Runtime Function Interposition and Dynamic Software Updating in Linux: A Literature Review

Parth Pawar*, Krishna Chamarthy**, Khushi Mittal**, Kalpak Gupte**, Dr. Ranjana Agarwal***(Department of Computer Engineering, MIT World Peace University, Pune, India, Email: parthpwr10112004@gmail.com)

----------**********----------

Abstract:

Modern software systems require high availability and minimal downtime. Traditional update mechanisms require applications to restart after modifications, which can interrupt system operations. Runtime software modification techniques allow developers to update, monitor, or analyze software systems without restarting them. This paper presents a literature review of runtime modification techniques in Linux systems. The study covers dynamic software updating frameworks, binary instrumentation systems, kernel live patching mechanisms, and modern observability tools such as eBPF. Key research contributions including POLUS, Kitsune, Luci, Dyninst, Valgrind, Intel Pin, and Ksplice are analyzed. The advantages and limitations of each approach are discussed, followed by a comparative analysis of runtime modification techniques. The review highlights research gaps in existing systems and identifies the need for lightweight user-space runtime function interposition frameworks capable of modifying running programs safely and efficiently.

Keywords — **Runtime Function Interposition, Dynamic Software Updating, Binary Instrumentation, Linux Systems, eBPF, Kernel Patching.**

----------**********----------

I. INTRODUCTION

Modern software applications must maintain continuous operation while receiving updates, bug fixes, and performance monitoring. Traditional update mechanisms require applications to restart after modifications are applied. This process can cause system downtime and service interruption, particularly in critical systems such as servers, cloud infrastructure, and large-scale distributed applications.

To address this issue, researchers have developed techniques that enable runtime modification of software systems. These approaches allow programs to be updated, analyzed, or monitored while they are running. Such techniques include dynamic software updating, binary instrumentation, runtime tracing, and kernel live patching.

Dynamic software updating frameworks allow developers to replace parts of a running program without restarting it. Binary instrumentation frameworks enable runtime analysis by inserting additional instructions into executable code during execution. Kernel patching systems allow security updates to be applied to the operating system kernel without rebooting the system.

Recent developments in observability tools have introduced technologies such as extended Berkeley Packet Filter (eBPF), which allow developers to monitor system behavior with minimal overhead.

Despite significant progress, many existing techniques suffer from limitations such as high runtime overhead, dependence on source code

modification, or restriction to kernel-level updates. Therefore, lightweight user-space runtime interposition frameworks remain an active area of research.

This paper reviews existing research in runtime software modification techniques and analyzes their advantages and limitations.

II. BACKGROUND

A. Dynamic Linking

Linux programs typically use the Executable and Linkable Format (ELF). Many applications rely on shared libraries that are loaded dynamically at runtime. The dynamic linker resolves function references when the program starts or when a function is first called.

This mechanism allows developers to override library functions using techniques such as LD_PRELOAD, which loads alternative library implementations before standard libraries.

B. Function Interposition

Function interposition refers to intercepting function calls and replacing their implementation during runtime. This technique is widely used in debugging, monitoring, and security frameworks.

C. Binary Instrumentation

Binary instrumentation involves inserting additional instructions into executable code during runtime to monitor or modify program behavior.

Popular binary instrumentation frameworks include:

- Valgrind
- Dyninst
- Intel Pin

D. Kernel Live Patching

These frameworks enable developers to analyze program execution without modifying source code.

Kernel patching systems allow updates to be applied directly to the running operating system kernel without rebooting the system.

E. Runtime Tracing

Modern Linux systems provide tracing frameworks such as eBPF, which allow developers to monitor system events with minimal performance overhead.

III. LITERATURE REVIEW

Several research efforts have proposed techniques for runtime modification, monitoring, and updating of software systems without interrupting program execution. These approaches can generally be categorized into dynamic software updating frameworks, binary instrumentation systems, kernel patching mechanisms, and runtime observability tools.

One of the early systems designed for runtime software updating is POLUS, proposed by Chen et al. (2007). POLUS introduced a flexible architecture that enables live software updates while the system is running. The framework allows both the old and new versions of program data structures to coexist temporarily during the update process. This design enables a smooth transition between versions and reduces the risk of system failure during updates. POLUS implements mechanisms that detect memory access violations through signal handling to synchronize updates and ensure consistency of program state. Although this method improves system availability by avoiding program restarts, it can introduce performance overhead due to signal-based synchronization mechanisms. Additionally, the complexity of maintaining multiple versions of data structures can increase implementation difficulty.

