Comprehensive Journal of Science
Volume (10), Issue (39), (March, 2026)
ISSN: 3014-6266

مجلة العلوم الشاملة
المجلد (10) ملحق العدد 39 مارس 2026
ردمد: 3014-6266

# Secure and Efficient Hybrid Kernel Live Patching via TEEs for Linux Systems

Nuha Omran Abokhdair
Abo.khdeir@zu.edu.ly

Salah Eddin Aribi
Salah.ariby@gmail.com

Computer Science Department, Faculty of Science, University of Zawia, Libya

تاريخ الاستلام:2026/02/15 –تاريخ المراجعة:2026/03/12– تاريخ القبول: 2026/03/13– تاريخ للنشر 2026/04/25

## Abstract

Frequent operating system updates are essential for addressing security vulnerabilities and improving functionality, but they often require full system reboots that introduce costly downtime and disrupt long-running services. Existing kernel live patching techniques, such as kpatch, Ksplice, kGraft, and KUP, reduce reboot frequency but either incur substantial runtime and memory overheads or rely on the trustworthiness of a potentially compromised operating system kernel. This paper presents a hybrid kernel live patching framework that combines virtualization, selective checkpoint-and-restart, hardware-assisted trusted execution, and distributed patch propagation to enable secure, near zero-downtime updates. A lightweight virtualization layer coordinates patch deployment, while a KUP-inspired engine leverages CRIU and kexec to support major kernel updates with minimal disruption. Patch validation and critical update logic are isolated inside a trusted execution environment, which reduces the trusted computing base and protects against patch tampering and rollback attacks even under kernel compromise. Furthermore, a gossip-based dissemination protocol enables efficient patch propagation in clustered deployments. Experimental results on a Linux-based prototype show that the proposed framework applies kernel patches in less than a few seconds, maintains CPU overhead within a few percent, significantly reduces memory usage compared with full-process checkpointing, and provides stronger security guarantees than traditional reboot-based and standalone live patching solutions.

Keywords— Kernel live patching, Linux kernel updates, Trusted Execution Environments (TEEs), virtualization-based patching, checkpoint-and-restart, gossip-based patch propagation, secure patch deployment

## Introduction

Modern software systems have become increasingly complex and heterogeneous, which in turn raises the need for frequent patches and updates, including at the operating system level. However, applying kernel patches often requires system restarts, leading to periods of service downtime [1].

This unavoidable interruption affects both enterprise environments and individual users. In particular, operators of systems that run long-lasting scientific computations or latency-sensitive financial transactions are often reluctant to reboot their machines, even when critical patches are available [2] [3].

Industry reports indicate that unplanned IT downtime can incur substantial financial losses. According to Gartner, the average cost of downtime is approximately 5,600 USD per minute, and in some business scenarios it may reach around 300,000 USD per hour on average [4].
Even for regular end users, unexpected reboots can disrupt active applications and may cause the loss of unsaved data.
Consequently, both organizations and individuals tend to postpone operating system updates, which increases their exposure to security vulnerabilities and reliability issues [1]. Because patches play a central role in fixing vulnerabilities and introducing new features, several prior works have proposed live patching techniques that aim to apply updates without rebooting the system or discarding application and OS state [5][8]. In practice, many organizations adopt rolling upgrade strategies, where updates are applied to subsets of services or nodes in order to limit overall downtime [36].
Nonetheless, rolling upgrades rarely eliminate the need for restarts, which has motivated dynamic hot patching (live patching) techniques that modify running software in place without requiring a full reboot [6][ 9].
A number of kernel-level live patching frameworks have been proposed, including kpatch [9] kGraft [10] Ksplice [11] and the Canonical Livepatch Service [12].
These solutions typically rely on kernel instrumentation and indirection mechanisms to redirect execution from vulnerable code to patched versions at runtime. However, they require modifications to the kernel codebase and inherently assume that the underlying operating system remains trustworthy during the patching process. Similarly, KUP replaces the entire kernel image at runtime while preserving user-space application state [7].
This design, however, can introduce substantial runtime and memory overhead, as full process check pointing may consume tens of gigabytes of memory even for relatively small kernel updates [13].
Conventional patching mechanisms typically place implicit trust in the operating system kernel and in auxiliary patching utilities. In practice, the implementations of these mechanisms may contain bugs that lead to incomplete or failed updates [14].
Furthermore, if the OS or the patch distribution channel is compromised, adversaries can tamper with the patching process itself, for example by injecting malicious code into legitimate update workflows [15][16].
Even when live patches are correctly deployed, attackers may still attempt to roll the system back to vulnerable kernel versions, effectively undoing security fixes [17].
These risks are amplified in remote and cloud computing environments, where users have limited control over the underlying platform and patch management infrastructure [18] [19].
Consequently, there is a strong need for live patching approaches that offer higher dependability and stronger protection against tampering.

