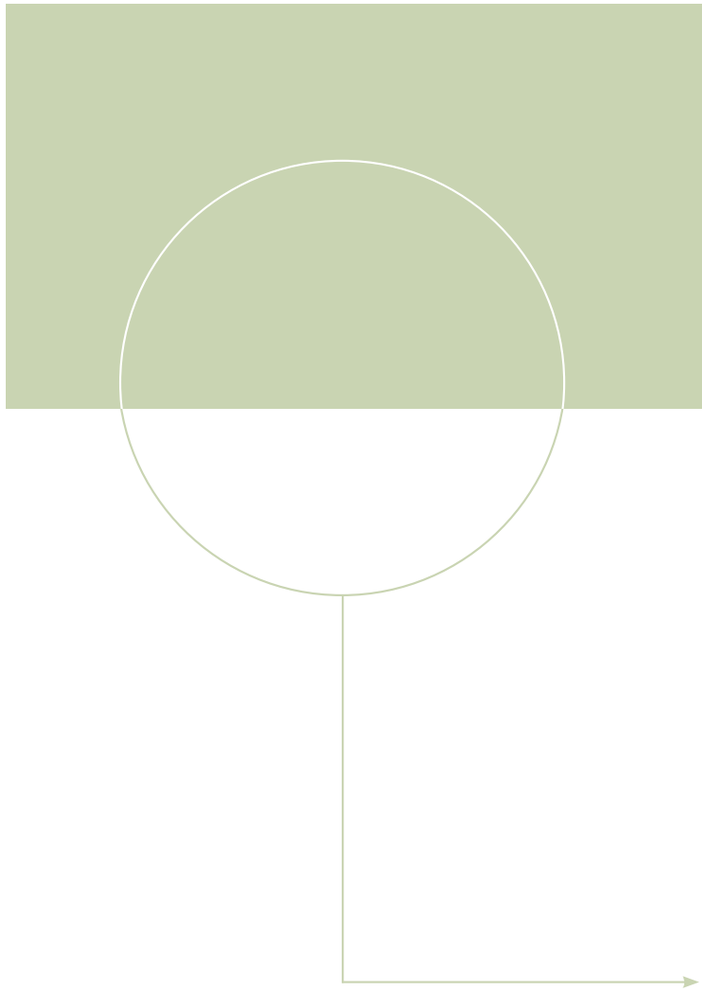

Master's thesis

2021

Kyle Richard Orlando

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Kyle Richard Orlando

Automating Virtual Patching via Application Security Testing Tools

July 2021

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Automating Virtual Patching via Application Security Testing Tools

Kyle Richard Orlando

Informatics
Submission date: July 2021
Supervisor: Jingyue Li
Co-supervisor:

Norwegian University of Science and Technology
Department of Computer Science

Automating Virtual Patching via Application Security Testing Tools

Kyle Richard Orlando

July 9, 2021

Abstract

Web Application Firewalls (WAFs) have become increasingly popular as a result of organizations' need to protect their web applications and services. One useful approach to WAF configuration is called virtual patching, in which one or more WAF rules act to quickly mitigate a security vulnerability that has not yet been addressed in the web application's source code. However, virtual patches tend to require a lot of manual configuration, which can become a serious security issue itself when done improperly. In this thesis, automating virtual patch creation via dynamic and static application security testing methods is explored. A utility called VPgen is developed that facilitates taking the output from state-of-the-art dynamic and static analysis tools and transforming it into rules and directives that can be interpreted by a WAF. The effectiveness of this approach in virtually patching two different vulnerable web applications is assessed and compared against ModSecurity deployed with its Core Rule Set. The results show that in addition to reducing configuration time, automating virtual patching via application security testing can reduce the number of false positives.

Sammendrag

Webapplikasjonsbrannmurer (WAF) har blitt stadig mer populært som et resultat av organisasjoners behov for å beskytte webapplikasjonene og tjenestene deres. En nyttig tilnærming til WAF-konfigurasjon kalles virtuell patching, hvor en eller flere WAF-regler raskt reduserer virkningene av et sikkerhetsproblem som ikke har blitt håndtert i applikasjonens kildekode. Men virtuell patching har en tendens til å kreve mye manuell konfigurasjon som kan bli et alvorlig sikkerhetsproblem dersom det gjøres feil. I denne oppgaven utforskes automatisk opprettelse av virtuell patcher ved hjelp av dynamiske og statiske metoder for sikkerhetstesting av applikasjoner. En del av denne masteroppgaven har vært å utvikle et verktøy kalt VPgen som legger til rette for å omforme resultatet av state-of-the-art dynamiske og statiske analyseverktøy til regler og direktiver som kan tolkes av en WAF. Effektiviteten av denne tilnærmingen til virtuell patching av to ulike sårbare webapplikasjoner blir vurdert og sammenlignet med ModSecurity, med dens Core Rule Set. Resultatene viser at i tillegg til å redusere konfigurasjonstiden kan automatisk virtuell patching, ved hjelp av sikkerhetstesting av applikasjoner, redusere antall falske positiver.