Another significant contribution in the area of dynamic software updating is Kitsune, proposed by Hayden et al. (2012). Kitsune is specifically designed to support runtime updates in programs written in the C programming language. The framework introduces the concept of dynamic update points, which are predefined locations in the program where updates can be safely applied. When an update is triggered, the program transitions to a new version through a state

transformation process that converts existing data structures into formats compatible with the updated program. Kitsune provides mechanisms for state migration, allowing developers to define transformation functions that map old data to new data structures. This approach allows complex program updates while preserving application state. However, Kitsune requires developers to manually insert update points and transformation logic into the program source code, which increases development complexity and reduces transparency.

A more recent approach to dynamic software updating is Luci, introduced by Heinloth et al. (2023). Luci integrates runtime updates directly into the dynamic linking process of Linux systems. The system modifies the dynamic loader to allow updated shared libraries to replace existing ones during program execution. This approach avoids the need for explicit update points or source code modifications, making it more transparent to developers. Luci also reduces runtime overhead by leveraging the existing dynamic linking infrastructure. However, the system primarily focuses on updating shared libraries and may not fully support complex updates involving large structural changes in application logic or data structures.

In addition to software updating frameworks, binary instrumentation systems provide powerful capabilities for analyzing and modifying running programs. One of the most widely used frameworks in this domain is Valgrind, introduced by Nethercote and Seward (2007). Valgrind operates by translating program instructions into an intermediate representation and executing them within a virtual instrumentation environment. This approach enables developers to detect memory errors, memory leaks, and concurrency issues during program execution. Valgrind provides several analysis tools, such as Memcheck and Cachegrind, which assist developers in identifying performance bottlenecks and memory-related bugs. Although Valgrind offers comprehensive analysis capabilities, it introduces significant runtime overhead due to instruction translation and instrumentation, which makes it unsuitable for production environments.

Another widely used binary instrumentation system is Intel Pin, developed by Luk et al. (2005). Pin provides a dynamic instrumentation framework that allows developers to build custom program analysis tools. The system intercepts program instructions during execution and inserts instrumentation routines to collect runtime information such as memory accesses, control flow, and system interactions. One of the key advantages of Pin is its flexibility, as it allows developers to create specialized analysis tools for tasks such as performance profiling, malware analysis, and program debugging. However, like other instrumentation frameworks, Pin introduces execution overhead due to the additional instrumentation code inserted into the running program.

Dyninst, proposed by Buck and Hollingsworth (2000), provides another approach for runtime code modification and instrumentation. Dyninst offers an application programming interface (API) that allows developers to insert or modify instructions in running programs without requiring a restart. The framework performs binary analysis to identify safe insertion points for instrumentation code and enables developers to monitor program behavior dynamically. Dyninst is widely used in performance analysis tools and debugging systems. However, implementing instrumentation using Dyninst requires detailed knowledge of binary structures and program execution, which can increase development complexity.

Bruening et al. proposed Transparent Dynamic Instrumentation (2012) as a method for performing runtime binary rewriting while maintaining program correctness and transparency. The framework uses dynamic translation techniques to intercept and rewrite instructions during execution. It ensures that the modified program behaves identically to the original program while enabling the insertion of analysis code. This approach is particularly useful for developing debugging and security analysis tools. Nevertheless, the use of

dynamic translation and code caching introduces additional memory usage and execution overhead.

Research on runtime modification techniques has also extended to operating system kernels. Ksplice, introduced by Arnold and Kaashoek (2009), enables Linux kernel updates to be applied without requiring a system reboot. Ksplice analyzes source-level patches and converts them into binary updates that can be applied directly to the running kernel. This approach significantly reduces system downtime and is widely used in enterprise environments where high availability is critical. However, Ksplice focuses primarily on kernel-level updates and does not address runtime modification of user-space applications.

More recently, advances in system observability have introduced new runtime monitoring mechanisms based on extended Berkeley Packet Filter (eBPF). Brendan Gregg (2019) introduced a set of tools known as BPF Performance Tools, which leverage eBPF to monitor system behavior with minimal performance overhead. These tools allow developers to trace system calls, monitor application performance, and analyze resource utilization in real time. Unlike traditional instrumentation frameworks, eBPF programs run inside the kernel in a controlled environment, which enables efficient monitoring of system events. However, eBPF-based tools are primarily designed for observation and tracing rather than modifying program execution or redirecting function calls.

