RuleCache: Accelerating Web Application Firewalls by On-line Learning Traffic Patterns

Xiaoyi Chen[1*], Qingni Shen[1], Peng Cheng[2], Yongqiang Xiong[2], Zhonghai Wu[1]
[1]*Peking University*, Beijing, China
[2]*Microsoft Research*, Beijing, China
{xiaoyi.chen, qingnishen, wuzh}@pku.edu.cn, {pengc, yongqiang.xiong}@microsoft.com

***Abstract*—Web Application Firewall (WAF) is widely deployed in cloud to protect web applications, whose performance becomes one of the major bottlenecks for web services. In this paper, we comprehensively analyze several root causes that downgrade WAF's efficiency. Inspired by that, we build a caching system RuleCache to devise optimization strategies for improving WAF's performance. Among, Rule Ordering Cache is online learning an optimal order of the ruleset for a better performance of blocking. Rule Result Cache reuses rule results of targets, saving large repetitive computations. Additionally, Rule Pre-pruning Cache aims to cut extra overhead by processing the static rules in the offline stage. Our evaluation demonstrates that the prototype can improve the performance by up to 3.85x, 1.57x, and 2.4x respectively with the above modules, and up to 5.5x in total.**

***Index Terms*—Web Application Firewall (WAF), rule set, genetic algorithm, performance gain**

I. Introduction

To meet the rapid growth need of Internet service, a considerably increasing number of web applications are deployed. Meanwhile, the incidents involving web security grow excessively over the recent years. Especially since 2020, the environment is just right for cyber criminals to strike during the COVID-19 pandemic. Thus, the owners and maintainers of the web applications are anxious to keep their services safe from the malicious attacks. One of the most popular countermeasures is the Web Application Firewall (WAF).

WAF is a firewall monitoring and filtering all the HTTP traffic going in and out of the web applications. It could effectively protect the web applications from attacks such as cross-site scripting (XSS), SQL injection and denial-of-service (DoS). The hardware WAFs, containing Netscaler MPX WAF [1], Barracuda WAF [2], Imperva SecureSphere Appliances [3], and F5 Big-IP ASM model 10200 [4], are efficient but very expensive. The software solution is another trend to implement WAF integrated with reverse proxy or web server such as Microsoft Azure Application Gateway [5], Amazon AWS WAF [6], and Cloudflare [7]. Compared with hardware WAFs, it is a good option to balance between flexibility and price especially with the rapid development of cloud.

Most of the WAF services are claiming to protect the web applications from the most 10 critical web application security risks presented by the Open Web Application Security Project (OWASP) foundation. The OWASP Top 10 [8] threats is a widely used standard to measure the coverage of WAF's protection. To target at the web application threats, especially the OWASP TOP 10, some rule sets came out to provide generic attack detection rules for WAF. As the most recognized open-source rule set, Core Rule Set (CRS) [9] is proposed as a pluggable set of rules for WAF which covers a comprehensive attacks include the top 10 most critical web application security risks.

However, by surveying the WAF solutions over the market, we found that the poor efficiency of the WAFs hardly satisfy the requirement of nowadays web applications. The main overhead of WAF comes from rule processing based on two observations: (1) The target computation of rule-matching can be non-trivial such as complex regular expressions (regex) [10], and there are a huge number of repetitive computations in target values among different requests, bringing redundant overhead. Additionally, the overhead of rule-matching significantly varies when processing different rules, because it largely depends on the complexity of regex patterns and inputs. Intuitively, partially caching time-consuming and high-frequency rule processing results can reduce the rule-matching latency by saving repetitive target computations. (2) For rulesets that do not impose a mandatory ordering on rule checking, the naive practice of sequentially going through rules can result in processing many unnecessary rules. Conceptually, if the triggering rule can be prioritized at the top of the ruleset, the system effectively achieves early-termination, thus minimizing unnecessary rule checking.

These findings enlighten us to design an efficient rule cache engine for future generation WAFs. Unlike the traditional web cache, our cache aims to cache the targets of requests. We design our RuleCache to devise optimization strategies mainly containing three modules (Figure 1). In the online engine, note that online learning represents learning online traffic sliced by the time window, i.e., the Analyzer, consisting of Rule Result Cache and Rule Ordering Cache, learns online traffic in one time window, and applies the results to the executor to guide WAF in the next time window. The executor co-locates together with the real-time WAF. Rule Result Cache caches the intermediate results of matching targets with higher *weight*, saving large repetitive computations brought by duplication. Rule Ordering Cache is online learning an optimal order of the ruleset for a better performance of blocking via genetic algorithm. Additionally, we also present other incremental

*The work was done during the author's internship at Microsoft Research.

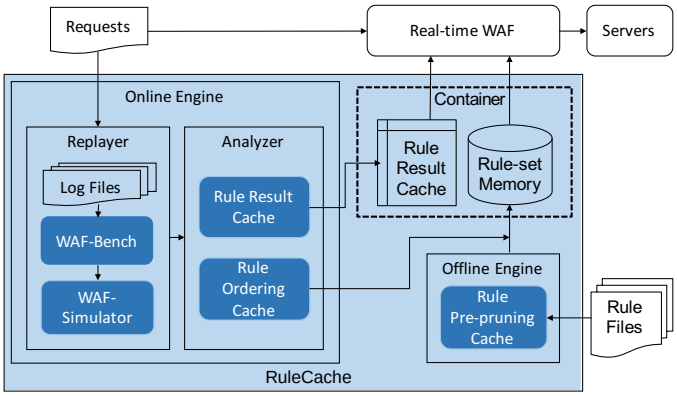


Fig. 1: Overall Architecture of **RuleCache**

optimizations. That is, Rule Pre-pruning Cache will analyze and pre-process the static rules during the offline stage.

To evaluate the benefits of **RuleCache**, we have integrated it into a popular open-source rule engine ModSecurity [11] and benchmark with its publicly available ruleset OWASP CRS [12]. We monitor the real traffic to make up our benchmark, containing $861,988$ requests from Microsoft Azure Application Gateway for six days from July 16, 2019 to July 21, 2019. Empirical results show that Rule Result Cache can achieve 7%-57% performance gain varying workload and cache percentage. The gain brought by Rule Ordering Cache grows from 22% to 74% in different ratios of malicious traffic. And Rule Pre-pruning Cache improves original performance by 1.4x. Our prototype can improve WAF's efficiency up to 5.5x in total. Therefore, it is an effective way to boost the current performance of WAF by designing and implementing one efficient rule cache engine.

