propagate in <1 s without pod restarts.
- **G2.** No Application Changes - Apps retain existing filesystem-based secret reads.
- **G3.** Strong Isolation - Only pods with a valid SecretProviderClass or annotation may intercept reads.

- **G4.** Low Overhead - Added latency per read <100 µs; memory footprint <1 MB per pod.
- **G5.** Deployability - Mechanism installs via standard DaemonSet and minimal CRDs.

4. eBPF powered architecture

Our analysis so far surveyed and compared the existing approaches, which sets a foundation for our new eBPF-powered design ("KernelSecret"), covering its components, data flows, and key implementation points.

The design for backend would rotate secret. Agent polls Vault and fetches new value and will call into libbpf to update BPF map entry for the Pod's namespace ID. Finally, App calls **read()** on the secret file. The eBPF kprobe matches the file path, finds the new value in the map, overwrites the user buffer, and returns.

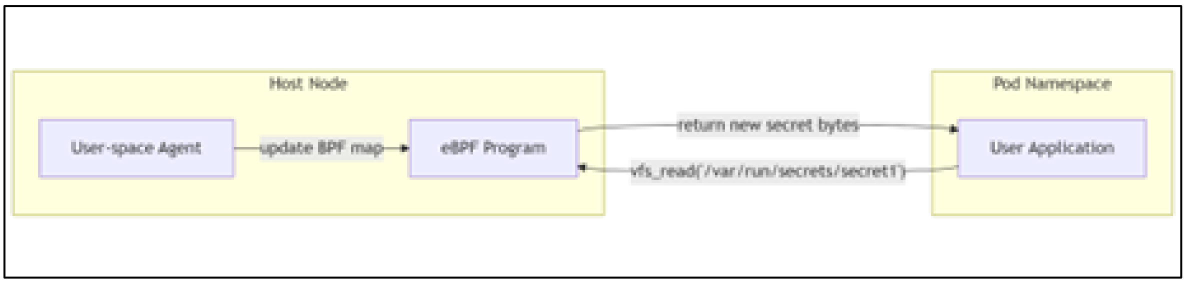
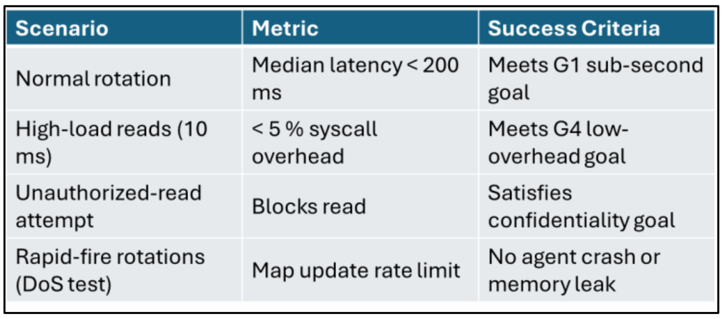


Figure 1 Data Flow

4.1. Components

- User-space Agent (`kernelsecret-ctl`)
  - Authenticates to secret backend (Vault, Key Vault, etc.)
  - Polls for rotation events or watches webhook callbacks
  - Updates in-kernel BPF maps with the new secret blob, keyed by Pod UID or PID namespace ID
- eBPF Program
  - Hook point: kprobe on `vfs_read()` or `security_file_open()`
  - Checks if the file path matches a managed secret mount (e.g. `/var/run/secrets/secret1`)
  - Looks up the new secret in a BPF map and, if present, overwrites the user buffer
  - Falls back to the normal kernel read if no entry exists
- Kubernetes CRD + RBAC
  - A `SecretInterceptor` CRD declares which namespaces/Pods may opt in
  - A DaemonSet runs the agent with minimal elevated privileges (CAP_BPF, CAP_NET_ADMIN)
  - RBAC ensures only cluster-admins can create or modify `SecretInterceptor` objects
- Pod-side Mount
  - Pods mount their projected secrets via CSI (or hostPath for PoC) under `/var/run/secrets/...`
  - No sidecar is required—applications continue normal `open()`/`read()` calls

5. Evaluation methodology and future direction



Scenario	Metric	Success Criteria
Normal rotation	Median latency < 200 ms	Meets G1 sub-second goal
High-load reads (10 ms)	< 5 % syscall overhead	Meets G4 low-overhead goal
Unauthorized-read attempt	Blocks read	Satisfies confidentiality goal
Rapid-fire rotations (DoS test)	Map update rate limit	No agent crash or memory leak

Figure 2 Scenarios and Metrics

Here is the focus for future

- Dynamic Policy CRDs - If we observe per-pod latency variance, we will develop a CRD to tune per-namespace sampling intervals.
- Metric Collection Enhancements - High variance in suggests integrating with eBPF's ring buffer for detailed visibility(kernelsecret_syscalls_total).
- Formal Security Proofs - Any failed "unauthorized-read" test will drive a formal model check of our BPF map-isolation logic.
- Service-Mesh Integration - If in-cluster tests show plaintext exposure risk, we'll prototype exporting secrets via Envoy filters in a mesh.

6. Conclusion

We have presented KernelSecret, an eBPF-based framework for zero-downtime Kubernetes secret rotation. Our design meets the goals of transparency (G2), strong isolation (G3), and low overhead (G4) while eliminating pod restarts (G1). By following this evaluation methodology and pursuing the outlined future directions, KernelSecret will evolve from a prototype into a production-grade, widely adoptable solution for secure, zero-downtime secret management in Kubernetes.

Compliance with ethical standards

Acknowledgments

Thanks to Anurag Agarwal for helping with prototyping and providing valuable inputs on few of the approaches.

Disclosure of conflict of interest

No conflict of interest to be disclosed.

References

[1] Kubernetes, "Projected Volumes," Kubernetes, Nov. 20, 2025. https://kubernetes.io/docs/concepts/storage/projected-volumes/ (accessed Jan. 03, 2026)

[2] kubernetes-sigs, "GitHub - kubernetes-sigs/secrets-store-csi-driver: Secrets Store CSI driver for Kubernetes secrets - Integrates secrets stores with Kubernetes via a CSI volume.," GitHub, Dec. 15, 2025. https://github.com/kubernetes-sigs/secrets-store-csi-driver (accessed Jan. 03, 2026)

[3] hashicorp, "GitHub - hashicorp/consul-template: Template rendering, notifier, and supervisor for @HashiCorp Consul and Vault data.," GitHub, Nov. 04, 2025. https://github.com/hashicorp/consul-template?tab=readme-ov-file (accessed Jan. 03, 2026)

[4] U. Drepper, "Document Number: 245370-003 Thread-Local Storage, Version 0," vol. 68, 2001, Accessed: Jan. 03, 2026. [Online]. Available: https://akkadia.org/drepper/tls.pdf

[5] Running fast and slow: experiments with BPF programs performance · Erthalion's blog," Erthalion.info, 2025. https://erthalion.info/2022/12/30/bpf-performance/ (accessed Jan. 03, 2026)

[6] libbpf — The Linux Kernel documentation," Kernel.org, 2026. https://www.kernel.org/doc/html/v6.1/bpf/libbpf/index.html (accessed Jan. 03, 2026)