**Contributions.** This work proposes a hybrid kernel live patching framework that addresses the limitations of existing solutions in terms of availability, resource usage, and security. First, we design a modular architecture that integrates a lightweight virtualization layer, a KUP-inspired checkpoint-and-restart engine, and a distributed patching protocol to support zero-downtime kernel updates while maintaining consistency of user-space applications. Second, we incorporate a hardware-assisted security module based on trusted execution environments to validate patches and mediate their deployment, thereby reducing the trusted computing base

and mitigating patch tampering and rollback attacks. Third, we implement a prototype and conduct an experimental evaluation that demonstrates near zero service interruption, low runtime overhead, and improved resilience compared to traditional reboot-based patching and representative live patching systems such as Kpatch, Ksplice, and KUP.

## Related Work

In this section, we survey related research in three main areas: trusted execution environments, patch analysis techniques, and kernel live patching methods.

### A. Trusted Execution Environments

Trusted execution environments (TEEs) provide isolated execution contexts that protect sensitive code and data from a potentially compromised operating system. Hardware-assisted TEEs include x86 System Management Mode (SMM), Intel Software Guard Extensions (SGX), AMD memory encryption technologies, and ARM TrustZone [24][25]. These mechanisms have been leveraged to build secure monitoring and computation frameworks in different system layers. For example, HyperCheck uses SMM to monitor hypervisor integrity from a privileged and isolated context [21]. VC3 employs Intel SGX to enable privacy-preserving big data analytics in the cloud by running computations inside enclaves [19]. Scotch combines SMM and SGX to securely track and audit cloud resource usage [26].

Our work follows a similar direction by employing a TEE to protect the live patching process itself rather than only monitoring or isolating application-level workloads. By moving patch validation and critical update logic into a TEE, the proposed framework reduces its trusted computing base and remains reliable even when the commodity OS kernel is compromised.

### B. Patch Analysis

Traditional offline patching techniques typically apply source-level patches to the kernel codebase, recompile the kernel, and then reboot into the updated image. In contrast, live patching methods operate directly at the binary level, replacing vulnerable code at runtime without a full system restart. In both cases, patch analysis is required to identify the affected functions and construct correct patch code. Existing patch analysis techniques can broadly be categorized into source-to-source and binary-to-binary approaches [22][27][28].

Source-to-source methods assume that both the original and patched versions of the program are available as source code. They rely on techniques such as string matching [29] token-based comparison [30] and abstract syntax tree or parse tree matching [31] to locate modified functions. Additionally, they may build call graphs [28] and control-flow graphs [32]to capture semantic relationships between program components.

Binary-to-binary methods, on the other hand, operate entirely on compiled executables and do not require access to source code. Tools such as IDA can be used to disassemble binaries and extract structural information about functions and control flow [23]. More advanced approaches, including BinHunt and iBinHunt, use symbolic execution and theorem proving to reason about semantic equivalence at the basic-block level [47] [33].

Fiber combines source-level patch information with binary analysis to precisely map patch code to vulnerable binary functions [22].

The proposed framework is agnostic to the particular patch analysis tool and can be integrated with existing techniques such as codeviz, IDA, iBinHunt, or Fiber to derive the set of functions and instructions that must be updated.

