Rampart: Protecting Web Applications from CPU-Exhaustion Denial-of-Service Attacks

Wei Meng[†], Chenxiong Qian[‡], Shuang Hao[*], Kevin Borgolte[§]
Giovanni Vigna[§], Christopher Kruegel[§], Wenke Lee[‡]
[†]*Chinese University of Hong Kong,* [‡]*Georgia Institute of Technology*
[*]*University of Texas at Dallas,* [§]*University of California, Santa Barbara*

Abstract

Denial-of-Service (DoS) attacks pose a severe threat to the availability of web applications. Traditionally, attackers have employed botnets or amplification techniques to send a significant amount of requests to exhaust a target web server's resources, and, consequently, prevent it from responding to legitimate requests. However, more recently, highly sophisticated DoS attacks have emerged, in which a single, carefully crafted request results in significant resource consumption and ties up a web application's back-end components for a non-negligible amount of time. Unfortunately, these attacks require only few requests to overwhelm an application, which makes them difficult to detect by state-of-the-art detection systems.

In this paper, we present RAMPART, which is a defense that protects web applications from sophisticated CPU-exhaustion DoS attacks. RAMPART detects and stops sophisticated CPU-exhaustion DoS attacks using statistical methods and function-level program profiling. Furthermore, it synthesizes and deploys filters to block subsequent attacks, and it adaptively updates them to minimize any potentially negative impact on legitimate users.

We implemented RAMPART as an extension to the PHP Zend engine. RAMPART has negligible performance overhead and it can be deployed for any PHP application without having to modify the application's source code. To evaluate RAMPART's effectiveness and efficiency, we demonstrate that it protects two of the most popular web applications, WordPress and Drupal, from real-world and synthetic CPU-exhaustion DoS attacks, and we also show that RAMPART preserves web server performance with low false positive rate and low false negative rate.

1 Introduction

Denial-of-Service (DoS) attacks are a class of attacks that aim to deteriorate the target system's availability and performance. They prevent the system from handling some or even all requests from legitimate users, by overwhelming its available resources, e.g., network bandwidth, disk space, memory, or CPU time. Consequently, users might experience long delays when interacting with the victim system, or they might be completely unable to access it. Availability and performance are essential to high-profile web servers, such as those operated by banks, news organizations, and governments, however, which are regular targets of DoS attacks [9, 21].

To degrade the performance of web servers, a common practice is to launch Distributed DoS attacks (DDoS) that flood the target system with numerous requests. Specifically, among other attacks, attackers might command thousands of computers (or more) to send attack traffic, or they might spoof the victim's IP address to launch reflected attacks [29, 34]. Fortunately for defenders, these attacks incur comparatively high cost for the attackers (e.g., acquiring a large-size botnet to mount the attack) and they can often already be detected by state-of-the-art network-level defense mechanisms [23–25, 30, 31].

Unfortunately, sophisticated DoS attacks gained significant traction recently. In sophisticated attacks, attackers use low-bandwidth, highly targeted, and application-specific traffic to overwhelm a target system [8, 12, 14, 22]. Different from traditional DDoS attacks that rely on flooding a victim system with an *extensive* amount of traffic, sophisticated DoS attacks require less resources and utilize a lower volume of *intensive* requests to attack the victim system's availability. Specifically, attackers target *expensive* or *slow* execution paths of the victim system. For example, an intensive attack might request the system to calculate computationally-expensive hashes for millions of times by specifying an unusually high iteration count for the `bcrypt` function. Particularly problematic is that sophisticated DoS attacks are difficult to detect by state-of-the-art defenses, such as source address filtering or traceback mechanisms, because they were designed to mitigate large-scale network-layer DDoS attacks [18, 23–25, 30, 31, 36, 37].

In this paper, we design and implement a defense mechanism, RAMPART, to protect a web application's back end from sophisticated DoS attacks. RAMPART aims to mitigate attacks that overwhelm the available *CPU resources* (CPU time) of a web server through *low-rate* application-layer attack traffic, which we call *CPU-exhaustion DoS attacks*. Therefore, we design RAMPART to accurately and efficiently detect and stop suspicious intensive attacks that may cause CPU exhaustion, and to be capable to block future attacks, without negatively affecting the application's availability for legitimate users.

Developing such a defense is challenging. First, attack requests can blend in well with normal requests: Similar to requests sent by legitimate users, they also arrive at a low rate. Moreover, attack requests are generally well-formed, and, thus, do not cause the application to crash or throw an exception except for possibly resource exhaustion exceptions (e.g., a stack overflow exception). In turn, it is difficult to differentiate these two kinds of requests, i.e., it is non-trivial to block only attack requests without also incorrectly blocking legitimate requests. Since a legitimate request can be mistakenly labeled as suspicious, the defense system has to quickly detect and revoke any false positive filter that blocks legitimate requests, to not reduce the application's availability unnecessarily.

To address these challenges, we leverage statistical methods and fine-grained context-sensitive program profiling, which allows us to accurately detect and attribute CPU-exhaustion DoS attacks. Specifically, RAMPART actively monitors all requests to precisely model the resource usage of a web application at the function-level. It then dynamically builds and updates statistical execution models of each function by monitoring the runtime of the function called under *different contexts*. Upon arrival of a new request, the request is then constantly checked against the statistical models to detect suspicious deviation in execution time at runtime. RAMPART lowers the priority of a request that it labeled as suspicious by aborting or temporarily suspending the application instance that is serving it, depending on the server's load. To prevent pollution attacks against the statistical models, RAMPART collects only profiling measurements of normal requests that do not cause a CPU-exhaustion DoS and that do not deviate much from the norm observed in the past. It also enforces a rate limit by network address.

RAMPART can deploy filters to prevent future suspicious requests from over-consuming the server's CPU time. It employs an *exploratory* algorithm to tackle the problems of *false positive requests* and *false positive filters*. Specifically, when a true positive attack request is detected, a filtering rule is deployed to block *similar* suspicious requests, which might include legitimate requests (false positives). RAMPART dynamically removes the deployed filter once the attack ends, to recover service for any legitimate users who might have been affected by the filter. Similarly, a false positive filter might be created if a legitimate request was incorrectly identified as suspicious. To not negatively impact an application's availability for future legitimate requests, RAMPART periodically evaluates (explores) all generated filter policies and deactivates false positive filters. In turn, this algorithm allows RAMPART to rapidly and intelligently discover false positive rules, while simultaneously thwarting true attacks.

We design RAMPART as a general defense against CPU-exhaustion DoS attacks. Importantly, to be protected by RAMPART, it is not necessary to modify a web application or its source code in any way. To emphasize the practicality of RAMPART, we implemented a prototype of RAMPART for PHP, which remains the most popular server-side programming language today [5]. Moreover, we thoroughly evaluated our prototype implementation, and we find that it incurs negligible performance overhead of less than an additional 3 ms for processing a request, i.e., roughly 0.1% of the median website load times [33].

Finally, we demonstrate that RAMPART can effectively preserve the availability and performance of real-world, non-trivial web applications when they are victim of CPU-exhaustion DoS attacks. We focus on two of the most popular open-source content management systems: Drupal and WordPress. For example, when launching *known* attacks without RAMPART's protection, then the average CPU usage increases from 32.21% to 95.05% for attacks on Drupal and from 42.21% to 94.14% for attacks on WordPress. However, if protected by RAMPART, then the average CPU usage remains comparatively stable at no more than 39.62% for Drupal and 51.40% for WordPress. Last, we demonstrate RAMPART's ability to protect the two applications from *unknown* vulnerabilities.

We make the following technical contributions:

- We present RAMPART, which is a defense that detects and mitigates sophisticated CPU-exhaustion DoS attacks against web applications by using statistical models and function-level program profiling.
- We implement RAMPART as an extension for the PHP Zend engine. Our prototype has negligible performance overhead and it can be readily deployed for 83% of websites worldwide without requiring source code modifications.
- We develop algorithms to reduce the false positive rate when detecting attacks and to mitigate any negative impact of a false positive. In turn, RAMPART has a low false positive rate of less than 1%.
- We thoroughly evaluate RAMPART with both real-world and synthetic vulnerabilities in two popular web applications, and we demonstrate that it effectively mitigates the impact of low-rate CPU-exhaustion DoS attacks and preserves application availability and server performance.

2 Rampart

In this section, we discuss the design of RAMPART, our defense mechanism to detect and mitigate sophisticated application-layer CPU-exhaustion DoS attacks (Section 2.1). Precisely, RAMPART performs context-sensitive function-level profiling to learn precise execution models for each endpoint of an application (Section 2.2). Whenever the server is overwhelmed, the system terminates or suspends anomalous prolonged application instances that it suspects to be suffering from an attack (i.e., instances it suspects are attempting to serve an attack request), to reduce the server's workload (Section 2.3). RAMPART employs a probabilistic algorithm to limit the false positive rate when stopping attacks (Section 2.4) and it constructs filtering rules to adaptively block future attacks using an exploratory algorithm (Section 2.5). Finally, we discuss how to optimize the performance of RAMPART (Section 2.6) and we detail our prototype implementation (Section 2.7).

2.1 Threat Model and Challenges

Threat Model. We consider a remote attacker that can send arbitrary HTTP(S) requests to a server serving a web application that is vulnerable to CPU-exhaustion DoS attacks. The attacker can exploit the vulnerability by sending carefully crafted requests that will consume a significant amount of the web server's CPU time. Her goal is to occupy all available CPU resources (cores) by sending multiple requests in parallel at a low rate. Attack requests are well-formed, and, thus, they cannot be easily distinguished from legitimate requests through statistical features, such as the size, or the values of the payload. She can also send legitimate requests to hide her attack among legitimate traffic. She does not, however, send numerous attack requests within a very short time window, i.e., flooding the target server, because volumetric attacks with a high attack rate can be easily detected by complementary network-based defenses, and a low attack rate is already sufficient to overwhelm the web server. Therefore, remote attackers who flood the web server with numerous requests at a time are outside the scope of our threat model.

