Making (Only) the Right Calls: Preventing Remote Code Execution Attacks in PHP Applications with Contextual, State-Sensitive System Call Filtering

Yunsen Lei[1,2] and Craig A. Shue[1]

[1] Worcester Polytechnic Institute {ylei3,cshue}@wpi.edu
[2] George Washington University yunsen.lei@gwu.edu

Abstract. PHP powers over 76% of websites worldwide, making security vulnerabilities in its applications particularly damaging. Unfortunately, such defects remain common: in 2021, nine of the top 15 most-exploited vulnerabilities identified by CISA involved remote code execution (RCE). Prior research has attempted to contain RCE through system call filtering (e.g., via `seccomp`), but these efforts are typically coarse-grained. They allow all system calls that could potentially be invoked anywhere in the application, providing attackers substantial opportunities for exploit.

We introduce a fine-grained, state-sensitive approach that builds an automaton for each PHP script, mapping different execution stages to carefully curated system call subsets. At runtime, our kernel module combines information from system call traces and PHP script-level events to apply these context-driven allow-lists. We demonstrate our method's effectiveness against real-world CVEs and against attackers crafting RCE payloads designed to mimic legitimate calls. Our model successfully detects these "stealth" attacks and maintains a low performance overhead of only 1%—a substantial improvement over the 5% overhead observed in prior work.

1 Introduction

PHP is a popular server-side language powering 75.4% of measured websites in 2024 [27]. PHP is the driving force behind WordPress, which itself powers 43.7% of measured websites worldwide [28]. Unfortunately, PHP faces the same security issues as other back-end server languages.

A critical and challenging to combat threat is remote code execution (RCE). An attacker typically exploits an application defect to deliver and execute malicious code that exceeds the application developer's intent, potentially spawning processes and accessing system resources without restriction. To address the risks posed by RCE, the research community has explored various mitigation strategies. A common approach involves a two-phase process of modeling and enforcement. During the modeling phase, the target application is analyzed to

determine its intended behavior. By requiring the program's runtime execution to comply with the defined model, defenders can constrain the scope in which arbitrary code can be executed.

Defenders can use a system call allow-list to enforce a program's behavior. The allow-list is first obtained by analyzing the target program for a set of reachable system calls and is then enforced through the `seccomp` module in Linux. The `seccomp` module can be viewed as a deterministic finite automaton with a single approve and reject state. System calls defined in the allow-list consist of transition rules that keep the program in the "approve" state. Conversely, system calls not in the allow-list form transition rules that move the program to the "reject" state. The `seccomp` module can choose to terminate a program if a disallowed system call is captured.

The effectiveness of the system call allow-list approach is directly related to how tight the constraints are. Prior efforts have built allow-lists that work at the entire PHP application granularity [12, 6] and at the HTTP request granularity (i.e., the script specified in the HTTP URL) [4]. Unfortunately, there are significant drawbacks to each: the first is too permissive, enabling attackers to exploit any allowed call; the second, which is built upon `seccomp`, requires restarting the PHP process or dedicating separate PHP engines per target script. Our investigation shows that even request-level profiles can permit stealthy "mimicry" RCE. We therefore re-examine the whole system call modeling and enforcement pipeline for PHP applications. We propose a protection system that constructs a detailed, stateful system call profile and enforces that profile using run-time sensors and kernel-mode system call filtering.

With this foundation, we explore important research questions: *How can we construct fine-grain profiles of PHP applications? To what extent can such fine-grained modeling and enforcement constrain an RCE attack on the PHP platform? What performance cost does this approach introduce to the existing PHP platform? Does such an approach negatively impact programs' legitimate execution?* In exploring these questions, we contribute the following:

A design for detailed PHP application profiling and sensing: We design an automata-based approach for modeling the state of a PHP application. We design tools to construct the automata from PHP scripts and from the executables and libraries associated with the PHP engine (Section 3).

A system for context-aware, state-sensitive system call filtering: We implement sensors and an enforcement module that recognizes application-level events that precede the underlying system calls. We use these to apply appropriate system call profiles to enforce (Section 4).

An evaluation of the security effectiveness and performance: Through real-world RCE vulnerabilities and legitimate workloads, we compare against `seccomp`-based defenses. Our approach completely stops the attacks across three tested classes (Section 5), adding under 3% overhead for script-level enforcement (Section 6).

2 Background for PHP Application Risks

In PHP applications, the risk of remote code execution is heightened by two intertwined factors. First, attackers have various techniques at their disposal to inject malicious code. Second, system call usage is prevalent in application scripts. Those prevalent system calls result in a larger list of permitted system calls, which can provide attackers with ample opportunities to mimic legitimate system operations and thus bypass existing security measures. In this Section, we first examine the common techniques for RCE on PHP applications and then discuss the impact of system call frequency.

2.1 Attack Techniques

Remote Code Execution (RCE) in PHP typically requires explicit injection of malicious code or data that the PHP engine then integrates into its execution, unlike intra-procedural attacks like return-oriented programming, which reuse existing instruction gadgets. We classify RCE based on how malicious code affects the system call profile across execution scopes. One category involves file inclusion or uploads, introducing new scripts and altering request-handling structures. Another relies on command injection or deserialization, executing malicious code within an existing script's scope.

```
1 <?PHP
2 $file_path = $_GET['file'];
3 include($file_path);
4 include('app_dir/' . 'dependency.php');
5 /* rest of the code ... */ ?>
```

Fig. 1: A file inclusion vulnerability through user-supplied path

File Upload and Inclusion Attackers often exploit arbitrary file upload vulnerabilities to place malicious scripts within the PHP application's source directory. Then, the attacker can execute their own uploaded script by accessing it directly via a URL or with a local file inclusion vulnerability (line 3 in Figure 1). For applications that need write access to its code directory (e.g., a WordPress updates its plugin), the attack may be more covert by overwriting existing scripts during the upload process (such as altering `dependency.php` on line 4 in Figure 1).