Our contributions are as follows:

- We explore the root causes of WAF's poor performance. That is, the target computation of rule-matching can be non-trivial and there are redundant repetitive computations in target values among different requests. Moreover, the naive practice of sequentially going through rulesets result in processing many unnecessary rules.
- We design an efficient rule caching engine **RuleCache** with several modules. Rule Result Cache partially caches the intermediate results of requests. Rule Ordering Cache deals with rule dependency, formulates the overhead of rule location, and therefore acquires the optimal solution.
- We implemented a prototype based on our design, improving WAF's performance by 1.57x, 3.85x respectively and 5.5x in total.

The rest of this paper is organized as follows. Section II provides background information of WAF. Section III explores the root causes which downgrade WAF's performance and motivates two scenarios to build a high-efficiency rule cache engine for WAF, proposing distinct challenges. Section IV designs **RuleCache** guided by above insights and then Section V implements an prototype of **RuleCache**. Section VI evaluates its efficiency. Lastly, Section VIII concludes this paper.

II. Background

In this section, we briefly introduce Web Application Firewall (WAF) and its general workflow of rule processing.

A. Web Application Firewall (WAF)

The last decade has seen an increasing popularity of web applications as primary platforms for services delivery over Internet [13]. Meanwhile, web security has become one of the most challenging problems. In 2019, overall web attacks on endpoints increased by 56 percent. More than 1.3 million unique web attacks (e.g., SQL injection, cross site scripting (XSS), cross-site request forgery (CSRF) [14]) were blocked every day [15]. Motivated by the urgent need for web security, research efforts have been devoted for mitigating attacks on web applications [16]–[18]. One of prevalent mechanisms is web application firewall (WAF), an appliance for monitoring HTTP requests and responses between clients and back-end servers [13], [19], [20].

There are two main WAF implementations over the market. The hardware WAFs, such as F5 Big-IP ASM [4] and Barracuda [2], are very expensive. Moreover, their annual updating cycle is too long to cope with new threats. The software solution is another way to implement WAF integrated with reverse proxy or web server, which is a better option to balance between flexibility and price. Besides, considering its privilege in scalability and adaptability to new environment, software WAF has already been a general trend with the rapid development of cloud. Furthermore, there is a strategic planning assumption that by year-end 2023, more than 70% of public web applications will use software WAFs delivered as cloud service [21].

B. Rule Processing

Rule-based software WAFs exploit a pluggable set of rules to process HTTP traffic. Core Rule Set (CRS) is the most recognized open-source rule set today [9], it covers a comprehensive attack detection include top 10 most critical web application risks presented by Open Web Application Security Project (OWASP) [8]. Thus, we take CRS for instance to illustrate WAF's workflow of rule processing.

1) Target: Compared with traditional network firewall, WAF parses each request into multiple targets (`VARIABLES`) and each target would be checked by different rules separately. There are more than 20 targets defined in CRS, e.g., `ARGS`, `COOKIE`, `XML:/*`, `REQUEST_HEADERS`, and etc..

2) Rule: The rule syntax of ModSecurity is as follows:

```
1  SecRule VARIABLES OPERATOR [ACTIONS]
```

`SecRule` is a header of each rule. `OPERATOR` is used to define matching conditions for targets, e.g., `@rx` (regular expression), `@streq` (same string) and `@ipmatch` (same IP). `ACTIONS` are performed if the condition is matched. `ACTIONS` can be classified into two: disruptive actions and

other actions. Disruptive actions include deny (packets rejected) and pass (packets allowed), while other actions (e.g., logging) have little effects on final decision.

When WAF processes a rule, it firstly extracts all targets from HTTP requests and transforms them respectively. Then it executes operators to check the match condition, and finally performs actions.

3) Rule Set: CRS divides 759 rules in total by attack types into 24 separated files called rule groups. They are loaded into WAFs with a sequential order every time. Most rule groups are independent with each other but rules have dependency within a rule group. Many rulesets do not fix a mandatory ordering of rules, and it dynamically re-orders rules according to how likely they would match the given input.

III. Observations for Existing WAFs

In this section, we study root causes of WAF's bad performance and then motivate two potential optimizations from the observations.

A. Poor Performance of WAFs

We classify WAFs into three categories, namely **black-list**, **white-list** and **White-and-Black-list**.

- Black-list WAFs have a pluggable set of rules to filter anomaly traffic by the signatures including WebKnight, IronBee, ModSecurity and Shadow Daemon [22]. [1]
- White-list WAFs such as NAXSI [23] only accept expected patterns in HTTP requests instead of recognizing the known attacks.
- White-and-Black-list (i.e., combination) WAFs such as Lua-resty-waf [24] use white-list rules before black-list rule set.

We conduct several experiments to learn the performance of existing WAFs in Table I. We use request per second (RPS) to measure the performance of WAFs and KA/N-KA represents whether the proxy is kept alive. To obtain the RPS of different WAFs, we utilize WAF-Bench [25] to send GET requests for 1 minute from back-end servers through each WAF on our testbed (§VI-A). The rulesets used in each WAF is also listed in the table. Among, "limited" represents that IronBee only imports limited rules of CRS, and "custom" represents that Lua-resty-waf imports some customized white-list rules.

Table I shows poor efficiency for black-list WAFs, while Lua-resty-waf and NAXSI achieve higher performance because white-list helps pass a wide range of requests. However, white-list has a high false positive rate of detection in the initial stage, and it needs continuous tuning by experienced engineers to maintain the white list. Therefore, to eliminate the gain of white-list, we import the same rules to compare their performance in Table II, which depicts that Lua-resty-waf achieves even worse performance than ModSecurity. Overall, the performance gap that WAF brings to Nginx — for instance, the slowdown of ModSecurity is up to 10x — has been turned out to be a bottleneck.

[1]Note that ModSecurity provides interface for customers to customize white-listing rules, but the majority of rule set it leverages are blacklist.

TABLE I: Performance Comparison of WAFs

Reverse Proxy	Mechanism	Ruleset	64B (KRPS)		10240B (KRPS)	
			N-KA	KA	N-KA	KA
Nginx	\	\	11.98	16.32	11.01	15.51
Nginx + IronBee	Black-list	Limited	1.01	1.33	0.97	1.31
Nginx + ModSecurity	Black-list	CRS	1.51	1.72	1.09	1.66
Nginx + NAXSI	White-list	Doxi	10.62	15.66	10.00	14.83
Nginx + Lua-resty-waf	Combination	Custom	5.33	6.56	5.04	6.12

TABLE II: ModSecurity VS. Lua-resty-waf

	64B (KRPS)		10240B (KRPS)	
	N-KA	KA	N-KA	KA
Mod (no rule)	6.18	7.53	5.77	7.04
Lua (no rule)	6.15	7.82	5.77	7.15
Mod (48 rules)	5.58	6.95	5.23	6.31
Lua (48 rules)	5.11	6.35	4.81	5.85

To explore its root causes, in this paper, ModSecurity with CRS is chosen as an example for WAF performance bottleneck study. Because of ModSecurity's mature and reliability, it becomes the most popular open-sourced WAF, which has been widely used by web application maintainers and cloud service providers like Microsoft Azure [26]. However, it should be noted that our work can be applied to most rule-based WAFs because their workflows are similar.