To detect and stop low-rate CPU-exhaustion DoS attacks efficiently, we have to address five core challenges:

Detection. Different from conventional DDoS attacks, low-rate application-layer DoS attacks are difficult to detect because they do not overwhelm a web server with large number of concurrent requests. In turn, existing state-of-the-art network-layer defense mechanisms [18, 23–25, 30, 31, 36, 37] cannot detect these sophisticated DoS attacks.

Attribution. It is not straight-forward how to attribute an attack to its corresponding request(s). In fact, it is particularly difficult because attack requests exercise legitimate functionality of the web application and they do not crash the application. Indeed, they do not even hijack the application's control flow.

Prevention. Developing a mitigation strategy that effectively stops the attacks while not negatively impacting the application's availability to normal users is not trivial. For example, simplistic URL-based requests filtering techniques are ill-suited because attackers send requests to endpoints that normal users may also visit. Relying on hand-crafted features and payload values is similarly problematic because they do not scale across applications or attacks, and because real attack payloads can depend on other parameters and they may even vary per user or time for some (unknown) vulnerabilities [1].

False Positives. Naturally, any defense mechanism relying on statistical properties may have false positives, i.e., legitimate requests that are blocked by a filter, or requests that might incorrectly be identified as attack requests, and, hence, might cause a false positive filter to be deployed. Considering the nature of low-rate application-layer DoS attacks, minimizing the false positive rate and the impact of false positive filters is a major challenge.

Performance. Lastly, our defense mechanism must not introduce significant performance overhead to the protected application. In particular, users must not notice any performance degradation when the application is running at normal load.

2.2 Web Application CPU Usage Modeling

RAMPART monitors and learns profiles (models) of a web application to establish the resources it normally requires. We use the models as reference to detect suspicious requests (Section 2.3). Web application commonly provide multiple endpoints for interaction. Users can request each of those endpoints under different contexts (e.g., anonymous or authenticated), and each requires different and diverse processing resources. Therefore, a profile at the application-level or request-level is not suitable to differentiate attack requests from normal requests.

To precisely model the resource usage of a web application in different states, RAMPART employs context-sensitive function-level program profiling. Specifically, RAMPART records the CPU time spent in a function (including time spent by the operating system's kernel on behalf of the function) instead of its wall clock time, because an application instance can be interrupted and rescheduled by the operating system before the function returns. RAMPART associates the measured execution time with a unique ID, representing the application's cur-

rent execution *state*. The ID is obtained from the calling context of the function and its name. In particular, we encode the execution state (ID) by calculating the hash value of the application's past states and the name of the function being invoked. We compute the state when a function c is invoked by its parent function p as follows: $\text{STATE}(c) = \text{HASH}(\text{STATE}(p), c)$.

As a result, the ID of a function frame depends on all of its parent callers. To keep track of previous application states, RAMPART maintains a shadow call stack, where each function frame stores the application state when it is called. We push a covering *main* function to the bottom of the call stack to measure the total CPU time spent in an endpoint. We employ the name of an endpoint (e.g., `/login`) as the initial state to differentiate functions with the same name (e.g., *main*) for different endpoints.

When calculating the ID, we do not consider sibling functions, because a varying numbers of sibling functions may have returned, and they represent a similar state in the program. In addition, executed sibling functions may not necessarily influence the execution of pending functions. For example, suppose that a parent function p calls a child function s for a random number of times at runtime in a loop, before calling another child function c. If we consider the previous sibling function s, we might have to maintain hundreds or thousands of records for different instances of it, even though they consume very similar amounts of resources. Moreover, we would have different IDs for c for each run of the program. Similarly, we do not use the argument values to encode the state of a function frame because they can also be dynamic.

2.3 CPU-Exhaustion DoS Attack Detection

A straw-man approach to detect CPU-exhaustion DoS attacks is to set a global timeout in the web application because a key characteristic of such attacks is that their requests take considerable time and consume numerous CPU cycles of the victim server. However, legitimate requests can also time out and could be mistakenly identified as attack attempts. For example, a user may upload a large file that could take a long time to transfer or process.

Instead of such a straw-man approach, RAMPART monitors the CPU usage of a web server to detect CPU-exhaustion DoS attacks, which works because attackers want to occupy as many CPU cores as possible, so that the victim server is less responsive. Compared with a (global) timeout, abnormally high CPU usage is a more accurate indicator. RAMPART continuously monitors the CPU usage of the server in a fixed interval T, and computes the average CPU usage r_S over the last S observations, where S is a parameter that a system administrator configures to control the detection sensitivity. If r_S is greater than a pre-defined threshold R_{CPU} (e.g., 90%), RAMPART raises an alarm, thus, indicating that the server is overloaded, and likely victim to a CPU-exhaustion DoS attack.

Intuitively, the requests that consumed the most CPU time can be identified as the culprits that caused the CPU-exhaustion. However, this can quickly lead to false negatives. Considering a similar upload example to before, i.e., a few users are uploading large files while a real attack is being launched. If the upload requests consumed slightly more CPU time than the attack requests, then these legitimate requests would be incorrectly detected as the responsible request (false positives) and the real attack requests would evade detection (false negative), although they might always take this long to process.

Instead, RAMPART leverages the function execution models it learned (Section 2.2) to detect suspicious requests that are statistically different from the historical profile. RAMPART periodically (e.g., every 250 ms) checks the CPU time spent in functions that have not returned yet, then it compares the time with the corresponding records in the profiling database, and, finally, it identifies one request as suspicious using the following method:

Let T_{min} and T_{max} be the minimum and maximum timeout thresholds. T_C is the CPU time of a function f in the stack; μ and σ are the mean and standard deviation of T_C with the ID $\text{STATE}(f)$ in the database; k is a parameter that represents the distance from the mean. We rely on the Chebyshev inequality (Equation 1) to estimate how likely one observation differs from the mean without assuming any underlying distributions. In particular, the probability of a random variable (X) that is k-standard deviations away from the mean is no more than $1/k^2$.

$$P(|X - \mu| > k\sigma) \leq \frac{1}{k^2} \tag{1}$$

$$T_C > min(max(\mu + k \times \sigma, T_{min}), T_{max}) \tag{2}$$

Thus, RAMPART labels a request as suspicious if T_C of function f is more than $k\sigma$ away from the mean (Equation 2). RAMPART can then terminate the application instances that serve such prolonged suspicious requests to release the occupied resources *only* when the web server is overloaded. Otherwise, it repeats the same process until all functions have returned. The minimum threshold T_{min} prevents RAMPART from reporting a request as suspicious if a deeper function with very short execution time (e.g., hundreds of microseconds) times out.

The above method effectively detects suspicious requests for which the required CPU time deviates significantly from what RAMPART observed previously. When serving attack requests, then T_C will be significantly higher for some frames in the call stack compared to legitimate requests. On the contrary, when serving the file-uploading requests and if T_C for all functions will be close

to the means, then these requests will not be marked as *suspicious* (the requests always take this long to process). If they are not close the means, however, then RAMPART aborts these requests *if* the server is overwhelmed, because they are indistinguishable from attack requests.

A limitation of RAMPART is that it requires at least one observation of a function call before it can rely on the function to determine if a request is suspicious. In practice, this training phase can be completed automatically by using a fuzzer, a crawler program to traverse the web application, or an existing test harness. In fact, developers can easily collect training data when testing their applications before deploying them to production. To reduce detection variance, we recommend letting RAMPART make at least N observations (e.g., we use $N = 5$, Section 4) for each endpoint. Although RAMPART might have not collected execution profiles for all states (function calls) of a web application, it knows the execution profile of each endpoint and it can start detecting attack requests.

Another limitation is that an attacker could pollute the profiling records of an application state she selects by gradually increasing the CPU time. We make such pollution harder by sampling requests to be written into the profiling database at random. Additionally, we restrict the number of samples that can be selected from a single network address or network prefix each day. To further increase the difficulty for an attacker to pollute or drift profiling records, one can consider strategies that assign higher importance (weight) to older measurement records when computing the mean and standard deviation (Equation 2).

2.4 Probabilistic Request Termination

RAMPART marks a request as suspicious when a function consumes significantly more CPU time than it normally does. It stops serving such suspicious requests when the server is overloaded, due to a real attack or a surge in visitor traffic. While this approach stops real attacks, it can also negatively impact normal users. For example, a user may make requests that RAMPART falsely detects as an attack because they take slightly more time than the threshold that RAMPART calculated (Equation 2). Such requests, together with real attack requests, would then be terminated by RAMPART until the CPU usage is reduced below R_{CPU}.

To reduce the impact of false positives, RAMPART can rely on a probabilistic algorithm to determine if a suspicious request should be dropped. The observation is that suspicious user requests usually do not consume as much CPU time as attack requests. Instead of aborting all suspicious requests immediately, RAMPART can be lenient initially and allow some requests to require slightly more time at a lower priority. Periodically, RAMPART then checks whether these requests have timed out and becomes stricter as the execution time of a timed-out function increases. In other words, a suspicious request that is *fast* is likely to be completely processed before it would be killed. On the contrary, a *slow* suspicious request is probably an attack (a true positive) and will be aborted eventually.

Algorithm 1 Probabilistic Algorithm

```
1.  procedure INIT
2.      c ← 0, ω ← 1, β ← 1
3.      T_o ← 10 ms, s ← 5 ms, R̂_CPU ← 75%
4.      σ ←STDDEV()
5.      i ←MAX(T_o, σ)
6.      TIMER(CHECK, i)
7.  procedure CHECK
8.      c ← c + 1
9.      r ← USAGE^CPU_avg()
10.     if r > R̂_CPU then
11.         p ← (c × ω + r × β)
12.         if RANDOM(0,100) ≤ p then
13.             ABORTREQUEST()
14.         else
15.             SUSPENDREQUEST(s)
```

We also consider the server workload when determining the probability to abort a suspicious request. Specifically, the probability increases with the average CPU usage so that less CPU time is allocated to slow suspicious requests. RAMPART suspends the allowed suspicious requests temporarily to free CPU time for other requests, i.e., allowed suspicious requests have lower priority.

RAMPART's algorithm to decide whether a request should be aborted or suspended is shown in Algorithm 1. The INIT procedure is executed at a function timeout event. $\hat{R}_{\text{CPU}}$ is the (upper) CPU usage threshold. σ is the standard deviation of CPU time of the function frame. T_o is the minimum interval that RAMPART periodically evaluates if the suspicious request should be suspended or aborted. A CPU timer that expires at every interval i is set in line 6. The number of timeouts for a timer is c. ω and β correspond to the weights of the counter and CPU usage. RAMPART suspends suspicious requests for the duration of s (wall clock time).