Overall, the reviewed literature demonstrates that numerous techniques exist for modifying or analyzing software systems during runtime. Dynamic software updating frameworks focus on replacing program logic without restarting applications, while binary instrumentation frameworks enable detailed runtime analysis of program behavior. Kernel patching systems provide mechanisms for applying operating system updates without rebooting, and modern tracing tools enable efficient monitoring of system events. Despite these advancements, existing approaches often introduce performance overhead, require source code modifications, or focus primarily on kernel-level updates. These limitations highlight the need for lightweight user-space runtime interposition frameworks capable of safely modifying running applications with minimal overhead.

IV. COMPARATIVE ANALYSIS OF EXISTING TECHNIQUES

The reviewed literature demonstrates that several approaches exist for modifying, analyzing, and updating software systems during runtime. These approaches differ in terms of implementation complexity, runtime overhead, level of control, and applicability to user-space or kernel-space environments.

Dynamic software updating frameworks such as POLUS, Kitsune, and Luci focus on updating applications while they are running. These systems attempt to maintain program state while replacing program logic with newer versions. POLUS achieves this by allowing multiple versions of program data to coexist temporarily during updates. Kitsune introduces update points and state transformation mechanisms to convert old program states to new ones. Luci further improves transparency by integrating update functionality directly into the dynamic linker.

Although these systems allow runtime updates, they often require significant changes to application source code or rely on complex state transformation logic. This requirement limits their usability in cases where source code is unavailable or difficult to modify.

Binary instrumentation frameworks such as Valgrind, Intel Pin, Dyninst, and Transparent Dynamic Instrumentation provide a different approach. These frameworks operate directly on compiled binaries and allow developers to insert additional instructions into running programs. This capability enables detailed analysis of program execution, including memory access patterns, control flow behavior, and system interactions.

While binary instrumentation frameworks are highly flexible and powerful, they typically introduce runtime overhead because each program

instruction may be translated or analyzed before execution. As a result, these frameworks are often used in debugging or testing environments rather than production systems.

Kernel live patching systems such as Ksplice provide mechanisms for updating the operating system kernel without requiring a system reboot. This approach is particularly valuable for applying security patches in production systems where downtime must be minimized. However, these systems are designed specifically for kernel-level updates and do not directly support runtime modification of user-space applications.

More recently, observability tools based on eBPF have emerged as efficient mechanisms for monitoring system behavior. eBPF programs can be dynamically attached to kernel or user-space events, allowing developers to trace system calls, monitor resource usage, and analyze application performance with minimal overhead. However, these tools are primarily designed for monitoring and tracing rather than modifying program execution.

The comparison of these techniques highlights the trade-offs between runtime modification capabilities, performance overhead, and implementation complexity.

V. COMPARISON OF REVIEWED SYSTEMS

TABLE I
COMPARISON OF RUNTIME MODIFICATION TECHNIQUES

System	Technique	Level	Advantages	Limitations
POLUS	Dynamic Software Updating	User-space	Supports runtime program updates	Signal-based synchronization overhead
Kitsune	Dynamic Software Updating	User-space	Supports complex state transformation	Requires source code modification
Luci	Loader-based Updating	User-space	Integrates with dynamic linker	Limited to shared library updates
Valgrind	Binary Instrumentation	User-space	Powerful debugging and memory analysis	Very high runtime overhead
Intel Pin	Binary Instrumentation	User-space	Flexible instrumentation framework	Performance overhead
Dyninst	Binary Instrumentation	User-space	Runtime binary patching	Complex binary analysis required
Transparent Dynamic Instrumentation	Binary Instrumentation	User-space	Safe runtime instruction rewriting	Additional memory overhead
Ksplice	Kernel Live Patching	Kernel	No reboot required for updates	Limited to kernel updates
BPF Performance Tools	Runtime Tracing	Kernel /User	Low overhead system monitoring	Cannot modify program behavior

This comparison illustrates that existing systems address different aspects of runtime software modification, but no single approach provides a lightweight and flexible solution for modifying running user-space programs with minimal overhead.

VI. RESEARCH GAP ANALYSIS

Although numerous techniques have been proposed for runtime software modification, several limitations remain.

First, many dynamic software updating systems require access to the application source code. Frameworks such as Kitsune depend on developer-defined update points and state transformation functions. This requirement reduces transparency and limits the applicability of these systems to legacy applications where source code may not be available.

