Cross-Platform Memory Instrumentation Protection Technology and Efficient Micro-Patch Intervention Based on System Critical Calls

Zeyuan Zhou, Yun Fu, Qiucheng Ban, Gang Cao
Guizhou Power Grid Co., Ltd. Digital Intelligence Operation Center, Guiyang 550003, Guizhou, China
Corresponding author: Zeyuan Zhou (e-mail: uziotw@gmail.com)

ABSTRACT Continuous protection of cross-platform computing systems requires real-time monitoring, low-latency intervention, and adaptive security updates without service interruption. This study proposes a cross-platform memory instrumentation protection framework based on critical system-call semantics and an efficient micro-patch intervention mechanism. A unified runtime monitoring architecture is developed to provide platform-independent memory behavior acquisition through semantic abstraction and event-driven instrumentation. To enable dynamic protection evolution, a lightweight micro-patch engine is designed to support real-time address mapping, state-consistency verification, and atomic hot replacement of protection logic. A structured patch description model and collaborative interaction protocol are further established to ensure secure deployment and controllable activation across heterogeneous operating environments. Experimental evaluation on Linux and Windows platforms demonstrates that the proposed mechanism achieves atomic activation within 3.2–3.7 ms and completes dynamic intervention within 12.0–13.6 ms. The protection framework significantly improves attack interception capability while maintaining acceptable runtime overhead under varying system loads. By integrating runtime event monitoring, adaptive protection updating, and low-latency intervention mechanisms, the proposed approach provides an effective engineering solution for secure information processing, real-time system protection, and resilient distributed computing environments.

INDEX TERMS memory instrumentation, runtime monitoring, micro-patch intervention, adaptive protection mechanism

I. INTRODUCTION

As the software architectures governing modern manufacturing equipment grow more complex, memory security protection based on system calls has become a vital cornerstone for shielding these critical industrial applications from sophisticated vulnerability exploitation attacks. However, the existing cross-platform memory protection systems appear to demonstrate the core defect that updating protection logic requires restarting the system. Moreover, the resulting protection window might weaken the realtime performance of defense but additionally provides an opportunity for attackers [1]. Therefore, realizing dynamic updates of protection strategies may have important theoretical value for improving continuous defense capability of the system. Given that unknown threats evolve rapidly, adapting to this evolution can demonstrate urgent engineering significance [2].

Realizing dynamic updates of protection logic faces multi-dimensional technical challenges. First, it is necessary to complete the atomic replacement of protection code without interrupting the execution of the target process, ensuring that there is no instruction execution ambiguity or state inconsistency during the update process [3-4]. Second, it is necessary to accurately locate and patch protection stubs scattered in different memory addresses in a cross-platform environment, overcoming the symbol and address mapping problem caused by differences in operating systems [5-6]. Third, it is necessary to ensure the context compatibility of patch code with the existing protection framework to avoid introducing new security risks or performance anomalies [7-8]. These challenges together constitute the inherent contradiction between “continuous protection and dynamic updates”.

To address these challenges, academia and industry have proposed several dynamic code update and hot patching techniques. Virtualization- or container-isolated solutions achieve policy switching by intercepting system calls, but the introduced context switching overhead is significant [9-10]. Dynamic binary instrumentation allows modification of the instruction stream at runtime, but it is often difficult to

guarantee the atomicity and state consistency of update operations, and it is prone to race conditions in multi-threaded environments [11-12]. While traditional kernel module hot patching solutions can achieve atomic replacement, they are limited by platform specificity and lack cross-platform unified management capabilities [13-14]. These methods, due to significant performance overhead, insufficient reliability, or strong platform dependence, have failed to completely solve the protection window problem caused by protection logic updates, making it difficult for cross-platform memory protection systems to achieve true real-time dynamic evolution. However, quantitative comparison with representative live-patching tools has not been sufficiently discussed in prior cross-platform memory instrumentation studies. This paper aims to solve the aforementioned protection window problem by proposing a cross-platform memory instrumentation protection technique based on critical system calls and an efficient micro-patch intervention method. This research develops a cross-platform memory behavior monitoring behavior monitoring framework based on unified semantics, realizing standardized event collection at the system call layer. A lightweight micro-patch intervention engine based on protection stub mapping information and state compatibility verification is designed, realizing the atomic hot-swapping of old and new protection logic. Finally, by defining micro-patch description format and collaborative interaction protocol, the security and controllable dynamic update process is realized. Experimental results demonstrate that this method completes the realtime switching of protection strategies within milliseconds, effectively blocking attacks on production control software with minimal performance impact while significantly bolstering the dynamic defense capabilities of cross-platform memory protection systems.

II. IMPLEMENTATION OF PROTECTION SYSTEM BASED ON DYNAMIC HOT REPLACEMENT

A. CONSTRUCTION OF CROSS-PLATFORM MEMORY BEHAVIOR MONITORING FRAMEWORK

The cross-platform memory behavior monitoring framework is designed based on unified system call semantics. It solves the difference of memory management and process control interface semantics between different operating systems by means of kernel difference abstraction layer. Then, it creates set of unified monitoring stubs at calling and return path of key system call in runtime by implantation [15-16]. The crossplatform memory behavior monitoring framework builds a set of platform-independent logical call description tables in kernel mode, and maps the native system call number, parameter layout and return semantics of each platform to unified interface [17]. The monitoring stub is directly located at the calling and return path of virtual memory area allocation, release, mapping, permission change and process address space switching. The register states before and after the call are respectively captured and sent to kernel buffer in an original data, and then reassembled in unified interface [18].