B. Redundant Target Computation

In this section, we reveal that poor performance sources from a huge number of redundant target computation and thus motivate our first strategy: Target-based Cache.

1) Motivation: Because of target-specific filtering in WAF, intuitively, complicated requests with multiple targets will cost more than simple ones. To prove this intuition, we send simple-requests traffic and real traffic to ModSecurity and quantify WAF's performance gap.

Here we randomly select 100,000 simple GET requests and real traffic respectively and send them from one client to ModSecurity-enabled Nginx (IIS and Apache have the similar results) to measure the proxy engine's performance.[2] `Paranoia Level` is a setting ranging from 1 to 4 which represents the sensitivity level of CRS.

From Table III, the overhead of real traffic is twice as much as simple requests in `Paranoia Level` 1; while the gap is up to 4x in `Paranoia Level` 4.[3]

It reveals that compared with simple requests, complex HTTP requests with multiple targets downgrade performance. And as `Paranoia Level` increases, the performance gap widens.

2) Insights: Inspired by this finding, caching the complex requests will help reduce the overhead.

We initially want to leverage existing Web Page Caching technology [27], [28] to achieve this goal. However, it is hard to apply it to WAF because Web Page Caching only considers Request URL but WAF focuses on different targets to do a comprehensive check (§II-B). Besides, most URLs

[2]The workloads and testbed are described in Section VI-A

[3]64B and 10240B only varies in the length of response

TABLE III: Performance of Traffic

Single Core ModSecurity (KRPS)	Simple Requests 64B	Simple Requests 10240B	Real Traffic
Paranoia Level 1 (low)	1.72	1.62	0.83
Paranoia Level 2 (mid)	1.61	1.49	0.53
Paranoia Level 3 (high)	1.54	1.36	0.42
Paranoia Level 4 (very high)	1.48	1.09	0.39

TABLE IV: Distribution of Target Values

	App ❶	App ❷	App ❸	App ❹
ARGS	7.4%	51.3%	18.9%	20.3%
ARGS_NAMES	51.2%	99.2%	64.8%	48.4%
COOKIE	2.2%	100%	13.5%	36.1%
COOKIE_NAMES	79.1%	100%	94.3%	56.0%
USER-AGENT	37.4%	100%	83.3%	63.3%
Whole Requests	1.16%	0.78%	1.66%	4.35%

are different which will bring a high cache miss ratio. In that case, typical web cache doesn't work, so we consider to build a Target-based Rule Cache.

We explore the distribution of target values. We analyzed the real product traffic for 1 hour from four different typical applications. App ❶ provides software solutions and services. App ❷ is an internal data fetching application of a global security company. App ❸ provides web service from an ISO Certified Company. App ❹ is a manufacturer's internal network for data transferring.

Table IV shows the total frequency of top 10 prevalent values in each target. HTTP requests share similar patterns. Especially the targets `ARGS`, `COOKIE` and `USER-AGENT` are very common to be shared. As a result, when rule detects these targets, it will do repetitive operations.

Target-based Cache can help cache these results so that ModSecurity does not need to repeatedly check the same field in different requests, which will cut much overhead.

3) Challenges: It turns out to be a conflict between too many target values and limited target cache pool. So how to select caching items is a challenge. To address this issue, we propose two objectives. According to that, our strategies are proposed in Section IV-A.

- **"The Fewer, The Better"**: Processing fewer rules will achieve better performance. Caching the whole request can directly deliver the results and process no rules. However, it is useless because most requests are different.
- **"The Higher, The Better"**: Caching higher frequency values will achieve a higher cache hit ratio. Target-based Cache partially caches the high-frequency target values and their intermediate results of rule-processing.

C. Redundant Rule Processing

Target-based Cache can reduce the overhead of repetitive detection, but traffic still need rule processing when the cache miss happens. However, the naive practice of sequentially going through rules can result in processing many unnecessary rules. In this section, we reveal the potential gain of ruleset re-ordering.

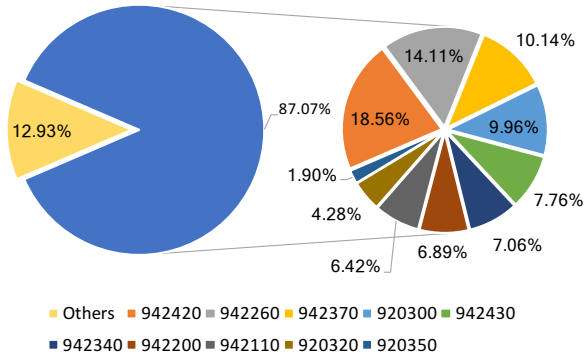


Fig. 2: Top 10 Triggered Rules

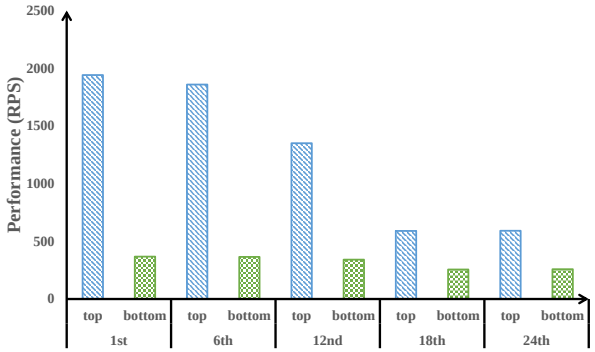


Fig. 3: Performance with Different Rule Location

1) Motivation: It is universally known that some attacks are more popular in real world. We sent real traffic extracted from workloads (§VI-A) to count the triggered frequency of rules and found that 87.07% of attacks are blocked by top 10 rules as shown in Figure 2. More interesting thing is that 7 rules among them have prefix 942 in the Rule ID, which shows that SQLI Injection dominates in the whole threats.

As a result, different rules have various impacts on real-world traffic. Just a few rules can block the majority of traffic.

2) Insights: Motivated by this finding, we propose an approach to re-schedule the rules. Under *self-contained mode*, ModSecurity blocks the request as soon as it triggers one rule. So if WAF could block an attack at a very early stage, then WAF could cut overhead. It enlightened us that the ordering of triggered rules matters performance.

We selected Rule ID 942330 from top 10 rules to compare the performance by adjusting its location. We moved the rule group (i.e., rule file) containing 942330 in the ruleset, and put 942330 at the top and bottom of the group. Then we flooded requests which only trigger 942330. Shown in Figure 3, performance is better when the group has a more front position and the rule is at the top. However, moving the rule group from 18^{th} to 24^{th} location does not make a difference. It is because that ModSecurity audits the requests by rules in 5 phases[4]. Rules in the last 6 files target at phase 3 and 4 while

[4]Request header, request body, response header, response body and logging.