Acknowledgement

I would first like to thank my advisor, Associate Professor Jingyue Li of the Department of Computer Science at the Norwegian University of Science of Technology NTNU. Professor Li was very flexible, supportive of, and patient with me throughout the entire thesis process, and for that I am very grateful. I should also mention that Professor Li was the one to originally propose web application firewalls as a potential research topic. This was a welcome relief since I initially could not make up my mind on which software security-related matter to pursue.

I would also like to thank Even Kronen Johansen, who has also delivered a thesis concerning improvements to web application firewalls. We were able to to bounce many ideas off of one another, and even though we ultimately decided to produce separate theses, the discussions we had while working together ended up being crucial. I only regret that due to the pandemic, we were never able to actually meet in person.

Additionally, I need to thank Henrik, Michael, Torstein, and my girlfriend Aly for the useful advice and proofreading they provided near the end of the writing process.

Finally, I would like to thank my parents for supporting me and enduring my endless ramblings and pacing as I worked hard to complete my research.

Sincerely,
Kyle Richard Orlando

Contents

Figures

Tables

Code Listings

Acronyms

AST Abstract Syntax Tree. 14, 15

CFG Control Flow Graph. 14, 15

CRS Core Rule Set. xiii, 12, 20, 22, 27, 28, 30, 31, 34, 39, 40, 45

DAST Dynamic Application Security Testing. 13, 15, 20, 25, 26, 28, 39–41, 45, 47

FI File Inclusion. 7, 8, 30

IAST Interactice Application Security Testing. 13, 14

LFI Local File Inclusion. 7, 32

LFI/RFI Local File Inclusion/Remote File Inclusion. 18

MLR Multivocal Literature Review. 17

NAT Network Address Translation. 35

OWASP Open Web Application Security Project. 3, 12, 14, 15, 27–29

PDG Program Dependency Graph. 14, 15

RASP Runtime Application Self-Protection. 13, 14

RCE Remote Command Execution. 30

RFI Remote File Inclusion. 7, 32

SaaS Software as a service. 8

SAST Static Application Security Testing. 13–15, 25, 26, 40, 43, 45, 47

SMT Satisfiability Modulo Theories. 15, 44

SQLi SQL Injection. xv, 5, 8, 21, 22, 24, 28, 30, 34, 37, 41, 47

VM Virtual Machine. xi, 34–36

WAF Web Application Firewall. 1–3, 8–10, 12, 13, 17, 18, 21–27, 35, 43, 44, 48

XSS Cross Site Scripting. 7, 8, 22, 24, 30, 39

ZAP Zed Attack Proxy. 29

Chapter 1

Introduction

1.1 Context

The COVID-19 pandemic and its world-wide disruptions has led to a major upheaval in terms of how people conduct business. According to surveys conducted by Pew Research, the percentage of Americans working from home increased from 20% before the outbreak to 71% by October 2020 [1]. This has caused a rapid increase in demand for enterprise cloud services. In Q3 of 2020, enterprise spending on cloud infrastructure services had reached $65 billion, which was a 28% increase from the Q3 of 2019 [2]. Between December 2019 and June 2020, organizations worldwide increased their cloud workload by 20%. Unfortunately, this also appears to correspond to an increase in cloud security incidents. Retail, manufacturing, and government sectors have seen an increase in the number of security incidents of 402%, 230%, 205% respectively [3].

An organization can mitigate these types of incidents by using some type of an intrusion detection and prevention system, such as a Web Application Firewall (WAF). WAFs can be particularly useful when the vulnerability is on the cloud provider's side, which renders the customer unable to identify and fix the vulnerability in the source code. However, manually configuring a WAF can lead to errors and security incidents itself. In 2019, a CloudFlare outage was caused by the addition of a new rule that added "a regular expression that backtracked enormously and exhausted CPU used for HTTP/HTTPS serving" [4]. Palo Alto Networks discovered "that 65% of publicly disclosed security incidents in the cloud were the result of customer misconfigurations" [5]. This signifies the need for robust automatic configuration of security controls.

Much of the existing academic research into WAFs focuses on improving attacks for/defenses against various types of injection vulnerabilities. Less focus has been on the automatic generation, configuration, and/or repair of WAFs. In addition, for many of the WAF approaches developed, the underlying web application is treated as a sort of blackbox, and as such the WAF cannot be specifically tailored. Although this has the benefit that the WAF is decoupled from the underlying application, this one-size-fits-all approach could lead to increased false positives

(i.e., blocked legitimate requests) [6] and reduced performance [7].

1.2 Research Contributions

This thesis will investigate how a WAF can be automatically configured and tailored to a specific web application, which can also be called automatic or automated virtual patching. A tool called **VPgen** will be designed, created, and evaluated. It takes as input a vulnerability report generated by a security analysis tool for a vulnerable web application, and it outputs a list of rules. These rules correspond to and protect the vulnerable resources and parameters of the web application. This approach will be evaluated by setting up the WAF with the generated rules and attacking it with both malicious and benign requests. The same will be repeated for the WAF configured with a standard rule set, and the results will be compared.