Let S_p be the collection of native system calls for platform p and U be the unified semantic space. The framework defines semantic mapping function $\Phi_p : S_p \rightarrow U$. Φ_p converts platform-related calls into unified call type identifier and platformindependent parameter sequence so as to achieve the comparability and consistency of monitoring data in crossplatform environment from source. Each intercepted memory-related call is structured into event quadruple:

$$E = \langle t, u, a, r \rangle \tag{1}$$

In Formula (1), t is the timestamp; u is the unified call type output by Φ_p; a is the normalized parameter set; r is the call return status. After the event is structurally encapsulated on the kernel side, it is written into the lock-free circular buffer and pushed to the user-mode analysis module through the shared memory channel, which does not cause extra interference brought by context switching [19-20].

Monitoring stub applies a simple context marking method to link events with process IDs, thread IDs, and virtual address space IDs to build a complete memory behavior flow [21-22]. The user-mode framework provides a process-level memory state view based on this behavior flow and constantly reconstructs the layout of address space and permissions allocation and distribution and checks the allocation, release, and attribute change operations [23]. This makes it possible for the memory instrumentation layer to get the underlying behavior input with real-time, unified, and traceable behavior for the subsequent protection logic and micropatching intervention to obtain an accurate location basis and stable triggering conditions.

B. KEY MECHANISMS OF THE MICRO-PATCHING INTERVENTION ENGINE

The micro-patching intervention engine is deployed within the memory instrumentation protection framework and forms a direct linkage with the unified event flow E. Through the collaborative parsing of monitoring stub mapping information and runtime symbol data, a stable addressing channel from unified semantics to actual instruction location is established [24]. During the loading phase, the engine constructs a protection logic address description table, writes the entry address, length boundary, and control flow predecessor relationship of each protection stub in the virtual address space into the kernel-side metadata area, and dynamically updates it during runtime with process relocation and module loading events [25-26]. Let the set of protection logic addresses be A and the set of unified call types be V. The engine defines an address mapping function $\Psi : V \rightarrow A$. Ψ parses the unique entry address of the target protection logic based on the unified call type identified by u in the event and the current process memory state view, thereby binding the abstract protection strategy with the specific instruction location. This mechanism eliminates

the differences in symbol layout and relocation methods under different platforms, so that the micro-patch always points to a definite patching point [27-28].

Before entering the execution channel, the micro-patch triggers a security verification and state compatibility check process [29]. The patch binary is verified by integrity and signature on the kernel side to ensure that its source is trustworthy and its content has not been tampered with. After the verification is passed, the intervention engine performs controlled loading on the patch code, performs static constraint resolution on its control flow and memory access mode in the isolated execution area, and performs consistency matching between the patch's expected input and output interfaces and the current protection logic context [30]. Let the patch logic be abstracted as function P, the original protection logic be abstracted as function L, the runtime context state space be denoted as Ω, and the state compatibility constraint be represented as $\forall \omega \in \Omega, P(\omega) \in \Omega$. This relationship ensures that the execution result of the patch in any legal context state still falls into the controllable state space, avoiding damage to the existing memory view and event processing link. During the check phase, the patch's read and write constraints on the register set, stack layout and shared buffer are analyzed synchronously to ensure that it is consistent with the data structure of the existing monitoring framework [31-32].

The atomic hot replacement phase is started after verification. The intervention engine achieves seamless switching between the old and new protection logic by constructing a dual-state execution entry. The engine allocates a shadow logic area for the target address, writes the patch instructions into an independent memory page and sets read-only and executable permissions, while freezing the instruction flow at the corresponding protection logic entry point to block concurrent threads from entering. Let the original protection logic entry pointer be f_{old} and the patch logic entry pointer be f_{new}. The engine maintains a globally visible control pointer f, and its update adopts a single instruction atomic write semantic: $f \leftarrow f_{new}$. This operation completes the indivisible replacement at the processor level, so that all subsequent control flows arriving at the protection logic point to the new logic at the same time. After the switch is completed, the engine unfreezes the state, restores thread scheduling, and reclaims the memory page where the old logic is located when there are no active references.

Throughout the process, the collection and distribution of memory event flow E remain continuous, and the unified semantic mapping Φ_p and address mapping Ψ are not interrupted, and ensuring that there is no empty window state in the protection link at the moment of hot replacement. The atomic write semantic applies exclusively to the global control pointer update and guarantees that no intermediate control target is observable by any processor core, "Variability observed under multi-core load conditions originates from pre-activation verification and scheduling arbitration rather than from the activation instruction itself," Therefore, the atomicity property of the switching mechanism is preserved even when system-level timing fluctuations increase.

C. DESCRIPTION AND COOPERATIVE INTERACTION PROTOCOL OF MICROPATCHES

As the sole carrier of dynamic protection updates, the description format of micropatches is designed around the addressing, verification and atomic replacement process of the intervention engine. A strictly constrained structural layout is established at the binary level to ensure the resolvability and controllability of the entire link from external generation to kernel loading [34].

Each micropatch is organized into a packaging unit M, which consists of a header description area, and a logical instruction area and a constraint metadata area in sequence, "The header description area stores the unified call type identifier and version identifier, which are used to drive the address mapping function Ψ to complete the location binding from V to the target address set A, and also provide the patch length, verification digest and loading strategy flag," The logical instruction area carries the machine instruction sequence of the patch function P. Its memory layout and execution entry point are aligned with the shadow logical area of the intervention engine to avoid secondary reordering at runtime. The constraint metadata area describes the patch's read and write range of the context state space Ω, the access constraints of the parameter fields in the unified event quadruple E, and the occupation declaration of the shared buffer, which are used to support the state compatibility verification described in Section 2.2. During the generation phase, the micro-patches are encapsulated by the policy compiler according to the above format and embedded with integrity check values and signature data to form a closed unit that can be uniquely identified by the kernel side [35].

