KINTSUGI: Empowering LLMs to Mitigate Web Vulnerabilities via Runtime Policy Injection

Yihao Peng* *Tsinghua University*†
Zizhen Zhu* *Tsinghua University*†
Jiatian Hu* *Tsinghua University*†
Jiaxu Wang* *Tsinghua University*†
Hai Wan‡ *Tsinghua University*†
Xibin Zhao§ *Tsinghua University*†

Abstract

The "response gap" between vulnerability detection and patching leaves web applications exposed to high-impact exploits such as remote code execution, command injection, and server-side request forgery. Current mitigations are flawed: code fixes often break functionality, while global runtime policies produce excessive false positives. We propose KINTSUGI, an automated runtime containment system that provides temporary protection for exploits that manifest as diverging syscall behaviors at the OS level. KINTSUGI uses a three-stage pipeline: First, by performing differential syscall analysis on normal and malicious requests, it locates a few vulnerability-related functions. Subsequently, leveraging LLMs, it precisely delineates the boundaries of the specific code snippets causing the malicious behavior within these functions and establishes policy trigger points. Finally, by analyzing the execution profiles from normal requests, it derives a deterministic, least-privilege syscall whitelist to serve as the runtime policy. This policy is enforced at the kernel level via eBPF and cgroups and is dynamically activated only when the request's execution flow enters the protected code snippet. Evaluation on 27 real-world CVEs across PHP, Python, and Java shows that KINTSUGI effectively neutralizes diverse exploits and their variants while preserving original application functionality. KINTSUGI achieves an average response time of 7.6 minutes. While the enforcement of surgical policies introduces a modest average latency overhead of 9.2% on targeted APIs, the impact on concurrent non-vulnerable traffic remains minimal, with an average throughput (RPS) drop of only 2.21%.

‡Corresponding author: wanhai@tsinghua.edu.cn
§Corresponding author: zxb@tsinghua.edu.cn
*These authors contributed equally to this work.
†Beijing National Research Center for Information Science and Technology (BNRist), Key Laboratory for Information System Security, Ministry of Education (KLISS), School of Software, Tsinghua University, Beijing 100084, China.

1 Introduction

As web applications become increasingly ubiquitous, they face a continuous barrage of cyber threats [3, 50]. Among these, exploits that hijack application logic to interact with the underlying operating system—such as remote code execution (RCE), command injection, unsafe deserialization, and server-side request forgery (SSRF)—are particularly destructive. While modern security infrastructures such as Web Application Firewalls (WAF) and Intrusion Detection Systems (IDS) [11, 22–25, 38, 54, 58] have become adept at capturing malicious requests, the capture of a specific attack is merely the beginning of the response cycle. The process of identifying the underlying vulnerability, developing, testing, and finally deploying a permanent patch remains heavily dependent on manual labor, time-consuming analysis, and deep domain expertise [35, 49, 52]. This critical window—which we term the "vulnerability response gap"—provides attackers with ample opportunity to exploit the same vulnerability through variants. Consequently, it is crucial to achieve automated, immediate mitigation using captured attack requests, providing temporary yet effective protection before a permanent fix is available [30, 36].

Previous solutions and their limitations. Existing research in automated temporary mitigation faces bottlenecks in two key stages:

1) Vulnerability Localization. Pinpointing the vulnerable execution point is the logical starting point. An ideal technique must identify which specific function or code snippet among thousands triggered the malicious behavior. However, existing methods struggle in web environments: (1) Crash-based methods [39, 59, 60] rely on analyzing core dumps after a program crash. They are fundamentally inapplicable to common web logic flaws (e.g., command injection) that do not trigger process crashes. (2) Static analysis methods [8, 44] often require a precise fingerprint (e.g., a specific malicious syscall log) as a starting point. In our scenario, the only input is a high-level http request, leaving the system unaware of where the malicious syscall is triggered. Furthermore, these

methods lack the precision to handle interpreted languages (Python, PHP) and complex framework call chains, leading to high false-positive rates.

2) Mitigation Implementation. Even with localized suspicious regions, current defense paradigms are difficult to deploy in web applications: (1) LLM-based Code Repair [18, 31, 40] attempts to rewrite program logic for permanent fixes. However, their capabilities are often confined to *local syntactic modifications*. When a vulnerability is embedded in functionally necessary logic (e.g., a critical deserialization API), a simple code-level fix is often insufficient. Real-world mitigation would require extensive *architectural refactoring* or a complete redesign of the business logic to maintain both safety and functionality. Such large-scale restructuring far exceeds the current autonomous capabilities of LLMs [12, 16, 41], often leading to broken applications or incomplete defenses if only local patches are applied. (2) Global Runtime Policies [7, 30, 56] enforce uniform syscall whitelists at the application level. This "one-size-fits-all" approach conflicts with the high-concurrency model of web servers. Since syscall streams from multiple requests are interleaved, global policies cannot accurately attribute a syscall to a specific request context, leading to false positives and business disruption. (3) Binary-oriented Localized Protection [10, 26, 27, 36, 57] relies on DWARF debugging info to monitor memory-safety errors. This methodology cannot be migrated to interpreted web applications that lack low-level debugging info and primarily suffer from non-memory logic vulnerabilities.

In summary, the dynamic nature, high concurrency, and non-crashing logic flaws of web applications render existing techniques either "blind" (unable to locate), "ineffective" (unable to apply), or "disruptive" (causing business downtime). We urgently need a new paradigm for request-level precision and fine-grained control.

Our solution. To bridge this gap, we propose KINTSUGI, an automated runtime containment system specifically designed for exploits that leave observable syscall footprints. Starting from a captured malicious request, KINTSUGI automatically localizes the code region responsible for malicious OS-level behavior and deploys a precise, temporary runtime patch. We address three core technical challenges:

1) Localizing vulnerability-related functions in web environments. Pinpointing a single function among thousands is like finding a needle in a haystack. We design a *function-level differential analysis* that operates offline. After capturing a malicious request, KINTSUGI matches it against normal requests from the same API endpoint as a baseline, KINTSUGI compares the execution traces and syscall sequences. By calculating the statistical variance in syscall distributions, we identify functions where behavior significantly "mutates." This trace-based analysis prunes the search space from thousands of functions to a handful of candidates.

2) Pinpointing malicious code snippets within functions. Once suspicious functions are identified, we must lock onto the specific lines driving the malicious behavior. This requires deep semantic understanding. We utilize the LLM as a *"Semantic Pinpointer."* We provide the LLM with the candidate code and the anomalous syscall information. The LLM's task is not to fix code but strictly to "circle" the boundaries of the code snippet most relevant to the malicious syscall. By limiting the LLM to semantic delimitation, we leverage its reasoning while avoiding security risks from generative hallucinations.

3) Generating robust, business-preserving mitigation policies. Relying on LLMs for security rules is unreliable. We propose a deterministic *"LLM-delimited, system-derived"* mechanism. Once the LLM fixes the boundaries, KINTSUGI analyzes the historical traces of that specific snippet under all past normal requests to extract a union of legitimate syscalls. This forms a *least-privilege whitelist*.

Through this design, KINTSUGI deploys a request-level behavioral control engine that surgically constrains the OS interactions of compromised execution paths. Leveraging low-level technologies such as eBPF for syscall filtering and cgroups combined with iptables for network access control, the engine dynamically activates these whitelists upon entering the protected code snippet and deactivates them immediately upon exit. This multi-backend enforcement ensures surgical isolation of critical resources with minimal performance overhead, providing modern web applications with a rapid and reliable capability for automated temporary protection.

Experiments and Evaluations. We evaluate KINTSUGI on 27 critical CVEs. The results demonstrate that KINTSUGI can isolate vulnerability-triggering functions with 86.5% context reduction and generate validated policies in minutes (median 102s). Performance benchmarks show that while active policy enforcement incurs a modest average latency overhead of 9.2% on targeted APIs (with a peak of 29.5% only in syscall-intensive scenarios), the system maintains a negligible impact on the rest of the application (2.21% average RPS drop). This request-scoped isolation allows KINTSUGI to be deployed in high-concurrency production environments where global security measures would be prohibitive.

Our Contributions:

- We introduce KINTSUGI, a novel containment paradigm for high-risk web exploits that shifts from risky code rewriting to *request-scoped runtime policy injection*, leveraging LLMs to surgically constrain OS-level malicious behaviors while preserving original business logic.
- We design a multi-stage mitigation framework that integrates *differential syscall analysis* to pinpoint vulnerability-related functions, a *hybrid policy generator* combining LLM-based delimitation with profile-guided derivation, and a *multi-backend runtime engine* for fine-grained, request-level enforcement.

- We implement a prototype of KINTSUGI and evaluate it on 27 real-world vulnerabilities[1]. Our results show that KINTSUGI neutralizes diverse exploits and variants with a median response time of 102 seconds, maintaining high-performance continuity with only a 2.21% average throughput impact on concurrent legitimate traffic.

2 Motivation

2.1 Accurate Vulnerability Localization

Precise localization is the prerequisite for automated mitigation, yet it remains challenging in web applications. For instance, in CVE-2021-26120 (an SSTI vulnerability in CMS Made Simple [1]), an attacker can trigger a pre-injected malicious template via a standard `GET /index.php` request. While an IDS may flag the request as malicious due to its downstream effects (e.g., an unexpected `execve` syscall), the request payload itself contains no malicious signatures. The core challenge is to trace this high-level malicious intent back to the specific vulnerable function within a complex execution path, enabling the deployment of a surgical mitigation.

Previous solutions. *1) Lack of crash signals.* RCE exploits typically do not trigger program crashes, rendering crash-based localization [39, 60] ineffective. *2) Dynamic complexity.* The interpreted nature of languages like PHP hinders static analysis from constructing accurate Data Dependency Graphs [44] or mapping high-level logic to low-level syscalls [8]. *3) Context explosion.* In our experiments, a single request's execution trace can exceed 58*k* tokens. Directly feeding such massive, noisy codebases into LLMs leads to a failure in pinpointing the exploit-triggering site, resulting in hallucinated or ineffective mitigation strategies.

KINTSUGI addresses this via *Syscall Deviation Analysis*. For CVE-2021-26120, KINTSUGI observes that while `loadCompiledTemplate` normally only invokes file-related syscalls (`openat`, `read`), the malicious execution of the same function triggers process-creation syscalls (`clone`, `execve`). This divergence allows KINTSUGI to pinpoint `loadCompiledTemplate` as the vulnerable function. By pruning the context from 58*k* to 117 tokens (a 99% reduction), KINTSUGI provides the LLM with a high-signal environment for precise reasoning.

2.2 Vulnerability Mitigation

Effective mitigation is difficult when malicious behavior stems from functionally necessary logic. In CVE-2023-41892 (an RCE in Craft CMS [42]), an attacker manipulates a file-loading function (e.g., `include`) via PHP Object Injection. Because file loading is essential for reading configurations and templates, simple removal or modification is impossible.