More specifically, the following contributions are made:

1. A novel approach to virtual patching that leverages an existing ruleset and popular application security testing techniques and tools
2. A tool that can take a vulnerability report from one of several different scanners, process it, and output virtual patches, i.e., specially tailored WAF rules
3. Improvements/fixes to an existing state-of-the-art static analysis tool in order to facilitate virtual patch generation

1.3 Thesis Outline

Chapter 2 introduces the fundamentals, concepts and related topics that are pertinent to this thesis. This includes a discussion of the most common security vulnerabilities found in web applications, as well as the tools and techniques used for attacking and defending web applications.

Chapter 3 mentions related works that have studied or have attempted to automate or improve WAFs.

Chapter 4 presents an overview of the research methodology, design, implementation, and evaluation of automated virtual patching via VPgen.

Chapter 5 presents the results from attacking a WAF with rules generated by VPgen. It will also present the results of attacking the WAF with a standard ruleset as a basis for comparison.

Chapter 6 discusses and interprets the results from the previous chapter. Various other aspects of the thesis are also discussed, such as the extensive efforts required to revitalize the state-of-the-art static analyzer used in this thesis.

Chapter 7 summarizes what this thesis has achieved, and presents ideas for future work.

Chapter 2

Background

This chapter will begin by summarizing the most common types web application security vulnerabilities in Section 2.1. Next, Section 2.2 will define and describe Web Application Firewall (WAF)s. Finally, Application Security Testing and its approaches will be presented in 2.3.

2.1 Critical Web Security Vulnerabilities

2.1.1 Rankings and Categorizations

OWASP Top 10

The Open Web Application Security Project (OWASP) maintains a list of "The Ten Most Critical Web Application Security Risks" called OWASP Top 10 [8]. The OWASP Top 10 2017 release, which is the most recent version, relied on what was possibly the most amount of data ever collected for developing an application security standard [8]. The full list, taken in verbatim from [8], is presented below.

A1:2017 - Injection Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

A2:2017 - Broken Authentication Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.

A3:2017 - Sensitive Data Exposure Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without

extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

A4:2017 - XML External Entities (XXE) Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.

A5:2017 - Broken Access Control Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

A6:2017 - Security Misconfiguration Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.

A7:2017 - Cross-Site Scripting (XSS) XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

A8:2017 - Insecure Deserialization Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.

A9:2017 - Using Components with Known Vulnerabilities Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.

A10:2017 - Insufficient Logging & Monitoring Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

CWE Top 25

The Common Weakness Enumeration (CWE™) is a large list of common software and hardware weaknesses [9] maintained by The MITRE Corporation, an American not-for-profit organization. As of CWE List Version 4.4, it contains 918 weaknesses [10]. CWE is endorsed by the CWE Community, which consists of representatives from major operating systems, security tool vendors, academia, and government institutions [9]. CWE produces a subset of this list called the Top 25 Most Dangerous Software Weaknesses (CWE Top 25). To create this list, the CWE team leverages specific vulnerability (Common Vulnerability and Exposures [CVE®]) and scoring data (Common Vulnerability Scoring System [CVSS]) from the National Institute of Standards and Technology (NIST) National Vulnerability Database (NVD) from the previous two calendar years [11]. A score is created for each weakness based on prevalence and weakness, and the top 25 weaknesses are published. Not all of these weaknesses are specific or relevant to web security, but many are. The 2020 CWE Top 25 list is presented in Table 2.1.

Comparison of Categorizations

The OWASP Top 10 list comprises general categories of the most critical risks to web applications [8]. The CWE Top 25 list covers a broader range of issues, though the weaknesses themselves are more specific and detailed [12]. CWE conveniently provides a "view" that shows the mappings from each OWASP Top 10 (2017) issue to one or more CWE entries [13], although not every CWE entry mapped to appears in the 2020 CWE Top 25.

2.1.2 Specific Vulnerabilities

This subsection contains brief descriptions and demonstrations of the specific web security vulnerabilities that are most relevant to this thesis. Most are examples of injection vulnerabilities.

SQL Injection

A SQL Injection (SQLi) attack, also known as a SQL Insertion attack, consists of specially crafted user input data that allows the attacker to interfere with one or more SQL queries that the web application uses to interact with the database. This can allow the attacker to read and modify data from the database, subvert application logic including login, or even inject OS commands [14, 15]. To protect against SQLi, software developers need to ensure that user-supplied input can only be interpreted as data. **This is generally the case for any injection vulnerability, including the others mentioned in this subsection**. For SQLi, this can be accomplished via prepared statements with parameterized queries, stored procedures, input validation, and/or input sanitization [16].