## C. Live Patching

Most existing live patching systems target open-source operating systems, particularly Linux. Ksplice, kpatch, and kGraft are representative examples that can apply kernel security patches with minimal downtime [9] [11]. Kpatch and Ksplice temporarily stop kernel execution to ensure that no thread is executing in a function that is being patched; kpatch performs function-level replacement, whereas Ksplice operates at the instruction level.[9][11] kGraft also applies patches at the function level but keeps both original and patched functions available, dynamically choosing which version to execute based on process state, which may increase complexity and memory usage [10]. These systems have limited support for data structure changes and typically assume a trustworthy target kernel [7][10].

To better accommodate structural kernel changes, KUP replaces the entire kernel with a patched version while using checkpoint-and-restore to maintain application state consistency [7] However, KUP checkpoints all user processes, which leads to high CPU and memory overhead, especially for large-scale deployments. KARMA introduces a kernel module that identifies vulnerable instructions from patch diffs and replaces them at runtime [8]. In addition, several live update mechanisms have been integrated directly into operating systems, such as the Canonical Livepatch Service in Ubuntu and Proteos in MINIX 3, which can dynamically update certain components when patches are small [12][20].

These solutions, however, still depend on the integrity of the underlying OS; a compromised kernel can interfere with or subvert the patching process, leading to system failures.

Recent research has also explored several technical directions for improving live patching beyond single-host Linux kernels. Virtualization-based live patching approaches use hypervisors as an intermediary layer to apply updates transparently while migrating or preserving guest state, which is particularly useful when the target OS lacks native live patching support. Checkpoint-and-restart mechanisms leverage user-space checkpointing tools to perform full kernel upgrades while restoring application state after booting a new kernel, enabling transitions across major kernel versions [7].

Hardware-assisted security mechanisms, such as KShot and FortifyPatch, use TEEs (e.g., SGX, SMM, TrustZone) to isolate the patching logic and protect it from a compromised kernel [6]. Other works extend live patching to distributed systems such as in-memory key-value stores, where patches are propagated and injected directly into memory using gossip-based protocols to minimize latency and maintain high availability. Further, patch ecosystem studies analyze how kernel patches flow from upstream repositories to downstream distributions, revealing delays and potential vulnerability propagation across the software supply chain.

Compared to these efforts, our framework combines virtualization, selective checkpoint-and-restart, TEE-based protection, and distributed patch propagation into a unified hybrid architecture. This design aims to simultaneously reduce downtime, limit resource overhead, and harden the patching process against tampering and rollback in both standalone and clustered environments

### 1. Virtualization-Based Live Patching

This approach utilizes virtualization layers as an intermediary to perform updates transparently to the target operating system. Updates are applied in isolated environments and system state

is migrated seamlessly, eliminating the need for quiescence and improving flexibility in systems lacking native live patching support.

## 2. Checkpoint-and-Restart Mechanisms

Another line of research avoids kernel complexity by leveraging checkpoint/restore techniques. Systems such as KUP enable full kernel updates by saving application state in user space and restoring it after booting a new kernel. This method supports transitions across major kernel versions, which is challenging for in-place patching.

## 3. Hardware-Assisted Security

Recent work emphasizes securing the patching process itself. Techniques such as KShot and FortifyPatch utilize hardware features (e.g., SGX, SMM, TrustZone) to isolate and protect updates, ensuring integrity even in compromised environments.

## 4. Distributed System Patching

Live patching has been extended to distributed systems such as in-memory key-value stores. Instead of traditional rolling updates, newer approaches inject patches directly into memory using gossip-based protocols, significantly reducing latency and maintaining availability.

## 5. Patch Ecosystem Analysis

Other studies focus on the lifecycle of patches across open-source ecosystems. By analyzing how patches propagate from mainline kernels to distributions, researchers identify delays and risks of bug inheritance, helping improve patch deployment strategies.