The CHECK procedure is called after INIT and whenever the evaluation timer expires. If the web server's average CPU usage r is greater than $\hat{R}_{\text{CPU}}$, then we calculate the probability p (in percent), and abort the request probabilistically (if it is larger than a random value, line 12). Otherwise, the request is suspended. In either case, the web server can serve other normal requests first.

2.5 CPU-Exhaustion DoS Attack Blocking

RAMPART can detect and stop CPU-exhaustion DoS attacks already, but the above design of RAMPART does not prevent such attacks from affecting the victim server. RAMPART lets an attack request be served until it has consumed a significant amount of CPU time. For example, we demonstrate in Section 4.1.1 that attackers can still occupy the web server's CPU and cause CPU-exhaustion DoS by continuously sending such requests. Thus, RAMPART needs to block follow-up attack requests to further mitigate CPU-exhaustion DoS attacks.

We face two challenges in designing a prevention strategy. First, it is difficult to extract features to properly distinguish attack requests from legitimate requests. According to our threat model (Section 2.1), the two kinds of requests can be very similar. The only reliable information RAMPART has learned about an attacker is the network address (which can be spoofed) and the endpoints that are used to exploit the vulnerability. Therefore, RAMPART builds filtering policies using the source IP (network) address, the requested URI, and the request parameters (e.g., the query string and post data, i.e., keys and values of PHP's `GET` and `POST` arrays) of an attack request. RAMPART then immediately rejects a follow-up request matching any filter without further processing it.

An attacker cannot evade the filter by supplying decoy parameters because each parameter is matched independently. She can, however, try to evade using spoofed IP addresses. However, IP address spoofing is an orthogonal problem because:

1. RAMPART is a host-based defense system;
2. IP address spoofing is commonly used in reflected DDoS attacks, which are out of scope of our work;
3. Defenses exist against network-based attacks (e.g., ingress filtering, unicast reverse path forwarding) [17].

Second, a filter should be deployed neither perpetually nor ephemerally. False positives cannot be completely eliminated due to randomness in web applications. On the one hand, a user could be blocked forever by a persistent filter, unless she switches to a different IP address not used by an attacker. On the other hand, if the lifespan of a filter is too short, then an attacker can wait and launch another round of attacks.

To address the above challenge, we design an *exploratory* algorithm to adaptively adjust the lifespan of a filter, instead of setting a fixed lifespan. Specifically, each filter is assigned with a *primary lifespan* when it is first created. A matching request is immediately dropped during the filter's primary lifespan. The filter transitions into an *inactive* state with a *secondary lifespan* when its primary lifespan expires. During the secondary lifespan, RAMPART lets the application serve one matched request at a time to *explore* the result of removing the filter. RAMPART aborts this request if a CPU-exhaustion DoS attack attempt is detected, and it *renews* the filter with a longer primary lifespan to penalize the attacker. Otherwise, the filter is removed because it might have been created as a false positive or the attacks have stopped.

We present the *exploratory* algorithm in Algorithm 2. The INIT-RULE procedure is invoked when a filtering rule is first created. T_p and T_s are the rule's default *primary* and *secondary* lifespans (in seconds), which are set the server's administrator. The primary lifespan expires at time t_{expiry}. $\hat{R}_{\text{CPU}}$ and $\check{R}_{\text{CPU}}$ are the *upper* and *lower* CPU usage thresholds. Together with parameter α and β, they control if RAMPART should *explore* a matched request (line 13-16). *exploring* represents RAMPART's exploration state and is initialized to `false`.

RAMPART calls the CHECK-RULE procedure when a new request arrives. RAMPART drops all incoming requests (line 10) that match the rule (line 8) if it is still active (line 9). After it transitions into the inactive state (line 11), RAMPART may start an exploration if no one is active (line 12). Other *matching* requests received during exploration are dropped (line 22). RAMPART decides if it should explore a request (line 12-15) with a probability depending on the current average server CPU usage r, and the parameters $\hat{R}_{\text{CPU}}$, $\check{R}_{\text{CPU}}$, α, and β (line 5-6). During exploration (line 16-20), the request is aborted immediately if it is detected as suspicious (line 17). The counter c is incremented by one to set a larger new primary lifespan (line 18-19). The rule is deleted if the secondary lifespan has expired (line 24).

This algorithm controls the upper bound of the rate that one attacker can cause CPU-exhaustion DoS on a web server with a unique combination of the fields in a filter. In particular, in any $T_p + T_s$ window, an attacker can cause at most *two* attacks, which RAMPART immediately detects and stops. She cannot evade detection by sending benign requests to hide attacks, because the rule would not be destroyed unless the attacker sends only *one* attack request in a $T_p + T_s$ window. She is further penalized for sending an attack request during the filter's second lifespan with a growing primary lifespan. Therefore, an optimal attacker can cause only *one successful attack* in every $T_p + T_s$ interval (other attacks are quickly stopped).

In turn, our algorithm allows RAMPART to recover the service's availability for a false positive user as soon as the server has sufficient resources. RAMPART is unlikely to detect a false positive user request it explores as suspicious again, because the server load is expected to be lower than the upper CPU usage threshold that is used to detect attacks. Otherwise, requests for one endpoint by a user leading to a false positive would temporarily

Algorithm 2 Exploratory Algorithm

```
1.  procedure INIT-RULE
2.      T_p ← 60, T_s ← 300, c ← 1
3.      exploring ← false
4.      t_expiry ← CURRENTTIME() + T_p
5.      Ř_CPU ← 25%, R̂_CPU ← 75%
6.      α ← (R̂_CPU + Ř_CPU) / (R̂_CPU − Ř_CPU), β ← 1
7.  procedure CHECK-RULE
8.      if ISRULEMATCHED(rule, request) then
9.          if CURRENTTIME() < t_expiry then
10.             DROPREQUEST(request)
11.         else if CURRENTTIME() < t_expiry + T_s then
12.             if exploring = false then
13.                 r ← USAGE^CPU_avg()
14.                 p ← α(R̂_CPU − r×β) / (R̂_CPU + Ř_CPU)
15.                 if RANDOM(0, 100) ≤ p then
16.                     exploring ← true
17.                     if ISATTACKDETECTED(request) then
18.                         c ← c + 1
19.                         t_expiry ← CURRENTTIME() + c × T_p
20.                     exploring ← false
21.             else
22.                 DROPREQUEST(request)
23.         else
24.             DELETERULE(rule)
```

be refused as the server is overloaded and it assigns the *suspicious* requests a lower priority. The user can still access other parts of the application as long as they do not depend on the blocked one.

2.6 Performance Optimizations

RAMPART is an in-line dynamic analysis system and, hence, may incur significant performance overhead. Next, we discuss how we optimized its performance.

First, RAMPART needs to make two system calls to measure the CPU time of a function call: one before the actual function call and one after it. Here, the system call overhead can be magnitudes larger than the raw execution time when profiling some built-in functions, e.g., arithmetic functions. Therefore, we want to avoid unnecessary system calls while profiling applications at a fine granularity. One might consider the unprivileged `RTDSC(P)` instruction of x86 processors to query the Time Stamp Counter (TSC) efficiently. Unfortunately, TSC is a global counter and shared among all processes running on the same processor, including unrelated processes, which is why we cannot use it as per-process CPU counter. Instead, we disable profiling for built-in functions, as they take almost constant or negligible time. The execution time of some functions, e.g., string manipulation, however, does strictly depend on its input and we need to take them into account. Fortunately, their execution time is included when RAMPART profiles their parent functions, thus, we do not measure them separately.

We also introduce a parameter `Max_Prof_Depth` to control the overall profiling granularity. It specifies the maximum number of function frames that RAMPART profiles. If `Max_Prof_Depth` is set to 1, then only the covering *main* function is profiled. If `Max_Prof_Depth` is large, more functions are profiled, which may be inefficient as the measured CPU time is inclusive. Practically, RAMPART still blocks CPU-exhaustion DoS effectively with low overhead when trading some profiling precision for performance (Section 3 and Section 4).

Second, some overhead may be the result of input and output operations on past measurements. To improve write performance, RAMPART writes measurements in batch after each request has been completely processed. To further mitigate contention, RAMPART offloads database operations to a dedicated daemon that regularly processes the measurement data.

RAMPART also sets a wall clock timer to periodically query for historical profiling records of function frames that have not yet returned. To improve performance here, RAMPART can clear the timer after the first query to avoid interrupts because it knows when the request will be marked as suspicious. Thus, RAMPART can wait until then or until the request was processed, whichever comes first.

Finally, RAMPART can optionally sample one measurement every X requests, and, in turn, avoid the system calls to write out measurements for $X-1$ requests. The first set of system calls remain required to measure the elapsed CPU time in case of an attack. Sampling also helps to defend against pollution attacks (Section 2.3).

2.7 Implementation

We implemented a prototype of RAMPART as an extension to the PHP Zend engine in roughly 2,000 lines of C code. The RAMPART PHP extension is loaded in each PHP process and thread for function profiling and to monitor CPU usage. We use the function `getrusage` provided by Linux to measure the CPU time of a function spent by both the user code and the system calls. The daemon for processing the profiling results is implemented in 400 lines of Python code. We implemented RAMPART for PHP because it remains the most popular server-side programming language today with a market share of 83% [5]. RAMPART is language-agnostic, and it can be implemented for other server-side programming languages as it does not rely on any language-specific features.

3 Performance Evaluation

RAMPART is an in-line defense and therefore introduces some performance overhead during normal execution, which we evaluate in this section. We also investigate the performance degradation when a web application is the victim of a CPU-exhaustion DoS attack. For our evaluation, we protect two open-source web applications: Drupal 7.13 and WordPress 3.9.0. We evaluate RAMPART on these specific applications and versions because of their popularity and because they contain known real-world CPU-exhaustion DoS vulnerabilities. Following, we first describe our experiment settings and the baseline performance of the two applications (Section 3.1), then we evaluate the performance overhead introduced by RAMPART (Section 3.2), and, last, we look at the performance degradation caused by sophisticated DoS attacks with and without RAMPART (Section 3.3).