Table 2.1: 2020 CWE Top 25 [11]

Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53
[18]	CWE-522	Insufficiently Protected Credentials	5.49
[19]	CWE-611	Improper Restriction of XML External Entity Reference	5.33
[20]	CWE-798	Use of Hard-coded Credentials	5.19
[21]	CWE-502	Deserialization of Untrusted Data	4.93
[22]	CWE-269	Improper Privilege Management	4.87
[23]	CWE-400	Uncontrolled Resource Consumption	4.14
[24]	CWE-306	Missing Authentication for Critical Function	3.85
[25]	CWE-862	Missing Authorization	3.77

Cross-Site Scripting

A Cross Site Scripting (XSS) attack is a type of injection attack in which malicious client-side script is injected into a web page. This script will be executed by the user's browser and thus can retrieve sensitive information such as cookies. It can even rewrite the the contents of the web page [17]. There are two main categories of Cross Site Scripting (XSS) attacks: Stored and Reflected [17].

- Stored XSS: The malicious script is stored on the web server, such as via a forum post or comment. This is also known as Persistent XSS.
- Reflected XSS: The malicious script is not stored, but is reflected back to the user's browser from the web server, usually via an error message or search result.

OWASP provides numerous XSS prevention guidelines [18], most of which concern encoding untrusted data for JavaScript, HTML, CSS, and URL data values, preferably with a dedicated security encoding library.

Path Traversal

A Path Traversal attack, also called Directory Traversal, aims to access files and directories outside of the website's root directory. This can be accomplished by modifying arguments and variables that reference resources with variations of "dot-dot-slash" (`../`) [19], which in many systems allows the user to traverse to the parent directory at the command line.

To protect against Path Traversal, user input should either be validated before being passed to any filesystem APIs, or it should be prevented from ever reaching any filesystem APIs [19].

File Inclusion

A File Inclusion (FI) attack attempts to force the web application into returning and executing a file of the attacker's choice. This is most often accomplished via PHP `include` statements, but this is also possible in technologies like JSP and ASP [20]. There are two types:

- Local File Inclusion (LFI): This attack is similar to a Path Traversal attack, except the goal is to execute the local file returned.
- Remote File Inclusion (RFI): This attack forces the web application to download and execute a remote file obtained via protocols like HTTP(S) and FTP.

The protections against FI are the same as for Path Traversal [19, 20].

Code Injection

A Code Injection attack attempts to inject server-side code that can be interpreted or executed by the web application [21]. More specific attacks include Java, PHP,

and ASP code injection. OWASP differentiates Code Injection from Command Injection by asserting that in Code Injection, "an attacker is only limited by the functionality of the injected language itself" [21].

As with other injection attacks, Code Injection can be prevented by properly validating and sanitizing user input.

Command Injection

A Command Injection attack attempts to exploit a vulnerability in web application that allows the attacker to execute OS commands on the host OS. These commands are usually executed in the context of a shell, which allows the commands to be executed with elevated permissions [22]. According to OWASP, Command Injection differs from Code Injection in that rather than injecting code, the attacker extends the default functionality of the web application, which executes system commands [22].

As with other injection attacks, Command Injection can be prevented by properly validating and sanitizing user input. Alternatively, there are usually safe, language-specific APIs that a developer can use to obviate such an attack.

2.2 Web Application Firewalls

A Web Application Firewall (WAF) is an application-level firewall that can be deployed to protect one or more web applications [23]. It generally runs in front of a web server through a reverse proxy [23], and it monitors, filters, and blocks packets of data as they travel to/from a web application [24]. It is often used to protect against various injection attacks (e.g., SQLi and XSS) [24], FI, and security misconfiguration. WAFs can be categorized in terms of implementation, security model, threat detection system, and license.

2.2.1 Implementation Types

According to Cloudflare, a WAF can be be implemented in several ways [25]:

Network-based WAFs These are generally hardware-based WAFs that have the lowest latency, but the highest cost. Phyiscal equipment will need to be stored and maintained.

Host-based WAFs These are software-based WAFs that are integrated into or embedded within the web application. They are cheaper than network-based WAFs, and provide for more configuration options, but this comes at the cost of higher latency, consumption of local resources, implementation complexity, and engineering time.

Cloud-based WAFs These are Software as a service (SaaS) and are thus require the least amount of effort to setup and maintain. The customer may pay an initial upfront cost, and then subsequently on a monthly or annual basis

for the service. The drawback is that the customer has little direct control over the WAF itself since it is operated by a third party.

2.2.2 Security Models

There are three main security models that a WAF can operate under [26] with the third security model being a combination of the first two.

Negative (blacklist) security model This model focuses on blocking known exploits or attack signatures. This model can be easier to create and update since it does not necessarily require knowledge of the underlying web application, but attackers can get around it by reworking exploits to be sufficiently different [27].

Positive (whitelist) security model This model only permits traffic deemed as safe according to a specific criteria. All other traffic is excluded. This can be viewed as implementing the input validation that the underlying web application(s) should have implemented [27]. This model can be more difficult to implement since it may need to be tuned to the underlying application, and thus will need to be updated whenever a new application feature is added [26].