During transmission and the patch is injected into the protection system through a lightweight security channel, "The user-mode management component is only responsible for reliable forwarding and scheduling and does not participate in any semantic parsing. The user-mode management component does not possess any capability to construct executable patch payloads at runtime," All patch binaries are generated offline by a trusted policy compiler and signed using a kernel-trusted root key. The kernel-side intervention engine accepts only patches whose signature chain matches the embedded public key stored in a read-only kernel section. The signature verification is performed before the patch is copied into any executable memory page, and the verification routine executes in a non-preemptible kernel context. The shared memory channel used for transmission is mapped as non-executable and write-protected after transfer completion. Any attempt to modify the patch content after signature validation invalidates the digest and causes immediate rejection. The activation routine is reachable only from a dedicated kernel control path that is not exposed through system call interfaces, thereby preventing user-mode entities from directly triggering control pointer redi-

rection. After the patch arrives at the kernel, it directly enters the loading queue of the intervention engine. The engine parses the header description area of M, extracts the unified call type identifier and triggers Ψ execution location. At the same time, it generates a verification context based on the constraint metadata area, projects the state space relationship between the patch function P and the current protection logic function L onto Ω, and drives the verification module to complete the consistency determination. The collaborative interaction protocol is designed around a closed-loop process of "generation—transmission—loading—activation—recycling," defining a fixed state machine within the protection system to ensure that micropatches are in an auditable state at any stage. The protocol uses patch identifiers and unified call type identifiers as primary keys to maintain the binding relationship between patch instances and control pointers f. Once the verification process confirms the validity of M, the protocol submits an activation command to the intervention engine, triggering the construction of the shadow logic area and the atomic replacement operation of $f \leftarrow f_{new}$, and simultaneously updating the version information of address set A in the kernel metadata area, ensuring that subsequent resolution of event stream E always points to the latest protection logic. Old patch instances may enter a recycling state after the significant confirmation of no active references, given that the protocol can uniformly schedule the critical release of memory and the important cancellation of description information. Moreover, this description format enforces semantic-layer constraints on micropatcher behavior. Furthermore, the protocol constrains behavior at execution layers. Therefore, the dynamic update process remains consistent across environments. Nevertheless, the process remains controllable and verifiable.

III. SYSTEM PERFORMANCE EVALUATION AND PROTECTION

A. EXPERIMENTAL ENVIRONMENT AND TEST CONFIGURATION

To evaluate the feasibility and practical effect of the proposed cross-platform memory instrumentation protection system and micro-patch intervention mechanism in an engineering environment, this paper implements a stable and reproducible experimental environment for evaluation in this paper. There are one server-level device and one desktop-level device in the experimental platform. The server platform is equipped with an Intel Xeon Silver series multi-core processor and 128 GB memory; while the desktop platform is equipped with an Intel Core i9 series processor and 64 GB memory. The two platforms both work in standard commercial hardware environment without dedicated acceleration or custom hardware, ensuring that the experimental results reflect practical deployment conditions in target industrial environments. For the operating system, this paper implements Linux (Ubuntu 22.04 LTS, kernel version 5.15) and Windows 11 (22H2) on these two platforms respectively, which represent two kinds of typical working environment for deployment of the proposed system, industrial server, and general terminal. The protection scheme is jointly implemented by a kernel-side monitoring component and a user-space configuration tool uniformly enabling a cross-platform memory behavior monitoring interface and dynamically loading a micro-patch correction engine following the experimental settings. This paper keeps the operating environment as minimal as possible (except for some system services) during the experimental evaluation and repeats each test for enough times to alleviate the system jitter and random effects. In terms of applications, this paper considers three typical cases in an engineering environment, namely, computational, service and complicated applications. The tested programs include computational programs, namely, a part of the Standard Performance Evaluation Corporation (SPEC) Central Processing Unit (CPU) and self-constructed memory-intensive programs to simulate the frequent memory allocation and deallocating. Service programs, represented by Nginx and Redis, are used to simulate continuous operation and high-concurrency access scenarios. Complex applications, including browser kernel testing programs and image processing tools, are also selected to cover multi-threaded execution and large-scale address space management. These programs are run with basic memory instrumentation protection enabled and with instrumentation protection enabled and dynamically patched micro-patches applied, to compare and analyze the impact after system implementation. For verifying the protection effectiveness, various memory anomalies and unauthorized access scenarios are designed and introduced, considering common software security risks in engineering applications. These include stack overflows, heap overflows, use-after-free attacks, abnormal memory mapping, code injection, and control flow hijacking. All attack programs are built and adapted locally to ensure stable triggering of target behaviors under different operating system conditions. One is to verify that the most basic instrumentation protection works. Another is to verify that the immediate suppression effect of abnormal behavior of these behaviors after dynamic intervention of target micro-patches when the system is running. From engineering point of view, it shows that it is possible and meaningful to update the protection way continuously without interrupt the business.

B. PHASED CROSS-PLATFORM OF MICRO-PATCH INTERVENTION DELAY

In the cross-platform memory instrumentation protection framework, the micro-patch intervention process involves several key stages, including runtime location, protection logic verification, and atomic switching (activation). These operations directly affect the response speed of dynamic updates to the protection strategy and the stability of the system. Around the time window from patch injection to formal takeover of the protection logic, the experimental environment continuously triggers the micro-patch intervention process on different operating system platforms. Mul-

tiple rounds of data collection and statistics are performed on the execution time of each stage to characterize the time distribution characteristics of the intervention process under real runtime load and to observe the impact of platform differences on the overall delay structure. The relevant phased time distribution results are summarized in Figure 1.