[1] https://github.com/zzz2602/kintsugi

Previous solutions. *1) Source code modification.* Methods that rely on code rewriting [18, 40] cannot easily replace core logic without risking functional regression. The official patch for this CVE [21], which merely added administrative permission checks, demonstrates the difficulty of addressing the underlying risk without extensive refactoring. *2) Policy-based defense.* Global policies [30] lack request-level awareness and cause false positives in high-concurrency environments. Furthermore, binary-level mitigations [36] are inapplicable to interpreted web languages that lack memory-safety signals.

KINTSUGI moves beyond logic rewriting by implementing *Runtime Policy Injection*. Instead of modifying business logic, KINTSUGI uses LLMs to identify the semantic boundaries of the compromised code and injects policy trigger points.

For CVE-2023-41892, KINTSUGI wraps the file-loading block with entry/exit markers. By analyzing historical traces, the system derives a least-privilege whitelist for this specific region: allowing file-read syscalls (`openat`, `read`) while strictly prohibiting command execution (`execve`). This surgical approach preserves original application functionality while neutralizing the exploit's capability, providing robust protection before a permanent fix is available.

3 Threat Model

KINTSUGI targets a specific class of remote attacks against web applications: those where an adversary exploits vulnerabilities such as Remote Code Execution (RCE), command injection, unsafe deserialization, or server-side request forgery (SSRF) to perform unauthorized operations at the operating-system level. The system operates under the assumption that such exploits will manifest as observable syscall deviations from normal application behavior. The goal of KINTSUGI is to provide immediate, automated, and temporary containment of these OS-level post-exploitation actions, effectively blocking known exploit paths before a permanent logic-level patch is deployed.

Assumptions. Our methodology is built upon the following key premises: *1) Attack Visibility:* We assume that a malicious request successfully triggering the vulnerability has been captured (e.g., by an IDS). *2) Baseline Availability:* KINTSUGI requires a set of normal execution traces from the same API endpoint to serve as a behavioral baseline. These can be retrieved from historical traffic logs or automated unit tests. *3) Behavioral Divergence:* Our core hypothesis is that a successful exploit will exhibit identifiable deviations in its execution path at the syscall level when compared to normal behavior. *4) Trusted Computing Base (TCB):* We assume the underlying Linux kernel (supporting eBPF and cgroups), the auditing modules, and the KINTSUGI framework itself are secure and reside within the TCB.

Scope. KINTSUGI is specifically designed for vulnerabilities whose exploitation inherently produces anomalous syscall

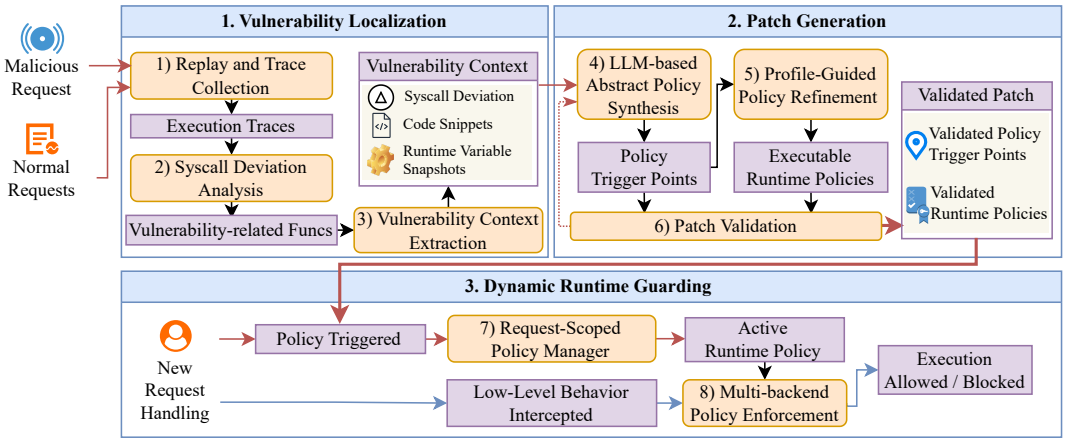

Figure 1: The overall architecture of KINTSUGI.

patterns—a property common to RCE, command injection, unsafe deserialization, SSRF, and similar OS-interacting exploits. This design makes the system effective for functionally necessary code blocks where traditional code deletion or logic rewriting would break core business features. The system does not target vulnerabilities that operate purely at the application-semantic level without OS-visible behavioral changes. These include broken access control, IDOR, most cross-site scripting (XSS), and business logic errors, as well as non-system attacks such as side channels, resource-exhaustion denial-of-service, and hardware-level threats.

4 System Design

4.1 System Design Overview

KINTSUGI is designed to provide automated, context-aware runtime containment for web exploits that produce observable syscall-level behavioral deviations. As illustrated in Figure 1, the architecture comprises three core stages: (1) Vulnerability Localization, (2) Patch Generation, and (3) Dynamic Runtime Guarding.

Vulnerability Localization. This stage operates offline to identify the vulnerable functions responsible for malicious behavior. After a malicious request is captured from the production environment, KINTSUGI initiates a comprehensive analysis in a controlled testing environment. KINTSUGI first performs *Replay and Trace Collection*, replaying the captured malicious request alongside normal baselines to record fine-grained execution traces. Subsequently, *Syscall Deviation Analysis* compares the malicious traces against baselines to pinpoint anomalous execution paths and identify *vulnerability-related functions*. Finally, the *Vulnerability Context Extraction* step aggregates these findings into a comprehensive *Vulnerability Context*, which includes syscall deviations, relevant code snippets, and runtime variable snapshots to support subsequent patch generation.

Patch Generation. Leveraging the LLM's semantic understanding, KINTSUGI performs *Abstract Policy Synthesis* to identify and inject policy trigger points into the source code. Instead of tasking the LLM with writing complex repair logic, we use *Profile-Guided Policy Refinement* to automatically derive deterministic runtime policies from normal traffic behavior. These patches undergo *Patch Validation* to ensure they block attacks without disrupting business logic.

Dynamic Runtime Guarding. Validated patches are deployed to production. When a request hits a trigger point, the *Request-Scoped Policy Manager* activates the corresponding defense rules. The *Multi-backend Policy Enforcement* engine utilizes eBPF and cgroups to intercept and evaluate low-level behaviors in real-time, achieving fine-grained, request-level isolation with minimal overhead.

In the following sections, we first detail the syscall-driven vulnerability localization and context extraction (Section 4.2). We then describe the generation and validation of runtime patches (Section 4.3). Finally, we present the dynamic guarding mechanism for fine-grained behavioral control (Section 4.4).

4.2 Vulnerability Localization

The localization phase identifies the exploit-triggering function within massive execution traces to provide semantic context for policy generation. To bridge the gap between high-level web logic and low-level syscalls in high-concurrency environments, KINTSUGI employs an execution-driven deviation analysis. Instead of relying on crashes or signatures, and inspired by anomaly detection methods [11, 15], we treat localization as a similarity search task to identify runtime behavioral shifts between malicious requests and normal baselines.

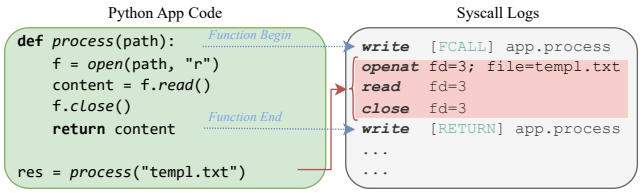

Figure 2: An example of the Function-Syscall Mapping.

4.2.1 Replay and Trace Collection

Once an attack is detected in the production environment, KINTSUGI replicates the scenario in a separate, controlled analysis environment. Within this offline setting, it performs selective instrumentation on the target application to obtain a function-level view of execution.

Multi-dimensional Data Capture. Our instrumentation mechanism implants hooks at function entry and exit points to record the call stack, parameter snapshots, and metadata (e.g., file paths and line ranges). Simultaneously, the system monitors kernel-level syscall events. To ensure temporal fidelity between user-space functions and kernel-space syscalls, the instrumentation hooks inject function context into the kernel event stream via specific syscalls.[2] This enables KINTSUGI to construct a precise "Function-Syscall Map" that records not only which syscalls (e.g., `execve`, `openat`) were triggered by each function but also their specific arguments (e.g., file paths, IP addresses). Figure 2 illustrates an example of this mapping.

Request-scoped Unit Partitioning. In concurrent web environments, syscall streams from multiple requests are interleaved in the kernel. KINTSUGI implements a context-aware partitioning mechanism to achieve request-level isolation:

1. *Thread-level Tracking:* Since modern web servers (e.g., Flask [47], Django [14], PHP-FPM [53]) typically use multi-threading or multi-processing, KINTSUGI uses the Thread ID (TID) as the primary isolation identifier.

2. *Logical Boundary Identification:* Within a single thread, we utilize socket I/O events to delimit http request boundaries. All function and syscalls occurring between the initial socket read and the final response transmission are grouped into a single execution unit.

3. *Asynchronous Model Support:* For asynchronous frameworks (e.g., FastAPI [45]) that utilize coroutines and task queues, KINTSUGI builds upon established causality tracking techniques [32, 37, 55]. By instrumenting points of task creation, scheduling, and coroutine context switching, KINTSUGI performs accurate unit partition by logically reassembling fragmented execution traces into a coherent request-level chain.

[2] In our implementation, we write this context directly to `/dev/null`.

Baseline Selection and Replay. By parsing the route information of the malicious request, KINTSUGI automatically retrieves normal requests targeting the same API endpoint from two complementary sources. The first source is *historical production logs*: http requests that were processed successfully without triggering security alerts, filtered by the target route pattern. The second source is the application's *official unit and integration test suite*, which exercises the endpoint across typical business scenarios, boundary conditions, and expected error paths.

We replay the attacker's prerequisite requests (e.g., authentication) followed by the malicious request in a controlled environment to ensure the application state (database, filesystem) remains consistent with the original attack scenario. This step assumes that prerequisite requests—typically documented in CVE reports or captured by IDS alert chains—are available for reproduction. In practice, reconstructing the exact server state may require snapshotting services or using known setup steps, which is a common prerequisite for any vulnerability analysis system that needs a reproducible trigger.

4.2.2 Syscall Deviation Analysis

This stage identifies the vulnerable function by analyzing behavioral deviations. We first apply path filtering to exclude third-party libraries (e.g., ORMs, logging components) and focus on the application's source code.

Deviation Quantization. KINTSUGI compares the malicious execution flow T_{mal} with the normal baseline T_{norm} by calculating the behavioral deviation of functions f at the same call-stack depth.

Inspired by prior anomaly detection research [15], which suggests that a single similarity metric is insufficient to capture complex web attack patterns, we adopt a similar multi-dimensional approach. We construct a feature vector $\mathbf{D}(f)$ across three dimensions:

1) Resource Set Deviation (D_{set}): Measures changes in the scope of operation targets. We extract the set of resources S associated with syscalls (including syscall types, file paths, and process names) and use the Jaccard index [34]:

$$D_{set}(f) = 1 - \frac{|S_{mal} \cap S_{norm}|}{|S_{mal} \cup S_{norm}|} \tag{1}$$

This effectively identifies unauthorized access to sensitive files or unexpected command execution.