Rule 942330 is at phase 2. Even though we move 942330 to the end of ruleset, ModSecurity still processes 942330 in phase 2.

From above experiments, we can infer that adjusting the rule's location in the ruleset can accelerate the blocking efficiency of ModSecurity. And the overhead of a ruleset is not static but related to traffic patterns. Based on this insight, we aim to design an online learning optimal algorithm to tune rules' priority.

3) Challenges: Designing a run-time reordering algorithm for WAF is non-trivial. Specifically, we identify the following challenges:

- **Challenge i:** Rules cannot be moved freely because of dependency. Most rule groups are independent, but re-ordering rules within a group may cause incorrectness.
- **Challenge ii:** Rules' priority cannot simply depend on the triggered frequency. There are other influence factors such as the processing time of rules.

D. Lessons Learned

Learning from this section, we propose two potential optimizations and address distinct challenges in each stage:

Target-based Cache is more complicated than Web Page Caching that it partially cache high-frequency target values. The challenge is how to determine cache replacement policy and collect related metrics.

Rule Set Re-ordering aims to improve the efficiency of rule processing. It is challenging to find the optimal order of rules with minimal overhead.

Next, we build upon these insights to design optimization strategies, working towards the ideal of WAF rule engine.

IV. Proposed Design

The insights learned in Section III are instrumental in our design of **RuleCache**. Figure 1 shows an overview of our proposed design. In the online engine, we utilize WAF-Bench [25] and WAF Simulator[5] to replay online traffic in one time window, delivering the results to Analyzer which consists of Rule Result Cache (§IV-A) and Rule Ordering Cache (§IV-B). These two modules will respectively generate an optimal rule-order and a rule-result cache table to guide ModSecurity in the next time window. Additionally, we add Rule Pre-pruning Cache (§IV-C) module in off-line engine to deal with the static rules in the compiling process.

When one request enters **RuleCache**, it firstly checks whether some targets hit the cache in rule-result table. If the result is "deny", it can deny the request and send back directly. If not, it will bypass the cache-hit targets and process the dynamic rule-set in an optimal order.

Next, we describe detailed optimization strategies of each module.

[5]A homemade tool to simulate WAF and meanwhile inspect the statistics of traffic patterns, e.g., target values' distribution.

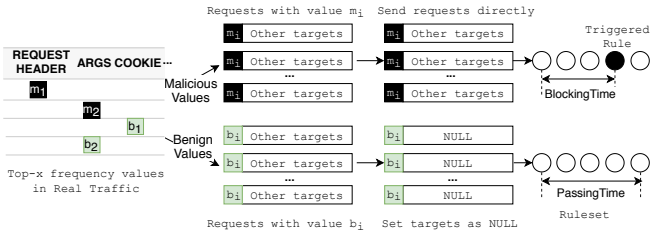


Fig. 4: Workflow of Metrics Collection

A. Rule Result Cache

To build a Target-based Cache (§III-B), we answer following two questions: how to define the cache replacement policy, and how to collect the detailed metrics.

1) Cache Replacement Policy: We need to cache the value of targets and its processing result (i.e., pass or deny) in this module. Then when one request queries its target value cached, **RuleCache** sets this target as NULL and simplifies the request. Our cache table is like white-list filtering to some extent. However, there are more than a million target values in just 10 minutes whereas the rule cache table is limited. So we need to define our cache replacement policy to cache the better ones. In preliminary, we define a term $weight$ to sort the priority of values.

An ideal cache replacement policy must take several metrics into account, such as temporal locality and update patterns [29]. Based on previous insights, we cache the target values with higher repetitive frequency and higher processing overhead to reduce more redundant computations. We define our policy as a $weight$-based policy. $weight$ is calculated as follows:

$$Weight = Frequency \times ProcessingTime \quad (1)$$

Where:

- *Frequency*: The target value's occurrence frequency in a time window.
- *Processing time*: The time to process one specific target value through all rules. However, some rules may have multiple targets. We only record the time of processing the certain target instead of the whole rule.

So we need to classify the values into two cases to collect the *processing time*. When the value is passed (i.e. benign value), its *processing time* is the overhead of dealing with it. When the value is blocked (i.e. malicious value), *processing time* is the overhead on tackling with the whole request.

2) Detailed Metrics Collection: To calculate the $weight$, we need to collect *Frequency* and *Processing time* in WAF Simulator. The process is shown in Figure 4.

Frequency. For each time period, we calculate instant frequency of target values and get the average frequency by moving average algorithm. In order to reduce the memory overhead, we just keep the TOP x frequency list of all the target values.

According to our analysis, we classified the target values into benign values and malicious ones. For the benign values, we pick top x_1 values whose frequencies are more than $f_1\%$. We notate them as b_i $(i \in [1, x_1])$. For the malicious values, we pick top x_2 values which account for more than $f_2\%$. We notate them as m_i $(i \in [1, x_2])$. Obviously, x is the sum of x_1 and x_2. We counted the value distribution of the targets by real traffic, and $f_1\%$ and $f_2\%$ are selected as 10%.

Processing time. Then, we send traffic to WAF Simulator and trace the processing time of the target values. Because real traffic varies, it is time-consuming for WAF Simulator to measure the processing time of all the values at runtime. Therefore, we need to rank them and only do measurements for more frequent values.

For the benign values, we extract the requests with b_i from real traffic to make up our dataset. Especially, to eliminate the processing time of other targets, they should be set as `NULL`. As shown in Figure 4, *Processing time* of benign value b_i is calculated as (2), where there are n requests in real traffic:

$$ProcessingTime(b_i) = \frac{PassingTime(b_i)}{n \times Frequency(b_i)} \quad (Frequency(b_i) > f_1\%) \tag{2}$$

For the malicious values, we choose top x_2 values. Then we also make a malicious dataset which contains all the requests with specific values and send these requests to WAF Simulator. Different with benign ones, we directly send these requests to WAF Simulator to obtain the blocking overhead. *Processing time* of malicious value m_i is calculated as follows:

$$ProcessingTime(m_i) = \frac{BlockingTime(m_i)}{n \times Frequency(m_i)} \quad (Frequency(m_i) > f_2\%) \tag{3}$$

B. Rule Ordering Cache

To address challenges in Section III-C, our approach has two steps: how to deal with the rule dependency, and how to determine the priority of rules.

1) Rule Dependency: Rule dependency can be categorized into two types: data hazards and control hazards.