Injection and Deserialization This method exploits functions that process unvalidated user inputs, allowing attackers to embed and execute malicious code. For instance, an attacker might craft a payload that is dynamically evaluated via functions like `eval` or injected into a template engine. In addition, unsafe deserialization can instantiate objects with carefully crafted fields, triggering unwanted function calls.

Figure 2 illustrates a Server-Side Template Injection (SSTI) attack on the Twig PHP template engine [24]. In a misconfigured or older version of Twig, attackers can construct the controllable parameter to inject a template that registers a callback function, then invoke it through another parameter. For example, setting `greeting` using `registerUndefinedFilterCallback` and `name` using `getFilter`. Such attacks are typically the result of improper configuration or unsafe template features rather than an inherent flaw in Twig itself.

```
1 <?PHP
2 $twig->render("Hello " .$_GET['greeting'] .$_GET['name']);
3 /* rest of the code ... */ ?>
```

Fig. 2: A Minimal Server Side Template Injection Example

Figure 3 illustrates a deserialization attack involving a PHP class called `LogWriter`, which includes a `destruct` magic method that writes content to a file when the object is unset, or the script ends. The separate script (lines 1–5) retrieves data from a request parameter and calls `process_data`, appearing harmless at first. However, an attacker can submit a serialized `LogWriter` object, manipulating its `filename` and `log_content` properties to write arbitrary files. In large applications using multiple libraries, attackers can construct complex object chains to trigger a series of malicious actions. This is akin to Return-Oriented Programming but is known as Property-Oriented Programming [10]. Tools like those in [20, 1] automate finding such "gadgets" in common PHP libraries.

```
1 <?PHP
2 require("dep.php");
3 $data = unserialize($_GET['data']);
4 process_data($data);
5 /* rest of the code ... */ ?>
-------------------------
6 <?PHP  # dep.php
7 class LogWriter {
8     public $filename;
9     public $log_content;
10    public function __destruct() { file_put_contents(
11        $this->filename, $this->log_content);}
12 function process_data($data) {
13   $data->value += 1; } ?>
```

Fig. 3: Code vulnerable to Unsafe Deserialization attacks

2.2 System Call Pervasiveness and Mimicry

All of these attacks arise from software defects. Ideally, flawless code would prevent them, but in reality, security vulnerabilities keep surfacing in production software. Consequently, we need a cross-application mitigation strategy that does not rely on a "perfect" application. System call filtering can help, but to be effective, it must be stringent enough to limit attackers' options. Although web applications legitimately require file and network operations during request processing, these same capabilities can be exploited for malicious purposes.

A basic `seccomp`-style filter, for instance, lacks awareness of the specific PHP application it protects; it must permit every system call reachable from the PHP engine or its extensions. APIs like `shell_exec` pose a particular risk, since they rely on powerful system calls (`fork`, `exec`), leaving wide avenues for arbitrary code execution if an RCE vulnerability is discovered.

Recent work moves toward dynamic system call specifications, adapting the filter to different execution stages (e.g., initialization vs. serving [13], or updated each `execve` call [11]). In the PHP domain, Saphire [4] filters system calls on a per-request basis. However, even a request-level scope can still be too broad: a single request may involve multiple scripts, all of which share the same system call set. While the filter permits system calls that are essential for legitimate request components, it cannot distinguish these from identical system calls initiated by malicious code.

The attack examples we have described above illustrate the need for more state-aware filters to prevent RCE attacks. In file inclusion or upload attacks, the malicious code executes as a new, separate script within a request's scope. For injection and deserialization, the malicious code is executed within an existing script. Effective defense mechanisms must, therefore, perceive the scope in which a system call is issued and enforce a multi-state model, each aligning with a specific execution context or phase of a request.

3 Design: Automata-Based Enforcement

We build an automaton for each PHP script to precisely capture valid system call behavior. Figure 4 depicts how, in a trusted environment, we use two sensors: one examines the compiled PHP binary and its libraries, and the other inspects the application's PHP source code. Based on their outputs, we generate an automaton per script.

Once these automata are ready, we instrument the PHP executable so that specific points in its execution trigger software interrupts, which then get handled by the kernel. In production, when clients connect, these interrupts provide the kernel with execution context, which is tracked via a stack that represents the application's state. Each system call is then checked against the relevant script automaton and the current execution context. If the automaton permits the call in that state, it is allowed; otherwise, it is rejected.

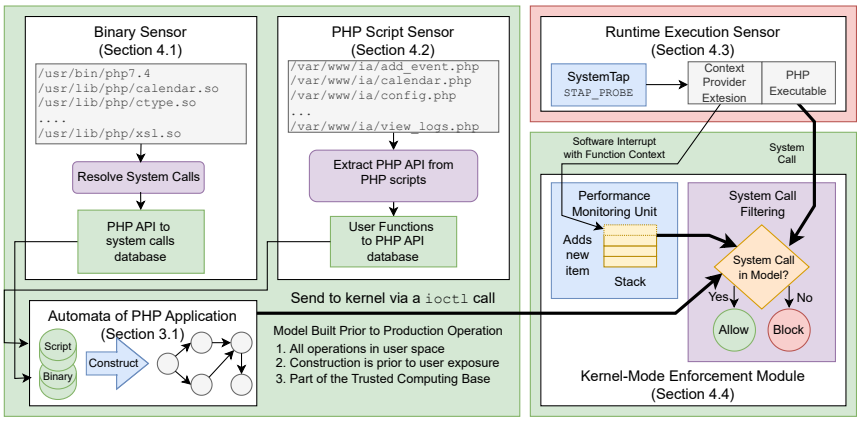

Fig. 4: The left side of the diagram shows the instrumentation and model construction that occurs prior to production operation. Through a set of sensors, we build an automaton for each PHP script. During operation (depicted on the right), we fuse that prior model with probe and system call data from the PHP executable to filter system calls that deviate from the model.