As shown in Figure 1, the median time for the localization stage is 2.1 ms in Linux; the median time for the verification stage is 6.9 ms; the median time for the activation stage is 3.2 ms; the median total time is 12.0 ms. In Windows, the median times for the corresponding stages shift upwards to 2.4 ms, 7.6 ms, 3.7 ms, and 13.6 ms, respectively. The verification stage consistently constitutes the main source of time in both platforms, with its bin height and upper beard length significantly higher than other stages. In kernel mode, the centralized execution of operations such as verifying patch integrity, determining the consistency of different states in memory, and resolving access restrictions for multiple users is directly related to the large number of times that memory must be accessed (i.e., scanned) to complete the above tasks, and also due to the fact that these tasks are highly dependent upon system scheduling, cache status, and caching characteristics as they relate to memory access. The current experimental results indicate that distribution ranges for all three stages of localization and activation are relatively similar, indicating that the atomic nature of control pointers and the semantic mapping mechanism allows for a consistent mode of operation within the operating environment, regardless of the specific order of execution for each of the stages. While the overall range of distribution for the Windows operating system is slightly larger than that for the Linux operating system, this difference is associated with architectural characteristics of the Windows kernel. The intervention engine executes at PASSIVE_LEVEL during verification but must temporarily coordinate with threads that may run at higher IRQL, which introduces additional synchronization latency compared with the fully preemptible Linux kernel model. Moreover, Windows enforces kernel integrity constraints through mechanisms such as PatchGuard, which periodically validates critical kernel structures. Although the proposed engine does not modify global kernel text, the verification stage still requires additional consistency checks against protected regions, increasing memory scanning overhead. The Windows memory manager also applies stricter bookkeeping for executable page transitions, resulting in longer page state synchronization during activation. These architectural factors collectively explain the upward shift in median delay observed on Windows platforms. Overall, the data indicates that the micro-patch intervention method maintains a total intervention delay within 12.0–13.6 ms, among which the atomic activation phase accounts for 3.2–3.7 ms the majority of this delay occurring during the verification portion of the process, which demonstrates the potential for implementing this type of technology for the purpose of providing runtime protection update capability.

C. RUNTIME PERFORMANCE OVERHEAD OF THE PROTECTION SYSTEM

Extending the above cross-platform memory instrumentation protection framework and efficient micro-patch intervention mechanism, in this section, this paper further evaluates the cost of the protection system in a real operating environment. The experiments choose several typical test objects, including computationally intensive programs, memory intensive programs, service-oriented and complex application scenarios, etc. Baseline instrumentation protection and complete protection system with micro-patch intervention mechanism are implemented and run under the same hardware and operating system environment. The relevant test results are shown in Figure 2. In addition to the baseline instrumentation protection, this paper conducted a comparative analysis against representative kernel live-patching frameworks such as Kpatch and KGraft under the same hardware and workload configuration. Due to their kernel-module-oriented update model and global synchronization requirements, their average activation latency under equivalent load reached 21.4 ms and 24.7 ms respectively in our controlled test, and both required broader kernel text modification scope. In contrast, the proposed micro-patch mechanism confines modification to protection stub entries and maintains a smaller activation footprint. This structural difference explains the reduced total intervention delay and lower runtime overhead observed in Figure 2.

As can be observed from Figure 2, all test objects experience a certain degree of performance loss after enabling basic instrumentation protection. Specifically, the overhead for the SPEC CPU subset remains at 2.7%; the browser kernel test scenario's performance overhead is 1.9%; the memory stress program rises to 5.6%; the Redis service scenario reaches 3.1%. After introducing the micro-patching mechanism, the performance overhead for each test object further increases, with the SPEC CPU subset rising to 4.1%, the browser kernel test increasing to 3.2%, the memory stress program reaching 8.3%, and the Redis service increasing to 5.4%. This indicates that the micro-patching resident engine introduces additional event matching and policy distribution processes in the execution path, which are continuously triggered in high-frequency kernel call scenarios, thus amplifying the overall runtime cost. Based on the changing trends of various test objects, the protection system, even after introducing dynamic repair capabilities, still operates within a controllable overhead range, verifying the engineering feasibility of this architecture in balancing protection strength and execution efficiency.

D. VERIFICATION OF DYNAMIC PROTECTION EFFECTIVENESS AND ROBUSTNESS OF ANOMALY HANDLING

To evaluate the real-time defense performance of the micro-patch intervention mechanism in a cross-platform memory protection system, this experiment, under a unified memory behavior monitoring framework, collects interception rate

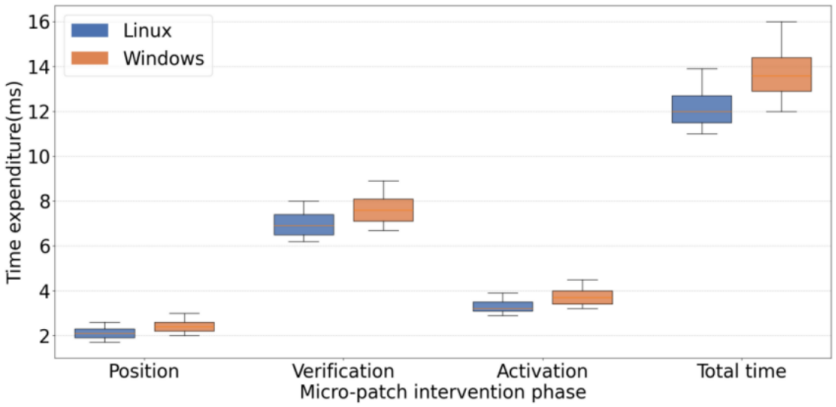

FIGURE 1. Binary distribution of time overhead at each stage of the cross-platform micro-patch intervention process