2) Logical Structure Deviation (D_{seq}): Measures syntactic changes in the execution path. We transform the triggered syscall sequence into an n-gram ($n = 3$) [51] frequency vector $\mathbf{v}$ and calculate the cosine similarity:

$$D_{seq}(f) = 1 - \frac{\mathbf{v}_{mal} \cdot \mathbf{v}_{norm}}{\|\mathbf{v}_{mal}\| \|\mathbf{v}_{norm}\|} \tag{2}$$

This metric is highly sensitive to insertions, deletions, or reordering of execution logic.

3) Environmental Entity Deviation (D_{ent}): Specifically targets server-side request forgery (SSRF) and unauthorized outbound connections. We extract sets of unique destination IPs (I) and ports (P) involved in the function. We define the entity ratio as $R(A,B) = \frac{\min(|A|,|B|)}{\max(|A|,|B|)}$ (where $R = 1$ if both sets are empty, and 0 if only one is empty). The deviation is then:

$$D_{ent}(f) = 1 - \frac{1}{2}\big(R(I_{mal}, I_{norm}) + R(P_{mal}, P_{norm})\big) \quad (3)$$

The final anomaly score $Score(f)$ is computed as the weighted average of the multi-dimensional distance components, after applying cross-sample normalization to ensure comparability across different functions and trace instances.

Call-stack De-redundancy. In nested web calls, a deep vulnerability trigger often causes all parent callers to exhibit statistical deviation. To achieve surgical precision, KINTSUGI employs a call-stack anchoring strategy: 1) Syscall deviations are recursively aggregated to their immediate parent functions. 2) The system prioritizes the deepest business function whose deviation exceeds a predefined threshold. This strategy locks onto the "immediate scene" of the deviation, focusing the LLM's attention on the most specific logic while preserving the full call stack for global context.

4.2.3 Vulnerability Context Extraction

Once the vulnerable function is identified, KINTSUGI automatically constructs a structured *Vulnerability Context*. This context provides the LLM with the minimum sufficient information required to resolve ambiguity, effectively mitigating hallucinations caused by input overload. The components of the vulnerability context are summarized in Table 1.

Table 1: Elements of the Vulnerability Context.

Element	Description
Identity	Full function name, file path, and line ranges.
Code Snippet	Complete function source code.
Call Stack	Execution path from the web entry point to the function.
Behavioral Gap	Syscall diff between malicious and normal execution.
Runtime Snapshots	Function parameter snapshots during the attack.

By condensing raw execution traces into this structured format, KINTSUGI significantly reduces the data volume for LLM processing while preserving the full causal chain from low-level behaviors to high-level semantics.

4.3 Patch Generation

After localizing the vulnerable function and extracting the execution context, KINTSUGI generates targeted mitigation patches. Unlike traditional approaches that attempt to rewrite business logic for a permanent fix, KINTSUGI produces a *declarative runtime policy patch*. The core challenge lies in leveraging the LLM's semantic understanding to define defense boundaries while overcoming model randomness and hallucinations. We design a hybrid generation workflow where the LLM guides semantic delimitation, while policy details are automatically derived from historical traffic.

4.3.1 LLM-Guided Semantic Boundary Identification

The primary task of patch generation is to determine where to apply interception. Enforcing strict syscall restrictions on an entire function often interferes with legitimate auxiliary logic, such as logging or exception handling. Therefore, we must identify the minimal high-risk code block within the source code that triggers the malicious behavior.

Prompt Design. We design a structured prompt that frames the LLM as a security expert. The prompt provides the LLM with: (1) *Vulnerability Context*, including the malicious payload and syscall deviations; (2) *Injection Primitives*, such as `syscall_filter_begin/end`; and (3) *Surgical Constraints*, which mandate wrapping only the specific sensitive statements (e.g., `eval`, `include`) to minimize side effects. The complete prompt template and its detailed construction are provided in Appendix C.

Refining Mitigation Placement. In practice, differential analysis may yield multiple high-risk candidate functions. KINTSUGI optimizes the placement of the mitigation using the following logic:

1) Batch Filtering: The LLM performs an initial scan of candidate functions to exclude those where syscall deviations result from normal business fluctuations (e.g., cache updates or timestamping) rather than vulnerabilities.

2) Call-stack Backtracing: This is an engineering optimization. Sometimes the function flagged by differential analysis is a low-level utility (e.g., a generic database wrapper or file handler). Applying restrictions there could lead to global service disruption. KINTSUGI guides the LLM to analyze the call stack and determine if the mitigation should be shifted "upwards" to a more context-aware Controller or Handler. This ensures the defense has access to the original user-input context, enabling more precise control.

Semantic Delimitation Output. The LLM outputs the modified code using a placeholder. For example, in a PHP environment, the LLM wraps dangerous statements like `eval()` or `include()` with trigger markers, as illustrated in Figure 3. At this stage, the patch is not yet active; the specific syscall whitelist is automatically populated in the subsequent phase. This "delimit-then-fill" design effectively decouples the LLM's semantic reasoning from the deterministic execution of the security policy.

Figure 3: Example of LLM-guided semantic delimitation.

4.3.2 Profile-Guided Policy Refinement

Once KINTSUGI identifies the semantic boundaries and injects the placeholder, the system must translate this into a deterministic, executable policy. Relying on LLMs to recall hundreds of low-level syscall names or complex network configurations is error-prone. Instead, KINTSUGI employs a *Profile-Guided Derivation* mechanism to generate robust whitelists from historical execution traces.

Automated Policy Derivation. KINTSUGI first maps the physical location of the injected markers to the source code line range $[L_{start}, L_{end}]$. It then retrieves normal execution traces (as described in Section 4.2) to construct a least-privilege baseline across two dimensions:

1) Syscall Whitelisting: The system extracts the union of syscall types (e.g., `openat`, `write`) and their associated resource targets (e.g., file paths) triggered within the code interval across all normal samples.

2) Network Boundary Derivation: KINTSUGI analyzes socket-related syscalls (`connect`, `sendto`) to determine the network footprint. If the normal profile only interacts with specific IP ranges, the system automatically derives a routing policy. By default, KINTSUGI categorizes network targets into *Internal* (Private IP ranges as defined in RFC 1918 [46], e.g., `10.0.0.0/8`) and *External* (Public Internet). If the normal execution only targets internal databases, the derived policy will strictly prohibit any outbound connections to the public internet during the execution of that block.

While KINTSUGI provides automated defaults, enterprise network topologies can be complex. The system therefore supports an optional *Interactive Refinement* step: before a network-isolation patch is deployed, an analyst can review the automatically derived egress rules and supplement them with organization-specific CIDR blocks, trusted internal services, or API gateway addresses. These additions are merged into the cgroup-based routing policy, causing the sandbox to allow the specified destinations in addition to the profiled ones. This ensures that the mitigation stays aligned with the actual deployment architecture without requiring the analyst to define policies from scratch.

The decision to deploy an eBPF-based syscall filter, a cgroup-based network restriction, or a hybrid of both is automatically determined based on whether the behavioral deviation involves file/process operations or network I/O.

Path Prefix Compression. Restricted code blocks frequently access many legitimate files—for instance, temporary files with randomized names. Writing each individual path into the whitelist would exhaust the limited eBPF map entries and incur high lookup overhead. We address this with a *Path Prefix Compression* step that condenses a concrete set of file paths into a compact set of directory-level prefixes. The idea is to replace paths that share deep common ancestors with a single prefix, thus drastically reducing rule count while granting only the minimal additional privileges required. A greedy algorithm selects the deepest prefixes that cover the largest number of original paths, and a density threshold prevents collapsing scattered files that would open unnecessarily broad access. The full algorithm is described in Appendix D.

4.3.3 Patch Validation

To ensure that the generated runtime patches neutralize threats without compromising application availability, KINTSUGI performs automated validation in an isolated staging environment before deployment. The validation process consists of two primary dimensions:

Security Verification. The system replays the captured malicious request against the patched application. A patch is considered successful if the enforcement engine effectively intercepts the anomalous operations (e.g., `execve`) within the delimited code block. Success is defined by the exploit being neutralized, and the generation of a corresponding audit log entry.

Compatibility Verification. To prevent over-restriction and false positives, KINTSUGI replays all collected normal traffic—both the historical requests and the complete unit test suite—against the patched application in a staging environment. A patch is automatically flagged as *over-protective* if any legitimate request causes a policy violation within the delimited code region (e.g., a benign `openat` call blocked by the syscall whitelist). When this occurs, the system discards the failing patch and re-enters the generation loop: the LLM may be prompted to adjust the code boundaries, or the profile-guided module may recompute the whitelist against an updated baseline, after which the patch is re-validated. This iterative refinement continues until a patch passes both security and compatibility checks, ensuring that occasional but legitimate behaviors observed in the training data are accommodated before the policy reaches production.

Only patches that satisfy both security and compatibility requirements are pushed to the production runtime engine. This ensures a surgical mitigation that excises the malicious capability while preserving the integrity of the business logic.

4.4 Dynamic Runtime Guarding

After patch generation and validation, KINTSUGI enters the dynamic guarding phase. Rather than imposing global restrictions, KINTSUGI employs an on-demand activation mechanism: kernel-level defense backends are dynamically triggered only when the execution flow enters a compromised code region. This design ensures context-aware protection and prevents policy conflicts in high-concurrency environments.

Why Kernel-Level Enforcement? We implement enforcement directly in the kernel (via eBPF and cgroups) rather than inside language interpreters for three practical reasons. First, web applications heavily depend on *opaque native dependencies*—PHP extensions, Python C modules, or Java Native Interface (JNI) libraries—that execute native code invisible to interpreter-level hooks. Kernel-level interception is the only uniform point that covers both interpreted and compiled code paths. Second, high-impact exploits such as deserialization-based remote code execution often spawn *child processes* through `execve` or `clone`. Once a shell or subprocess starts, it runs entirely outside the interpreter, and no runtime-level sandbox can reliably constrain it. eBPF programs, by contrast, are inherited across process forks and can restrict the entire process tree. Third, a single kernel-level engine provides *cross-ecosystem consistency*: it handles PHP, Python, and Java uniformly, whereas interpreter-level sandboxing would require separate, language-specific mechanisms, each with its own bypass surface. This choice does introduce deployment requirements (privileged eBPF installation) which we discuss further in Section 6.

4.4.1 Request-Scoped Policy Activation

To achieve isolation between concurrent requests, KINTSUGI maps the web application's execution model to the kernel's process/thread context.

Policy Signaling and Mounting. Once deployed, the trigger points injected into the source code serve as the communication interface between the user-space application and the kernel-space engine. When a request hits a trigger point, it transmits a control packet containing a policy identifier and a whitelist hash to the kernel via the `prctl` syscall.

Context-Aware Enforcement. Upon receiving a signal, KINTSUGI extracts the execution flow's identity (`tgid`/`tid`) and binds it to the request-specific whitelist via a global eBPF map. This binding initiates a stateful lifecycle: the thread is marked as restricted before entering high-risk logic, enforcing syscall or cgroup-level constraints that transitively apply to any child processes. Upon exiting the vulnerable region, the system atomically clears the restriction status.