3.1 Threat Model: Trusted Models and Kernel

Like other approaches that aim to constrain untrusted PHP server applications, we rely on a trusted kernel for security decisions and assume we can examine the PHP binary and scripts in a trusted environment to build our models.

We trust that the constructed model can be read by the kernel without malicious alteration (e.g., from a trusted account on the server). We trust that the kernel can safely load our constructed model and that we can instrument the PHP executable before any client connections. Once a client connects, an attacker may attempt corruption through system calls.

The defender's goal is to block at least one necessary system call the attacker needs to compromise the environment. The attacker's goal is to corrupt the environment through a set of system calls without any of the required system calls being blocked by the defender. The attacker may utilize mimicry techniques to operate using system calls that are likely to be allowed by the module as desired, but they must eventually attempt to corrupt the environment to enact the attack.

3.2 From a PHP Script to an Automaton

When an HTTP server (e.g., Nginx or Apache) receives a request (e.g., `GET`, `POST`), it uses configuration data to determine which PHP script to run. We call this the *target script*. During execution, the target script may include other scripts—*included scripts*—which can likewise include additional scripts. Scripts

```php
1 <?PHP # req1.php
2 if (isset($_GET['admin'])) {
3   include('admin/'. 'dep2.php'); }
4 else {
5   include('inc/'. 'dep3.php'); }
6 $code = $GET['file'];
7 eval($code);
8 /* rest of the code ... */ ?>
-----------------------------------
9  <?PHP # dep2.php
10 include(dep4.php);
11 logAdminAccess('access.log');
12 loadAdminTemplate();
13 /* rest of the code ... */ ?>
-----------------------------------
14 <?PHP # dep3.php
15 include(dep4.php);
16 loadVisitorTemplate();
17 /* rest of the code ... */ ?>
```

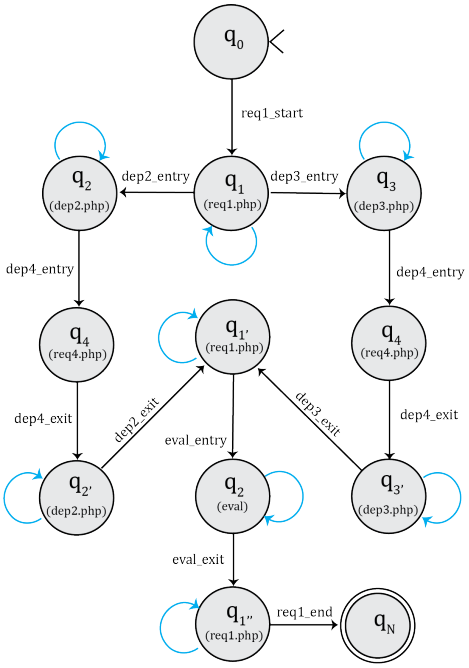

Fig. 5: The code on the left is an example of a vulnerable PHP application consisting of two request handling scripts: `Req1.php` and `Req2.php`. The diagram on the right shows how the program can be transformed into a DFA-based system called filtering that leverages the application context in its decisions.

with global-scope code run immediately upon inclusion, while others only define classes or functions.

We can represent the target script's inclusion and execution of a dependent script as a unique and identifiable execution stage. These execution stages can be conceptualized as a state in a Deterministic Finite Automaton (DFA). Formally, our DFA is defined as $(Q, \Sigma, \delta, q_0, q_N)$:

States Q. Each state q_i represents an execution phase in the script. We create a state for the global scope code of each script (including included scripts) and additional states for specific function scopes. Each state q_i is associated with a system call allow-list A_i. This variable state granularity definition enables flexible system call policies.

Alphabet Σ. This consists of both system call symbols (identified by their kernel call numbers) and application-level events (e.g., entering a script, exiting a script, or calling a function).

Transition Function δ. Given a state and an input event, it determines the next state. System calls allowed in the current state lead to self-transitions

(e.g., $\delta(q_i, s_k) \rightarrow q_i$ for $s_k \in A_i$) while events like `script_entry` or `script_-script_exit` move execution to different states.

Initial State q_0. The PHP engine is awaiting a request to process.

Final State q_N. Reached when the script completes processing the request.

Figure 5 shows a sample automaton for a target script `req1.php`. Depending on the `admin` parameter, `dep2.php` or `dep3.php` is included; each of these in turn includes `dep4.php`. Afterward, `req1.php` reads a `GET` parameter (`file`) and calls `eval`. The DFA starts in q_0, transitions to a script-specific state (e.g., q_1) upon request start, then moves to new states when it includes other scripts or enters function scopes. At each state, only the system calls listed in its allow-list are permitted. The final state q_N indicates the end of request handling.

The DFA approach naturally covers script-level system call specifications, but it can also apply at the function level. As shown in Figure 5, we can enforce a dedicated filter for security-critical functions like `eval`, effectively blocking any system calls within those functions. Such granular policies help mitigate risks without unduly restricting legitimate functionality.

A more detailed DFA yields tighter state-level allow-lists, further limiting the system calls available to attackers. By confining powerful calls to fewer regions of trusted code, the defender reduces the chance an attacker can exploit vulnerabilities and successfully execute malicious operations.

3.3 Profiling: PHP APIs to System Calls

The automaton from the previous section can support fine-grained enforcement. To construct such a model, we need to understand when system calls will naturally occur in a PHP application. This is a two-step process, and this section describes the first step: discovering the system calls, if any, associated with each PHP API function.