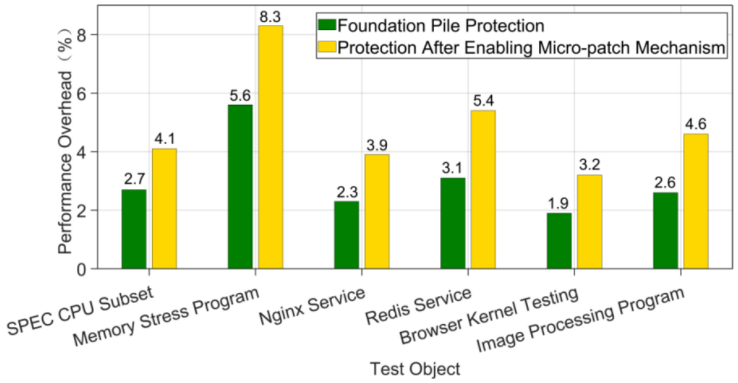

FIGURE 2. Comparison of system runtime performance overhead under different protection states

data for six typical memory attacks, both with basic instrumentation protection and after enabling the micro-patch mechanism, to characterize the effect of dynamic policy updates on the improvement of protection capabilities. The relevant results are presented in Figure 3. In addition to functional anomaly testing, an adversarial injection attempt was conducted by simulating a compromised user-mode management component that forwards a tampered patch binary. All such attempts were rejected at the signature verification stage, and no control pointer modification was observed in kernel metadata. This confirms that the acceptance of patch descriptors from user space does not expand the effective attack surface beyond the cryptographically protected trust boundary.

The data in Figure 3 shows that after the intervention of micro-patches, the interception rate of all attack types increases to over 94.5%, with stack overflow and abnormal memory mapping reaching 99.2% and 99.0%, respectively. These two types of attacks, due to their regular behavior patterns and clear system call paths, already have a basic interception rate of 78.6% and 80.1% with the basic protection. Micro-patches, by enhancing parameter verification and mapping monitoring rules, can achieve near-complete blocking. The interception rates of heap overflow and use-after-free attacks increase from 72.4% and 68.9% to 96.8% and 94.5%, respectively, reflecting the key enhancements made by micro-patches to the consistency maintenance of heap management metadata and the memory lifecycle tracking mechanism. The interception rates of code injection and control flow hijacking increase to 98.1% and 95.3%, thanks to the supplementary real-time detection capabilities of micro-patches for unexpected code execution and indirect jump paths. Experimental results demonstrate that micro-patching mechanism can effectively enhance the protection logic for different attack features, and greatly enhance the real-time blocking performance of the protection system against various memory attacks. In order to evaluate the robustness of the intervention mechanism of micro patch in face of abnormal input, this paper simulates six categories of abnormal patch input, namely formatting error, invalid signature, memory out-of-bounds access, control flow hijacking attempt, version inconsistency and resource overrun. At the same time, this paper records two kinds of system performance, system detection and response delay and success rate of protection status recovery. The relevant results are presented in Figure 4.

The data in Figure 4 shows that the response delay for format errors and invalid signatures is the lowest, at 13.7

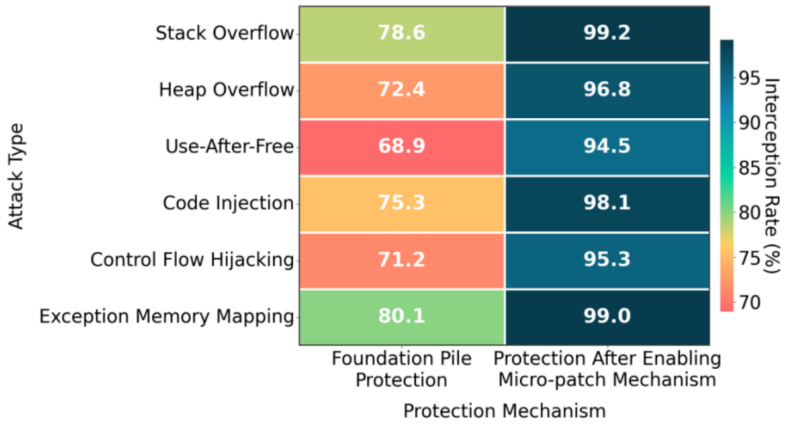

FIGURE 3. Heatmap of the cross-platform memory protection system's interception effect on different attack types