Such fine-grained, request-scoped activation ensures that defense policies are strictly confined to the compromised execution path, preserving the performance of concurrent legitimate traffic.

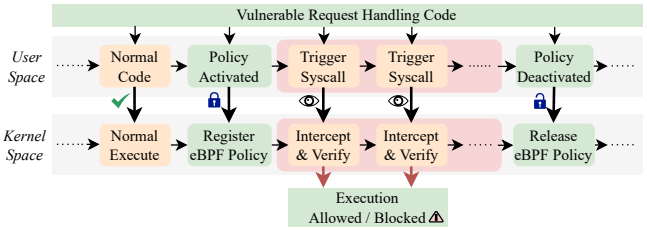

Figure 4: An example of eBPF-based syscall filtering.

4.4.2 Kernel-Level Syscall Filtering via eBPF

As illustrated in Figure 4, upon policy activation, KINTSUGI leverages eBPF to intercept dangerous syscalls in the kernel. To achieve fine-grained parameter control under strict kernel execution constraints, we address both instruction complexity and performance overhead.

Whitelist-based Type Filtering. KINTSUGI's core defense logic is deployed at kernel syscall tracepoints. When a restricted thread initiates a syscall, the eBPF program retrieves its corresponding whitelist. If the syscall type is unauthorized (e.g., `execve` during template parsing), the engine intercepts the execution and forces a return value of `-EPERM`. This approach neutralizes malicious behavior without terminating the process, ensuring service continuity.

Fine-grained Parameter Constraints. Simple type filtering cannot prevent logical abuses, such as using a legitimate `openat` to read sensitive system files. Consequently, KINTSUGI implements path-prefix filtering. However, direct string matching in the kernel faces two bottlenecks: *1) Memory Overhead:* Frequent copying of long paths from user-space to eBPF stack space causes significant performance degradation under high concurrency; *2) Verifier Constraints:* The eBPF verifier limits loop iterations and instruction counts, making variable-length string comparisons inefficient.

To resolve this, we design a lightweight matching mechanism based on pre-computed hashes: *1) Policy Hashing:* During patch generation, path prefixes in the whitelist are pre-computed into fixed-length hashes (e.g., FNV-1a [17]). This significantly compresses the policy storage in kernel maps. *2) Minimized On-demand Copying:* Instead of copying the entire path, the engine reads only the first L_{max} bytes from user-space, where L_{max} is the length of the longest prefix in the whitelist. This truncated read minimizes data transfer. *3) Single-pass Hash Matching:* The eBPF engine performs an incremental hash of the truncated fragment and executes a direct equality comparison against the pre-stored hashes. This transforms non-deterministic string operations into fixed-length memory copies and numerical comparisons, bypassing verifier limitations and ensuring minimal overhead.

4.4.3 Network Resource Constraint via Cgroup

Web attacks such as SSRF often involve unauthorized outbound connections. To restrict network access within compromised code blocks, KINTSUGI utilizes Linux control groups (cgroups) for request-level resource management.

Network Semantic Isolation. Traditional firewall rules are typically bound to processes or IPs, failing to distinguish between concurrent web requests. KINTSUGI utilizes the `net_cls` subsystem to construct a dynamic network sandbox. When a request hits a trigger point, the system migrates the execution flow (via its TID) to a restricted cgroup node. This node is assigned a specific ClassID, allowing the kernel to tag every packet originating from that thread.

Request-level Dynamic Routing. In conjunction with `iptables`, KINTSUGI enforces fine-grained network isolation by tagging packets with the thread-specific ClassID. Once a request enters a protected code block, the engine automatically applies the network policies derived in Section 4.3. These policies typically include RFC 1918 [46] private range restrictions or custom enterprise topology rules defined during the refinement phase. By routing traffic based on these request-scoped tags, KINTSUGI ensures that any unauthorized outbound connections such as SSRF or data exfiltration are intercepted at the kernel level without impacting the connectivity of concurrent legitimate requests.

Hierarchical Inheritance and Cleanup. To handle asynchronous tasks or child processes, KINTSUGI leverages the hierarchical inheritance of cgroups. All descendant tasks of a restricted thread automatically inherit the same ClassID, preventing attackers from bypassing restrictions via reverse shells. Once the code block exits, the thread is moved out of the `net_cls` node, lifting the restrictions. This cgroup-based approach complements eBPF filtering, providing comprehensive behavioral constraints from low-level calls to high-level communication without disrupting network configurations.

Comparison with Global Sandboxing. A natural alternative is to enforce the same syscall and network restrictions globally for the entire application. However, web servers process concurrent requests in interleaved fashion; a single system-call stream observed by a global monitor is a mixture of multiple independent http transactions. Global filtering therefore cannot determine *which request* triggered a particular syscall, making it impossible to apply different policies to different requests without introducing false positives on benign traffic. By anchoring policy activation to LLM-injected trigger points inside the application code, KINTSUGI binds restrictions to the specific execution context that reaches a vulnerable snippet. This request-scoped strategy preserves the concurrency and functional integrity of the rest of the server, as evidenced by the low overhead on non-targeted APIs reported in Section 5.6.

5 Evaluation

We designed our experiments to answer the following research questions:

1) How effectively does KINTSUGI block diverse real-world web attacks compared to LLM-direct repair baselines? (Section 5.2)

2) To what extent can the syscall-based differential analysis narrow down the candidate vulnerable functions and reduce the context size for subsequent LLM analysis? (Section 5.3)

3) How do the semantic boundary identification, profile-guided policy derivation, and the choice of different LLM backends affect the mitigation? (Ablation Study, Section 5.4)

4) How does KINTSUGI perform a full-lifecycle response against a real-world vulnerability? (Case Study, Section 5.5)

5) What is the end-to-end latency of the patch generation workflow and the runtime performance impact on high-concurrency applications? (Section 5.6)

5.1 Evaluation Setup

Benchmark Construction. To evaluate KINTSUGI's effectiveness on the class of exploits it targets, we construct a benchmark consisting of 27 publicly disclosed and reproducible CVEs that involve OS-level behavioral divergence. These span three major web development stacks—PHP (10), Python (10), and Java (7)—and cover remote code execution, command injection, unsafe deserialization, SSRF, and related categories. As summarized in Table 2, the selected CVEs map to 8 out of the *OWASP Top 10 (2025)* [4] categories. Vulnerabilities that do not typically produce syscall-level deviations (e.g., cryptographic failures, logging misconfigurations, pure access-control flaws) are outside the intended scope of KINTSUGI and are therefore not included in the benchmark. For each CVE, we provide a fully functional environment using Docker to ensure reproducibility. For asynchronous scenarios (e.g., Plone's worker queues Y03), we instrument task dispatch and execution points to maintain causality, ensuring request-level policy consistency across thread boundaries.

Patching Patterns. To demonstrate KINTSUGI as a robust alternative when permanent code-level fixes are challenging, we categorize official patches into five types: (A) *Sanitization*, filtering malicious inputs; (B) *Access Control*, adding authentication checks; (C) *Functional Deprecation*, removing vulnerable features; (D) *Structural Refactoring*, large-scale logic changes (e.g., replacing `pickle` with `json`); and (E) *Dependency Update*, upgrading third-party libraries. While types D and E represent significant challenges for automated source-to-source repair tools (e.g., LLM-direct baselines), KINTSUGI is specifically designed to mitigate them via surgical runtime policy injection.

Traffic Generation. Baseline traffic is derived from official unit tests, while malicious traffic uses public PoCs. We manu-

Table 2: Overview of the evaluation benchmark.

ID	CVE ID	App.	Vuln. Type	OWASP	Repair
PHP Ecosystem					
P01	2015-8562	Joomla	Deserialization	A08	Type C
P02	2016-5734	phpMyAdmin	Code Injection	A05	Type D
P03	2018-1000533	GitList	Argument Injection	A05	Type A
P04	2018-7600	Drupal	RCE (Input Handling)	A05	Type A
P05	2021-26120	CMSMS	SSTI	A05	Type B
P06	2022-46169	Cacti	Command Injection	A07	Type D
P07	2023-40033	Flarum	SSRF	A01	Type A
P08	2023-41892	CraftCMS	Object Injection	A05	Type A
P09	2024-25737	VuFind	SSRF (Proxy)	A01	Type A
P10	2024-25738	VuFind	SSRF (Config)	A06	Type A
Python Ecosystem					
Y01	2017-18638	Graphite	SSRF	A01	Type C
Y02	2018-16509	Pillow	Supply Chain	A03	Type E
Y03	2021-33926	Plone	Info Leak (RSS)	A10	Type C
Y04	2022-4223	pgAdmin	Injection	A01	Type B
Y05	2023-37941	Superset	RCE (Pickle)	A08	Type D
Y06	2023-47116	Label Studio	SSRF (DNS Rebind)	A10	Type D
Y07	2023-5002	pgAdmin	Injection	A05	Type D
Y08	2024-39705	NLTK	Deserialization	A08	Type D
Y09	2025-2945	pgAdmin	Code Inj. (Eval)	A05	Type D
Y10	2025-31116	MobSF	SSRF (DNS Rebind)	A10	Type D
Java Ecosystem					
J01	2018-1273	Spring	SpEL Injection	A05	Type A
J02	2022-22947	Spring	Actuator Injection	A02	Type B
J03	2022-22963	Spring	SpEL Injection	A05	Type C
J04	2022-25845	Fastjson	Deserialization	A08	Type C
J05	2022-42889	Commons	Interpolation RCE	A03	Type C
J06	2023-49070	OFBiz	Deserialization	A08	Type B
J07	2024-45507	OFBiz	RCE (XML Load)	A01	Type A

ally craft *Variant Payloads* (obfuscation/encoding) to test the robustness of syscall whitelists against polymorphic attacks.

LLM Configurations. To leverage the latest advancements in reasoning and code understanding, we employ a hierarchy of LLMs as our reasoning backends. For our primary experiments (RQ1, RQ2, RQ4, RQ5), we use *DeepSeek-V3.2* (685B, Dec 2025), a state-of-the-art open-weights model optimized for complex reasoning tasks. To evaluate the robustness of KINTSUGI across different model capacities (RQ3), we also incorporate a suite of models released in 2025: *DeepSeek-V3.1* (685B, Nov 2025) for longitudinal comparison, and the *Qwen-3* series including *Qwen3-235B-A22B* (Jul 2025) and the compute-efficient *Qwen3-32B* (Apr 2025).

Implementation Details. We implemented a KINTSUGI prototype with a Python-based control plane and an eBPF/cgroup-powered defense plane. To support diverse backends, we developed lightweight instrumentation plugins for PHP (Zend extension), Python (`sys.settrace`), and Java (Java Agent). Detailed implementation mechanics are provided in Appendix E. All experiments are conducted on a server with an AMD Ryzen 9 9950X3D 16-Core CPU and 128GB RAM, running Ubuntu 24.04.3 LTS with Linux Kernel 6.14.0.