The PHP API consists of internal PHP functions and methods, as well as additional functions and methods provided by dynamically linked shared libraries. These API implementations are written in lower-level languages and compiled into executable and shared object files. To profile the system calls invoked by each PHP API function or method, we analyze their implementations to determine the system calls they use. This analysis can be recursive since one API function or method may invoke others, forming a control flow graph (CFG) with inter-procedural calls. We annotate the CFG with any system calls discovered during the analysis and store this information in a mapping database for quick retrieval of system call details for each API function or method.

A significant challenge in this profiling process lies in resolving function references. At the PHP engine level, function calls may target other functions where the exact call target is initially unknown. This occurs when the call instruction uses registers or memory locations to specify the target function. We tackle these details in Section 4.1.

3.4 Profiling: User-defined Functions to APIs

The second step in mapping a PHP application to its system calls involves identifying all PHP API calls made by the application's scripts. Unlike built-in APIs, functions and methods defined within scripts are written in PHP and referred to as *user-defined functions*. To capture all relevant PHP API calls, we recursively analyze these user-defined functions, as they may invoke other user-defined functions or PHP APIs.

To identify PHP API calls within scripts, we transform all application scripts into control flow graphs (CFGs). Each basic block in a CFG consists of a sequence of instructions represented as PHP Abstract Syntax Tree (AST) nodes. By analyzing these AST nodes, we perform recursive graph traversal to discover all referenced PHP internal APIs.

PHP can resolve function names and method functions dynamically, which our profiling must support. For the method names, we adopt a static analysis process that tries to backtrace a variable AST node's definition or initial assignment. This static analysis helps resolve a variable's class type, which gives us a full `class_name::method_name` association. We leverage dynamic profiling to resolve the function names that remain unknown. When constructing the DFA, we map these API invocations identified in each script with their underlying system calls. We add those calls to the relevant nodes associated with each script's corresponding state.

3.5 Enforcement: Context and Filters

The existing `seccomp` module lacks the visibility needed to match a system call to a specific scope in a process's execution. The module only knows the process ID that is associated with a system call. That module also statically associates a system call filter with a process, making the filter insensitive to the state within a process. In contrast, our design needs to keep the kernel enforcement module in lockstep with the state of the user space program. This requires synchronized context from the user space process to the kernel enforcement module.

Our approach uses a software interrupt to signal transitions. In addition, we pass an application-level event's specification to the kernel in advance. The specification tells the enforcement module how to collect an event's context data when a corresponding interrupt handler is invoked. To trigger the interrupt at the desired execution stage, we instrument a PHP extension to insert the interrupt's instruction. The PHP extension is developed to overwrite a set of PHP execution hooks. Each hook is essentially the entry and exit point of a PHP application-level event (e.g., a script entry or request start).

Before production usage, the filtering module registers callback functions that the interrupt handler will invoke to collect user context. To get a system call, we adopted the same process as `seccomp`, which tags the target process with a flag to indicate the need to examine a system call when invoked.

4 Implementation: Sensors and CFGs

Our automata-based enforcement design requires profiling work and detailed run-time sensors to operate. We now describe the details of implementing this design. We start with the instrumentation of the PHP executables and libraries and then describe the script-level instrumentation. We then describe our run-time sensors and the kernel enforcement.

4.1 PHP API to System Calls

Our goal is to map each built-in PHP API function or method to its associated system calls. The PHP engine can be built with various dynamically linked libraries; in the instance (PHP 8.4) we used for evaluation, there were 89 such libraries.

We implemented this mapping using the `angr` tool [23], specifically the `CFGFast` API, to build control flow graphs (CFGs) for the PHP binary and its linked libraries. These CFGs form the foundation of our binary analysis. First, we identify all function symbols defined in the PHP binary. We then locate basic blocks in the CFG that end with `call` instructions, which can be classified as either direct or indirect calls.

For direct calls, we resolve targets by directly identifying function definitions or through entries in the process linkage table (PLT). PLT entries reference dynamically linked functions and are typically resolved at runtime via lazy binding, where initial calls redirect to corresponding global offset table (GOT) entries. Using the GOT entry address, we access the relocation table (`.rela.plt`) to obtain the symbol type and an index into the dynamic symbol table (`.dynsym`). With this index, we identify the function's symbolic name, the owning library, and ultimately its offset within the library's symbol table. This method allows us to map PHP API functions to their dynamically-linked implementations.

Indirect calls typically reference memory locations or registers, often involving function pointers or dynamic resolution (e.g., via `dlsym`). During the static analysis phase, we label unresolved indirect calls and record their call site addresses. Then we use `DynamoRIO` [8] to instrument these calls to log the relative addresses of the call site and target, along with the module path. Running PHP's test suite lets us correlate this data to identify the target function's library and offset.

These methods allow us to establish an initial mapping between PHP internal APIs and their callee functions. We then traverse each callee function's CFG to identify system calls. On `x86_64` systems, system calls are invoked using the `syscall` instruction, which stores the system call number in the `eax` register. If the `eax` register is populated with an immediate value (e.g., `mov eax, 0x01`), determining the system call number is straightforward. In cases where the `eax` register is populated with a non-immediate value, we emulate the basic block containing the `syscall` instruction—or the preceding block—to identify the stored system call number. This information is then compiled into a database, mapping PHP internal APIs to their implementing system calls.

This mapping between PHP internal APIs and their associated system calls is independent of the API's arguments. Although the actual system calls invoked by a PHP API could vary depending on its parameters, our mapping captures the complete set of possible system calls for each API.

4.2 User-defined Function CFGs to APIs

For PHP APIs, we examine each user-defined function to identify the internal PHP APIs it invokes. Our static analysis starts by using `php-cfg` [17] to build CFGs. We modify `php-cfg` to embed original source line numbers within the SSA nodes, making it easy to integrate dynamic profiling data.