Hybrid model This model utilizes both of the previous models.

2.2.3 Popular WAFs

Commercial, open-source, and academic WAFs have been developed. Gartner, an IT Industry Analyst, produces a "Magic Quadrant" for WAFs [28], which is a semi-annual market research report that indicates the current participants and trends within WAF technology. See Gartner's Magic Quadrant for Web Application Firewalls in Figure 2.1 for their list of the most important and relevant commercial WAF vendors on the market. Note that this is just a subset of vendors.

Open-source WAF options are more limited. ModSecurity is the most popular solution, having a companion rule-set [29], an active mailing list [30], two actively maintained releases [31], and even a published handbook [27]. It also appears often in academic literature [6, 32–35]. ModSecurity will be further discussed in Section 2.2.4. Other open-source WAFs that have been referenced in academic literature include:

- AQTRONiX WebKnight [35–37]
- Guardian [35, 38]
- Shadow Daemon [37, 39]
- NAXSI [40, 41]
- lua-resty-waf [41, 42]

Figure 2.1: Magic Quadrant for Web Application Firewalls [28].

2.2.4 ModSecurity

ModSecurity is an open-source, rule-based WAF that was first released as an Apache module in November 2002 [27]. Although only originally available for the Apache web server, ModSecurity was eventually ported to IIS and nginx starting in version 2.7.0 [27]. In version 3.0.0, the ModSecurity platform was completely rewritten in order to fully decouple it from the Apache web server. This complete rewrite was named Libmodsecurity, and it serves as an interface to "ModSecurity Connectors" [31]. A ModSecurity Connector functions as a connection point between Libmodsecurity and a web server. In order for a given web server to be able to communicate with LibModSecurity, a Connector must be implemented. Connectors already exist for common web servers such as nginx, IIS, and Apache.

Although ModSecurity 3 was released in December 2017, it is still not considered to be as stable as ModSecurity 2, which is still maintained and updated (version 2.9.4 was released in June 2021). This is partially because ModSecurity 3 still does not have an up-to-date reference manual [43] or documentation on which features have been successfully ported or added [31]. The other reason is that ModSecurity 3 continues to have serious bugs and security issues such as DoS [44] and complete bypass [45].

ModSecurity provides the following features [27, 46]:

- Complete HTTP Traffic Logging
- Active Monitoring and Attack Detection
- Virtual Patching
- Flexible Rule Engine

ModSecurity 2 also has two separate deployment options:

Embedded ModSecurity can be deployed as part of an existing web server infrastructure, such as an additional Apache module. This has configuration, load balancing, overhead, and complexity benefits. However, ModSecurity will have to share server resources.

Reverse Proxy ModSecurity can be independently deployed as a sort of HTTP router that stands in front of a web server. This has the benefit of adding a separate isolated security layer that has its own dedicated resources. However, this adds an additional point of failure, and thus some redundant reverse proxies will need to be added.

Rule Language

ModSecurity uses its own Turing Complete rule language [47]. The rule language consists of three types of directives [27]:

Configuration directives These directives specify how ModSecurity should process data. An example is `SecRuleEngine`, which controls whether the ModSecurity rule engine is on, off, or detection only.

Rule directives These directives specify what ModSecurity should do with the processed data, such as pass or block. The most important and obvious example of this is `SecRule`, which creates a rule that will analyze provided variables using a selecter operator and optionally perform certain actions. This will be described in greater detail below.

Other These are advanced or less commonly used directives that may not fit into the other categories. An example is `SecHashEngine`, which enables the ModSecurity hash engine for cryptographically signing links.

According to the ModSecurity Handbook, rules defined by a `SecRule` directive conform to the same format that consists of four parts [27]:

```
SecRule VARIABLES OPERATOR [TRANSFORMATION_FUNCTIONS] ACTIONS
```

Variables Identify the part(s) of an HTTP transaction that the rule should work with.

Operators Specify how the part(s) identified by one or more variables should be analyzed.

Transformation functions Optionally, a rule can specify transformation functions. These can modify the input before the operators act.

Actions Specify the action(s) that should be taken when a rule has matched, such as blocking or passing a request.

Core Rule Set (CRS)

The OWASP ModSecurity Core Rule Set (CRS) is a set of generic attack rules that can be used by ModSecurity or any other compatible WAF [29]. It aims to protect web applications from the attack types mentioned in the OWASP Top 10 List in addition to other common attacks (see Section 2.1), while also minimizing the amount of false alarms.

2.2.5 Virtual Patching

When an organization discovers a security vulnerability in its deployed web application, it needs to address that vulnerability as quickly and as thoroughly as possible to prevent bad actors from potentially wreaking havoc. The most obvious resolution strategy is for the organization to identify the vulnerable source code, fix the vulnerable source code, and then deploy and install a patch containing that fix. However, this strategy may not always be possible or timely. OWASP gives several reasons for this [48, 49]:

Third party software If the vulnerability is caused by or lies within a vendor's commercial module or application, then the organization, as a customer, may not have access to the relevant source code. The organization has to wait for the vendor's official patch, which may not be available as quickly as the customer would like.