5.2 RQ1: Effectiveness of Vulnerability Mitigation

In this section, we evaluate the end-to-end mitigation capabilities of KINTSUGI and compare it against two primary baselines: *LLM-Direct Repair* and *Global Policy Enforcement*.

Baseline Configurations. To ensure a rigorous comparison against the state-of-the-art, we define the following baselines:

- *Context-rich LLM-Direct Repair:* We select state-of-the-art source-to-source repair methodologies, specifically drawing inspiration from multi-stage reasoning and CoT prompting techniques in APPATCH [40] and SAN2PATCH [31]. To establish a strong baseline and ensure a fair comparison, we enhance this model by providing it with the exact same fine-grained runtime context (e.g., vulnerable functions and syscall deviations) that KINTSUGI's localization stage identifies. This configuration tests the upper-bound performance of SOTA code-rewriting approaches when given the vulnerability-location information.
- *Global Policy Enforcement (Phoenix [30]-style):* This baseline applies KINTSUGI 's generated syscall and network whitelists at a coarse, application-wide level (via global `seccomp` or `iptables`) rather than KINTSUGI's fine-grained, per-request injection. This directly evaluates the necessity of our contextual enforcement mechanism over simpler, static approaches.

Evaluation Metrics. We define three key metrics: (1) *Success Rate (SR):* The percentage of cases where the mitigation successfully blocks the PoC and its variants. (2) *Pass Rate (PR):* The percentage of official unit tests that pass after the patch is applied, measuring business continuity. (3) *Localization Consistency (LC):* Whether the KINTSUGI injection point resides within the same logical scope as the official fix.

Overall Mitigation Results. As summarized in Table 3, KINTSUGI achieved a 100% Success Rate and 100% Pass Rate across all 27 CVEs. On average, the system requires only 1.07 iterations to generate a valid policy, indicating that the profile-guided refinement is highly deterministic.

In contrast, *LLM-Direct Repair* achieved an overall SR of only 63.0%. We observed that while LLMs can generate simple patches, they often fail to handle complex logic migrations. For instance, in `pickle`-related RCEs (Y05, Y08), LLMs failed to safely replace serialization logic, whereas KINTSUGI successfully neutralized the threat by surgically restricting syscalls within the existing `pickle.load` context.

Localization and Robustness. KINTSUGI achieved an LC of 70.4%. Manual analysis reveals that KINTSUGI often enforces policies at the *Sink* (vulnerable execution point) rather than the *Source* (input entry) targeted by official fixes. This behavioral-level defense makes KINTSUGI resilient against *Variant Payloads* (e.g., obfuscation); while syntactic filters

Table 3: Mitigation Effectiveness and Localization Consistency. *SR* (Success Rate) denotes the percentage of exploits blocked; *Pass* denotes the percentage of unit tests passed. *Loc. Consist.* measures if the injection point matches the logical scope of the official fix.

Repair Type	Count	KINTSUGI		Loc. Consist.	LLM-Direct Repair		Avg. Iter.
		SR	PR		SR	PR	(KINTSUGI)
Type A (Sanitization)	8	100.0%	100.0%	75.0%	62.5%	62.5%	1.12
Type B (Access Control)	4	100.0%	100.0%	50.0%	75.0%	50.0%	1.00
Type C (Deprecation)	6	100.0%	100.0%	66.7%	66.7%	66.7%	1.00
Type D (Refactoring)	8	100.0%	100.0%	75.0%	62.5%	62.5%	1.10
Type E (Dependency)	1	100.0%	100.0%	100.0%	0.0%	100.0%	1.00
Overall	**27**	**100.0%**	**100.0%**	**70.4%**	**63.0%**	**63.0%**	**1.07**

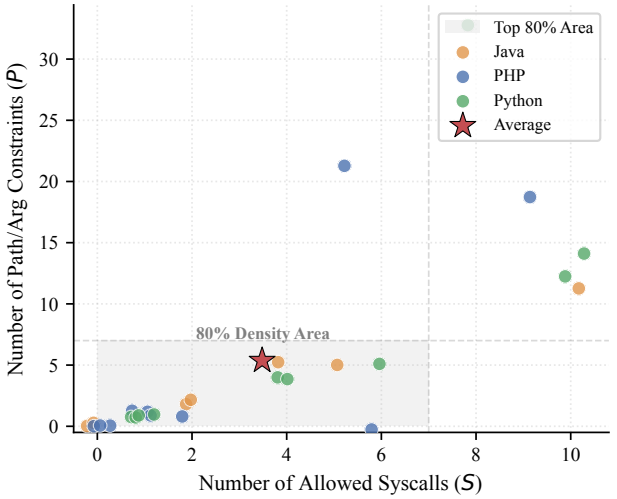

Figure 5: Granularity of generated policies across different ecosystems. The red star ($\star$) denotes the global average ($S = 3.48, P = 5.37$). The shaded area represents the 80th percentile density, showing that most policies are surgically precise with fewer than 6 syscalls and 7 argument constraints.

can be bypassed, the underlying syscall patterns remain constant. Furthermore, for vulnerabilities like DNS Rebinding (Y06, Y10), this sink-based approach avoids the complex, error-prone logic refactoring that LLMs struggle with.

The Failure of Global Policies. The *Global Policy* baseline failed to maintain business continuity (0% PR) in all cases due to semantic conflicts. For example, blocking internal IP access globally to prevent SSRF also severed legitimate database connections. Similarly, restricting `execve` globally prevented the interpreter's bootstrapping. These failures confirm that *Request-scoped Policy Injection* is essential for maintaining availability in heterogeneous web environments.

Policy Granularity Analysis. Figure 5 illustrates the precision of KINTSUGI's policies. On average, a policy consists of 3.48 syscalls and 5.37 argument constraints. PHP policies are the most concise (Avg. $S = 2.5$), while Python/Java require more constraints due to framework complexity. The high density in the 80th percentile area ($S \leq 5, P \leq 6$) demonstrates that KINTSUGI consistently derives surgically precise boundaries.

Table 4: Efficiency of Differential Localization across ecosystems.

Lang.	Tot. Func	Tot. Tok	Cand. Func	Cand. Tok	Reduc.
PHP	434.1	59,290	2.1	900.5	98.0%
Python	126.9	18,797	1.9	765.6	71.6%
Java	70.6	13,883	1.0	618.1	91.3%
Global	**226.1**	**32,520**	**1.7**	**777.3**	**86.5%**

5.3 RQ2: Accuracy and Efficiency of Differential Localization

The effectiveness of KINTSUGI's mitigation relies on its ability to isolate vulnerable functions from the vast execution space of modern web applications. We evaluate this by comparing KINTSUGI's differential analysis against a *Full-Context LLM Localization* baseline, where we provide the LLM with all internal application functions executed during the malicious request.

Complexity Reduction. As summarized in Table 4, a single http request in our benchmark traverses an average of 226.1 unique functions, spanning 32,520 tokens of source code. In extreme cases such as Drupal (P04), the execution path involves over 1,600 functions and exceeds 200k tokens, which far surpasses the reliable reasoning window of even the most advanced LLMs.

KINTSUGI's syscall-based differential analysis achieves surgical precision in filtering this noise. On average, it reduces the search space to just 1.74 high-risk candidate functions, representing a 90.7% reduction in function count and an 86.5% reduction in token volume globally. As shown in Table 4, the reduction is particularly significant in the PHP ecosystem (98.0% token reduction), where large frameworks often introduce massive amounts of boilerplate code that is

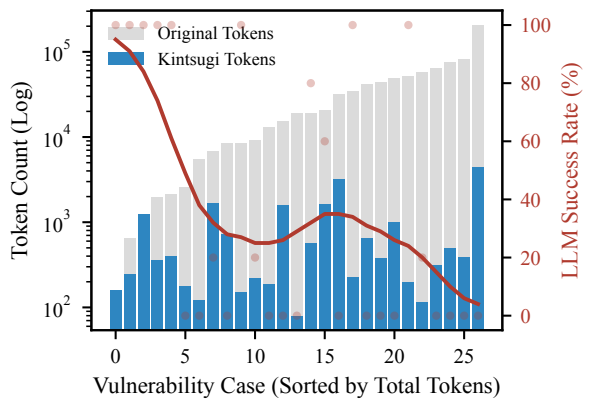

Figure 6: Impact of context size on localization. Grey bars represent the original token count (log-scale), while blue bars represent the reduced tokens after KINTSUGI's localization. The red line shows the smoothed trend of LLM success rate, which decreases as the original context size grows.

irrelevant to the vulnerability.

Impact on LLM Reasoning. To further investigate how this reduction affects mitigation, we visualize the relationship between context size and LLM localization accuracy in Figure 6. In this experiment, we sort the 27 cases by their original token count (grey bars).

The results demonstrate a clear inverse correlation: as the context size increases, the LLM's success rate in identifying the vulnerability (red trend line) exhibits a sharp decline. When the context is small ($< 2k$ tokens), the LLM can occasionally locate the vulnerable functions; however, when faced with real-world framework-level execution traces ($> 10k$ tokens), the success rate fluctuates and collapses towards zero. By contrast, KINTSUGI's localization (blue bars) consistently compresses the context to a manageable size (typically $< 1k$ tokens), regardless of the application's complexity.

5.4 RQ3: Ablation Study

To understand the contribution of each component in KINTSUGI, we conduct an ablation study focusing on two key design choices: (1) the robustness across different LLM backends, and (2) the reliability of our profile-guided policy derivation versus direct LLM policy generation. We select 6 representative cases (P04, P07, Y06, Y08, J02, J05) covering all ecosystems and enforcement backends.

Impact of LLM Backends. We evaluate KINTSUGI using four state-of-the-art open-weights LLM models (Table 5). The results demonstrate that KINTSUGI is robust across various backends, consistently achieving a 100% mitigation success rate. While simpler cases like Y08 and J02 consistently converge in a single iteration across all models, more complex scenarios (e.g., P04 and Y06) require multiple iterations to resolve. We observed that less capable models may occasionally struggle with minor syntactic mapping errors or function selection in these complex cases, leading to a higher iteration count. However, KINTSUGI's automated validation loop effectively captures these regressions and guides the models toward correct boundaries in subsequent steps, demonstrating that our iterative framework effectively compensates for the inherent stochasticity of different black-box models.

Table 5: Ablation of LLM Backends (Measured by Iteration Count).

Case ID	DS-V3.2	DS-V3.1	Qwen3-235B-A22B	Qwen3-32B
P04	**1.2**	1.4	1.8	1.2
P07	**1.0**	1.0	1.2	1.4
Y06	**1.6**	3.0	3.0	1.6
Y08	**1.0**	1.0	1.0	1.0
J02	**1.0**	1.0	1.0	1.0
J05	**1.0**	1.0	1.0	2.0

Profile-Guided vs. Direct LLM Policy Generation. We compare KINTSUGI's hybrid approach against a "Direct LLM Policy" baseline (LLM writes syscall/argument whitelists directly). Within a 10-iteration limit, the baseline failed in all 4 syscall-based cases due to: (1) *Hallucination*: suggesting non-existent or architecture-incompatible syscalls; and (2) *Semantic Over-fitting*: predicting argument constraints from training data rather than actual runtime behavior, causing either bypasses or service breakage. Conversely, KINTSUGI uses LLMs only for semantic boundary identification, leaving the deterministic policy derivation to profile-guided analysis—a combination that ensures both precision and stability.