Our analysis starts by traversing the abstract syntax tree (AST) to detect function and class definitions. We then inspect call-related AST nodes, extracting names for direct function calls and static methods. For dynamic method calls (e.g., `$a->method()`), we track each variable back to its last assignment to resolve its class. Specifically, we handle class resolutions arising from `new` instructions, resolvable function returns, or global variables. However, PHP's dynamic typing and callback registration mechanism can still obscure function names or class types. To resolve these, we employ a runtime profiler that logs user-defined and internal calls along with their script locations. This dynamic profiling complements our static analysis, resolving calls that cannot be determined statically.

In practice, we find that we can successfully resolve all calls in the PHP applications we evaluated, including popular and complex PHP applications like WordPress. In the event that the instrumentation fails to resolve a function, our implementation records the failure to enable manual resolution.

4.3 SysTap Probes for Runtime Context

We developed a minimal PHP extension called `context_collector` that uses SystemTap's `STAP_PROBE` macro to help track runtime events without modifying the core PHP engine.

This extension gathers runtime context by declaring pointers to memory locations for storing event-related data. Each variable is wrapped with a `STAP_PROBE` macro from the SystemTap tool [22]. This macro inserts an inline `NOP` instruction after the variable declaration and uses the `.pushsection` assembler directive to record probe specifications—such as the `NOP`'s address and register allocations—in the note section of the extension's ELF binary. The ELF binary is loaded as a shared library in the PHP engine.

The `context_collector` uses PHP's built-in hooks to intercept the original function handler. In our custom handler, we embed the `STAP_PROBE` macro to collect context such as the request URL, script name, and function name. This custom function handler still invokes the original handlers to maintain normal application behavior.

At runtime, probes are activated using the `perf_event_open` system call, which replaces the `NOP` instruction with a software interrupt (`INT3`). Probes

can be disabled by closing their file descriptors, which restores the original `NOP` instruction. This design minimizes overhead when the PHP process does not enforce system call filtering.

4.4 Kernel Enforcement: Automata Checks

To collect context information from the software interrupts enabled in Section 4.3, we developed a kernel module that registers an interrupt handler for our custom `STAP_PROBE` probes with the Performance Monitoring Unit (PMU). The PMU manages the creation of the underlying probe structure, which invokes the handler function. The kernel module's header file is accessible to the `context_collector` extension, ensuring consistent encoding and parsing of context data between the two components.

The enforcement module processes two types of events: regular system calls and application events triggered by software interrupts. For system calls, the module queries the current system call profile to decide whether to allow or deny the operation. For application events, the module advances the automaton based on event attributes such as request URLs or script names.

To manage system call profiles, our implementation uses a fixed-size bitmap, mapping each system call to a specific bit. We utilize the kernel's `DECLARE_BITMAP` and `bitmap_` APIs to create, set, and check the bitmap values. Each profile requires $\lceil NR\_syscall/8 \rceil$ bytes of memory, providing constant-time lookups and fixed memory usage. This approach is more efficient than traditional `seccomp`, where each `seccomp_rule_add` invocation increases the BPF bytecode size and results in linear growth for lookup times.

4.5 Artifacts Availability

The source code and scripts supporting this paper are publicly available at: `https://github.com/yunsenlei/phpsys_filter`.

5 Security Evaluation

We aim to answer two research questions: *To what extent can fine-grained modeling and enforcement constrain an RCE attack on the PHP platform? Does such an approach negatively impact programs' legitimate execution? To what extent can the approach protect itself from attacks?*

5.1 Prevention: Real-World Vulnerabilities

We tested our defense on two VMs running `Nginx` and PHP 7: one used our approach, and the other used Saphire. We focused on PHP application vulnerabilities using WordPress CVE-based exploits.

Attackers have significant flexibility after injecting malicious code. Attackers commonly use the `system` API to spawn new processes, enabling malicious

Table 1: Comparison of Saphire and our approach across RCE attacks from reported WordPress CVEs.

	CVE	Saphire	Our Approach
Process Launch	CVE-2018-7602	**Yes**	**Yes**
	CVE-2018-7600	**Yes**	**Yes**
	CVE-2020-35729	**Yes**	**Yes**
	CVE-2023-39362	**Yes**	**Yes**
	CVE-2023-39147	**Yes**	**Yes**
System Call Mimicry	CVE-2018-12613	**No**	**Yes**
	CVE-2020-8644	**No**	**Yes**
	CVE-2021-26120	**No**	**Yes**
	CVE-2022-1329	**No**	**Yes**
	CVE-2023-28115	**No**	**Yes**

actions beyond PHP's scope. A more advanced approach, system call mimicry, uses legitimate PHP APIs to replicate benign behavior at the system call level, avoiding obvious malicious calls like `sys_exec`.

Table 1 compares these strategies (labeled Process Launch and System Call Mimicry) across Saphire and our approach. Both block overt calls like `sys_exec` (already excluded from their allow-lists). However, our context-aware state machine better detects mimicry attacks, addressing subtle threats more effectively.

Saphire uses a request-level system call profile and blocks attacks in two scenarios: (1) the requested URL has an empty allow-list profile, automatically rejecting any system calls or (2) the attack uses system calls not included in the allow-list for that request. Scenario (1) applies to RCE attacks via file uploads where the malicious script, such as `uploaded_script.php`, is accessed directly through its URL. This script is unprofiled during the allow-listing phase, making such attacks easy to detect. Our evaluation primarily focuses on Scenario (2).

For instance, RCE via file inclusion allows attackers to embed malicious scripts into legitimate requests, exploiting permissive system call profiles. Using CVE-2022-1329 [19] as an example, a WordPress vulnerability permits non-admin users to modify plugin source code. This enables attackers to execute malicious scripts within any legitimate request. Our state machine model prevents such attacks by detecting script inclusions that violate defined transition rules and enforcing script-level system call profiles. This remains effective even if uploaded scripts overwrite legitimate ones.