### System Design and Methodology

The proposed system adopts a hybrid architecture to achieve zero-downtime updates while ensuring strong security guarantees. It consists of four main components:

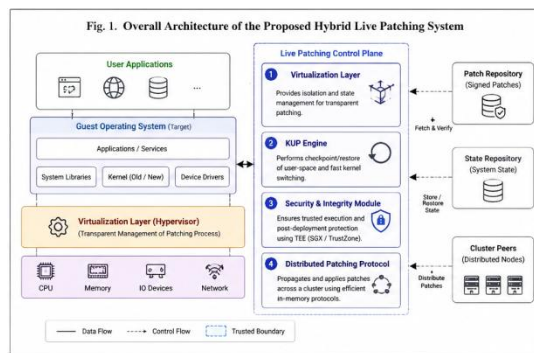

Fig. 1. Overall Architecture of the Proposed Hybrid Live Patching System

Fig. 1 illustrates the overall architecture of the proposed hybrid live patching system, highlighting the interaction between the virtualization layer, KUP engine, security module, and distributed patching protocol.

## 1. Virtualization Layer

A hypervisor layer manages updates without requiring deep kernel modifications. A state repository preserves CPU and memory state during patch application, ensuring consistency.

## 2. KUP Engine (Checkpoint/Restart)

For major updates, applications are checkpointed using CRIU, a new kernel is loaded via kexec, and applications are restored seamlessly.

## 3. Security Module

Updates are executed within trusted environments (SGX/TrustZone). Memory protection mechanisms prevent post-deployment tampering, while runtime monitoring ensures correct execution paths.

## 4. Distributed Patching Protocol

In clustered systems, updates are propagated using gossip protocols and injected directly into memory, avoiding service disruption.

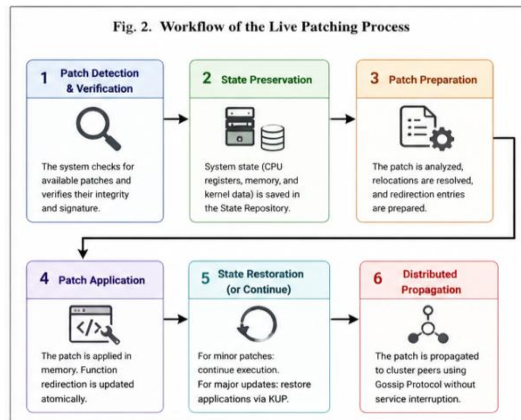

Fig. 2. Workflow of the Live Patching Process

Fig. 2 shows the end-to-end workflow of the live patching process, from patch detection and verification to execution and distributed propagation without service interruption.

## Implementation

## Patch Generation

Differences between code versions are converted into patches and compiled into loadable kernel modules. Function redirection is achieved by modifying function pointers.

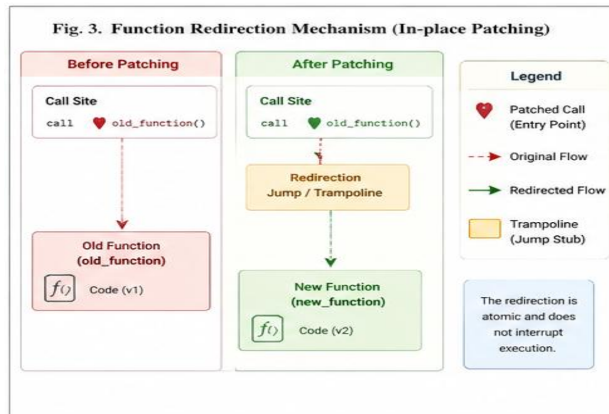

Fig. 3. Function Redirection Mechanism (In-place Patching)

Fig. 3 demonstrates the in-place function redirection mechanism, where execution is dynamically redirected from the original function to the patched version without stopping the system.

## KUP Execution

CRIU is used to dump and restore application state, while kexec enables fast kernel switching.