3.1 Setup and Baseline Performance

For our experiments, we use two machines, one being web server and one being the client. Both machines are running Debian Stretch (Linux Kernel 4.9.0). The web server runs Apache 2.4.25 with PHP 7.0.19-1 on an Intel Xeon X3450 quad-core CPU with 2.67 GHz and 16 GB RAM. The client is an Intel Xeon W3565 quad-core CPU with 3.2 GHz and 16 GB RAM. Both machines are on the same local area network (LAN) to eliminate any randomness that might result from sending requests over the Internet.

We created 256 user accounts after a fresh installation of each application, and we saved the application database to disk so that we can recover the state for reproducibility. Afterward, we used some accounts to interact with the two applications. We used OWASP Zed Attack Proxy (ZAP) as a network proxy to capture the interactions between the clients (users) and the applications. We also crawled all the endpoints of each web application with ZAP's spider program, and we stored the correspond requests for replay. We then removed requests for static files (e.g., JavaScript, Cascading Style Sheets, etc.) and we merged the remaining requests (generated by humans and the spider program) into the *user trace* for each application. Based on this user trace, we developed a traffic generator that can replay the trace's requests sequentially. It mimics multiple parallel users (replaying multiple interactions in parallel), of whom each is assigned one user account.

To evaluate overall server performance, we measure performance of each web application with various traffic loads (number of users). After each round of experiments, we reset the application to its initial state. We repeated each experiment five times to report average performance metrics ($N = 5$). Importantly, the traffic generator sends two consecutive requests with a 0.1 s pause in-between to simulate a large number of concurrent connections. In practice, however, the interval between consecutive requests sent by a legitimate user are much larger. For each request, we record the timestamps when it was sent (T_{start}) and when the corresponding response was received (T_{end}), and we compute the *request processing time* ($RPT = T_{end} - T_{start}$). Throughout the experiments, we also monitor the server's CPU usage.

The baseline performance of the server running the two applications is shown in Table 1. Naturally, the average server CPU usage increases as the traffic load increases. With modest loads of no more than 32 user instances, the average RPT (ARPT) of WordPress did not vary much. However, both applications exhibited significant performance degradation in their ARPT once load became heavier (64 user instances and higher). For a fair evaluation, we use 32 user sessions in the remaining experiments.

3.2 Performance Overhead

Based on the same parameters, we measure the overhead that our prototype implementation may incur. We report ARPT and average CPU usage in Table 2 for various values of `Max_Prof_Depth`, which is RAMPART's parameter to control how many function frames are profiled. Unsurprisingly, if more function frames are profiled (higher `Max_Prof_Depth`), then performance degrades more. Specifically, for Drupal, the parameter does not negatively affect the ARPT, but its increase correlates with higher CPU usage. For WordPress, the server performance remains close to its baseline performance (Table 1) while `Max_Prof_Depth` was less than five, but performance degrades when more function frames are profiled.

To investigate how `Max_Prof_Depth` might influence server performance, we recorded the number of profiled function frames and the time spent processing the measurement results by our analysis daemon. For each analysis iteration, our single-threaded analysis daemon sampled up to 100 measurement files because it could not process all files in real time if `Max_Prof_Depth` was greater than nine. The time to process 100 measurements, the average number of unique profiled function frames, and the average number of profiled function frames are shown in Table 2. The daemon's performance decreases and it can handle less files per second as more functions are profiled, which is the case because more measurement data is being generated by RAMPART per received request that the daemon must analyze.

We find that `Max_Prof_Depth` $= 5$ results in a reasonable performance for both applications. For Drupal, RAMPART's CPU overhead is 3.31% and we do not ob-

Application	Benchmark	User Instances					
		8	16	32	64	96	128
Drupal	ARPT (ms)	277.5	361.8	398.1	502.4	607.3	717.5
	CPU (%)	19.47	24.83	32.21	47.18	59.97	70.53
WordPress	ARPT (ms)	20.8	21.7	22.5	38.9	85.6	144.7
	CPU (%)	13.47	22.63	42.21	73.03	86.72	90.11

Table 1: Server performance under different user traffic loads.

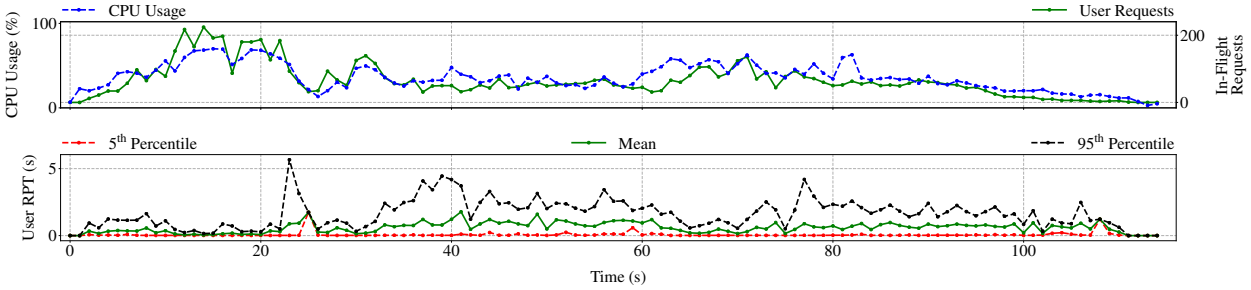

Figure 1: CPU usage and request processing time (RPT) over time for 32 users sending requests every 0.1 seconds to Drupal.

serve any overhead in Drupal's request processing time. For WordPress, the CPU overhead is 5.65% and RAMPART introduces an additional 0.2 ms (0.83%) for the request processing time on average. Overall, WordPress incurs slightly higher overhead than Drupal because more functions are profiled (Table 2).

Finally, we investigate the RPT of Drupal with 32 concurrent user instances with RAMPART enabled (Figure 1). The bottom of the figure shows the 5th percentile, mean, and 95th percentile of the RPTs for requests *sent* for each one second interval. The x-axis is the time elapsed since the start of experiment and the y-axis is the RPT. The number of in-flight requests (RIF) in each one-second window are shown in a green solid line, and the average server CPU usage is shown in a blue dashed line in the top figure. Evidently, CPU usage remains modest throughout the experiment. Following, we show how a only few attack requests can quickly exhaust the CPU (Section 3.3), and how RAMPART preserves server performance (Section 4).

3.3 DoS Attack Performance Degradation

We measure the performance degradation of the server when a CPU-exhaustion DoS attack was launched against a web application. Specifically, we evaluate two kinds attacks for both web applications: `XML-RPC` for both Drupal and WordPress (CVE-2014-5266 [4]), `PHPass` for Drupal (CVE-2014-9016 [2]) and Wordpress (CVE-2014-9034 [3]). The `XML-RPC` attacks allow remote attackers to cause a CPU-exhaustion DoS by sending a large XML document containing a significant number of elements. The `PHPass` attacks allow remote attackers to cause a CPU-exhaustion DoS by supplying a long password that is improperly handled by the password hashing functions. We also evaluated several other CVEs (e.g., CVE-2012-1588, CVE-2013-2173, and CVE-2014-5019), which can similarly cause CPU-exhaustion DoS, which we omit due to space limitations.

We use our traffic generator to send attack traffic from the client machine to the server. Each generated attack payload takes Drupal and WordPress between 10 and 30 seconds to process. We then launch multiple attackers concurrently via our traffic generator. For each attacker session, the generator sends two consecutive requests with five seconds break in-between. Assuming that the RPT for an attack request is 25 seconds, then the attack traffic rate with 30 attacker sessions is one attack request per second. This rate is significantly lower than that of a typical DDoS attack (tens of thousands of requests per second or more). Indeed, such sophisticated application-layer DoS attacks require significantly fewer resources to be successful.

In our experiments, we configure the *user* traffic generator to run 32 user sessions (Section 3.2), and the *attack* traffic generator to operate 8 or 16 attacker sessions. We launch the *attack* traffic generator five seconds after we started the *user* traffic generator. As in our baseline performance experiments, we repeat each experiment five times to measure the average performance metrics, i.e., the server's CPU usage, the number of in-flight requests each second (RIF), and the request processing time (RPT) of user sessions and attacker sessions. RAMPART is disabled for all of these experiments.

Application	Benchmark	Max_Prof_Depth 1	3	5	7	9	11	13
Drupal	ARPT (ms)	397.6	389.0	400.9	393.0	413.6	412.6	410.9
	CPU (%)	34.53	34.80	35.62	36.32	38.52	40.94	44.20
	Number of Unique Functions	12	76	567	1,421	2,473	4,019	5,405
	Number of Functions	341	2,167	12,677	31,152	53,263	80,186	110,606
	Processing Time (ms)	11.3	29.5	142.5	321.8	543.7	886.7	1,147.1
WordPress	ARPT (ms)	23.7	23.7	23.5	24.6	29.1	36.4	41.6
	CPU (%)	44.25	43.12	49.08	56.56	61.60	69.37	68.41
	Number of Unique Functions	17	199	846	3,186	7,909	13,337	17,410
	Number of Functions	422	4,479	15,314	42,957	89,080	136,910	170,904
	Processing Time (ms)	11.4	46.1	169.1	572.8	1,470.2	2,653.7	3,529.0

Table 2: Web server performance and daemon statistics for RAMPART with 32 users for different `Max_Prof_Depth` values.