For RCE via injection and deserialization, malicious code alters existing script behavior without introducing new scripts, often bypassing detection due to broad permissions. For example, CVE-2021-26120 targets the Smarty PHP template engine [26], exploiting a flaw in the `Smarty_Internal_Template` object that grants unauthorized access to its parent `Smarty` object. This allows attackers to overwrite template caches, executing malicious code when affected pages are reloaded. Our state machine detects the attack by monitoring function-level events, such as code evaluation and template rendering. This granularity distin-

guishes system calls triggered by these events, enabling our approach to detect and prevent malicious actions that bypass traditional defenses.

5.2 Profile Correctness: Legitimate Use

In a system call allow-list, the filter flags calls outside the list as "positive" and permitted calls as "negative." The previous section measured how each approach blocks unauthorized calls ("true positives") and avoids letting them through ("false negatives"). Here, we assess how they allow legitimate calls ("true negatives") and prevent mistaken blocks ("false positives"), indicating the profile's completeness. A complete profile correctly maps all valid system calls to the application or its states, preventing false positives.

We compare the system calls extracted by our approach with those from Confine [12], Sysfilter [6], and Saphire [4]. Confine automates extraction from containers, so we use a single PHP container for comparison. Sysfilter extracts calls from the program binary, while Saphire targets PHP's internal APIs. Like Saphire, we apply our filter only after PHP begins handling a request, using all mapped PHP internal API calls.

Table 2: A comparison of the system calls extracted across four approaches for the entire PHP engine. A lower number of allowed system calls may provide fewer attack opportunities.

Approach	Number of system calls extracted
Confine [12]	194
`sysfilter` [6]	159
Saphire [4]	123
Our work	119

In Table 2, we compare the number of system calls extracted by each approach. Confine and `sysfilter` generate more calls due to their broader analysis scope (a container or entire PHP process). By contrast, Saphire and our method attach filters when PHP handles a request, capturing only the calls triggered by scripts. Saphire's profile also includes additional `epoll` calls required between requests. Our 119 calls form a superset of all potential calls a PHP script can make before automaton-based modeling. In practice, scripts use fewer calls; for example, WordPress scripts average 32 calls in their allow-list.

In addition to the comparison, we tested our profiles for false positives by generating various request workloads targeting the application. A false positive would incorrectly block a legitimate request, resulting in an error response. To comprehensively assess this, we created multiple WordPress user accounts with varying privilege levels, from administrator to regular users. We employed automated scripts to execute actions permitted by each user role: administrative tasks included modifying site configurations, whereas normal user activities involved

viewing, commenting, and posting content. Using Xdebug [7] with `php-code-coverage` [3], we measured code coverage. In our experiments with WordPress, benign requests achieved a code coverage of 54%, and no system calls were mistakenly rejected.

5.3 Potential Attacks on the Sensors and Enforcement Systems

Our approach relies on system call traces and user-level context events, which attackers may attempt to disrupt or compromise. With a trusted kernel threat model, attackers have limited means to prevent the kernel from detecting system calls. The threat model also assumes that effective attacks must use system calls, making it infeasible for adversaries to avoid them entirely.

Prior work has highlighted concerns for system call interposition frameworks, particularly regarding time-of-check/time-of-use (TOCTOU) issues. As with `seccomp`, we avoid pointers and evaluate the system call arguments (e.g., an immediate number) directly [25]. This avoids the conditions needed for a TOCTOU attack.

Attackers might also target mechanisms capturing user-level context events: two potential evasion strategies [21] include 1) **Memory Permission Alteration**: Changing Virtual Memory Area (VMA) flags to prevent the kernel's `install_breakpoint` function from operating. This can be done by modifying ELF binaries or remapping process memory. 2) **False Context Injection**: Using `ptrace` to manipulate the execution of a PHP application and inject false context data. Both strategies require root privileges, making them harder to execute in practice.

Our evaluation, assuming an attacker with root privileges, confirmed the feasibility of these evasion techniques. However, as these attacks require root access, they are considered beyond the scope of this study. An attacker with such privileges could execute most server-side operations without bypassing the detection system.

6 Performance Evaluation

This section explores the research question: *What performance cost does this approach add to the PHP platform?*

6.1 Experiment Setup

We consider four cases that use different filtering approaches:
1. No Filter: Base case: no system call filter used.
2. Request Aware: Our filtering approach is configured to sense the application's current request during runtime and filters system calls based on a per-request profile (as in Saphire).
3. Script Aware: Our filtering approach is configured to sense the interleaving of PHP script and filter system calls based on a per-script profile.

4. Saphire: This shows Saphire's system call filtering approach. It uses `seccomp` as the underlying enforcement module and filters system calls at the per-request level.

We conduct experiments on Ubuntu 22.04 with 4 CPU cores at 2.4 GHz and 8 GB of RAM, using PHP 7.4 for Saphire compatibility. We select WordPress [30] due to its popularity and realistic workloads, and use `Apachebench` to generate requests with varying concurrency and URL patterns. Our performance metric is the request response time.

6.2 Single URL Workload

In Saphire, when users concurrently access a single URL, the system call profile for a PHP worker is established during the first request. This profile remains unchanged for subsequent requests, leading to overhead primarily driven by the operations of the underlying `seccomp` module. Although the request is unchanged in this scenario, our approach still dynamically updates the system call profile for each incoming request or script event. We then apply this updated profile to enforce the appropriate system calls.