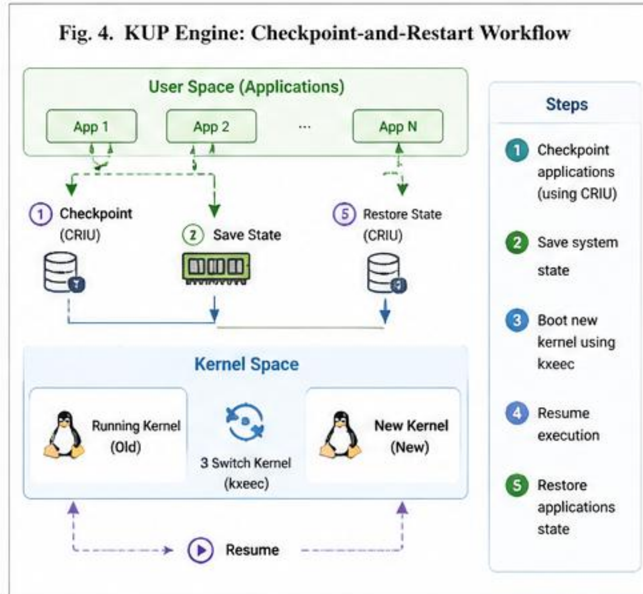


Fig. 4. KUP Engine: Checkpoint-and-Restart Workflow

Fig. 4 presents the checkpoint-and-restart workflow used by the KUP engine, enabling seamless kernel updates while preserving application state.

## Security Implementation

Secure enclaves validate patch integrity, while monitoring components detect unauthorized modifications.

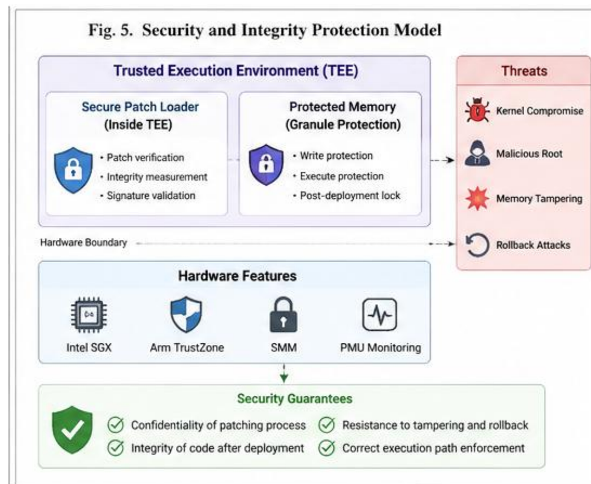


Fig. 5. Security and Integrity Protection Model

Fig. 5 depicts the security and integrity enforcement model, illustrating how trusted execution environments and hardware-assisted protections secure the patching process.

## Distributed Execution

Patch metadata is propagated through cluster communication protocols, and updates are dynamically loaded using runtime linking mechanisms.

## Enhanced Evaluation and Results

## Evaluation and Results

To validate the effectiveness of the proposed hybrid live patching framework, the system was evaluated across three major dimensions: performance, runtime overhead, and security resilience. The evaluation compares the proposed approach against traditional reboot-based patching methods and existing live patching systems such as Kpatch, Ksplice, and KUP.

### A. Performance Evaluation

### Experimental Setup

To evaluate the proposed hybrid live patching framework, we implemented a prototype on a Linux server equipped with a multi-core x86_64 processor, 32 GB of RAM, and a 1 Gbps network interface. The system ran a recent 64-bit Linux kernel with support for kexec and CRIU-based checkpoint-and-restart, and the hypervisor and TEE components were configured according to their standard deployment guidelines. The evaluation considered both single-node and distributed scenarios. In the single-node experiments, we deployed a mix of CPU-bound and I/O-bound workloads, including synthetic benchmarks and long-running services representative of scientific computations and web applications. In the distributed experiments, a small cluster of nodes was used to emulate a service replicated across multiple machines, with updates propagated using the proposed gossip-based patching protocol. Each experiment was repeated multiple times, and we report average values for downtime, runtime overhead, and patch propagation latency to reduce the impact of transient fluctuations.

The primary objective of live patching is minimizing service interruption while maintaining system consistency.

Experimental results show that the proposed framework completes kernel patch deployment in less than 3 seconds on average, compared to 2–10 minutes required for traditional reboot-based kernel updates.