Contribution of Differential Localization. As evidenced in Section 5.3, without differential noise filtering, LLM localization accuracy collapses as context grows. This confirms that syscall-based differential analysis is the indispensable prerequisite for any LLM-driven mitigation in framework-heavy environments.

5.5 RQ4: Case Study

To illustrate KINTSUGI's full-lifecycle response, we conducted a detailed case study on a critical RCE in Apache Superset (Y05) involving complex `pickle` deserialization. Our analysis demonstrates how KINTSUGI successfully isolates the vulnerable sink (`pickle.loads`) from a trace of over 19k tokens and enforces a surgical eBPF-based policy that blocks malicious `execve` calls without affecting legitimate dashboard functionality. The complete step-by-step analysis, including the localization process and policy synthesis details, is provided in Appendix F.

Table 6: Offline performance statistics across 27 CVEs.

Metric	Mean	Median	P95	Max
Data Prep (s)	309.7	33.8	1,677.0	1,992.4
Localization (s)	82.7	7.1	538.3	731.3
Synthesis (s)	63.2	44.8	124.5	240.8
Total Latency (s)	**455.6**	**102.0**	**1,981.2**	**2,829.1**
Token Usage	**8,381**	**5,341**	**19,943**	**35,168**

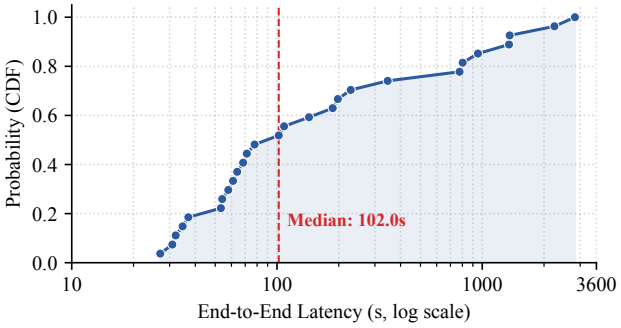

Figure 7: Cumulative Distribution Function (CDF) of end-to-end response latency across 27 CVEs. The x-axis is in log scale to accommodate the long-tail distribution. Over 50% of cases are mitigated within 102 seconds.

5.6 RQ5: System Performance and Runtime Overhead

End-to-End Response Latency. We evaluate the offline efficiency of KINTSUGI by measuring the time elapsed from capturing a malicious request to deploying a validated runtime policy. As summarized in Table 6, KINTSUGI achieves a median end-to-end latency of 102.0 seconds.

The latency is primarily composed of three stages: data preparation, differential localization, and policy synthesis. The median localization time is only 7.1s, and policy synthesis takes 44.8s. Even in the most complex case, the total latency remains under 48 minutes (2,829.1s), while the average response time is approximately 7.6 minutes (455.6s). Figure 7 presents the Cumulative Distribution Function (CDF) of the total latency. Notably, over 50% of the vulnerabilities can be mitigated within less than 2 minutes. This rapid response capability is critical for closing the "window of vulnerability" before a permanent vendor patch is available.

Token Usage and Cost. The average token usage per vulnerability is 8,381 (Median: 5,341). Based on the model's pricing structure from OpenRouter [2], the average financial cost per mitigation is approximately 0.0023 USD. For the 95th percentile (19,942 tokens), the cost per case is approximately 0.0055 USD.

Runtime Performance Overhead. To evaluate the impact of KINTSUGI on production traffic, we perform stress testing using `wrk` on the representative subset defined in Section 5.4. The benchmarks are conducted with 50 concurrent connections across 8 threads for a duration of 30 seconds per run, ensuring the system reaches a steady state. These cases provide a comprehensive cross-ecosystem evaluation, covering both eBPF-based syscall filtering (P04, J02, J05, Y08) and cgroup-based network isolation (P07, Y06). We measure the Throughput (RPS) and Latency for two types of requests: (1) *Targeted APIs* that trigger the injected runtime policies, and (2) *Other APIs* that represent normal business logic without vulnerabilities.

Table 7: Runtime performance overhead across different enforcement backends. *Targeted* refers to APIs containing policy triggers, while *Other* refers to baseline business APIs.

ID	Backend	Targeted API		Other API	
		RPS Drop	Lat. Inc.	RPS Drop	Lat. Inc.
P04	eBPF	8.73%	10.00%	1.28%	1.41%
P07	Network	<0.1%	<0.1%	<0.1%	<0.1%
J02	eBPF	0.92%	0.85%	<0.1%	<0.1%
J05	eBPF	22.02%	29.53%	3.71%	3.78%
Y06	Network	6.19%	3.67%	3.19%	2.17%
Y08	eBPF	1.36%	11.14%	5.08%	1.33%
Avg.		**6.54%**	**9.20%**	**2.21%**	**1.45%**

As shown in Table 7, the performance impact of KINTSUGI is localized. For *Targeted APIs*, the average throughput drop is 6.54% with a latency increase of 9.20%. The overhead peaks in J05 (29.53% latency increase), representing a worst-case scenario where the protected code block involves the initialization of a scripting engine (`Nashorn`). This process triggers a high density of syscalls—including extensive file I/O for class-loading and memory management—all of which must be individually intercepted and validated by the eBPF backend within the restricted context.

Crucially, the impact on *Other APIs* remains minimal, with an average RPS drop of 2.21% and a 1.45% latency increase. This confirms the effectiveness of our request-scoped activation: eBPF and cgroup constraints are dynamically engaged only within marked boundaries, allowing KINTSUGI to protect high-concurrency environments with negligible impact on non-vulnerable traffic.

6 Limitation and Discussion

Availability of Behavioral Baselines. The effectiveness of KINTSUGI's policy derivation relies on the availability of normal execution traces. In modern DevSecOps environments, such baselines are readily accessible through multiple channels, including regression test suites during CI/CD, historical traffic logs from production, and traffic mirroring in canary deployments. Our requirement for a behavioral base-

line aligns with the standard assumptions of state-of-the-art anomaly detection and provenance-based investigation systems [11, 15, 55].

Scope of Applicability. KINTSUGI provides runtime containment for web exploits whose successful execution produces observable syscall deviations from normal application behavior. This class prominently includes remote code execution, command injection, unsafe deserialization, and server-side request forgery. The system's design follows the established paradigm of audit-log-based forensics and provenance analysis [25, 38], in which behavioral divergence at the OS level serves as the fundamental prerequisite for automated detection and response. Consequently, KINTSUGI is not intended to neutralize vulnerabilities that operate purely at the application-semantic layer without OS-visible side effects, such as broken access control, IDOR, or most cross-site scripting (XSS). In practice, it complements app-layer defenses that address those classes rather than replacing them.

Mitigation vs. Permanent Patching. KINTSUGI is intended as an immediate containment measure that operates at the impact site of an exploit. For complex exploit chains, such as second-order stored injections, the system typically enforces policies at the point where the payload triggers OS-level actions, rather than at the original injection source. While this effectively neutralizes post-exploitation behavior, it treats the symptoms rather than the root cause. The system therefore serves as a stop-gap shield, giving security engineers time to develop and test a permanent logic-level fix.

Deployment Constraints. The kernel-level enforcement backends (eBPF and cgroups) require KINTSUGI's control plane to be installed with elevated privileges on the host. In managed cloud environments or shared hosting platforms where customers cannot load custom eBPF programs or modify cgroup hierarchies, deploying KINTSUGI may be restricted. This trade-off is inherent to any security mechanism that places enforcement points at the OS kernel layer, and it limits the system's applicability to infrastructure where operators retain low-level access to the kernel. Additionally, the offline analysis pipeline assumes that application source code and instrumented runtime environments are available; in fully closed-source or SaaS-only deployments, localizing and patching may not be feasible without vendor cooperation.

Behavioral Envelope and Mimicry. Because KINTSUGI derives its runtime policies from previously observed normal behavior, an exploit that stays entirely within the historical OS-level activity envelope could evade containment. For example, a multi-stage attack that initially only performs file reads already present in the baseline would not be intercepted at that stage. In practice, the threat model that KINTSUGI addresses involves exploits that need to execute qualitatively new OS actions—such as spawning a shell, binding a socket, or accessing sensitive system files—to achieve their intended impact. For such exploits, the small, code-snippet-scoped whitelists leave little room to maneuver without introducing anomalous syscalls. Nevertheless, the system offers containment only against post-exploitation behaviors that deviate observably from the learned profile; it is not a general intent-recognition mechanism. This represents an inherent trade-off in behavioral containment approaches that operate at the syscall level.

Baseline Incompleteness. The quality and completeness of the derived policies depend on how well the collected normal traces represent the application's legitimate runtime behavior. If a valid but rare execution path is absent from both the production logs and the test suite, the initial auto-derived policy may block that path when encountered in production. The staged validation loop described in Section 4.3 is designed to catch such mismatches during pre-deployment testing, but only for behaviors already present in the available traces. Increasing trace diversity—for example, by supplementing unit tests with canary-traffic replay or extended soak testing—can further reduce this risk. In its current design, KINTSUGI accepts a coverage-quality trade-off inherent to any behavioral profiling system.

7 Related Work

Vulnerability Localization. Traditional localization relies on program slicing [39], spectrum analysis [59], or delta debugging [60]. These methods typically require program crashes as logical anchors, making them ineffective against non-crashing web vulnerabilities (e.g., RCE). Domain-specific tools like ICSPatch [44] and AIVL [9] utilize Data Dependence Graphs or System Call Flow Graphs for binary auditing. However, they struggle with the dynamic control flows of interpreted languages and lack request-context awareness in high-concurrency environments. KINTSUGI achieves precision via function-level differential analysis of runtime traces without relying on static models or crash triggers.

Vulnerability Patching and Repair. Automated repair has transitioned from constraint-based methods (e.g., ExtractFix [19], VulnFix [61]) to LLM-driven syntactic rewriting (e.g., VulRepair [18], APPATCH [40], SAN2PATCH [31]). While effective for simple bugs, these approaches risk breaking core business logic when vulnerabilities reside in functionally necessary code. Furthermore, complex logic repairs often require architectural refactoring beyond current LLM capabilities [12]. KINTSUGI sidesteps these risks by enforcing runtime constraints instead of performing potentially destructive source code modifications.

Syscall Sandboxing and Least-Privilege Approaches. Constraining process capabilities at the syscall boundary is a foundational defense strategy. Early systems such as Systrace [43] and the kernel-level access-control framework of Bernaschi et al. [6] showed that intercepting syscalls can prevent unauthorized operations, but they required manually authored policies. Later work automated policy generation

through static analysis and temporal specialization, producing application-specific or container-wide syscall sets from binaries or interpreter-to-kernel mappings [7, 13, 20, 33, 56]. Falco [5] monitors syscall events for threat detection, while IDS-style frameworks like HOLMES [38] and MORSE [25] identify attack patterns from syscall traces.