Table 3: Average request response time (in milliseconds)

Filtering Approach \ Concurrent Users	10	25	50	100
No Filter	248	646	1302	2631
Request-Aware	252	657	1318	2667
Script-Aware	252	659	1339	2724
Saphire	260	677	1371	2765

Table 3 shows the request response time. Our request-aware filtering approach can generally keep up with the baseline across different concurrency levels, which only add 1% of overhead on average. Saphire adds around 5% overhead.

6.3 Complex Workload

When multiple users simultaneously request different URLs, each request must receive the correct system call profile. Our method automatically switches profiles at runtime using script entry/exit events. In contrast, Saphire must either restart the PHP process or dedicate separate worker pools to different request URLs. This can lead to resource imbalances. When configuring pools of PHP workers, a defender with Saphire needs to consider the size of the pool. Each script that can be requested must have a pool associated with it. It can be challenging for the defender to size the pool to optimally match the dynamics of user

access patterns. In the worst-case scenarios, a small pool of workers is constantly active while large portions of workers in a different pool are idle and underutilized. For instance, if 50 concurrent users hit two URLs in a 1:4 ratio, one pool may be overused while another remains idle, slowing overall performance. Our approach instead loads all possible profiles into a single enforcement module, avoiding such inefficiencies.

7 Related Work

The risks of remote code execution (RCE) are well-recognized, and the research community has made significant efforts to understand PHP applications and mitigate these risks. We broadly classify related work into three areas: profiling and enforcing models for PHP applications, restricting system calls in applications, and uncovering vulnerabilities in PHP applications.

7.1 Modeling PHP Applications for Protection

Static and dynamic analyses, combined with runtime protection, are often used to detect or prevent RCE attacks. These methods require runtime behavior to adhere to a pre-constructed model. ZENIDS [14] exemplifies this approach by recording a PHP application's behavior with benign user inputs and building an execution profile using an inter-procedural control flow graph. While ZENIDS captures informative user-level context, such as request data, it lacks insight into system-level interactions, which are the primary focus of attackers in RCE scenarios. Saphire [4] takes a different approach by extracting a request-level system call profile for PHP applications to prevent RCE attacks. Saphire is effective when malicious actions include system calls not covered by a request-level profile. However, its coarse-grained approach struggles to handle more subtle attack strategies that require fine-grained detection.

Our work uses a similar approach to these modeling efforts, focusing on fine-grained restrictions for system calls during program execution. By integrating detailed runtime state sensing and a custom enforcement module, we leverage this context to constrain system calls more effectively.

7.2 Restricting System Calls in Applications

System calls mediate unprivileged user-space access to privileged system resources. Wagner and Dean [29] introduced four modeling approaches: (1) a basic allow-list, as in `seccomp`, which must include all calls a program might make; (2) a call graph model, incorporating control flow; (3) an abstract stack model; and (4) a digraph model capturing transitions between consecutive calls.

Most existing work targets general programs rather than PHP. For instance, SFIP [5] enforces digraph-based transitions and tracks system call origins (instruction addresses). However, this lacks sufficient context for PHP's interpreter-based calls, which often stem from a single internal API. Our approach tracks

which specific PHP function or script initiates a system call, allowing more precise checks. Similarly, temporal system call specialization [13] refines profiles based on whether a server is initializing or serving requests. Other recent efforts use eBPF for programmable system call security [18], but still lack the fine-grained, script-level state awareness our method provides. In contrast, we automatically derive a detailed automaton for each PHP script and apply state-sensitive filters without manual labeling of program stages.

7.3 Discovering PHP Script Vulnerabilities

Prior work has identified RCE as a key threat in PHP applications. Huang et al. introduced WebSSAIR [16] to detect insecure information flows enabling script inclusion from user input. UChecker [15] focuses on file upload RCE by modeling and verifying exploit conditions via an SMT solver. Backes et al. [2] use a code property graph (CPG) that combines abstract syntax trees, control flows, and dependencies. By querying the CPG for patterns such as tainted inputs reaching sink functions, developers can detect vulnerabilities before deployment.

Our work is compatible with these prior efforts. The efforts in this section can be used to detect and correct the underlying errors in software. Until those vulnerabilities are identified and patched, our work can decrease the likelihood that an adversary can successfully exploit these vulnerabilities to implement an attack. Our approach can offer these benefits because, in contrast to the work in this section that aims to identify the software vulnerabilities, our script-level approach aims to detect and block system calls that do not match existing profiles. For more detailed sensitivity, developers can identify specific functions warranting heightened resolution (e.g., `eval`, template functions).

8 Discussion and Concluding Remarks

Applicability to other languages: The profiling approach is specifically designed for the PHP language for serving web applications. This allows us to fully explore any practical challenges of implementing the approach, such as unintended filtering or performance issues. However, our instrumentation and enforcement techniques generalize beyond PHP. In particular, we use the `perf_event_open` interface, which leverages Linux `uprobe` events. Many other frameworks (e.g., Node.js, Python) provide similar hooks at the HTTP request level, function entries, or script imports, making it straightforward to adopt our approach for other languages.

PHP Just-in-Time Compilation: PHP 8 introduced Just-in-Time (JIT) compilation as part of the Opcache extension. However, JIT-compiled code does not integrate seamlessly with runtime profiling tools that rely on intercepting PHP function calls. To address this limitation, third-party developers have created new tracing APIs [9]. In PHP 8.4, JIT compilation is not enabled by default, reducing its prevalence in most configurations. Our instrumentation extension remains compatible with the latest version of PHP when JIT is disabled.

Conclusion: Our work has explored a practical security problem in PHP applications. We have designed an automata-based approach that enables detailed, state-based filtering. In evaluation, we have found it is more sensitive and can prevent attacks that go unnoticed by current state-of-the-art techniques. We found the performance overheads for the approach are low, enabling practical deployment.