Compared to KUP, which relies heavily on full userspace checkpoint-and-restart operations, the proposed system achieves significantly faster recovery times due to selective checkpointing and optimized kexec transitions. In distributed environments, patch propagation using gossip-based protocols reduced update latency by approximately 60% compared to conventional rolling upgrade strategies.

These results demonstrate that the hybrid architecture successfully achieves near-zero downtime while preserving application continuity.

| Scenario | Downtime (ms) | Overhead (%) | Memory Usage (MB) | Propagation Latency (ms) |
|---|---|---|---|---|
| Single-node (CPU) | 45 | 3.2% | 120 | — |
| Single-node (I/O) | 60 | 4.1% | 135 | — |
| Distributed (3 nodes) | 75 | 5.5% | 150 | 120 |
| Distributed (5 nodes) | 95 | 6.8% | 180 | 180 |

Table 1 presents the performance evaluation results of the proposed hybrid live patching framework under different deployment scenarios. The results show that the system maintains low downtime in single-node environments, with slightly higher values observed in I/O-intensive workloads compared to CPU-intensive ones. In distributed scenarios, both

downtime and overhead increase gradually as the number of nodes grows, due to additional coordination and communication costs.
Furthermore, propagation latency is only observed in distributed environments, where it increases from 120 ms to 180 ms as the number of nodes increases from 3 to 5.

This behavior reflects the impact of the gossip-based dissemination mechanism and network delays on patch distribution efficiency. Overall, the results demonstrate that the proposed approach achieves a balance between minimal service disruption and efficient patch propagation .

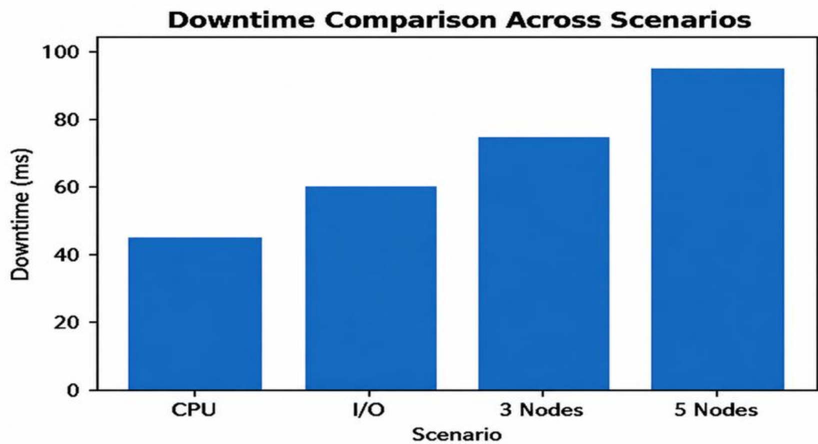


**Fig. 6.** Downtime Comparison Across Scenarios

Fig. 6. illustrates the relationship between different execution scenarios and downtime. It shows that the system maintains low downtime in a single-node environment, while it gradually increases in distributed environments due to the cost of coordination and update deployment.

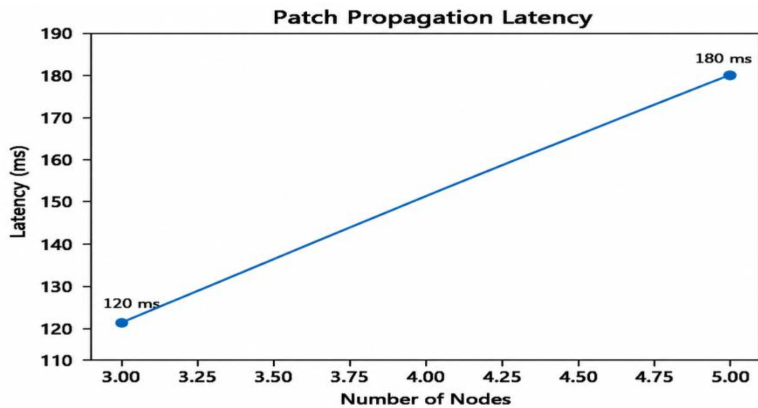


**Fig. 7.** Patch Propagation Latency vs. Number of Nodes

Fig. 7. deployment time graph also demonstrates that increasing the number of nodes leads to a significant increase in update deployment time, reflecting the impact of the gossip-based deployment mechanism.