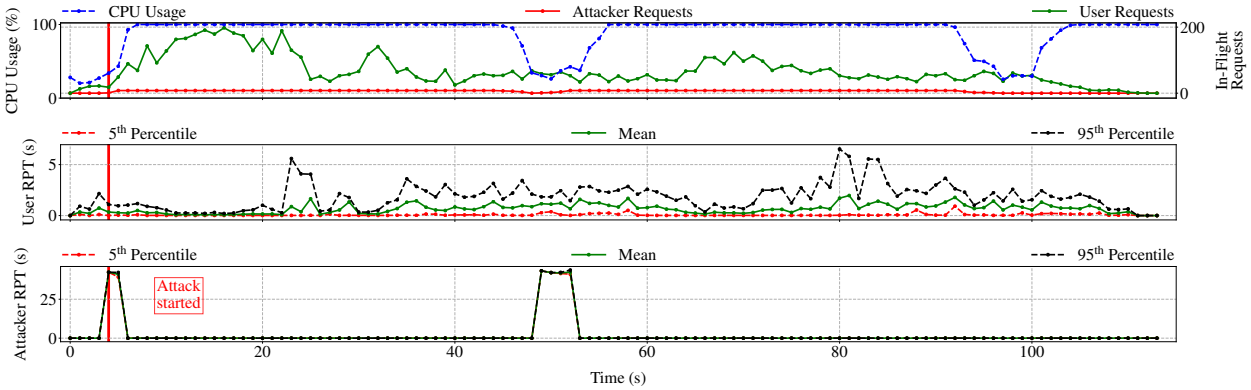

Figure 2: CPU usage and RPT over time for 8 `PHPass` attackers on Drupal without RAMPART.

For each figure, the middle and bottom graphs show the 5th percentile, mean, and 95th percentile of the RPT of *user requests* (middle) and *attack requests* (bottom) that were sent in each one second window. The green and red solid lines in the top figure represent the RIF of user sessions and attacker sessions, and the blue dashed line shows the server's CPU usage. A red solid vertical line in each three graphs indicates when we started the attack.

Launching 8 `PHPass` attacker sessions attack against Drupal (Figure 2), the server spends on average 42 seconds on processing one attack request. The CPU remains almost fully occupied once we launch the attack, except for the five seconds break when we paused the attack. In fact, the results show that an attacker sending only 0.17 requests per second (8 / (42 + 5)) can already exhaust CPU resources of a vulnerable server. Performance degrades severely with 16 parallel attacker sessions, at which point the CPU usage stays close to 100% throughout the experiment. Corresponding to doubling the number of attacker sessions, the server has to spend almost twice as much time (82 seconds, or 1.95x) to serve each request, likely because of the operating system's process scheduling. For 16 attackers, the required attack rate is 0.18 requests per second (16 / (82 + 5)).

The results for the other three attacks, `XML-RPC` on Drupal, `PHPass` on WordPress, and `XML-RPC` on WordPress, are shown in Figure 3, Figure 4, and Figure 5.

The mean CPU usage and the ARPT for all the experiments is summarized in Table 3. For Drupal, the two attacks consume between 52.4% and 62.84% additional CPU time and they cause a 36% slowdown in processing user requests. The ARPT of WordPress is more sensitive to both attacks, causing an increase of 40% to 118% in ARPT and consuming between 41.65% and 51.93% additional CPU time.

4 Mitigation Evaluation

For RAMPART to be an effective defense, it must successfully preserve the availability of a web application from CPU-exhaustion DoS attacks. Therefore, we first investigate whether RAMPART can correctly detect and stop attacks exploiting known real-world CPU-exhaustion DoS vulnerabilities (Section 4.1). Next, we look at whether RAMPART can effectively protect web applications from unknown CPU-exhaustion DoS attacks (Section 4.2).

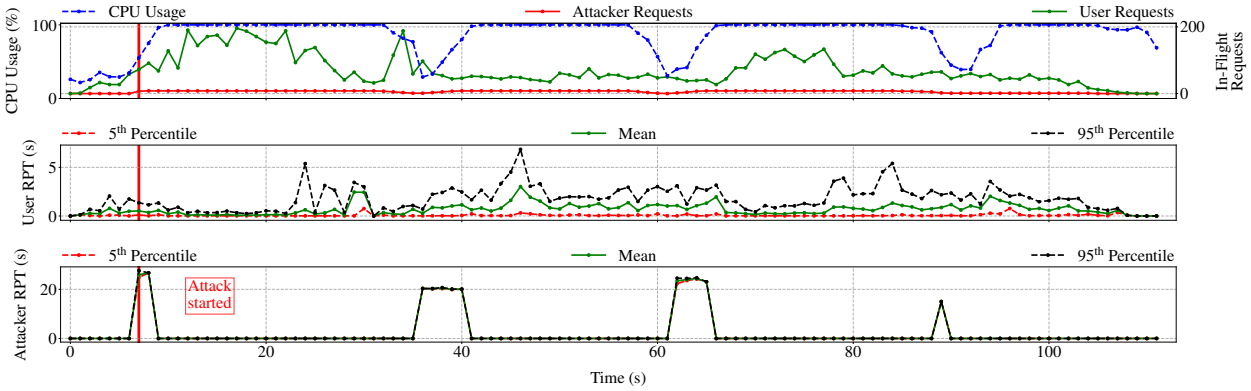

Figure 3: CPU usage and RPT over time for 8 `XML-RPC` attackers on Drupal without RAMPART.

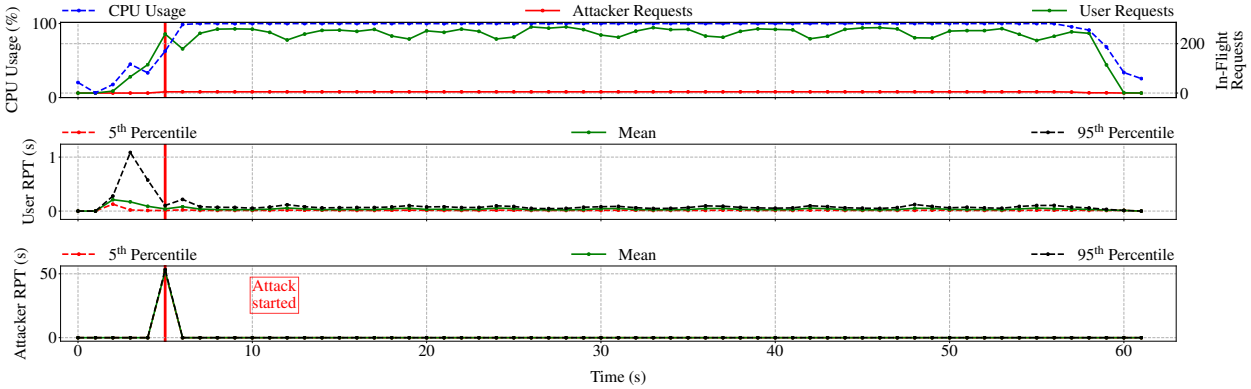

Figure 4: CPU usage and RPT over time for 8 `PHPass` attackers on WordPress without RAMPART.

We also study if RAMPART may mistakenly mark a legitimate request as an attack request, i.e., a false positive, and what the consequences are. For example, a user may initiate *slow* requests that appear similar to attack requests. Blocking such requests while an active attack is occurring is acceptable because there is no good way to differentiate such requests from the attack requests (Section 2.1). However, it is unnecessary and undesirable to constantly reject such legitimate requests when the application is not under attack.

4.1 Mitigation of Known Attacks

We evaluate how RAMPART can mitigate attacks exploiting the real-world vulnerabilities that we studied (Section 3.3). We are particularly interested in understanding:

1. How well does RAMPART help preserve server performance and availability when attacks occur?
2. How long stays an aborted attack request alive before it is terminated by RAMPART?
3. How many attack requests are not aborted by RAMPART, i.e., what is the false negative rate (FNR)?
4. How many user requests are aborted, i.e., what is the false positive rate (FPR)?

To answer these questions, we perform the following experiments: First, we evaluate RAMPART's ability to detect attack requests in the *stop-only* experiments (Section 4.1.1). Here, RAMPART uses the probabilistic algorithm (Algorithm 1) to lower a suspicious request's priority by either aborting or suspending it, but it does not deploy any filters to block requests. In turn, RAMPART checks all the requests sent by attackers. Next, we evaluate whether RAMPART can preserve server performance by *stopping* and *filtering* suspicious requests. In the *stop-and-filter* experiments (Section 4.1.2), RAMPART additionally uses the exploratory algorithm (Algorithm 2) to synthesize and deploy filters to block future attack requests. Here, we set the primary lifespan (T_p) to 10 seconds and the secondary lifespan (T_s) to 30 seconds. We assign a unique local IP address to each user/attacker session, so that RAMPART can distinguish the different instances.

We evaluate two threshold values (50% and 75%) for the CPU usage threshold $\hat{R}_{\text{CPU}}$, which RAMPART uses to determine if a server is under attack. We report the average request processing time (ARPT), average server CPU usage, FPR, and FNR for user requests and attack requests over five runs per configuration. The RPT of false positive requests that RAMPART aborted are not included in the user ARPT.

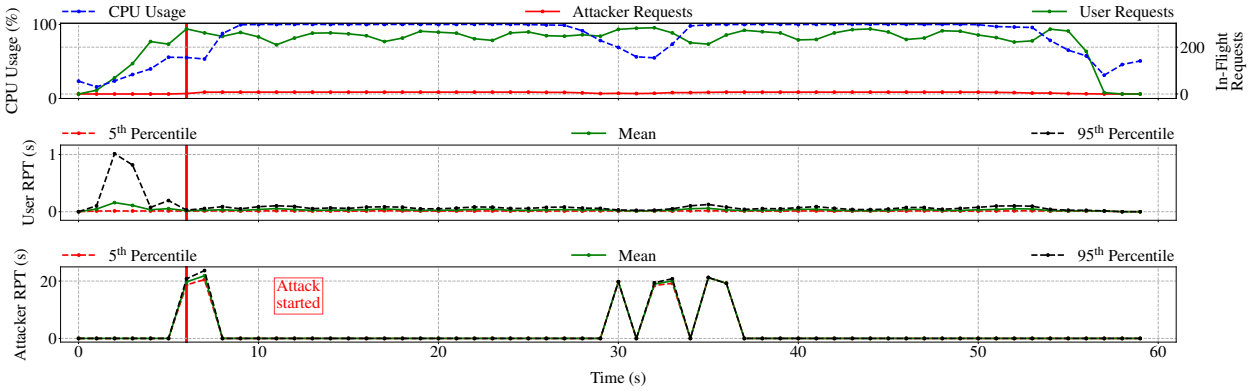

Figure 5: CPU usage and RPT over time for 8 `XML-RPC` attackers on WordPress without RAMPART.