These efforts operate *proactively*: they compute a reduced syscall surface before deployment under normal operating assumptions. KINTSUGI, in contrast, is designed for *reactive, post-exploitation containment*—an attack has already been observed, and the system must generate a temporary, request-scoped policy within minutes. Rather than statically analyzing the entire application, KINTSUGI derives its restricted syscall set from the *deviation* between a captured attack trace and normal execution at the same endpoint, anchoring enforcement to a specific code snippet. This attack-driven, request-level approach complements proactive hardening by filling the response gap before a permanent fix is available.

Policy-based Runtime Defense. Runtime defense intercepts malicious behavior via external rules. Phoenix [30] uses Seccomp and provenance analysis for stateful filtering, while VulShield [36] provides non-intrusive virtual patching for binaries. Global enforcement of syscall or network policies, however, cannot attribute individual syscalls to specific requests in high-concurrency web servers, leading to false positives. Binary-level protections [10, 26, 36] are also inapplicable to interpreted languages that lack DWARF metadata. KINTSUGI bridges this gap by binding high-level request context with low-level enforcement, enabling per-request policy activation without disrupting concurrent legitimate traffic.

eBPF for Dynamic Security. eBPF is increasingly used for efficient security enforcement. Recent works leverage eBPF to enhance Seccomp with stateful filtering [29] or dynamic policy switching [48]. KINTSUGI extends this paradigm by using LLMs to inject “Policy Trigger Point” into source code, enabling eBPF to provide request-level isolation and fine-grained behavioral control with minimal overhead.

8 Conclusion

We presented KINTSUGI, a runtime containment system that bridges code-level semantics with kernel-level enforcement for web exploits that produce observable syscall deviations. Through *request-scoped runtime policy injection*, KINTSUGI uses LLMs for precise semantic delimitation, avoiding error-prone code rewriting. The system automatically localizes OS-level behavioral anomalies via differential analysis and deterministically derives least-privilege policies from historical traffic. Evaluation on 27 CVEs spanning RCE, command injection, deserialization, and SSRF shows that KINTSUGI neutralizes these high-impact exploits with a median response time of 102 seconds. It maintains a minimal performance footprint on non-targeted traffic, with a 2.21% RPS drop in concurrent workloads and a 9.2% latency overhead on protected APIs.

A Ethical Considerations

Stakeholders and Impacts. We identify the following primary stakeholders:

- *Application Owners and Users:* These stakeholders benefit from the "Beneficence" of our work, as KINTSUGI provides immediate protection against active exploits during the high-risk window before a permanent patch is deployed, thereby safeguarding user data and ensuring service availability.
- *Security Researchers and Developers:* Our methodology assists developers by automating the tedious process of vulnerability localization and providing a "stop-gap" defense that reduces the pressure of emergency patching.
- *The Broader Internet Ecosystem:* By neutralizing active exploits (such as RCE) at the kernel level, KINTSUGI prevents compromised servers from being used as pivot points or botnet nodes, serving the "Public Interest."

Potential Harms and Mitigations. A potential tangible harm is the risk of "False Positives"—where an over-restrictive runtime policy might disrupt legitimate business logic. We mitigate this by: (1) using differential analysis to ensure surgical precision, (2) deriving whitelists from actual historical "normal" traffic, and (3) implementing an automated validation pipeline in a staging environment before any policy is deployed to production.

Another consideration is the dual-use nature of automated tools. While an attacker might theoretically use our localization techniques to better understand a vulnerability, KINTSUGI requires access to internal source code and high-fidelity execution traces (e.g., eBPF/instrumentation data), which are typically only available to legitimate system administrators and defenders. **Research Procedures.** Our experiments were conducted using publicly available CVEs and widely used open-source web frameworks (e.g., CMS Made Simple, Craft CMS). No private user data was harvested or utilized during our evaluation; all "normal traffic" baselines used for differential analysis were generated using standard automated testing suites and synthetic workloads.

Decision to Proceed. We believe the benefits of providing an automated, request-scoped defense mechanism far outweigh the potential risks. The current manual response cycle for web vulnerabilities is a major source of systemic risk, and KINTSUGI offers a transparent, deterministic, and verifiable way to reduce this risk without the "black-box" dangers of fully autonomous code rewriting.

B Open Science

KINTSUGI is committed to open science, reproducibility, and transparency. The public artifact is available on Zenodo:

https://doi.org/10.5281/zenodo.20551685

The artifact provides the implementation and evaluation materials for KINTSUGI. It contains the source code of the mitigation pipeline, PHP/Python/Java CVE case-study harnesses, normal and malicious traffic scripts, runtime tracing components, eBPF/syscall-filtering components, network-filtering components, and documentation for setting up and running representative examples.

At a high level, the artifact takes vulnerable CVE environments, paired normal and malicious requests, and optional LLM API configuration as inputs. It then traces application runtime behavior, identifies vulnerability-related execution differences, uses LLM-assisted reasoning to locate mitigation points, and derives runtime policies that block malicious behavior while preserving benign requests. The generated outputs include vulnerability-specific policy trigger points, syscall/network filtering configurations, execution logs, and validation results.

The archive is organized as a single package with a top-level README, license, and the KINTSUGI source tree. It includes 20 Dockerized PHP/Python CVE cases and seven Java CVE harnesses used in our evaluation. Large generated outputs, local build caches, container images, and nonessential experimental data are not included; required software, system assumptions, and dependencies are documented in the artifact.

By releasing these materials, we aim to make the design, implementation, and evaluation setup of KINTSUGI transparent and reusable by the research community.

C Detailed Prompt Design for Patch Generation

In Section 4.3, we discussed the high-level strategy for LLM-guided semantic boundary identification. This appendix provides the concrete structure and implementation details of the prompt used to guide the LLM backends.

The prompt is designed to minimize hallucination by providing a rigid structure and clear operational constraints. As shown in Figure 8, the prompt template consists of four primary segments:

- **Role Definition:** It establishes the LLM as a "Network Security Expert" to bias the model's internal attention toward security-relevant code patterns and vulnerability patterns.
- **Contextual Data:** We feed the LLM with the differential analysis results, including the specific syscalls that deviated from the baseline and the corresponding stack traces. This allows the model to focus on the execution path that actually triggered the anomaly.
- **Tooling API:** We provide the exact syntax for the runtime markers (e.g., `syscall_filter_begin`). By providing these as "tools," we ensure the LLM treats them as atomic primitives rather than generating arbitrary code.
- **Surgical Constraints:** We explicitly instruct the model to avoid modifying any existing business logic. The goal is to produce a "non-intrusive" patch that only adds monitoring boundaries.

Role
Network Security Expert

Task
Identify the code block triggering malicious behavior and wrap it with policy triggers.

Tools
`syscall_filter_begin($WHITELIST$);`,
`syscall_filter_end($WHITELIST$);`

Constraints
- `$WHITELIST$` is a mandatory placeholder for automated derivation.
- The scope must be as narrow as possible (surgical precision).
- Do not modify business logic; only inject markers.

Context
Payloads, Stack Trace, Syscall Diff, Source Code, Snapshots.

Figure 8: Structure of the Patch Generation Prompt.

D Path Prefix Compression Algorithm

This appendix provides the formal details of the Path Prefix Compression algorithm discussed in Section 4.3.

D.1 Algorithm Design

The compression task is formulated as a greedy optimization problem. Given a set of file paths S, we define two primary metrics to guide the selection of prefixes:

- **Compression Gain (*Gain*):** The number of paths covered by a prefix minus one. Higher gain reduces the rule count more effectively.
- **Compression Cost (*Cost*):** Defined as the negative depth of the prefix path (i.e., $-Depth(prefix)$). Shallower paths (closer to the root) have higher costs as they grant broader permissions to the sandbox.

In each iteration, the algorithm selects the prefix that minimizes cost (prioritizing the deepest possible path) and maximizes gain. Finally, a *density check* is performed: scattered file paths are only collapsed into their parent directory if the number of siblings exceeds a threshold τ; otherwise, exact paths are preserved to maintain maximum security.

D.2 Formal Pseudocode

Algorithm 1 details the iterative process of transforming specific paths into a least-privilege prefix set.

Algorithm 1 Path Prefix Compression

Input: Path set S, target size T, density threshold τ
Output: Compressed prefix set P
$P \leftarrow S$
while $|P| > T$ **do**
  $\mathcal{C} \leftarrow \{p \mid p \text{ is ancestor of some } s \in P\}$
  $\mathcal{F} \leftarrow \{p \in \mathcal{C} \mid |\{s \in P : s \text{ starts with } p\}| \geq 2\}$
  if $\mathcal{F} = \emptyset$ **then break**
  $p^* \leftarrow \text{argmin}_{p \in \mathcal{F}} Cost(p)$ ▷ Break ties by max $Gain(p)$
  $Covered \leftarrow \{s \in P \mid s \text{ starts with } p^*\}$
  $P \leftarrow (P \setminus Covered) \cup \{p^*\}$
$Files \leftarrow \{s \in P \mid s \text{ is a file path}\}$
for each unique parent directory d of $Files$ **do**
  $Siblings \leftarrow \{s \in Files \mid s \text{ resides in } d\}$
  if $|Siblings| \geq \tau$ **then** $P \leftarrow (P \setminus Siblings) \cup \{d\}$
return P

E Detailed Implementation

The KINTSUGI prototype consists of approximately 10,000 lines of Python code, managing the full lifecycle from localization to deployment.

Multi-language Instrumentation and Tracing. To support diverse web backends, we developed lightweight tracing plugins:

- **PHP:** A custom Zend extension hooks the core executor to capture runtime call stacks and function entries.
- **Python:** We utilized built-in tracing interfaces and wrapped `threading` and `os.fork` to maintain trace integrity across concurrent and multi-process execution.
- **Java:** We employed Java Agent technology to perform bytecode instrumentation at class-loading time, enabling context-aware logic injection.

All plugins utilize a unified event protocol that synchronizes user-space function events with kernel-space syscall streams via specific syscall anchors (e.g., writing context to `/dev/null`).

Kernel-level Defense Engine. The defense engine is built on eBPF using the BCC framework [28]. It implements kernel-space monitors that intercept critical syscalls via `kprobes`. The system maintains a thread-level `BPF_HASH` map to store dynamically generated whitelists. To bypass eBPF instruction limits for path matching, we implemented the FNV-1a hash algorithm for efficient kernel-side prefix comparison. We leveraged `prctl` as the low-latency communication channel between user-space and the kernel for instantaneous policy activation.

Request-level Network Isolation. For network-layer threats like SSRF, we integrated the Linux cgroup `net_cls` subsystem. Upon entering a compromised code block, the engine dynamically migrates the specific thread (via its TID) to a restricted control group. Combined with `iptables` rules, this achieves surgical interception of unauthorized outbound connections without affecting other concurrent requests.