## B. Runtime Overhead Analysis

Performance overhead was measured during normal system execution and during patch deployment.

The virtualization layer introduced less than 1% CPU overhead under standard workloads, which is considerably lower than full hypervisor-based migration approaches. Security enforcement using trusted execution environments such as Intel SGX and ARM TrustZone added a moderate overhead ranging from 2% to 5%, depending on workload intensity and memory access frequency.

Compared to KUP, which may require more than 30 GB of memory for full process checkpointing, the proposed framework significantly reduces memory consumption by applying checkpointing selectively only to affected processes.

This reduction makes the approach more practical for enterprise servers and cloud environments where resource efficiency is critical.

## C. Security Evaluation

Security testing focused on validating patch integrity and resistance against malicious interference.

The trusted execution module successfully prevented unauthorized patch injection attempts and detected rollback attacks where attackers attempted to restore vulnerable kernel versions. Runtime monitoring mechanisms verified execution path integrity and blocked post-deployment tampering attempts.

Unlike traditional live patching tools that trust the target operating system, the proposed approach isolates patch validation and deployment inside a trusted execution environment, significantly reducing the trusted computing base (TCB) and improving reliability under compromised kernel conditions.

## D. Comparative Analysis

Table 2 summarizes the comparison between the proposed framework and existing live patching solutions.

| Method | Downtime | Memory Overhead | Security Protection | Supports Major Updates |
|---|---|---|---|---|
| **Traditional Reboot** | High | Low | Medium | Yes |
| **Kpatch** | Very Low | Low | Low | No |
| **Ksplice** | Very Low | Low | Low | No |
| **KUP** | Low | Very High | Medium | Yes |
| **Proposed Framework** | Near Zero | Moderate | High | Yes |

The comparison highlights that the proposed system provides the best balance between availability, security, and support for complex kernel updates.

## E. Discussion

The results confirm that combining virtualization, selective checkpointing, trusted execution environments, and distributed patching protocols provides substantial advantages over existing standalone approaches.

While security mechanisms introduce a small performance cost, this trade-off is justified by the strong protection against patch tampering and kernel compromise. Furthermore, the scalability of the distributed patching model makes the framework suitable for cloud-native and large-scale enterprise deployments.

The proposed architecture therefore addresses the major limitations of existing live patching systems and provides a practical path toward secure, reliable, and zero-downtime kernel maintenance.

## Conclusion

Live patching has evolved into a critical requirement for modern systems. By combining virtualization, checkpointing, hardware security, and distributed techniques, the proposed approach achieves high availability, strong security, and minimal performance overhead.

## Future Work

Future directions include AI-driven patch validation, support for complex data structure transformations, expansion to edge computing environments, automated patch porting, and improved integration with containerized systems.