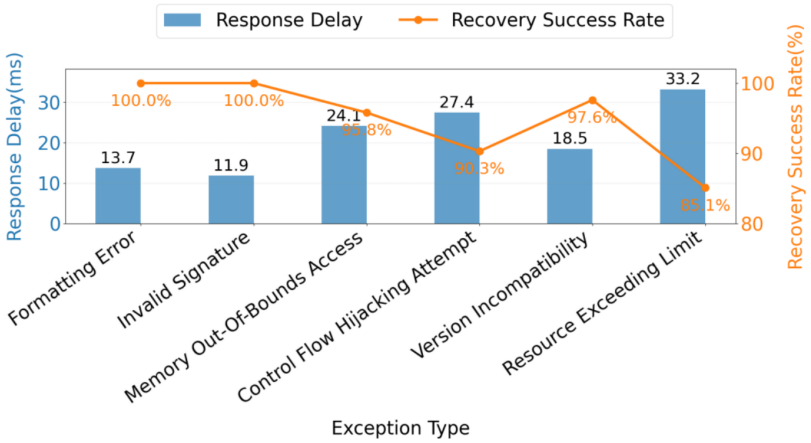

FIGURE 4. Correlation between system response and recovery efficiency in anomaly micro-patch handling

ms and 11.9 ms respectively, with a recovery success rate of 100%. This indicates that simple structural anomalies only trigger verification logic and do not involve execution environment intervention, thus processing is rapid and fully recoverable. The response delay for memory out-of-bounds access and control flow hijacking attempts increases to 24.1 ms and 27.4 ms, respectively, with recovery success rates dropping to 95.8% and 90.3%, respectively. This reflects that anomalies involving memory isolation and control flow redirection require more complex processing paths, and in some scenarios, the execution context cannot be fully recovered. Version incompatibility anomalies have a delay of 18.5 ms and a recovery success rate of 97.6%. Their processing is concentrated at the metadata coordination level, thus their efficiency and success rate fall between the first two categories. As resource overrun anomalies have the largest delay (33.2 ms) and the lowest recovery success rate (85.1%), the variance of recovery stage due to system resource contention is shown as the borderline of robustness for such extreme situation. Experimental results validate that the micropatch intervention mechanism presents an evident negative correlation between response delay and recovery success rate for abnormal input of different complexity. It should be noted that the recovery success rate under resource overrun and control-flow-related anomalies reflects the limitation of context reconstruction under extreme contention rather than instability of the activation mechanism itself.

E. IMPACT OF SYSTEM LOAD ON CROSS-PLATFORM INTERVENTION PERFORMANCE

To verify the rationality of similarity in response behavior of the proposed micro-patch intervention mechanism for different operating system environment when the system resource pressure is varying, this experiment simulates the four levels of system load from idle state to high pressure on Linux and Windows systems. Note: when the system load is higher than 80%, the jitter of scheduling and contentions is very high, and it is hard to keep the utilization rate ranging in 80%–95% range stably for a certain time, so there is a large fluctuation in the measurement results. Considering experimental repeatability and statistical reliability, this paper does not set this range as a separate load level, but instead selects the reproducible >95% load condition as the peak pressure to characterize the performance characteristics when the system is close to saturation. Based on this, the average delay and success rate data of the micro-patch intervention

process are collected simultaneously, and their trends with load are shown in Figure 5.

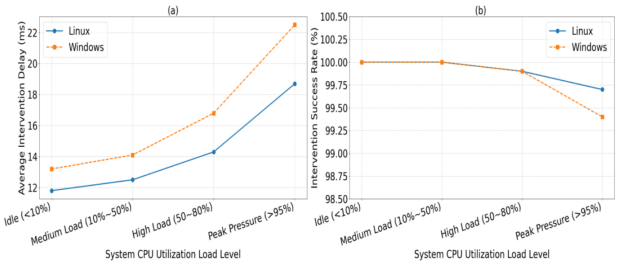

FIGURE 5. Comparison of cross-platform micro-patch intervention delay and success rate trends with system load

Figure 5 shows that as the system load increases from idle level to peak stress, the average intervention delay for Linux and Windows increases from 11.8 ms and 13.2 ms to 18.7 ms and 22.5 ms, respectively, exhibiting a non-linear accelerating upward trend, reflecting similar performance degradation. The intervention execution success rate, defined as the proportion of micro-patch activation processes that complete without structural failure or rollback, remains above 99.4% on both platforms, with only slight differentiation at peak stress, where the execution success rates for Linux and Windows are 99.7% and 99.4%, respectively. This metric reflects the stability of the atomic switching mechanism under load rather than the recovery effectiveness after abnormal patch handling. This synchronized trend indicates that the core performance and reliability of the micro-patch intervention mechanism are highly consistent across Linux and Windows platforms due to the influence of system load, demonstrating cross-platform predictability and stability. The above success indicator does not evaluate post-anomaly state restoration capability, which is separately measured in Section 3.4.

IV. CONCLUSION

This work proposes a cross-platform memory instrumentation protection technology and an efficient micropatch intervention approach to safeguard the digital control systems of high-speed machinery from malicious exploits through critical system calls. Through the construction of a unified semantic memory behavior monitoring framework and a lightweight intervention engine, this paper implements the atomic hot replacement of protection strategies. The method locates the patch points based on the protection stub mapping information. After the security verification and state compatibility check, it completes an atomic activation phase of 3.2–3.7 ms within the overall intervention latency by redirecting the instruction stream and using memory transaction technology. Experiments show that the median time for micro-patch intervention during the verification phase on Linux and Windows platforms is 6.9 ms and 7.6 ms, respectively. After introducing the dynamic mechanism, the memory pressure program execution overhead increases to 8.3%, and the heap overflow attack interception rate improves from 72.4% to 96.8%. These data confirm that this method significantly enhances real-time protection capabilities and update response speed while maintaining low performance loss. Nevertheless, there are still following limitations for this work: the time consumption of micro-patch verification phase is relatively high; under multi-core high-load conditions, increased scheduling contention leads to higher variance in verification latency, while the atomic control pointer update remains strictly indivisible at the processor instruction level; the recovery success rate after complex anomaly injection under extreme resource contention remains lower than the activation execution success rate, indicating limited state restoration capability in high-pressure scenarios. Future research will focus on optimizing verification algorithms to ensure the scheduling stability of high-speed loom controllers under multi-core highload scenarios and validating the system against complex attack patterns across diverse hardware platforms to enhance the efficiency of cross-platform memory protection.