References

1. Ambionics Security: PHPGGC: PHP Generic Gadget Chains. `https://github.com/ambionics/phpggc` (2023), gitHub repository
2. Backes, M., Rieck, K., Skoruppa, M., Stock, B., Yamaguchi, F.: Efficient and flexible discovery of php application vulnerabilities. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 334–349 (2017). https://doi.org/10.1109/EuroSP.2017.14
3. Bergmann, S.: php-code-coverage: Library for collecting test coverage statistics for php code. `https://github.com/sebastianbergmann/php-code-coverage` (2023), gitHub repository
4. Bulekov, A., Jahanshahi, R., Egele, M.: Saphire: Sandboxing PHP applications with tailored system call allowlists. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2881–2898. USENIX Association (Aug 2021)
5. Canella, C., Dorn, S., Gruss, D., Schwarz, M.: Sfip: Coarse-grained syscall-flow-integrity protection in modern systems (2022)
6. DeMarinis, N., Williams-King, K., Jin, D., Fonseca, R., Kemerlis, V.P.: sysfilter: Automated System Call Filtering for Commodity Software. In: International Symposium on Research in Attacks, Intrusions and Defenses (RAID) (2020)
7. Derick Rethans: Xdebug: Debugger and profiler tool for php. `https://xdebug.org/` (2023), available: Xdebug Official Website
8. DynamoRIO Contributors: DynamoRIO: Dynamic Instrumentation Tool Platform. `https://dynamorio.org/`, accessed: 2023-12-17
9. Engineering, D.: PHP 8 observability (2021), `https://www.datadoghq.com/blog/engineering/php-8-observability-baked-right-in/`, accessed: 2024-12-01
10. Esser, S.: Utilizing code reuse or return oriented programming in php application exploits. In: Proceedings of the Black Hat Conference. Las Vegas, NV, USA (2010)
11. Gaidis, A.J., Atlidakis, V., Kemerlis, V.P.: Sysxchg: Refining privilege with adaptive system call filters. In: Conference on Computer and Communications Security. p. 1964–1978. CCS '23, Association for Computing Machinery (2023). https://doi.org/10.1145/3576915.3623137
12. Ghavamnia, S., Palit, T., Benameur, A., Polychronakis, M.: Confine: Automated system call policy generation for container attack surface reduction. In: 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020). pp. 443–458. USENIX Association, San Sebastian (Oct 2020), `https://www.usenix.org/conference/raid2020/presentation/ghavanmnia`
13. Ghavamnia, S., Palit, T., Mishra, S., Polychronakis, M.: Temporal system call specialization for attack surface reduction. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1749–1766. USENIX Association (Aug 2020), `https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia`
14. Hawkins, B., Demsky, B.: Zenids: Introspective intrusion detection for php applications. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). pp. 232–243 (2017). https://doi.org/10.1109/ICSE.2017.29

15. Huang, J., Li, Y., Zhang, J., Dai, R.: Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 581–592 (2019). https://doi.org/10.1109/DSN.2019.00064
16. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing web application code by static analysis and runtime protection. In: Proceedings of the International Conference on World Wide Web. p. 40–52. WWW '04, ACM, New York, NY, USA (2004). https://doi.org/10.1145/988672.988679
17. ircmaxell: php-cfg: A library to build and work with a control flow graph in php. `https://github.com/ircmaxell/php-cfg` (2023), accessed: 19-Nov-2023
18. Jia, J., Zhu, Y., Williams, D., Arcangeli, A., Canella, C., Franke, H., Feldman-Fitzthum, T., Skarlatos, D., Gruss, D., Xu, T.: Programmable system call security with ebpf (2023), `https://arxiv.org/abs/2302.10366`
19. National Vulnerability Database (NVD): Cve-2022-1329 detail: Elementor website builder plugin for wordpress vulnerability. `https://nvd.nist.gov/vuln/detail/CVE-2022-1329` (Apr 2022), accessed: 19-Nov-2023
20. Park, S., Kim, D., Jana, S., Son, S.: FUGIO: Automatic exploit generation for PHP object injection vulnerabilities. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 197–214. USENIX Association, Boston, MA (Aug 2022), `https://www.usenix.org/conference/usenixsecurity22/presentation/park-sunnyeo`
21. Quarkslab: Defeating ebpf uprobe monitoring. `https://blog.quarkslab.com/defeating-ebpf-uprobe-monitoring.html` (2024), `https://blog.quarkslab.com/defeating-ebpf-uprobe-monitoring.html`, accessed: 2025-04-20
22. Red Hat, IBM, Intel: SystemTap Language Reference (2023), `https://lrita.github.io/images/posts/systemtap/langref.pdf`, accessed: 2024-02-05
23. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy (2016)
24. Symfony: Home - twig - the flexible, fast, and secure php template engine. `https://twig.symfony.com/`, accessed: 19-Nov-2023
25. The Linux Kernel Documentation: Seccomp BPF (SECure COMPuting with filters) (2024), `https://docs.kernel.org/userspace-api/seccomp_filter.html`, accessed: 2024-07-07
26. The Smarty Project Contributors: Smarty: A template engine for php. `https://www.smarty.net/` (2023), accessed: 2024-02-07
27. W3Techs: Usage statistics and market share of php for websites. `https://w3techs.com/technologies/details/pl-php` (2024), accessed: 2-Dec-2024
28. W3Techs: Usage statistics and market share of wordpress. `https://w3techs.com/technologies/details/cm-wordpress` (2024), accessed: 2-Dec-2024
29. Wagner, D., Dean, R.: Intrusion detection via static analysis. In: Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001. pp. 156–168 (2001). https://doi.org/10.1109/SECPRI.2001.924296
30. WordPress: Blog tool, publishing platform, and cms – wordpress.org. `https://wordpress.org` (2023), accessed: 19-Nov-2023