Data hazards occur when rules modify some values of requests in a processing chain. Ignoring potential data hazards can result in detecting faults. There are two `ACTIONS` causing data hazards, belonging to RAW (Read After Write) dependency:

- *Set Var*: One's target relies on another's action results. Take Rule 912150 and 912160[6] for instance. The target of 912160 is `DOS_COUNTER`, which has been changed by the action of 912150. So 912150 must be put ahead of 912160.
- *Expire Var*: It causes right-value dependency. For example, the left-value of Rule 910110 is assigned by a variable whose value depends on the previous rules.

[6]You can get rule details from https://github.com/SpiderLabs/owasp-modsecurity-crs/tree/v3.2/dev/rules.

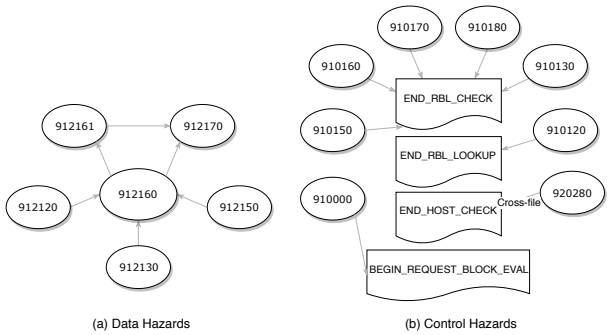


Fig. 5: Rule Dependency Graph

```
# Rule 912150
SecRule TX:EXTENSION "!@within %{tx.
    static_extensions}" "setvar:ip.
    dos_counter=+1"

# Rule 912160
SecRule IP:DOS_COUNTER "@ge %{tx.
    dos_counter_threshold}" "phase:5,\ "

# Rule 910110
"expirevar: ip.reput_block_flag=%{tx.
    reput_block_duration},\ "
```

Figure 5(a) illustrates part of rules with data hazards. The vertex denotes a rule, and the directed edge means the delivery of results. Vertex 912130 points to 912160 means that Rule 912160 depends on results of 912130.

For the rules with data hazards, we still separate them into different groups but maintain their certain order when doing optimizations. It could make the dynamic ruleset more flexible.

Control hazards are caused by the action *skipAfter* which tells ModSecurity to jump over rules until it reaches the **End Marker** in rule-set. There are two types of rules causing control hazards:

- *Static*: The rules target at `PARANOIA_LEVEL`.
- *Dynamic*: The rules are categorized into intra-file skipping and inter-file skipping. (i) Intra-file skipping rules will jump to their **End Marker**. (ii) Inter-file skipping rules jump to their **End Marker** in another file.

Figure 5(b) shows the rules with dynamic control hazards. The vertex denotes a rule, and the directed edge means the direction of skipping. The end of edge is not a rule but an **End Marker**. For example, 910120 points to `END_RBL_LOOKUP` means that 910120 skips to the **End Marker**.

Solutions. For the rules with data hazards, we move them freely but maintain the original order of themselves.

For the intra-file skipping rules with control hazards, we cannot insert anything between them and their **End Marker**. So we maintain them as a whole group. For the inter-file skipping rules, it is very challenging to solve the hazards. Fortunately, CRS only has one inter-file skipping rule called 910000. It initially checks if the client's IP has already been blacklisted and jumps to the end directly. So we still put it at the very beginning.

In conclusion, we split 24 rule groups into 119 rule subgroups in total.

2) The Formulation of Overhead: Given real traffic during a time period, our optimization goal is to find an optimal rule-order with the minimal overhead.

The notation is as follows. There are N rule-subgroups called A_1 to A_n. $\{G_1, G_2, ..., G_n\}$ is a random sequence of A_1 to A_n, representing the actual order of rule-subgroups. In four phases of ModSecurity, we apply our optimization algorithm separately to get the best order[7]. We represent the hit probability of group G_i in phase 1 as $P_1(G_i)$, average triggered time of group G_i as $T_1(G_i)$, average non-triggered time of group G_i as $NT_1(G_i)$, and so forth.

We applied Naive Bayes Model [30] to estimate the mathematical expectation of all the rule-subgroups' overhead. To sum up, the expectation of overhead is formulated as (IV-B2).

$$\begin{aligned} E_{overhead} = \sum_{j=1}^{4} \sum_{i=1}^{n+1} \{ [\prod_{l=1}^{j-1} \prod_{k=1}^{n} (1 - P_l(G_k))] \\ \cdot \prod_{k=1}^{i-1} (1 - P_j(G_k)) \cdot P_j(G_i) \\ \cdot [\sum_{l=1}^{j-1} \sum_{k=1}^{n} NT_l(G_k) + \sum_{k=0}^{i-1} NT_j(G_k) + T_j(G_i)] \} \end{aligned} \tag{4}$$

Especially,

$$P_l\left(G_{n+1}\right) = 1, \;\; T_l\left(G_{n+1}\right) = 0$$

The subscript l above means the l^{th} phase.

3) The Optimization of Rule Location: To make the expectation of overhead as low as possible, we adopted heuristic algorithm to search for the optimal solution of (IV-B2). Among multiple algorithms such as Simulated Annealing, Particle Swarm Optimization and Genetic Algorithm, we decided to choose Genetic Algorithm (GA) because GA is known to perform well in solving combinatorial optimization problems and its gene is naturally suitable for re-ordering.

Genetic algorithm is a meta-heuristic optimization inspired by the process of natural selection including operators such as *mutation* and *crossover* [31]. The evolution is an iterative process, with the population in each iteration called a *generation*. In each generation, the *fitness* of every individual in the population is evaluated. In this case, the fitness value of GA was set to the countdown of overhead in (5) so that the largest fitness value could lead to the minimal overhead.

$$Fitness = 1/E_{overhead} \tag{5}$$

We regarded a possible order of rule-subgroups $\{G_1, G_2, ..., G_n\}$ as a chromosome. When two chromosomes crossover, they exchange the corresponding gene fragments.

[7]Rules in logging phase have to be processed in the end. So we don't re-order them. FYI, the introduction of rule phases is in Section III-C.

However, it may cause conflicts because the particularity of an order. All the elements in an order must be unique, but duplication may arise after two orders exchange their elements. Thus we developed a method to check and eliminate the conflicts after crossover.

Also, some optimizations were applied to GA in order to improve its efficiency. We always maintained the best one in four chromosomes when crossover instead of randomly picking. Besides, we added a step of reversing to reverse the whole gene fragment, in order to avoid falling into local optimal solutions.

C. Other Optimizations

For other incremental optimizations, we present an off-line processing method Rule Pre-pruning Cache to reduce runtime overhead.

By counting the rules in CRS, we found there are 759 rules in total, and among them 169 rules are about paranoia level control which is up to 22%. However, when we inspected the rules, we found that the paranoia level control is actually achieved by a kind of rules which have nothing to do with the HTTP traffic.

