International Journal on Advanced Computer Engineering and Communication Technology

ISSN: 2278-5140

Volume 15 Issue 2 (2026) | Pages: 227-230

Production-Oriented Extensions for a User-Space Runtime Function Interposition Framework

Parth Pawar[1], Ranjana Agrawal[2]

[1,2]Dept. of Comp. Science, Dr. Vishwanath Karad MIT World Peace University, Pune, India

[1]1032221829@mitwpu.edu.in, [2]ranjana.agrawal@mitwpu.edu.in

Peer Review Information

***Type**: Article*
***Received**: 25 May 2026*
***Revised**: 21 June 2026*
***Accepted**: 15 July 2026*
***Published**: 11 Aug 2026*

Abstract

The runtime function interposition allows for changing of running applications without a complete service restart. The patching mechanism is basic; while it allows function redirection to be done in a way that is safe for the instructions, it is also important for practical deployment to have monitoring, authentication, validation, rollback and administrative control. It is an extension of a user space runtime interposition framework for x86-64 Linux userspace and production mechanisms for patch deployment that provide more safety and observability. The proposed method will take runtime measurement metrics, administrative interface, patch registration and backup management, authentication and signature verification, canary validation, and automatic rollback and place them in the current patching pipeline. Process control is implemented via ptrace, instruction-boundary via Capstone, and function redirection via inline trampolines. The extended workflow first resolves the target symbol, checks the patch, suspends and audits the application threads and saves the original instructions, adds the trampoline and restores the application's execution. The patch can then be kept in place in the post deployment environment or automatically reverted to the original implementation if a canary failure occurs. These extensions provide better observability, recovery and operational reliability of live patching while maintaining the light weight user space architecture.

Keywords: Runtime Interposition; Live Patching; Linux; ptrace; Capstone; Trampoline; Rollback; Canary Validation; Observability; ELF

How to Cite This Article

Pawar, P., & Agrawal, R. (2026). Production-oriented extensions for a user-space runtime function interposition framework. *International Journal on Advanced Computer Engineering and Communication Technology,* 15(2), 227–230.

Introduction

As high-availability applications are often accessed by live users, they may require fixes or functionality enhancements without disrupting the live application. Typical software updates involve stopping the application and restarting it, causing existing connections to be lost and services to be unavailable. The current system does this by changing the code of an already running Linux process via user space runtime function interposition. It works by using three essential elements: ptrace, instruction-aware analysis, thread auditing, and inline trampolines to divert execution (without restarting the application).

A practical deployable live-patching solution, however, needs more than just capabilities to change into executable memory. Administrators need to know if a patch is applied and the state it is in, to validate patch operations, to verify application health, and to quickly recover from unexpected behavior that results from the patch.

This is addressed by adding extensions to the current patching mechanism in Project to meet these requirements. These extensions include an embedded metrics and administration server, patch registry and patch backup management, authentication and signature verification, canary validation and automatic rollback. So the goal is not to supplant the existing patching process, but to create a working overlay that will enhance runtime patch deployment visibility, control, and recoverability.

Production-Oriented Methodology

Patch Initialization and Target Resolution

The first step in patching is to identify the desired function that needs to be changed. The framework uses a layering mechanism for symbol resolution, which is available in runtime code, to resolve the address of the target symbol. A DWFL/libelf based resolution is attempted first and the nm based one is attempted second if the primary ones are not available. This makes it easier to support address-space layout randomization, position-independent executables and shared libraries.

Authentication and Patch Verification

The framework then completes the defined authorization and integrity checks, before changing the target process. A token mechanism can optionally ensure that the patch operation is performed by an authorized operator. Additional signature verification may also be set to verify integrity and authenticity of the patch payload. If authentication or signature verification fails, then the patching operation will be stopped without changing anything.

Thread Suspension and Safety Audit

Once validated successfully, the framework attaches to the target process's threads with ptrace and stops the running of the target process. Instruction-aware disassembly is used to analyze the original bytes of the target function. Capstone has been used to determine the boundary for a complete x86-64 instruction, to prevent the trampoline from overwriting a partial instruction. The second phase in the framework then audits the thread instruction pointers. If the RIP of a thread is inside an unsafe memory region that will be overwritten, the thread will be advanced by controlled single-stepping until it is out of the unsafe memory region. This stage makes sure that memory does not get modified during the execution of instructions being modified by an active thread.