			Attack			
		No Attack	PHPass [Attackers]		XML-RPC [Attackers]	
Application	**Benchmark**		8	16	8	16
Drupal	ARPT (ms)	398.1	461.2 (1.16x)	519.6 (1.31x)	458.3 (1.15x)	541.7 (1.36x)
	CPU (%)	32.21	88.95	95.05	84.61	94.91
Wordpress	ARPT (ms)	22.5	37.0 (1.64x)	49.0 (2.18x)	31.5 (1.40x)	41.7 (1.86x)
	CPU (%)	42.21	89.71	94.14	83.86	92.09

Table 3: Average request processing time of requests and server CPU usage with RAMPART's defense turned off.

4.1.1 Stop-Only Experiments

We summarize the results of the stop-only experiments in Table 4. We observed *no false negative* in our experiments, i.e., *all* attack requests were detected and eventually aborted, which demonstrates that RAMPART accurately detects CPU-exhaustion DoS attacks.

However, some user requests were also aborted by RAMPART as false positives in the Drupal `PHPass` experiment with 8 attacker sessions. Upon closer investigation of the logs and traffic traces of Drupal, some requests took the server more than several seconds to process, even when it was not under attack (black spikes in Figure 1). Some of those requests were marked as suspicious because several function frames deviated from their execution models. However, the overall impact was limited:

1. *Not all* such requests were aborted by RAMPART.
2. Requests of *only a few* users were aborted, although all users sent the same requests.

This is the case because RAMPART *only* terminated application instances serving a suspicious request when the server was overloaded. Nevertheless, the FPR is always less or equal to 0.33%, i.e., less than 18 out of 5,344 user requests were mistakenly aborted by RAMPART.

At the same time, RAMPART helps to preserve server performance and availability substantially, compared to the attack results without RAMPART (Table 3). The ARPT for user requests (ARPT-U) during the `PHPass` attacks on Drupal and WordPress are close to their baseline counterparts (Table 1). However, ARPT-U during the `XML-RPC` attacks on the web applications did not improve significantly. On the other hand, the ARPT for attack requests (ARPT-A) is long, with attack requests being processed for up to 2,294 ms (Drupal) and 787 ms (WordPress) before RAMPART aborted them. This explains why average CPU usage did not drop back to the baseline (Table 1) but remained slightly higher. We also observe that `PHPass` attack requests consumed more CPU resource with a higher CPU usage threshold $\hat{R}_{\text{CPU}}$.

Finally, we look at 8 attacker sessions launching the `PHPass` attack against Drupal with $\hat{R}_{\text{CPU}}$ set to 50% (Figure 6). The magenta dashed lines in the middle and bottom graphs represent the number of aborted user requests (middle) and attack requests (bottom). In the first 20 seconds of the experiment, RAMPART quickly aborted all attack requests because the server's CPU usage was above the threshold. Some requests were aborted even when the CPU usage in the top figure appears to be lower than the 50% threshold, which is because RAMPART monitors CPU usage at a shorter interval (10 ms), while the CPU data in the top figure was collected each second using the `mpstat` command. When the server load decreased, the attack requests could occupy the CPU for up to five seconds until the CPU usage crossed the threshold again. In turn, this behavior demonstrates the need for deploying filters to block suspicious requests to prevent CPU usage oscillation. Nevertheless, RAMPART detects and blocks attacks much earlier with a CPU threshold close to but above the expected CPU usage during normal operation.

		CPU Threshold for Attack							
		50%				75%			
		PHPass [Attackers]		XML-RPC [Attackers]		PHPass [Attackers]		XML-RPC [Attackers]	
Application	**Benchmark**	8	16	8	16	8	16	8	16
Drupal	ARPT-U (ms)	392.3	397.9	463.6	536.3	378.1	408.9	465.0	506.4
	ARPT-A (ms)	2,093	1,988	988.6	1,089	2,294	2,017	1,175	1,368
	CPU (%)	45.10	51.76	39.41	43.73	48.76	53.62	38.79	39.65
	FPR (%)	0.10	0.15	0.00	0.00	0.02	0.33	0.00	0.00
WordPress	ARPT-U (ms)	24.9	27.1	28.1	36.8	24.4	27.0	26.8	39.0
	ARPT-A (ms)	404.3	472.4	546.2	772.9	521.0	515.2	526.8	787.6
	CPU (%)	53.74	58.98	53.06	55.27	56.09	60.30	50.64	54.61
	FPR (%)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 4: Server performance in the stop-only experiments.

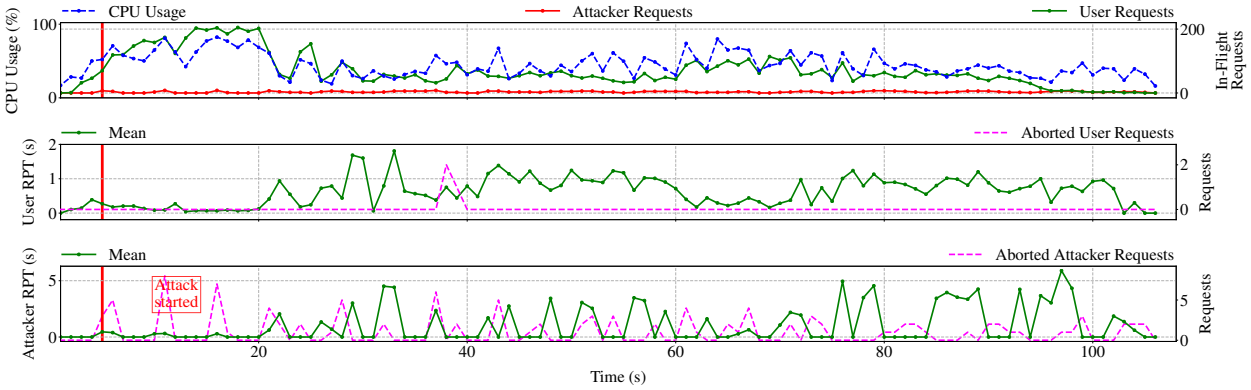

Figure 6: CPU usage and RPT over time for 8 `PHPass` attackers on Drupal with RAMPART in the stop-only experiment.

4.1.2 Stop-and-Filter Experiments

We present the results of the *stop-and-filter* experiments in Table 5. Analog to the stop-only experiments, we observed *no false negative* in the stop-and-filter experiments. However, the FPR increased compared to the stop-only experiments because RAMPART drops any request matching a filter created from false positive requests until the filter's primary lifespan has expired. In fact, these events are evident in the Drupal `PHPass` experiment with 8 attacker sessions and $\hat{R}_{\text{CPU}} = 50\%$ (orange dashed line in Figure 7, which represents the number of requests that were dropped because of a filter). Around the 35$^{\text{th}}$ second and 39$^{\text{th}}$ second, two user requests were detected and aborted as false positives and two matching filters were created. As a result, 16 additional requests from these two users were also dropped in the following T_p seconds. The primary lifespan of the last rule then expired at the 49$^{\text{th}}$ second. RAMPART then explored a matching request (the blue dashed line) at around the 58$^{\text{th}}$ second according to the exploratory algorithm (Algorithm 2) and it detected that the filtering rule was a false positive. RAMPART's FPR in stop-and-filter mode is still negligible at less than 0.69%.

Although RAMPART's stop-and-filter mode blocked some legitimate requests, it also immediately blocked the majority of attack requests (86.5%) and entirely prevented them from consuming any additional CPU time. The remaining 21 attack requests (13.5%) were also all detected as suspicious and aborted. In fact, 8 of the aborted requests were the initial requests sent by the 8 attackers, i.e., the earliest that any defense could have detected them as suspicious. RAMPART explored the remaining 13 requests and eventually also detected them as suspicious. Since the attackers sent requests at an interval of five seconds, which is shorter than T_s, RAMPART incremented the primary lifespan of a filter as penalty each time an exploring request was detected as suspicious.

Because RAMPART blocked most of the attack requests immediately, it preserved the web server's performance as if no attack had occurred (Table 5). In particular, the average CPU usage and the ARPT of user requests are much closer to their baseline (Table 1) compared to the stop-only experiments (Table 4). The ARPT of attack requests is an order of magnitude smaller. Overall, the results illustrate that RAMPART can effectively protect web applications from known CPU-exhaustion DoS attacks using the exploratory algorithm (Algorithm 2).

		CPU Threshold for Attack							
		50%				75%			
		PHPass [Attackers]		XML-RPC [Attackers]		PHPass [Attackers]		XML-RPC [Attackers]	
Application	**Benchmark**	8	16	8	16	8	16	8	16
Drupal	ARPT-U (ms)	394.7	427.1	423.4	460.4	400.9	418.6	437.4	471.6
	ARPT-A (ms)	203.6	228.3	148.1	172.2	258.9	166.6	160.4	181.0
	CPU (%)	38.51	38.76	36.30	37.68	38.84	39.62	36.30	37.73
	FPR (%)	0.60	0.00	0.25	0.00	0.69	0.00	0.15	0.00
WordPress	ARPT-U (ms)	24.1	26.1	25.6	26.8	24.4	26.1	24.5	25.1
	ARPT-A (ms)	142.1	234.4	205.9	220.5	152.8	242.3	226.3	180.2
	CPU (%)	45.92	51.40	49.89	50.74	49.15	50.98	50.91	52.14
	FPR (%)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5: Server performance in the stop-and-filter experiments.

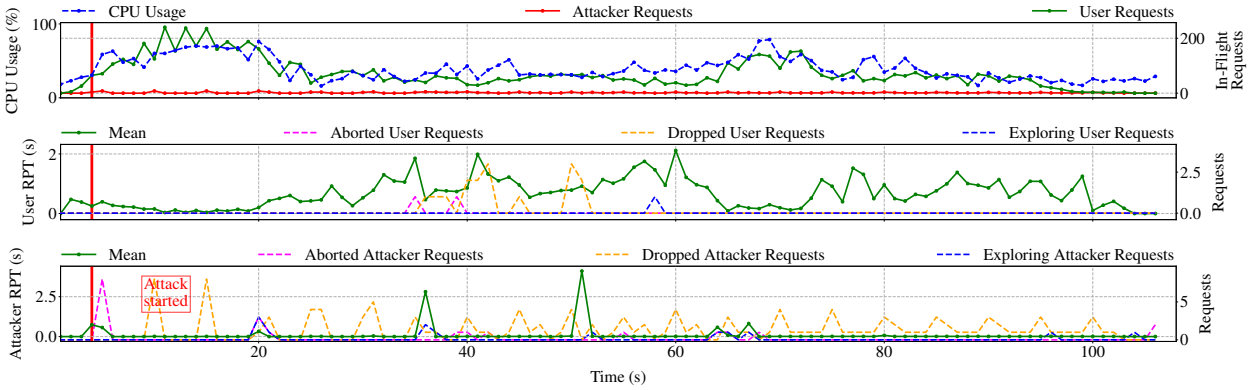

Figure 7: CPU usage and RPT over time for 8 `PHPass` attackers on Drupal with RAMPART enabled in the stop-and-filter experiment.