The paranoia level rules are dispersing all over the whole rule-set to split the rules into a few regions which corresponding to the paranoia levels. ModSecurity processes the rules in sequence until it reaches the rules related to paranoia level. And then, ModSecurity chooses to continue processing rules below the configured paranoia level or jump over the rules that beyond the level.

Taking advantage of this characteristic, we removed the rules whose level is higher than target paranoia level along with the paranoia level control rules manually.

Similar with `Paranoia Level` control rules, there are other types of rules also non-related to HTTP traffic, which are enumerated as follows:

- **Action Classification Cache**: Action classification is very heavy in ModSecurity because ModSecurity will put all the actions in one table and classifies them at runtime per request. To save the runtime overhead, we cached the action type after classification at first time. When ModSecurity handles following requests, it does not need to classify the actions in rules at runtime.
- **Set Var Cache**: "Set Var" action assigns value to variable at runtime. However, we observed that part of the "Set Var" actions assign the static value to variables in the context. ModSecurity does not know the variable and value of "Set Var" until it parses the string of "Set Var" in rules at runtime. Our solution is to cache the variable and value of each "Set Var". If the value stays unchanged, ModSecurity directly assigns the value to variable without parsing the string at runtime. Otherwise, ModSecurity updates the cache after parsing the "Set Var" at runtime.
- **Skip After Cache**: "Skip After" is an action that tells ModSecurity to jump over rules until it reaches an **End Marker**. The paranoia level control rules use "Skip After" actions to jump over rules that do not beyond

the current paranoia level. Before ModSecurity reaches the **End Marker**, it has to check every rule at runtime which causes overhead. Our Rule Pre-pruning Cache keeps an index of the **End Marker**. When ModSecurity executes the "Skip After" action, it could jump to the index of **End Marker** immediately without checking the rules.

- **Logging**: ModSecurity logs the intermediate data when processes the rules and prints out data in log file if necessary. However, the logging in ModSecurity is a two step action. ModSecurity composes the log message at first. Then it checks the log level at runtime and decides whether to print out the message into file. The log message is always composed even it's not required. We cached the log decision for ModSecurity at the first time, and ModSecurity does not compose log message if the logging function is off.

This design helps ModSecurity to acquire an additional gain in Section VI-D.

V. Implementation

To evaluate our design, We implemented a prototype of **RuleCache** containing offline engine and online engine. Then we deployed the modules in ModSecurity 2.9.1.

In the offline engine, Rule Pre-pruning Cache detects the rules with static results by compiler. In the online engine, we firstly implemented a tool WAF-Bench which could help us to generate and send requests. We then utilized a modified version of ModSecurity called WAF Simulator to generate the raw data needed in Analyzer. There are two modules in Analyzer, namely Rule Result Cache and Rule Ordering Cache. We adopted an optimized Genetic Algorithm in order to look for the optimal order in Rule Ordering Cache. Finally, we built a cache result table in Rule Result Cache to cache high-frequency target values. The whole prototype was implemented in C.

VI. Evaluation

In this section, we evaluate the performance gains achieved by three modules in our experiments. The results are from end-to-end measurements in Nginx with **RuleCache**.

A. Workloads

Testbed setup. Our testbed consists of 6 servers located under a ToR switch as shown in Figure 6. We use 1 server as a client, 1 server as a proxy and 4 servers as backend servers which have 8 virtual machines for each (i.e., 32 VMs in total). Each server is a Dell PowerEdge R730 with two 16-core Intel Xeon E5-2698 2.3 GHz CPUs and 256GB RAM. Every server has one Mellanox ConnectX-3 Pro 40G NIC. The switch is Arista DCS-7060CX-32S-F with Trident chip platform. The web servers run Nginx 1.11.5 in proxy mode with ModSecurity 2.9.1. The rule-set in our testing is CRS 3.0 and `Paranoia Level` is set to 2 as default.

Workload datasets. Microsoft Azure Application Gateway, a WAF-enabled L7-load balancer for customers, serves live user traffic for a diverse set of web applications. We monitor the real traffic to make up our datasets, contain 861,988 requests from Microsoft Azure Application Gateway for six days from July 16, 2019 to July 21, 2019.

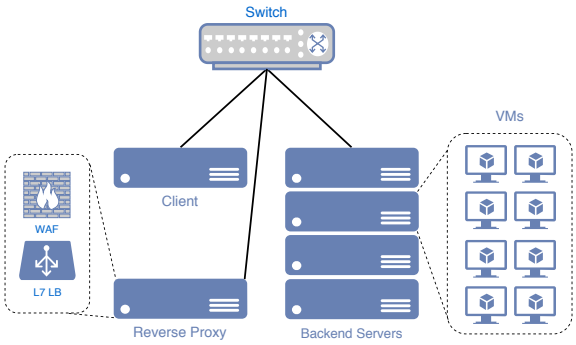


Fig. 6: Testbed Topology

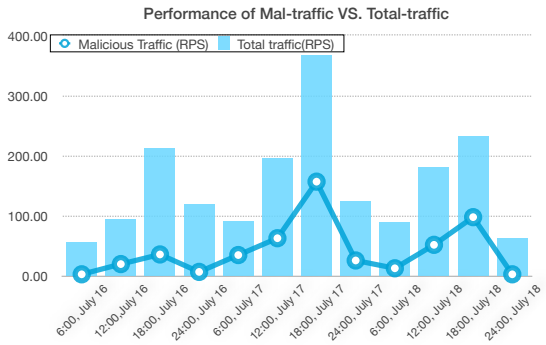


Fig. 7: Malicious Traffic VS. Total Traffic

Additionally, to simulate some cases that we cannot collect enough specific traffic in real world, we use WAF-Bench to generate random requests according to given patterns, e.g., simple GET requests and requests of certain attack types.

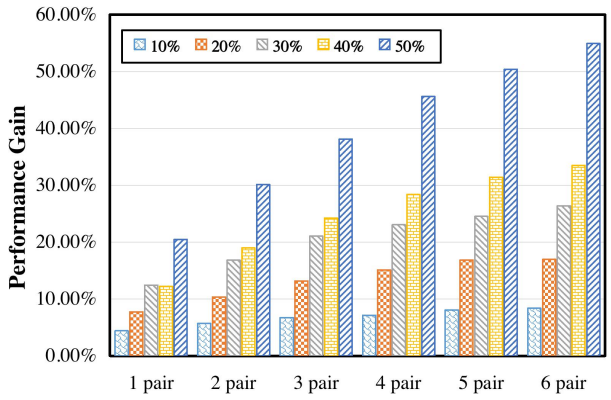
The analysis of real traffic is shown in Figure 7, justifying that the proportion of malicious traffic varies in the real world, ranging from 4.79% to 42.24%.