Backup and Trampoline Injection

Executable memory is not modified until it is backed up by the original function bytes to a per-patch backup file: /tmp/livepatch_<PID>_<function>.bin. The backup contains a deterministic recovery mechanism in case the patch must be removed. After the original implementation has been safely backed up and all relevant threads have been audited the framework writes the trampoline into the target function. The trampoline is used to jump to the replacement implementation in the patch payload. The threads are then picked up again.

Production Extensions

Runtime Metrics and Administration

An embedded HTTP server offers Runtime observability and Administration. The server has exposed endpoints like /metrics, /health, /ready, /admin/patches and /admin/rollback. The /metrics endpoint delivers operational metrics, and the health and readiness endpoints let you check the health and readiness of the application. The administrative endpoints offer visibility on the live patches and offer a controlled rollback. This extension will make patching framework easier to monitor, without the need of a separate monitoring service.

Patch Registry and Backup Management

A patch registry keeps track of patches applied to the system, such as the patch name, process ID, backup file, and patch status. If you apply a patch, the patch is added or modified in the registry. If you use unpatch or rollback, the corresponding entry will be deleted or changed. This enables a consistent picture of the patch life cycle and relates the administrative interface to the patch state.

Authentication and Signature Verification

Security features have been added to ensure that patch operations cannot be made without permission and that patch payloads cannot be altered. Token-based authorization may limit operations for the administrators to certain users. Optional cryptographic signature verification can be used to verify the patch payload before it is injected into the target process. Hence, a patch is required to go through the security checks that have been set up before the framework will move on to processing modification.

Canary-Based Validation

Once a patch is applied and application threads restarted, the framework can optionally do a canary health check on a configured endpoint. This stage aims to find the failures that may not become apparent right after the patch is installed. If the application continues to respond properly, then the patch is still active. When canary check fails periodically for the set timeframe the framework performs the rollback process.

Automatic Rollback

Rollback re-uses the bytes that were stored when the patch was installed. After manually requesting rollback or failing a canary, the framework suspends the appropriate threads, restores the saved instructions, removes the backup as needed, updates the patch registry and resumes application execution. The same safety features are employed during patch removal as they were during patch installation.