The results for the remaining three experiments with $\hat{R}_{\text{CPU}} = 50\%$, namely, `XML-RPC` on Drupal, `PHPass` on WordPress, and `XML-RPC` on WordPress, are shown in Figure 8, Figure 9, and Figure 10.

4.2 Mitigation of Synthetic Attacks

Compared to static vulnerability analysis tools that look for specific features in the source code, RAMPART does not require an application's source code, nor does it require any knowledge about specific CPU-exhaustion DoS vulnerabilities. Instead, RAMPART is a generic defense that automatically detects known and unknown application-level CPU-exhaustion DoS attacks at runtime dynamically.

We demonstrate RAMPART's ability to detect and mitigate such attacks in web applications. Beyond the vulnerabilities that we explored, we automatically inserted CPU-exhaustion DoS vulnerabilities into the source code of the two web applications at random locations. We configured RAMPART to record all invoked functions when serving a request for the two web applications, and we then inserted a vulnerability (Listing 1) into a function that was randomly chosen. The vulnerable code calculates the hash value of a variable `$v` by repeatedly invoking the `md5` function (line 11). The number of iteration in the loop is controlled by the parameter `$exp`, which an attacker can set through the `dos-exp` query parameter. In our experiment, attacker requests set `$exp` to 24 to cause CPU-exhaustion DoS (i.e., 2^{24} `md5` invocations).

For each application, we randomly chose 50 vulnerabilities (requests) and launched 16 attacker sessions. We set the average CPU threshold R_{CPU} to 75%. All 50 vulnerabilities in WordPress were successfully exploited, while only 21 vulnerabilities in Drupal could be exploited because the other 29 vulnerable functions were not invoked. They could not be invoked because they require to be set up by other requests beforehand, which we did not replay.

We report the results with and without RAMPART (Table 6). The average CPU usage threshold to determine if RAMPART successfully mitigated an attack against Drupal is 45% and for WordPress it is 55%. RAMPART successfully mitigates all attacks with $\hat{R}_{\text{CPU}} = 50\%$. However, some attack requests were incorrectly classified as benign. These false negatives occurred for Drupal because the server load was light (less than the 50% threshold) when those requests arrived. Although RAMPART did not abort those requests, it flagged them as suspicious.

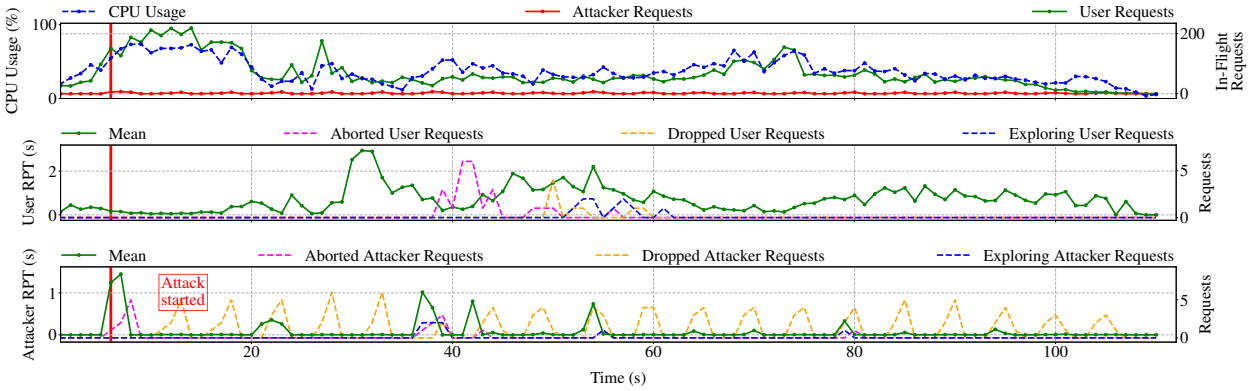

Figure 8: CPU usage and RPT over time for 8 `XML-RPC` attackers on Drupal with RAMPART enabled in the stop-and-filter experiment.

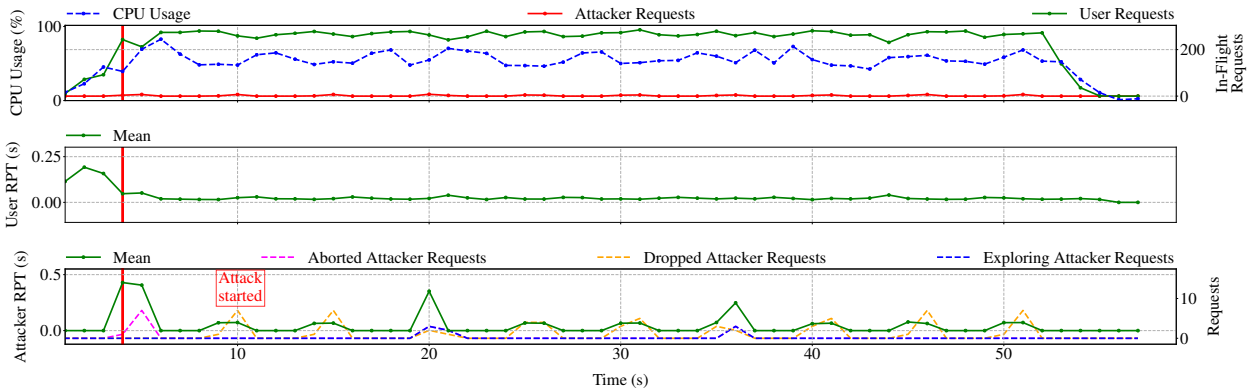

Figure 9: CPU usage and RPT over time for 8 `PHPass` attackers on WordPress with RAMPART in the stop-and-filter experiment.

Application	Benchmark	RAMPART Enabled	RAMPART Disabled
Drupal	Successful Attacks	0	21
	ARPT-U (ms)	436.5	519.7
	ARPT-A (ms)	290.5	29,631
	CPU (%)	39.15	90.56
	FPR (%)	0.03	N/A
	FNR (%)	1.31	N/A
WordPress	Successful Attacks	0	50
	ARPT-U (ms)	25.8	38.9
	ARPT-A (ms)	157.5	37,966
	CPU (%)	51.05	92.91
	FPR (%)	0	N/A
	FNR (%)	0	N/A

Table 6: Web server performance in the synthetic attack experiments with RAMPART being enabled and disabled.

Overall, the synthetic attacks experiments demonstrate that RAMPART can detect and mitigate CPU-exhaustion DoS attacks regardless of the location of the vulnerable code, i.e., it can detect and mitigate attacks not only for front-facing code, but it can also detect and mitigate attacks for (third-party) library functions. Our prototype is implemented as an extension to the PHP engine (and can be similarly implemented for other languages), and, thus, it can adapt to any change of an application's source code without requiring any manual interaction or reconfiguration. RAMPART can automatically detect new vulnerabilities that might be introduced by unintentional source code modifications. On the contrary, a developer using a static vulnerability detection tool would need to run it each time she modifies the code. Considering RAMPART's effectiveness and low overhead, RAMPART is a practical defense to protect applications from CPU-exhaustion DoS attacks.

5 Related Work

We compare RAMPART to the most relevant work, i.e., sophisticated DoS vulnerability detection, program profiling techniques, and anomaly detection.

DoS Vulnerability Detection. CPU-exhaustion DoS attacks received significant attention from researchers over the past years. Existing research focused on finding vulnerabilities (bugs) that can be exploited to launch sophisticated DoS attacks. In turn, prevention of the attacks is manual by fixing the detected bugs before an application is deployed. *Safer* performs static taint analysis and control-dependency analysis to identify loops and recursive calls whose execution can be controlled by a remote attacker [10]. Similarly, *SaferPHP* uses static taint anal-

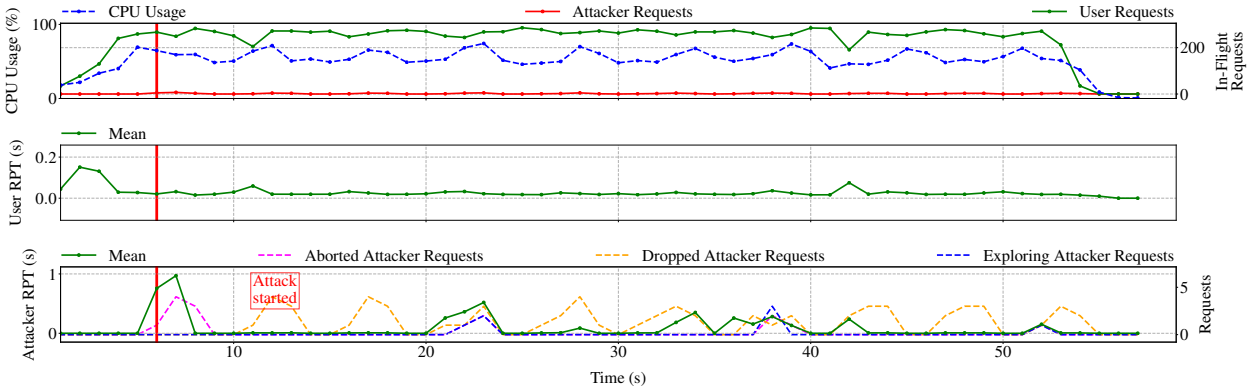

Figure 10: CPU usage and RPT over time for 8 `XML-RPC` attackers on WordPress with RAMPART in the stop-and-filter experiment.

```php
<?php

$v = time() + 86400 * 30;
$exp = 0;

if(isset($_GET["dos-exp"])) {
  $exp = $_GET["dos-exp"];
}

for($i = 0; $i < pow(2, $exp); $i++) {
  $v = md5($v);
}

?>
```

Listing 1: Snippet of vulnerable PHP code.

ysis to find loops whose execution can be influenced by network inputs [32]. It then uses symbolic execution to detect whether the network inputs can trigger the loops to run infinitely. Xiao et al. proposed ΔInfer, which is an approach to detect workload-dependent performance bottleneck loops by inferring iteration counts of the loops using complexity models [35]. *Torpedo* detects second-order DoS vulnerabilities using taint analysis and symbolic execution [26]. *SlowFuzz* is a dynamic testing tool that generates inputs triggering worst-case algorithmic behavior for several well-known algorithms [27].