B. Rule Result Cache

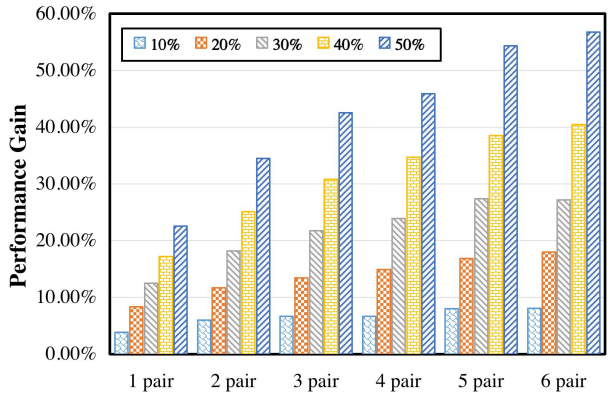
It is necessary for us to evaluate the performance of multiple frequencies and sizes of targets, because the real traffic patterns are apparently different (§III-B).

By testing the *weight* of multiple targets, we found that the target values of `ARGS` and `COOKIE` always have priority in our *weight*-based cache, while the the target values of `USER-AGENT` weigh less than them though it also has high frequency. It is because that shown in Table V, 104 and 84 rules of 759 rules in total inspect `ARGS` and `COOKIE`, respectively, but only 17 rules inspect `USER-AGENT`, causing the imbalance processing time through the whole rule set. Therefore, the list of targets in our evaluation only include `ARGS`, `COOKIE`, and we show their performance gain with the increasing size of cookies and arguments in Figure 8.

For the cached `COOKIE` in Figure 8(a), as the number of cookie pairs varies from 1 to 6, the performance gain



(a) Performance Gain of Cookie Caching



(b) Performance Gain of ARGS Caching

Fig. 8: Performance Gain with Different Frequency of Target Values in Rule Result Cache

TABLE V: Top 10 Rule Targets in CRS

Rule Targets	Count	Rule Targets	Count
TX:PARANOIA_LEVEL	169	REQUEST_COOKIES	84
ARGS	104	REQUEST_FILENAME	53
ARGS_NAMES	102	RESPONSE_BODY	28
XML:/*	94	USER-AGENT	17
REQUEST_COOKIES_NAMES	84	TX:sql_error_match	16

increases from 20.46% to 54.94% when the 50% of total traffic have the cached cookie field. The performance gain of 3 cookie pairs grows from 7.13% to 45.61% when the percentage of traffic with cached cookie field increase from 10% to 50%. In Figure 8(b), we also find that `ARGS` has the similar performance gain as `COOKIE`. Therefore, Rule Result Cache improves the performance of WAF by around 7% to 57% with different workloads and cache percentage.

C. Rule Ordering Cache

There are two modes for checking rules in CRS, namely self-contained mode and anomaly scoring detection mode. To clarify, we apply our Rule Ordering Cache under both modes. If self-contained mode is enabled, ModSecurity will block the request immediately once it triggers the rule. For anomaly scoring mode, ModSecurity checks the anomaly score against a threshold at last. But we modify it to be checked every time when it triggers a rule.

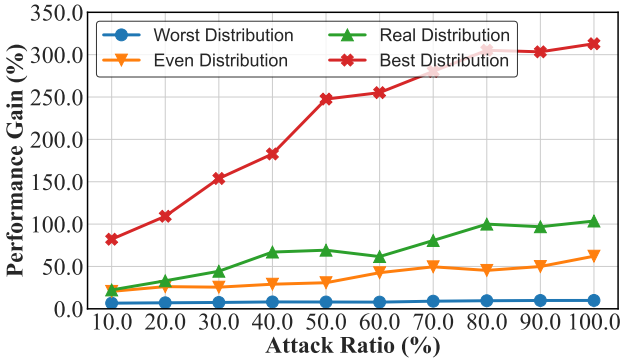


Fig. 9: Performance Gain in Rule Ordering Cache with Different Proportions of Malicious Traffic

To evaluate the optimization strategy, we generated several workload datasets with various attack ratios and attack distribution to explore the benefits of our design.

Attack ratio means the proportion of malicious traffic. We set it as 10%, 20%, ..., 100% respectively in each set.

Attack distribution represents the probability distribution for each rule-group being triggered. We did experiments with traffic in the following distributions:

- *Best Distribution.* In the original, all the malicious traffic hit the last rule-group. After our optimization, the last group is moved forward to accelerate the efficiency of blocking.
- *Worst Distribution.* Attacks are all triggered at the very beginning originally. Adjusting their order cannot bring much gain.
- *Even Distribution.* Attacks trigger all the rule-groups in an average frequency.
- *Real Distribution.* The probability distribution is produced by fitting that of real traffic. Actually, some rule-groups are hit more frequently than others in the real world.

For the diverse sets varying in attack ratios and distributions, the end-to-end performance gain of optimal order is shown in Figure 9.

It comes to a conclusion that as the proportion of malicious traffic increases from 10% to 100%, the performance gain grows from 22.27% to 103.61% when the traffic fits the real distribution. However, it is almost impossible to reach 100% attacks in the real world. According to Figure 7, the ratio of malicious traffic is ranging from 4.79% to 42.24%, then we assume that the ratio is up to 50%, and therefore Rule Ordering Cache obtains the gain of around 22.27% to 69.20% with real distribution of traffic. Considering best and worst distribution, it can improve the performance up by 1.08x to 2.48x.

TABLE VI: Performance Gain in Rule Pre-pruning Cache

	64B				10240B			
	N-KA		KA		N-KA		KA	
IIS-Ori	774	-	808	-	724	-	748	-
IIS-Cache	1802	133%	1939	140%	1576	118%	1728	131%
ModSec-Ori	1525	-	1697	-	1484	-	1657	-
ModSec-Cache	2425	59%	2894	71%	2343	58%	2795	69%

D. Rule Pre-pruning Cache

Besides our two main optimizations, we also have enabled a prototype cache engine on ModSecurity with the working cache set we mentioned in Section IV-C. To recap, this engine is an offline engine, thus we do not follow the setting of traffic workloads when evaluating Rule Result Cache and Rule Ordering Cache. The experimental results are shown in Table VI. Our cache engine could achieve up to 140.0% performance improvement compared to original ModSecurity.

E. Overall System Evaluation

1) Performance Gain: We build a **RuleCache** system including all of them and measure the performance using real-traffic. After running 1,000,000 real-traffic requests, our system obtains the performance of 2813.26 RPS. Comparing to the original ModSecurity, overall system can achieve about 5.5x performance gain.

2) Security Analysis: We send same real-traffic to original ModSecurity and **RuleCache** and compare their generated logs. They are completely consistent so that the cache of rule results and re-arranging of rule orders do not reduce correctness. **RuleCache** has been turned out to fulfill performance optimization without sacrificing security, which satisfies people's need today in pursuing efficiency.