Automated Validation Pipeline. The localization engine processes offline audit logs in parallel. To ensure reliability, we built a containerized validation pipeline that performs bidirectional verification (security and compatibility) in an isolated environment before production deployment.

F Detailed Case Study (CVE-2023-37941)

To illustrate KINTSUGI's full-lifecycle response, we examine a critical RCE in Apache Superset (Y05). The vulnerability resides in the dashboard permalink mechanism, where configuration data is retrieved from a metadata database and deserialized using Python's `pickle` module.

Localization and Contextualization. A single malicious request to the permalink endpoint traverses 162 unique functions and generates over 19k tokens of code. Direct LLM analysis of this volume is prone to high hallucination rates. KINTSUGI's differential analysis identifies a stark deviation in the function `GetKeyValueCommand.get()`. While a normal request only triggers `fcntl` and `write`, the malicious request triggers a sequence of `clone`, `execve`, and `vfork`. This behavioral signal allows KINTSUGI to isolate the vulnerable functions to 11 lines of code within the `key_value` module.

Policy Synthesis and Enforcement. Providing this localized context to the LLM allows it to pinpoint the exact sink: `pickle.loads(entry.value)`. As shown in Figure 9, KINTSUGI injects a request-scoped boundary around the deserialization call. The profile-guided refinement phase automatically populates the whitelist based on legitimate permalink access, which involves only basic I/O and synchronization.

```
// Mitigated logic in
superset/key_value/commands/get.py
sysfilter_begin({'fcntl': [], 'write': []})
result = pickle.loads(entry.value)
sysfilter_end()
```

Figure 9: Surgical policy injection for Y05

Outcome. When the attacker triggers the permalink with a payload designed to execute `touch /tmp/success`, the eBPF backend intercepts the `execve` call at the kernel level. Because the thread is in a "restricted state" tied to the `GetKeyValueCommand` context, the syscall is blocked.

Acknowledgments

This research is sponsored in part by science and technology innovation project of Hunan Province (No. 2023RC4014) and the NSFC Program (No. 6212780016).

References

[1] Open source content management system | cms made simple. URL: https://www.cmsmadesimple.org/.

[2] Openrouter. URL: https://openrouter.ai.

[3] Owasp api security project. URL: https://owasp.org/www-project-api-security/.

[4] Owasp top 10:2025. URL: https://owasp.org/Top10/2025/.

[5] Falco: Cloud native runtime security, 2016. URL: https://falco.org/.

[6] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V Mancini. Operating system enhancements to prevent the misuse of system calls. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 174–183, 2000.

[7] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. Saphire: Sandboxing {PHP} applications with tailored system call allowlists. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2881–2898, 2021.

[8] Changhua Chen, Tingzhen Yan, Chenxuan Shi, Hao Xi, Zhirui Fan, Hai Wan, and Xibin Zhao. The last mile of attack investigation: Audit log analysis towards software vulnerability location. *IEEE Transactions on Information Forensics and Security*, 2024.

[9] Changhua Chen, Tingzhen Yan, Chenxuan Shi, Hao Xi, Zhirui Fan, Hai Wan, and Xibin Zhao. The last mile of attack investigation: Audit log analysis towards software vulnerability location. *IEEE Transactions on Information Forensics and Security*, 2024.

[10] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[11] Zijun Cheng, Qiujian Lv, Jinyuan Liang, Yan Wang, Degang Sun, Thomas Pasquier, and Xueyuan Han. Kairos: Practical intrusion detection and investigation using whole-system provenance. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3533–3551. IEEE, 2024.

[12] Jonathan Cordeiro, Shayan Noei, and Ying Zou. An empirical study on the code refactoring capability of large language models. *arXiv preprint arXiv:2411.02320*, 2024.

[13] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 459–474, 2020.

[14] Django Software Foundation. Django: The web framework for perfectionists with deadlines. https://www.djangoproject.com/, 2005.

[15] Asbat El Khairi, Marco Caselli, Andreas Peter, and Andrea Continella. Replicawatcher: Training-less anomaly detection in containerized microservices. In *Network and Distributed System Security Symposium, NDSS 2023*. Association for Computing Machinery, 2024.

[16] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023.

[17] Glenn Fowler, Landon Curt Noll, Phong Vo, Donald Eastlake, and Tony Hansen. The FNV Non-Cryptographic Hash Algorithm. IETF Draft, jan 2011. URL: https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-03.

[18] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 935–947, 2022.

[19] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–27, 2021.

[20] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1749–1766, 2020.

[21] GitHub Advisory Database. Craft cms remote code execution vulnerability · cve-2023-41892. https://github.com/advisories/GHSA-4w8r-3xrw-v25g.

[22] Akul Goyal, Gang Wang, and Adam Bates. R-caid: Embedding root cause analysis within provenance-based intrusion detection. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3515–3532. IEEE, 2024.

[23] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. *arXiv preprint arXiv:2001.01525*, 2020.

[24] Xueyuan Han, Xiao Yu, Thomas Pasquier, Ding Li, Junghwan Rhee, James Mickens, Margo Seltzer, and Haifeng Chen. Sigl: Securing software installations through deep graph learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2345–2362, 2021.

[25] Md Nahid Hossain, Sanaz Sheikhi, and R Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *2020 IEEE symposium on security and privacy (SP)*, pages 1139–1155. IEEE, 2020.

[26] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 618–635. IEEE, 2016.

[27] Zhen Huang and Gang Tan. Rapid vulnerability mitigation with security workarounds. In *Proceedings of the 2nd NDSS Workshop on Binary Analysis Research, ser. BAR*, volume 19, 2019.

[28] IO Visor Project. BCC: BPF compiler collection. https://github.com/iovisor/bcc, 2015.

[29] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. Programmable system call security with ebpf. *arXiv preprint arXiv:2302.10366*, 2023.

[30] Hugo Kermabon-Bobinnec, Yosr Jarraya, Lingyu Wang, Suryadipta Majumdar, and Makan Pourzandi. Phoenix: Surviving unpatched vulnerabilities via accurate and efficient filtering of syscall sequences. In *Network and Distributed Systems Security Symposium (NDSS)*, 2024.

[31] Youngjoon Kim, Sunguk Shin, Hyoungshick Kim, and Jiwon Yoon. Logs in, patches out: Automated vulnerability repair via tree-of-thoughtllm analysis. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4401–4419, 2025.

[32] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, volume 16, 2013.

[33] Yunsen Lei and Craig A Shue. Making (only) the right calls: Preventing remote code execution attacks in php applications with contextual, state-sensitive system call filtering. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–65. Springer, 2025.

[34] Michael Levandowsky and David Winter. Distance between sets. *Nature*, 234(5323):34–35, 1971.

[35] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

[36] Yuan Li, Chao Zhang, Jinhao Zhu, Penghui Li, Chenyang Li, Songtao Yang, and Wende Tan. Vulshield: Protecting vulnerable code before deploying patches. In *NDSS*, 2025.

[37] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Mpi: Multiple perspective attack investigation with semantic aware execution partitioning. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1111–1128, 2017.

[38] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, Ramachandran Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. In *2019 IEEE symposium on security and privacy (SP)*, pages 1137–1152. IEEE, 2019.

[39] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. Vulchecker: Graph-based vulnerability localization in source code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6557–6574, 2023.

[40] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. Appatch: Automated adaptive prompting large language models for real-world software vulnerability patching. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4481–4500, 2025.

[41] Bridget Nyirongo, Yanjie Jiang, He Jiang, and Hui Liu. A survey of deep learning based software refactoring. *arXiv preprint arXiv:2404.19226*, 2024.

[42] Pixel & Tonic, Inc. Craft cms: A flexible, user-friendly cms for developers and content managers. https://craftcms.com/, 2026. Accessed: 2026-01-19.

[43] Niels Provos. Improving host security with system call policies. In *USENIX Security Symposium*, pages 257–272, 2003.

[44] Prashant Hari Narayan Rajput, Constantine Doumanidis, and Michail Maniatakos. Icspatch: Automated vulnerability localization and non-intrusive hotpatching in industrial control systems using data dependence graphs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6861–6876, 2023.

[45] Sebastián Ramírez. FastAPI: A modern, high-performance, web framework for building APIs with Python. https://fastapi.tiangolo.com/, 2018.

[46] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address allocation for private internets. RFC 1918, February 1996. URL: https://www.rfc-editor.org/rfc/rfc1918.

[47] Armin Ronacher. Flask: A lightweight WSGI web application framework. https://palletsprojects.com/p/flask/, 2010.

[48] Matthew Rossi, Michele Beretta, Dario Facchinetti, and Stefano Paraboschi. Poster: Transparent temporally-specialized system call filters. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, pages 1797–1799, 2025.

[49] Yaman Roumani. Patching zero-day vulnerabilities: an empirical analysis. *Journal of Cybersecurity*, 7(1):tyab023, 2021.

[50] Jahanzeb Shahid, Muhammad Khurram Hameed, Ibrahim Tariq Javed, Kashif Naseer Qureshi, Moazam Ali, and Noel Crespi. A comparative study of web application security parameters: Current trends and future directions. *Applied Sciences*, 12(8):4077, 2022.

[51] Claude E Shannon. The redundancy of english. In *Cybernetics; Transactions of the 7th Conference, New York: Josiah Macy, Jr. Foundation*, pages 248–272, 1951.

[52] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. Undo workarounds for kernel bugs. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2381–2398, 2021.

[53] The PHP Group. PHP-FPM (FastCGI Process Manager). https://www.php.net/manual/en/install.fpm.php, 2010.

[54] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, Carl A Gunter, et al. You are what you do: Hunting stealthy malware via data provenance analysis. In *NDSS*, 2020.

[55] Ruihua Wang, Yihao Peng, Yilun Sun, Xuancheng Zhang, Hai Wan, and Xibin Zhao. Tesec: Accurate server-side attack investigation for web applications. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2799–2816. IEEE, 2023.

[56] Wenya Wang, Xingwei Lin, Jingyi Wang, Wang Gao, Dawu Gu, Wei Lv, and Jiashui Wang. Hodor: Shrinking attack surface on node. js via system call limitation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2800–2814, 2023.

[57] Zicheng Wang, Yueqi Chen, and Qingkai Zeng. Pet: Prevent discovered errors from being triggered in the linux kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4193–4210, 2023.

[58] Chunlin Xiong, Tiantian Zhu, Weihao Dong, Linqi Ruan, Runqing Yang, Yueqiang Cheng, Yan Chen, Shuai Cheng, and Xutong Chen. Conan: A practical real-time apt detection system with high accuracy and efficiency. *IEEE Transactions on Dependable and Secure Computing*, 19(1):551–565, 2020.

[59] Dandan Xu, Di Tang, Yi Chen, XiaoFeng Wang, Kai Chen, Haixu Tang, and Longxing Li. Racing on the negative force: Efficient vulnerability root-cause analysis through reinforcement learning on counterexamples. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4229–4246, 2024.

[60] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on software engineering*, 28(2):183–200, 2002.

[61] Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 691–702, 2022.