Although these systems can detect CPU-exhaustion bugs before the applications are deployed, they commonly rely on additional manual analysis to confirm vulnerabilities or reduce false positives. They also incur additional opportunity cost because developers need to run them whenever the application's code or any of its dependencies are updated. Most important, they do not prevent attacks after an application has been deployed.

Instead of using static program analysis, RAMPART dynamically monitors a web application's state and determines automatically if the current state deviates significantly from the expected state. In turn, RAMPART automatically adapts to any change to the application or its libraries without requiring source code. RAMPART achieves a low false positive rate by leveraging a probabilistic algorithm and by updating the filtering rules intelligently with an exploratory strategy, and it exhibits false negatives only if an attack is not severe enough to consume significant CPU resource.

Program Profiling. The program profiling implementation of RAMPART is inspired by prior work related to flow-sensitive and context-sensitive profiling [6, 7, 13, 15, 16]. Here, a function's execution time is counted in different contexts based on the calling context tree. That is, they accumulate all functions that are called on the current execution path, to distinguish the same function called under different contexts. For RAMPART, we adopt a similar profiling strategy: We compute a hash value to encode the current execution state. Correspondingly, we can profile the running time of each called function in different contexts, and we can build a statistical execution model for each function. Moreover, during profiling, we compare the profiled functions to their statistical models, which allows us to identify the request that caused the CPU-exhaustion DoS attack, and which enables RAMPART to block similar requests in the future.

Anomaly Detection. RAMPART employs anomaly detection techniques to detect suspicious requests. The simplest anomaly detection approach is to set a static threshold for each feature, and to generate alerts when some or all the feature values are below or above their thresholds. Instead of a static threshold, RAMPART learns a dynamic threshold for function execution time because it is impractical to determine a static threshold for each function accurately and a priori, as their execution time can vary greatly in different execution contexts. Prior work employed supervised learning algorithms to build anomaly detection models [11, 19, 20, 28], which stands in contrast to RAMPART: We leverage anomaly detection models using statistical methods, but without requiring any labels during training.

6 Conclusion

Sophisticated Denial-of-Service (DoS) attacks targeting application-layer vulnerabilities can cause significant harm by severely degrading the performance and availability of a victim server over a prolonged period with only few carefully crafted requests.

In this paper, we present RAMPART, which is a system that protects web applications from sophisticated DoS attacks that would otherwise overwhelm the server's available CPU resources through carefully crafted attack requests. RAMPART performs context-sensitive function-level program profiling and learns statistical models from historical observations, which it then employs to detect and stop suspicious requests that could cause CPU-exhaustion DoS. RAMPART also adaptively synthesizes and updates filtering rules to block future attack requests. We thoroughly evaluated RAMPART's effectiveness and performance on real-world vulnerabilities as well as synthetic attacks for two popular web applications, Drupal and WordPress. Our evaluation demonstrated that RAMPART is robust against a varying number of attackers and that it can effectively and efficiently protect web applications from CPU-exhaustion DoS attacks with negligible performance overhead, low false positive rate, and low false negative rate.

7 Acknowledgments

We thank the anonymous reviewers for their helpful suggestions and feedback to improve the paper. This material is based on research supported by DARPA under agreement FA8750-15-2-0084, NSF under agreement CNS-1704253, ONR under grants N00014-09-1-1042, N00014-15-1-2162 and N00014-17-1-2895, and the DARPA Transparent Computing program under contract DARPA-15-15-TCFP-006. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views, findings, conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of DARPA, NSF, ONR, or the U.S. Government.

References

[1] CVE-2013-2173, Feb. 2013. URL https://nvd.nist.gov/vuln/detail/CVE-2013-2173.

[2] CVE-2014-9016, Nov. 2014. URL https://nvd.nist.gov/vuln/detail/CVE-2014-9016.

[3] CVE-2014-9034, Nov. 2014. URL https://nvd.nist.gov/vuln/detail/CVE-2014-9034.

[4] CVE-2014-5266, Aug. 2014. URL https://nvd.nist.gov/vuln/detail/CVE-2014-5266.

[5] Usage Statistics and Market Share of PHP for Websites, Nov. 2017. URL https://w3techs.com/technologies/details/pl-php/all/all.

[6] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, June 1997.

[7] T. Ball. Efficiently Counting Program Events with Support for On-line Queries. *ACM Trans. Program. Lang. Syst.*, 16(5):1399–1410, Sept. 1994.

[8] U. Ben-Porat, A. Bremler-Barr, and H. Levy. Vulnerability of Network Mechanisms to Sophisticated DDoS Attacks. *IEEE Transactions on Computers*, 62(5):1031–1043, May 2013.

[9] British Broadcasting Company (BBC). Thai Government Websites Hit by Denial-of-Service Attack, 2015. URL http://www.bbc.com/news/world-asia-34409343. BBC News.

[10] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov. Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities. In *Proceedings of the 22nd IEEE Computer Security Foundataions Symposium (CSF)*, Port Jefferson, NY, 2009.

[11] N. V. Chawla, N. Japkowicz, and A. Kotcz. Editorial: Special Issue on Learning from Imbalanced Data Sets. *SIGKDD Explor. Newsl.*, 6(1):1–6, June 2004.

[12] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2003.

[13] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, Boston, MA, June 1982.

[14] M. Guirguis, A. Bestavros, I. Matta, and Y. Zhang. Reduction of Quality (RoQ) Attacks on Internet End-Systems. In *Proceedings of the 24th IEEE International Conference on Computer Communications (INFOCOM)*, Miami, FL, Mar. 2005.

[15] R. J. Hall. Call Path Profiling. In *Proceedings of the 14th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, May 1992.

[16] R. J. Hall and A. J. Goldberg. Call Path Profiling of Monotonic Program Resources in UNIX. In *Proceedings of the USENIX Summer 1993 Technical Conference on Summer Technical Conference - Volume 1*, Cincinnati, OH, June 1993.

[17] Internet Society. Addressing the Challenge of IP Spoofing, Sept. 2015. URL https://www.internetsociety.org/doc/addressing-challenge-ip-spoofing.

[18] J. Ioannidis and S. M. Bellovin. Implementing Pushback: Router-Based Defense Against DDoS Attacks. In *Proceedings of the 9th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2002.

[19] M. V. Joshi, R. C. Agarwal, and V. Kumar. Mining Needle in a Haystack: Classifying Rare Classes via Two-phase Rule Induction. In *Proceedings of the 2001 ACM SIGMOD/PODS Conference*, Santa Barbara, CA, May 2001.

[20] M. V. Joshi, R. C. Agarwal, and V. Kumar. Predicting Rare Classes: Can Boosting Make Any Weak Learner Strong? In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, Edmonton, Alberta, Canada, July 2002.

[21] T. Kitten. DDoS Attacks Against Banks Increasing, 2015. URL http://www.bankinfosecurity.com/ddos-a-8497.

[22] A. Kuzmanovic and E. W. Knightly. Low-rate TCP-targeted Denial of Service Attacks: The Shrew vs. The Mice and Elephants. In *Proceedings of the 14th ACM SIGCOMM*, Karlsruhe, Germany, Aug. 2003.

[23] X. Liu, X. Yang, and Y. Lu. To Filter or to Authorize: Network-Layer DoS Defense Against Multimillion-node Botnets. In *Proceedings of the ACM SIGCOMM*, Seattle, WA, Aug. 2008.

[24] X. Liu, X. Yang, and Y. Xia. NetFence: Preventing Internet Denial of Service from Inside Out. In *Proceedings of the ACM SIGCOMM*, New Delhi, India, Aug. 2010.

[25] D. Moore, G. M. Voelker, and S. Savage. Inferring Internet Denial-of-Service Activity. In *Proceedings of the 10th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2001.

[26] O. Olivo, I. Dillig, and C. Lin. Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[27] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[28] C. Phua, D. Alahakoon, and V. Lee. Minority Report in Fraud Detection: Classification of Skewed Data. *ACM SIGKDD Explorations Newsletter*, 6(1): 50–59, June 2004.

[29] C. Rossow. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[30] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *Proceedings of the 11th ACM SIGCOMM*, Stockholm, Sweden, Aug. 2000.

[31] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-Based IP Traceback. In *Proceedings of the 12th ACM SIGCOMM*, San Diego, CA, Aug. 2001.

[32] S. Son and V. Shmatikov. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security (PLAS)*, San Jose, CA, June 2011.

[33] J. Stevens. How Slow is Too Slow in 2016?, Feb. 2016. URL https://www.webdesignerdepot.com/2016/02/how-slow-is-too-slow-in-2016/.

[34] R. van Rijswijk-Deij, A. Sperotto, and A. Pras. DNSSEC and Its Potential for DDoS Attacks: A Comprehensive Measurement Study. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, Vancouver, Canada, Nov. 2014.

[35] X. Xiao, S. Han, D. Zhang, and T. Xie. Context-sensitive Delta Inference for Identifying Workload-dependent Performance Bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, Lugano, Switzerland, July 2013.

[36] A. Yaar, A. Perrig, and D. Song. Pi: A Path Identification Mechanism to Defend against DDoS Attacks. In *Proceedings of the 24th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2003.

[37] A. Yaar, A. Perrig, and D. Song. SIFF: A Stateless Internet Flow Filter to Mitigate DDoS Flooding Attacks. In *Proceedings of the 25th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2004.