Second, binary instrumentation frameworks introduce significant runtime overhead. Systems such as Valgrind and Intel Pin analyze or translate program instructions during execution, which can slow down program performance considerably. This overhead makes them unsuitable for production environments where performance is critical.

Third, many runtime update mechanisms focus primarily on kernel-level patching. Systems such as Ksplice enable operating system updates without rebooting but do not address runtime modification of user-space programs.

Fourth, modern observability tools such as eBPF provide powerful tracing capabilities but are designed mainly for monitoring rather than modifying program execution.

These limitations highlight the need for new approaches that provide lightweight runtime modification of user-space programs without requiring source code modifications or introducing significant performance overhead.

VII. FUTURE RESEARCH DIRECTIONS

Future research in runtime software modification should focus on developing techniques that combine the flexibility of binary instrumentation with the efficiency of lightweight runtime patching systems.

One potential direction is the development of user-space runtime function interposition frameworks that allow specific functions within a running process to be redirected dynamically. Such systems could use debugging interfaces such as ptrace combined with instruction-level analysis to safely modify program execution.

Another promising direction is the integration of runtime patching mechanisms with modern observability frameworks. Combining runtime monitoring tools such as eBPF with dynamic modification techniques could enable adaptive systems capable of responding automatically to performance issues or security threats.

Additionally, improving the safety and reliability of runtime patching mechanisms remains an important research challenge. Techniques for detecting safe instruction boundaries, preventing race conditions in multi-threaded applications, and ensuring consistency during updates are critical for developing practical runtime modification systems.

VIII. CONCLUSIONS

Runtime software modification techniques provide important capabilities for updating, analyzing, and monitoring software systems without interrupting program execution. This paper reviewed several major approaches in this area, including dynamic software updating frameworks, binary instrumentation systems, kernel live patching techniques, and modern observability tools.

Systems such as POLUS, Kitsune, and Luci demonstrate the feasibility of updating running programs, while instrumentation frameworks such as Valgrind, Intel Pin, and Dyninst enable detailed runtime analysis. Kernel patching systems such as Ksplice and tracing frameworks based on eBPF further extend the capabilities of runtime system management.

Despite significant advancements, many existing approaches introduce performance overhead, require source code modifications, or focus primarily on kernel-level updates. Future research should focus on developing lightweight user-space runtime interposition mechanisms capable of modifying program behavior safely and efficiently during execution.

ACKNOWLEDGMENT

The authors would like to express their sincere gratitude to Dr. Ranjana Agarwal, Department of Computer Engineering and Technology, MIT World Peace University, Pune, for her valuable guidance, encouragement, and continuous support throughout the development of this literature review. The authors also thank the faculty members of the Computer Engineering department for providing the necessary academic environment and resources required for conducting this study. Finally, we acknowledge the researchers whose work has contributed significantly to the field of runtime software modification and instrumentation techniques.

REFERENCES

[1] H. Chen, J. Yu, R. Chen, B. Zang, and P. C. Yew, "**POLUS: A Powerful Live Updating System**," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, Minneapolis, MN, USA, 2007, pp. 271–281.

[2] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, "**Kitsune: Efficient, General-Purpose Dynamic Software Updating for C**," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented*

Programming, Systems, Languages, and Applications (OOPSLA), Tucson, AZ, USA, 2012, pp. 249–264.

[3] B. Heinloth, P. Wägemann, and W. Schröder-Preikschat, "**Luci: Loader-Based Dynamic Software Updates for Off-the-Shelf Shared Objects**," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2023.

[4] N. Nethercote and J. Seward, "**Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation**," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, 2007, pp. 89–100.

[5] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood,"**Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation**," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, USA, 2005, pp. 190–200.

[6] B. Buck and J. K. Hollingsworth,"**An API for Runtime Code Patching**," *International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[7] D. Bruening, Q. Zhao, and S. Amarasinghe,"**Transparent Dynamic Instrumentation**," in *Proceedings of the ACM SIGPLAN Conference on Virtual Execution Environments (VEE)*, London, UK, 2012.

[8] J. Arnold and M. F. Kaashoek,"**Ksplice: Automatic Rebootless Kernel Updates**," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Nuremberg, Germany, 2009, pp. 187–198.

[9] B. Gregg,**BPF Performance Tools: Linux System and Application Observability**, Boston, MA, USA: Addison-Wesley Professional, 2019.