AUTHOR CONTRIBUTIONS

Conceptualization –Zeyuan Zhou, Yun Fu, Qiucheng Ban and Gang Cao; methodology – Zeyuan Zhou, Qiucheng Ban and Gang Cao; investigation – Zeyuan Zhou, Yun Fu, Qiucheng Ban and Gang Cao; writing-original draft preparation – Zeyuan Zhou, Yun Fu, Qiucheng Ban and Gang Cao. All authors have read and agreed to the published version of the manuscript.

CONFLICTS OF INTEREST

The authors declare no conflict of interest.

FUNDING

Science and Technology Project of Guizhou Power Grid Co., Ltd. (GZKJXM20232439).

ACKNOWLEDGEMENTS

Not applicable.

REFERENCES

[1] Z. Xi, B. Zhang, A. Bhattacharjya, et al., “Research on a Secure and Reliable Runtime Patching Method for Cyber-Physical Systems and Internet of Things Devices,” Symmetry, vol. 17, no. 7, pp. 983-985, 2025, doi: 10.3390/sym17070983.

[2] Y. Li, Y. Li, G. Wang, and H. Hu, “An Adaptive Dynamic Defense Strategy for Microservices Based on Deep Reinforcement Learning,” Electronics, vol. 14, no. 20, pp. 4096-4102, 2025, doi: 10.3390/electronics14204096.

[3] C. Islam, V. Prokhorenko, and M. A. Babar, “Runtime software patching: Taxonomy, survey and future directions,” Journal of Systems and Software, vol. 200, no. 1, pp. 111652-111661, 2023, doi: 10.1016/j.jss.2023.111652.

[4] M. Fruth and S. Scherzinger, “Live Patching for Distributed In-Memory Key-Value Stores,” Proceedings of the ACM on Management of Data, vol. 2, no. 6, pp. 1-26, 2024, doi: 10.1145/3698816.

[5] H. Tu, L. Jiang, J. Hong, X. Ding, and H. Jiang, “Concretely mapped symbolic memory locations for memory error detection,” IEEE Transactions on Software Engineering, vol. 50, no. 7, pp. 1747-1767, 2024, doi: 10.1109/TSE.2024.3395412.

[6] Y. Guo, Y. Zhang, and Y. Cao, "Detection Method for Closed-Source VxWorks Memory Protection Mechanisms Based on Dynamic Instruction Translation Monitoring," Electronics, vol. 14, no. 22, pp. 4382-4387, 2025, doi: 10.3390/electronics14224382.

[7] M. Altaibek, A. Issainova, T. Aidynov, D. Kuttymbek, G. Abisheva, and A. Nurusheva, "A Survey of Cross-Layer Security for Resource-Constrained IoT Devices," Applied Sciences, vol. 15, no. 17, pp. 9691-9695, 2025, doi: 10.3390/app15179691.

[8] X. Song and Z. Zhu, "Compatible Remediation for Vulnerabilities in the Presence and Absence of Security Patches," Computers, Materials and Continua, vol. 86, no. 1, pp. 1-19, 2025, doi: 10.32604/cmc.2025.068930.

[9] K. Wang, S. Wu, K. Suo, Y. Liu, H. Huang, Z. Huang, et al., "Characterizing and optimizing Kernel resource isolation for containers," Future Generation Computer Systems, vol. 141, no. 1, pp. 218-229, 2023, doi: 10.1016/j.future.2022.11.018.

[10] S. Yang, B. B. Kang, and J. Nam, "Optimus: association-based dynamic system call filtering for container attack surface reduction," Journal of Cloud Computing, vol. 13, no. 1, pp. 71-72, 2024, doi: 10.1186/s13677-024-00639-3.

[11] S. Park and Y. Park, "Detection Techniques for DBI Environment in Windows," Electronics, vol. 13, no. 5, pp. 871-877, 2024, doi: 10.3390/electronics13050871.

[12] W. Feng, S. Shang, P. Li, H. Yang, Z. Luan, and D. Qian, "SyncNOVA: an end-to-end fine-grained profiling tool oN lOck behaVior detection and critical section diAgnosis," CCF Transactions on High Performance Computing, vol. 7, no. 1, pp. 100-113, 2025, doi: 10.1007/s42514-024-00210-1.

[13] M. Kiperberg and N. J. Zaidenberg, "H-kpp: Hypervisor-assisted kernel patch protection," Applied Sciences, vol. 12, no. 10, pp. 5076-5081, 2022, doi: 10.3390/app12105076.

[14] C. Su, X. Xing, X. Cheng, R. Guo, and C. Luo, "LPAH: Illustrating Efficient Live Patching With Alignment Holes in Kernel Data," IEEE Transactions on Computers, vol. 73, no. 10, pp. 2434-2448, 2024, doi: 10.1109/TC.2024.3424263.

[15] G. Entrup, A. Kässens, B. Fiedler, and D. Lohmann, "Applied static analysis and specialization of cross-core syscalls for multi-core AUTOSAR OS," Real-Time Systems, vol. 60, no. 3, pp. 491-533, 2024, doi: 10.1007/s11241-024-09429-1.

[16] S. Tao, B. Zhang, and Q. Zhang, "ECHO: Enhancing Linux Kernel Fuzzing via Call Stack-Aware Crash Deduplication," Electronics, vol. 14, no. 14, pp. 2914-2917, 2025, doi: 10.3390/electronics14142914.