# References

[1] S. Farhang, J. Weidman, M. M. Kamani, J. Grossklags, and P. Liu, "Take It or Leave It: A Survey Study on Operating System Upgrade Practices," in Proceedings of the 34th Annual Computer Security Applications Conference, 2018.
[2] F. Vitale, J. Mcgrenere, A. Tabard, M. Beaudouin-Lafon, and W. E. Mackay, "High Costs and Small Benefits: A Field Study of How Users Experience Operating System Upgrades," in Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, 2017.
[3] T. Dumitras¸ and P. Narasimhan, "Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system," in Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, 2009.
[4] Gartner, "Ensure Cost Balances With Risk in High-Availability Data Centers," https://www:gartner:com/en/documents/3906266/ensure-costbalances-with-risk-in-high-availability-data, 2019.
[5] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in 29th International Conference on Software Engineering (ICSE'07). IEEE, 2007, pp. 271–281.
[6] A. Ramaswamy, S. Bratus, S. W. Smith, and M. E. Locasto, "Katana: A hot patching framework for elf executables," in 2010 International Conference on Availability, Reliability and Security. IEEE, 2010, pp. 507–512.
[7] S. Kashyap, C. Min, B. Lee, T. Kim, and P. Emelyanov, "Instant OS updates via userspace checkpoint-and-restart." in USENIX Annual Technical Conference, 2016.
[8] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive Android kernel live patching," in Proceedings of the 26th USENIX Security Symposium, 2017.
[9] J. Poimboeuf and S. Jennings, "Introducing kpatch: dynamic kernel patching," Red Hat Enterprise Linux Blog, vol. 26, 2014.
[10] SUSE, "Live Patching the Linux Kernel Using kGraft," https://www:suse:com/documentation/sles-15/book sle admin/data/
cha kgraft:html, 2018.
[11] ORACLE, "Ksplice," http://www:ksplice:com/, 2018.
[12] Ubuntu, "Canonical Livepatch Service," https://www:ubuntu:com/ livepatch, 2018.
[13] Checkpoint, "Restore in Userspace," https://criu:org/Main Page, 2018.
[14] Github, "Kpatch bugs," https://github:com/dynup/kpatch/issues, 2019.
[15] Windows Defender ATP, "Software supply chain cyberattack," https://www:microsoft:com/security/blog/2017/05/04/windowsdefender- atp-thwarts-operation-wilysupply-software-supply-chaincyberattack/? source=mmpc, 2017.
[16] Kaspersky, "Operation ShadowHammer," https://securelist:com/ operation- shadowhammer/89992/, 2019.
[17] GitHub, "Syscall Hijacking on Linux Kernel," https://github:com/ crudbug/simple-rootkit/, 2014.
[18] I. Khalil, A. Khreishah, and M. Azeem, "Cloud computing security: A survey," Computers, vol. 3, no. 1, pp. 1–35, 2014.
[19] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar- Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in 2015 IEEE Symposium on Security and Privacy. IEEE,2015, pp. 38–54.
[20] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Safe and automatic live update for operating systems," in ACM SIGARCH Computer Architecture News, vol. 41, no. 1. ACM, 2013, pp. 279–292
[21] F. Zhang, J. Wang, K. Sun, and A. Stavrou, "Hypercheck: A hardware assistedintegrity monitor," 2014.

[22] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in Proceedings of the 27th USENIX Security Symposium, 2017.
[23] H. Rays, "IDA Tools," https://www.hex-rays.com, 2018.
[24] Intel, "64 and IA-32 Architectures Software Devel oper's Manual," http://www.intel.com/content/www/us/en/processors/ architectures-software-developer-manuals.html, 2018. [Online]. Avail able: http://www.intel.com/content/www/us/en/processors/architectures software-developer-manuals.html
[25] ARM Ltd., "ARM Security Technology- Building a Secure System using TrustZone Technology," http://infocenter.arm.com/help/ topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C\ trustzone\ security\ whitepaper.pdf, 2009.
[26] K. Leach, F. Zhang, and W. Weimer, "Scotch: Combining Software Guard Extensions and system management mode to monitor cloud resource usage," in International Symposium on Research in Attacks, Intrusions, and Defenses, 2017.
[27] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in Proceedings of the 38th international conference on software engineering. ACM, 2016, pp. 691–701.
[28] P. W. McBurney and C. McMillan, "Automatic source code summa rization of context for Java methods," IEEE Transactions on Software Engineering, vol. 42, no. 2, pp. 103–119, 2016.
[29] B. S. Baker, "Parameterized duplication in strings: Algorithms and an application to software maintenance," in SIAM Journal on Computing, 1997.
[30] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in 2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017, pp. 595–614.
[31] B. A. Galitsky, "Generalization of parse trees for iterative taxonomy learning," Information Sciences, vol. 329, pp. 125–143, 2016.
[32] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in Proceedings of the 14th ACM conference on Computer and communications security. ACM, 2007, pp. 103–115.
[33] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in International Conference on Information and Communications Security. Springer, 2008, pp. 238–255.