Overall Methodology Flow

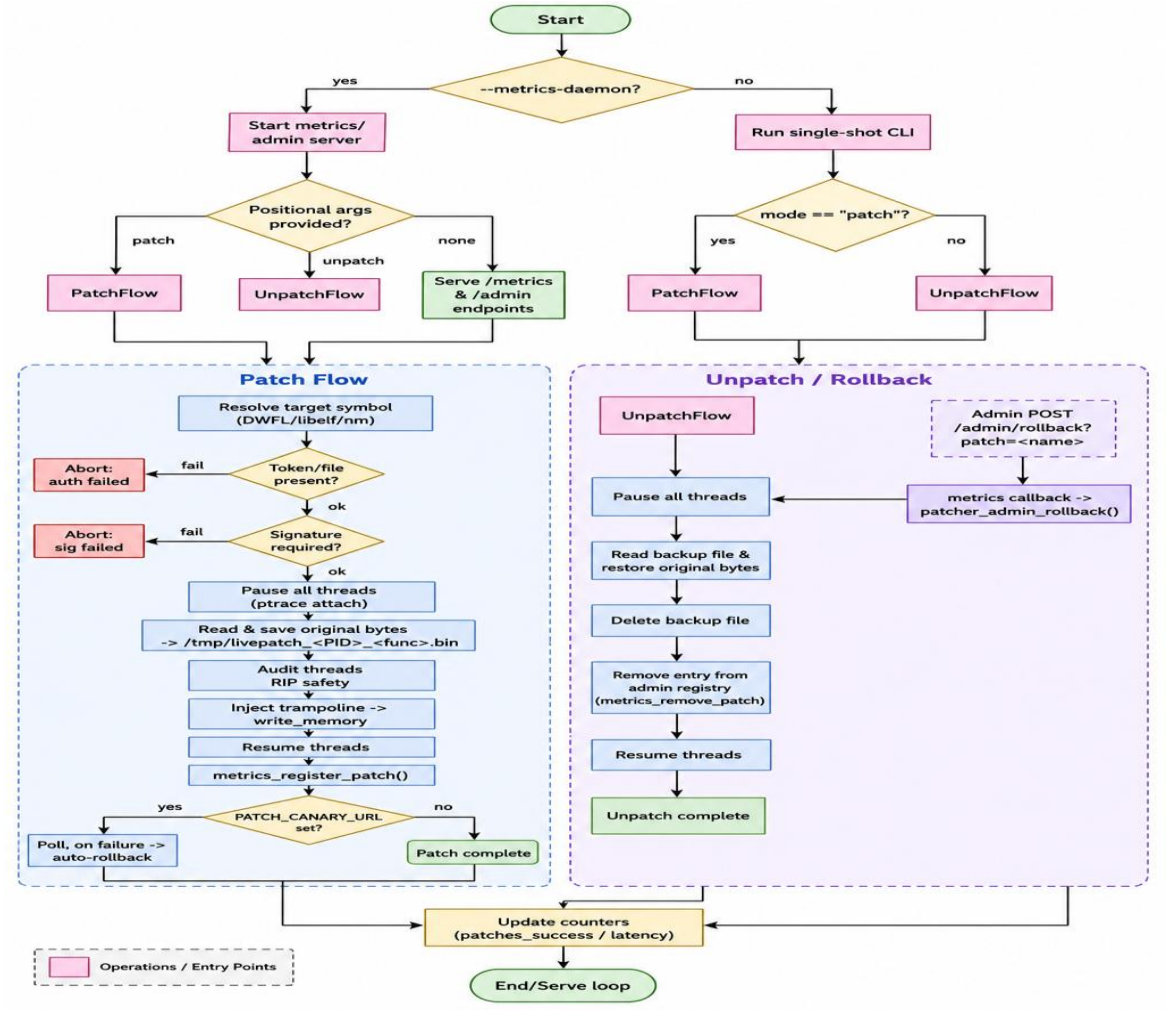

Fig. 1. *Production-Oriented Runtime Interposition and Patch Management Workflow*

The overall Project workflow is as follows: validate the target and patch, suspend the process threads, find a safe instruction boundary, save the original bytes, audit the instruction pointers of the threads, inject the trampoline, resume the process threads, register the patch, optionally validate the health of the application via a canary endpoint. If it's a healthy canary, the patch remains in place, otherwise the failed canary can trigger an automatic rollback. The unpatch and administrative rollback paths restore the original bytes that are saved and update the patch registry.

Operational Benefits

- Observability: The metrics, health checks, readiness checks and patch registry give administrators visibility into the state of the system that is running.
- Security: Authentication and signature checks ensure patch payload integrity and that patch is not tampered with.
- Recoverability: Original instructions are kept prior to modification, so that the framework can go back to the previous implementation.
- Validation: Post-Patch Monitoring: Canary is an external indicator that the application is healthy after patch deployment.
- Automation: Automatic rollback minimizes the time needed to roll back from a failed patch.
- Administration: An embedded HTTP interface offers a centralized way to inspect patches and trigger admin rollback.

Conclusion

Project extends the interposition function used to extend runtime functions to the user space to a more operationally complete live-patching system. The underlying mechanisms for safe code modification (ptrace, Capstone, thread-auditing, backup, trampoline) are unchanged, and the new production layer offers monitoring, administrative control, security validation, health-based verification, and automated recovery.

This will result in a holistic approach for runtime patching (validate, suspend, backup, audit, inject, resume, monitor, retain/rollback). This increases the usability of live patching of long-running, multi-threaded Linux applications without the original framework's heavy weight user space design.

References

1. Linux Kernel Documentation, "ptrace(2) — process trace and control," Linux Programmer's Manual.
2. Capstone Engine, "Capstone — The Ultimate Disassembler," Capstone Disassembly Framework.
3. System V Application Binary Interface, "AMD64 Architecture Processor Supplement," x86-64 calling convention and register usage.
4. M. Arnold and B. G. Ryder, "A Framework for Hot Patching of Running Programs," literature on dynamic software updating and runtime patching.
5. J. Arnold and M. F. Kaashoek, "Ksplice: Automatic Rebootless Kernel Updates," Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys), 2009.
6. Red Hat, "kpatch — Dynamic Kernel Patching," documentation on live kernel updates.
7. N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," Proceedings of PLDI, 2007.
8. B. Heinloth et al., "Luci: Loader-Based Dynamic Software Updates," USENIX Annual Technical Conference, 2023.