Long installation time Even if a patch is quickly and readily available, extensive and time-consuming regression testing is often needed prior to deployment into production.

Lack of resources Developers may already be allocated to other projects, and/or it may be deemed too expensive to fix the custom code causing the vulnerability.

Legacy Code The organization may be necessarily utilizing a commercial application or module that is no longer actively supported by the vendor.

Outsourced Code The organization may be outsourcing some or all of their application development, adding an additional layer of complexity. Asking for a vulnerability fix may require an entirely new project and additional cost.

WAFs can mitigate these issues, often completely, in strategy known as virtual patching. OWASP's Virtual Patching Cheat Sheet [49] provides the following definition for virtual patching: "A security policy enforcement layer which prevents and reports the exploitation attempt of a known vulnerability". Unlike typical WAF strategies, which see the firewall deployed site-wide with only some application-specific tuning, virtual patching is meant to apply only to certain resources and parameters, resulting in rules and signatures that are specific to the application under protection.

As with WAFs in general, virtual patching can use the security models mentioned in Section 2.2.2.

In order to create whitelist virtual patches, it must be known what the valid and expected input values are for a given parameter or resource. This is generally the recommended strategy [49], especially since it can be applied to every parameter and resource in the web application regardless of the existence of vulnerabilities. This can be considered a form of defense-in-depth [50].

For blacklist virtual patches, the goal is to create rules that block the specific types of attacks that can exploit the underlying application's vulnerabilities. However, one must take care not to create an exploit patch that is too specific, e.g., a patch that only blocks a specific payload/string, since these types of rules can be easily bypassed by tweaking something inconsequential like the number of spaces.

2.2.6 Evasion Strategies

There are many strategies for evading WAF protections. OWASP itself has several pages dedicated to evasion concepts and payloads [51, 52]. Other payloads and strategies can come from open-source code repositories [53] or tweets from security analysts [54]. Some of these strategies can be used as a basis for improving WAFs and WAF rulesets, which will be discussed in Chapter 3.

For XSS, many evasions involve encoding or obfuscation of the malicious script. The obfuscation can be a syntax error, an obscure encoding, or an esoteric subset of Javascript like JSFuck. Others require the usage of obscure or obsolete HTML attributes and events [55]. Mimicry JavaScript attacks, a variation of XSS attacks, use slight transformations (i.e., changing the leaf values of abstract syntax tree) of an application's benign scripts as attack vectors for malicious purposes. This bypasses WAFs that use script-whitelisting mechanisms. Script-whitelisting mechanisms creates unique identifiers for every valid script during a training phase, which takes place before an app goes live. These identifiers combine elements that are extracted from either the script, i.e., part of the AST, or its execution env, such as the URL that triggered execution. The identifiers are stored in a whitelist. During productions, only scripts that generate identifiers in the whitelist will be identified and approved for execution [56].

2.3 Application Security Testing

Application Security Testing refers to the tools, techniques, and services used to test applications for security flaws. This information is then used to address those flaws and thus harden the application against any potential security threats. There are several types of application security testing [57] including:

- Static Application Security Testing (SAST)
- Dynamic Application Security Testing (DAST)
- Interactice Application Security Testing (IAST)
- Runtime Application Self-Protection (RASP)

SAST and DAST are the most well-known approaches to application security

testing. They will be discussed below in sections 2.3.2 and 2.3.1. IAST tools run as a software agent, which allows for data from running applications to be collected and analyzed [57]. This data helps provide the developer a better understanding of their application's security situation, and it can also be leveraged by other security testing tools. RASP tools are embedded inside of the application itself and can block attacks.

2.3.1 Static Application Security Testing

Static Application Security Testing (SAST) tools take a white-box approach to security testing. They automatically analyze the source code of application in order to reason about the run-time of a program without actually executing it [58]. SAST tools usually leverage compiler technology to construct Abstract Syntax Tree (AST)s from the source code. Then, analysis must be performed locally (within a function), modularly (within a file or module), and globally (across the entire application) [59]. SAST tools scale well and produce reports that are very useful to software developers who want to know where exactly in the source code the issue is occurring [60]. However, many types of security vulnerabilities, such as authentication problems, access control issues, and misconfigurations, cannot easily be found from static source code analysis [60]. False positives are common, and even figuring out the exploit for true positives can be difficult. OWASP maintains a large table of commerical, open-source, and free SAST tools [60].

SAST techniques and concepts most relevant to this thesis are described below.

Abstract Syntax Tree (AST)

A tree that represents the syntactic structure of source code written in a specific programming language. Unlike parse trees, these do not represent concrete program syntax. Inner nodes of the tree represent operators and leaf nodes represent operands [61].