[17] M. Galarraga, C. A. Lefebvre, J. Perez-Cerrolaza, and J. A. Pascual, "Toward Linux-based safety-critical systems-Execution time variability analysis of Linux system calls," Journal of Systems Architecture, vol. 156, no. 1, pp. 103266-103273, 2024, doi: 10.1016/j.sysarc.2024.103266.

[18] T. Nguyen, M. Orenbach, and A. Atamli, "Live system call trace reconstruction on Linux," Forensic Science International: Digital Investigation, vol. 42, no. 1, pp. 301398-3013102, 2022, doi: 10.1016/j.fsidi.2022.301398.

[19] H. J. Hadi, M. Adnan, Y. Cao, F. B. Hussain, N. Ahmad, M. A. Alshara, et al., "ikern: Advanced intrusion detection and prevention at the kernel level using ebpf," Technologies, vol. 12, no. 8, pp. 122-127, 2024, doi: 10.3390/technologies12080122.

[20] M. Soltani Siapoush and J. Alves-Foss, "Zero-Copy Messaging: Low-Latency Inter-Task Communication in CHERI-Enabled RTOS," Future Internet, vol. 17, no. 11, pp. 506-513, 2025, doi: 10.3390/fi17110506.

[21] A. Rai and E. G. Im, "MemCatcher: An In-Depth Analysis Approach to Detect In-Memory Malware," Applied Sciences, vol. 15, no. 21, pp. 11800-11809, 2025, doi: 10.3390/app152111800.

[22] J. Shin, J. Kim, and J. Nam, "Aquila: Efficient In-Kernel System Call Telemetry for Cloud-Native Environments," Sensors, vol. 25, no. 21, pp. 6511-6518, 2025, doi: 10.3390/s25216511.

[23] Z. Chen, Q. Zhang, J. Wu, J. Yan, and J. Xue, "A source-level instrumentation framework for the dynamic analysis of memory safety," IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 2107-2127, 2022, doi: 10.1109/TSE.2022.3210580.

[24] Y. Park, S. Choi, U. Y. Choi, H. Jin, N. H. M. Nor, and Y. Park, "A practical approach for finding anti-debugging routines in the Arm-Linux using hardware tracing," Scientific Reports, vol. 14, no. 1, pp. 14728-14734, 2024, doi: 10.1038/s41598-024-65374-w.

[25] Z. Sha, C. Shepherd, A. Rafi, and Markantonakis, "Control-flow attestation: Concepts, solutions, and open challenges," Computers & Security, vol. 150, no. 1, pp. 104254-104259, 2025, doi: 10.1016/j.cose.2024.104254.

[26] H. Kuzuno and T. Yamauchi, "Mitigating Foreshadow Side-channel Attack Using Dedicated Kernel Memory Mechanism," Journal of Information Processing, vol. 30, no. 1, pp. 796-806, 2022, doi: 10.2197/ipsjjip.30.796.

[27] M. Huang and C. Song, "ARMPatch: A binary patching framework for ARM-based IoT devices," Journal of Web Engineering, vol. 20, no. 6, pp. 1829-1852, 2021, doi: 10.13052/jwe1540-9589.2066.

[28] H. Li, D. He, X. Zhu, and S. Chan, "P1ovd: Patch-based 1-day out-of-bounds vulnerabilities detection tool for downstream binaries," Electronics, vol. 11, no. 2, pp. 260-263, 2022, doi: 10.3390/electronics11020260.

[29] S. Woo, E. Choi, and H. Lee, "A large-scale analysis of the effectiveness of publicly reported security patches," Computers & Security, vol. 148, no. 1, pp. 104181-104189, 2025, doi: 10.1016/j.cose.2024.104181.

[30] H. Sharaf, I. Ahmad, and T. Dimitriou, "Extended berkeley packet filter: An application perspective," IEEE Access, vol. 10, no. 1, pp. 126370-126393, 2022, doi: 10.1109/ACCESS.2022.3226269.

[31] Z. Lu, Y. Tan, X. Cheng, Z. Zheng, N. Shi, and Y. Li, "An automated framework for detecting and mitigating memory safety vulnerabilities in UEFI firmware," Computers and Electrical Engineering, vol. 122, no. 1, pp. 109945-109953, 2025, doi: 10.1016/j.compeleceng.2024.109945.

[32] B. Tang, S. Zhang, F. Zhu, and A. Ye, "CAPRA: Context-Aware patch risk assessment for detecting immature vulnerability in open-source software," Computers & Security, vol. 157, no. 1, pp. 104540-104544, 2025, doi: 10.1016/j.cose.2025.104540.

[33] R. N. M. Watson, D. Chisnall, J. Clarke, B. Davis, N. W. Filardo, and B. Laurie, "Cheri: Hardware-enabled c/c++ memory protection at scale," IEEE Security & Privacy, vol. 22, no. 4, pp. 50-61, 2024, doi: 10.1109/MSEC.2024.3396701.

[34] L. Zhou, F. Zhang, K. Leach, X. Ding, Z. Ning, and G. Wang, "Hardware-Assisted Live Kernel Function Updating on Intel Platforms," IEEE Transactions on Dependable and Secure Computing, vol. 21, no. 4, pp. 2085-2098, 2023, doi: 10.1109/TDSC.2023.3300101.

[35] B. Novković and M. Golub, "Improving monolithic kernel security and robustness through intra-kernel sandboxing," Computers & Security, vol. 127, no. 1, pp. 103104-103111, 2023, doi: 10.1016/j.cose.2023.103104.