VII. Related Work

Existing performance analysis works on WAFs always focus on the detection accuracy against web attacks, remaining the poor efficiency of rule-processing an open problem.

Sobola et al. [32] provide insight on detection capability of ModSecurity with CRS v.3.2 at default level, and how well it can protect web server against Denial of Service (DoS) attacks. Additionally, it measures the performance on web server in terms of Throughput, Transaction rates, Concurrency, whereas lacks the analysis of WAF's poor throughput. Thang et al. [33] apply effective algorithms to train WAFs automatically for increasing its efficiency in detecting attacks. Concretely, this work introduces parameterization of the task for increasing the accuracy of query classification by the random forest method, thereby creating the basis for detecting attacks at the application level. Mukhtar et al. [34] investigate the efforts to detect and prevent the SQL injection attacks, and they also assess the efficiency of Modsecurity in preventing SQL injection attacks. More recently, Arnaldy et al. [35] propose an optimal web server that applies a package namely Reverse Proxy which is used to optimize a web server and WAF which is used to maintain the security of a web server.

VIII. Conclusion

In this paper, we explored the current design and implementations of modern WAF solutions. Based on the exploration results, we analyze several root causes of performance slowdown and get some insights such as target-based cache and rule set reordering. These insights enlighten our design of **RuleCache** system, including Rule Result Cache, Rule Ordering Cache and Rule Pre-pruning Cache. Our evaluation demonstrates that the prototype can achieve 5.5x performance gain. Therefore, an efficient rule cache engine should be an effective way to boost the current performance of WAF.

Acknowledgments

We thank anonymous reviewers for their constructive feedback and suggestions. This work is partly supported by National Natural Science Foundation of China (Grant No. 61672062, 61232005).

References

[1] C. Systems, "Netscaler mpx waf," 2018. [Online]. Available: https://www.citrix.com/products/netscaler-adc/platforms.html
[2] B. Networks, "Barracuda web application firewall," 2018. [Online]. Available: https://www.barracuda.com/products/webapplicationfirewall
[3] Imperva, "Imperva securesphere appliances," 2018. [Online]. Available: https://www.imperva.com/resources/datasheets/DS_SecureSphere_Appliances.pdf
[4] O. B. Ryan Liles, "Waf product analysis: F5 big-ip asm model 10200," 2017.
[5] M. Azure, "Microsoft azure application gateway," 2018. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/application-gateway/
[6] A. W. Services, "Aws waf," 2019. [Online]. Available: https://aws.amazon.com/waf/
[7] Cloudflare, "Cloud web application firewall," 2019. [Online]. Available: https://www.cloudflare.com/waf/
[8] OWASP. (2021) Owasp top 10 - 2021 report.
[9] SpiderLabs, "OWASP ModSecurity Core Rule Set," 2019. [Online]. Available: https://modsecurity.org/crs/
[10] Z. L. Li, M. C.-J. Liang, W. Bai, Q. Zheng, Y. Xiong, and G. Sun, "Accelerating rule-matching systems with learned rankers," in *ATC (USENIX Annual Technical Conference)*. USENIX, July 2019.
[11] S. Trustwave, "Modsecurity: Open source web application firewall," 2021. [Online]. Available: https://www.modsecurity.org
[12] SpiderLabs, "Github: Spiderlabs," 2019. [Online]. Available: https://github.com/SpiderLabs/owasp-modsecurity-crs/tree/v3.1/dev
[13] M. Becher, *Web application firewalls*. VDM Verlag, 2007.
[14] A. Logic, "2018 critical watch report," Alert Logic, Inc., Tech. Rep., 2018.
[15] Symantec, "Internet security threat report," Technical report, Symantec, Tech. Rep., 2019.
[16] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *2010 IEEE symposium on security and privacy*. IEEE, 2010, pp. 332–345.
[17] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 40–52.
[18] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S P'06)*. Washington, DC, USA: IEEE, May 2006, pp. 6 pp.–263.
[19] R. Balock and T. Jaffery, "Modern web application firewalls fingerprinting and bypassing xss filters," Technical report, Rhainfosec, Tech. Rep., 2013.
[20] T. Krueger, C. Gehl, K. Rieck, and P. Laskov, "Tokdoc: A self-healing web application firewall," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010, pp. 1846–1853.

[21] C. N. Jeremy D'Hoinne, Adam Hils. (2018, September) Magic quadrant for web application firewalls. [Online]. Available: https://www.gartner.com/doc/reprints?id=1-5ELTARA&ct=180904&st=sb

[22] C. Kumar, "Open source web application firewall for better security," 2016. [Online]. Available: https://geekflare.com/open-source-web-application-firewall/

[23] N. System, "Naxsi," 2019. [Online]. Available: https://github.com/nbs-system/naxsi

[24] R. Paprocki, "Lua-resty-waf," 2018. [Online]. Available: https://github.com/p0pr0ck5/lua-resty-waf

[25] Microsoft, "Waf-bench," 2019. [Online]. Available: https://github.com/Microsoft/WAFBench

[26] M. Azure, "Web application firewall (waf)," 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/application-gateway/application-gateway-web-application-firewall-overview

[27] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso, "Analysis of caching and replication strategies for web applications," *IEEE Internet Computing*, vol. 11, no. 1, pp. 60–66, 2007.

[28] J. Erman, A. Gerber, M. T. Hajiaghayi, D. Pei, and O. Spatscheck, "Network-aware forward caching," in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 291–300.

[29] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Computing Surveys (CSUR)*, vol. 35, no. 4, pp. 374–398, 2003.

[30] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers," *Machine Learning*, vol. 29, no. 2, pp. 131–163, 1997.

[31] M. Mitchell, *An introduction to genetic algorithms*. Cambridge, MA, USA: MIT press, 1998.

[32] T. D. Sobola, P. Zavarsky, and S. Butakov, "Experimental study of modsecurity web application firewalls," in *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing,(HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*. IEEE, 2020, pp. 209–213.

[33] N. M. Thang, "Improving efficiency of web application firewall to detect code injection attacks with random forest method and analysis attributes http request," *Programming and Computer Software*, vol. 46, no. 5, pp. 351–361, 2020.

[34] B. I. Mukhtar and M. A. Azer, "Evaluating the modsecurity web application firewall against sql injection attacks," in *2020 15th International Conference on Computer Engineering and Systems (ICCES)*. IEEE, 2020, pp. 1–6.

[35] D. Arnaldy and T. S. Hati, "Performance analysis of reverse proxy and web application firewall with telegram bot as attack notification on web server," in *2020 3rd International Conference on Computer and Informatics Engineering (IC2IE)*. IEEE, 2020, pp. 455–459.