Control Flow Graph (CFG)

A graph that describes all of the code execution paths within a program as well as the necessary conditions for those paths. Control flow graphs can be constructed from ASTs [61].

Program Dependency Graph (PDG)

A graph that represents dependencies among statements and predicates within a program. It consists of data dependency edges, which each represent the influence of one variable on another, and control dependency edges, which each represent the influence of a predicate on a variable [61].

Code property graphs

A graph that merges ASTs, CFGs, and PDGs into a joint data structure that can be used for modeling security vulnerabilities within a program [61].

Taint analysis

A type of information flow analysis used in the security domain that traces user data from "sources" to locations of interest called "sinks" [62]. Any variables that the input data modifies is considered "tainted" until it has been properly sanitized

Symbolic execution

A way to systematically explore many execution paths concurrently without relying on concrete inputs. Instead, inputs are abstractly represented using symbols, and constraint solvers are used to find violations [63].

SMT solvers

According to Barrett et al.: "Satisfiability Modulo Theories (SMT) refers to the problem of determining whether a first-order formula is satisfiable with respect to some logical theory" [64]. SMT solvers attempt to solve these types of problems. These solvers are particularly in string constraint solving for symbolic execution. Examples include the Z3 Theorem Prover [65] and the Z3str2 String Constraint Solver [66].

2.3.2 Dynamic Application Security Testing

Dynamic Application Security Testing (DAST) tools, also known as vulnerability scanners, take a black-box approach to security testing. These automated tools scan an application for potential security vulnerabilities by injecting malicious input and observing the response from the application. These are usually used in conjunction with SAST tools [67]. OWASP also maintains a large table of commercial, free, and open-source DAST tools [68]. In addition, there is a project called the Web Application Vulnerability Scanner Evaluation Project (WAVSEP) that was designed to help assess the quality of various DAST tools [69].

Fuzzing

Fuzzing is defined by Sutton et al. as "a method for discovering faults in software by providing unexpected input and monitoring for exceptions" [70]. Vulnerability scanners heavily employ various types of fuzzing in order to find potential security flaws.

2.3.3 Automatic Exploit Generation

Automatic exploit generation tools use some combination of the whitebox and blackbox techniques described above to find bugs, determine whether they are exploitable, and then produce a working attack string to achieve that exploit [71]. Subsequently, this exploit could be used to then patch the vulnerability [72]. Examples of automatic exploit generators include AEG [71], CRAXweb [73], Chainsaw [74], and NAVEX [75]. The latter of this list is particularly notable since its focus is on dynamic web applications and its code repository is public [76].

Chapter 3

Related Works

In this chapter, related works will be briefly mentioned or summarized.

3.1 Multivocal Literature Review

3.1.1 WAF Strengths and Weaknesses

In order to determine which types of attacks the above WAFs were most and least effective against, a MLR was conducted following a set of guidelines designed for software engineering [77].

Planning the review

First, it needed to be determined whether or not a MLR was actually needed. After a few manual searches using Scopus, Google Scholar, and Google, the following questions [77] were answered:

#	Question	Answer
1	Is the subject "complex" and not solvable by considering only the formal literature?	**Yes**
2	Is there a lack of volume or quality of evidence, or a lack of consensus of outcome measurement in the formal literature?	**Yes**
3	Is the contextual information important to the subject under study?	**Yes**
4	Is it the goal to validate or corroborate scientific outcomes with practical experiences?	**No**
5	Is it the goal to challenge assumptions or falsify results from practice using academic research or vice versa?	**No**
6	Would a synthesis of insights and evidence from the industrial and academic community be useful to one or even both communities?	**Yes**
7	Is there a large volume of practitioner sources indicating high practitioner interest in a topic?	**Yes**

Figure 3.1: Should I conduct an MLR?

Since most answers were yes, an MLR is appropriate. Next, a research/review question with sub-questions was formulated:

1. How many studies have evaluated WAFs in terms of the types of attacks they are able to protect against?
    a. Which WAFs have been evaluated?
    b. Which types of attacks are most effective, and how do they map to OWASP's Top 10 Web Application Security Risks?

Conducting the review

The following search engine(s) were used for academic/formal literature:

- Scopus
- Google Scholar

The following search engine(s) were used for gray literature:

- Google

Stopping criteria:

- First 50 search hits
- Continue only if last page reveals anything new or interesting

Inclusion criteria:

- Discusses the effectiveness of an attack/attacks against an available WAF/WAFs
- Comprehensible English
- 2014 or newer

Query:

```
"web application firewall" AND (fail OR survey OR comparison OR "false negative" OR "evade" OR "evasion" OR "bypass" OR detect)
```

Taking into account the stopping and inclusion criteria, there were only 13 results. Thus, it was decided to repeat the following query for every WAF listed in [28] (starting with Amazon) and ModSecurity:

```
"amazon" "application firewall" "bypass"
```

This yielded an additional 22 results once the stopping and inclusion criteria were applied.

Results

Table 3.1 shows which types of attacks were able to successfully bypass a given WAF. The attacks are categorized by OWASP Top 10 - 2017 categories. An x means that a type of attack indicated by the column was successful against the WAF indicated by a row, i.e., the WAF was bypassed. Local File Inclusion/Remote File Inclusion (LFI/RFI) is also a category.

Tables 3.2 and 3.3 show the number of formal and gray papers that pertained to each WAF and each OWASP Top 10 - 2017 category.

	OWASP Top 10 - 2017										
WAF	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**	**LFI/RFI**
Akamai	x						x				
Amazon											
Barracuda		x					x				
Cloudflare	x					x	x				
Comodo	x					x					x
Fortinet											
F5 Big IP	x				x		x	x			
Guardian	x				x	x	x				x
Imperva Incapsula							x				
Microsoft Azure											
ModSecurity	x	x			x	x	x			x	x
PHPIDS	x					x					x
QuickDefense	x					x	x				x
Radware											
Signal Sciences											
Sucuri	x					x	x				
WebKnight	x				x	x	x				

Table 3.1: Successful attacks against WAFs categorized by OWASP Top 10 - 2017.

OWASP Top 10 - 2017	**Formal Papers**	**Gray Papers**	**Total**
A1	11	12	23
A2	2	1	3
A3	0	0	0
A4	0	0	0
A5	5	1	6
A6	1	5	6
A7	3	7	10
A8	0	1	1
A9	0	0	0
A10	1	0	1
LFI/RFI	3	1	4

Table 3.2: Number of papers that pertain to each OWASP Top 10 - 2017 category.

WAF	Formal Papers	Gray Papers	Total
Akamai	0	3	3
Amazon	0	0	0
Barracuda	0	2	2
Cloudflare	0	9	9
Comodo	0	1	1
Fortinet	0	0	0
F5 Big IP	0	5	5
Guardian	1	0	1
Imperva Incapsula	0	2	2
Microsoft Azure	0	0	0
ModSecurity	12	9	21
PHPIDS	1	2	3
QuickDefense	0	2	2
Radware	0	0	0
Signal Sciences	0	0	0
Sucuri	0	4	4
WebKnight	1	2	3

Table 3.3: Number of papers that pertain to each WAF.

3.2 Virtual Patching

Although there have been attempts to automate whitelist virtual patching, it remains a largely manual process. Betarte et al. [33] developed a tool called DEPSA that could translate "security requirements expressed in a high-level language over a model of the vulnerable application" into both whitelist and virtual patches (specifically, ModSecurity rules). However, the application needs to be modeled and the requirements specified, both of which are largely manual processes.

Automated blacklist virtual patch creation seems to have more tools, however most are proprietary commercial solutions that rely on the output of a proprietary Dynamic Application Security Testing (DAST), and some of those are a decade old and likely obsolete. None have been independently assessed or evaluated in academic literature. The OWASP ModSecurity Core Rule Set (CRS) has virtual patching scripts that parse vulnerability reports generated by open-source DAST tools such as OWASP ZAP and Arachni Web Scanner [68, 78], but they are also a decade old and no longer function as originally intended, and have not been rigorously evaluated. They rely on the anomaly scoring feature of the CRS. Essentially, the anomaly score would be incremented by another fixed amount when the location and parameter matched a known vulnerability. The Arachni solution [68] was particularly interesting because ModSecurity itself would initiate the scan whenever a resource was visited that had not been previously scanned. Finally, there is a blog post by a computer security company that describes in-depth an approach to

blacklist virtual patching via static analysis [79], but this has not been evaluated by or referenced in academic literature.

Salemi et al. [80] explores automatically generating WAF rules via Runtime Application Self Protection (RASP) logs. However, the WAF is intentionally redundant in this approach since RASPs already detect and block malicious requests.

Ryan Barnett at Breach Security wrote a white paper about virtually patching the vulnerable OWASP WebGoat application [81], but this seems to have been done manually.

3.3 Automatic WAF Repair

Krueger et al. [82] creates a reverse proxy called `TokDoc` that intercepts requests, parses them into token-value pairs, and determines whether or not a token needs to be "healed" based on the learned profile of normal content. "Healing" involves dropping, encoding, or replacing the token. It is an anomaly-based solution, not a rule/signature-based one.

Appelt et al. [32, 83] puts forth an approach for repairing a WAF based on successful SQLi attacks. The approach starts with defining an attack grammar for SQLi attacks, and using that to create a diverse set of random attacks. These attacks, called the tests, are sent to a web application protected by WAF, and they are subsequently labeled by whether they passed ("P") or were blocked ("B"). The tests are then decomposed into slices (substrings) according to the grammar, and each slice is given a unique id. Slices are assigned a value of "1" if they are part of a successful attack, otherwise they receive a "0". These are then modeled by decision trees (see Figure 3.2) to derive string patterns. Finally, the regular expressions matching those string patterns are evolved via genetic algorithms with the optimization objectives of minimizing blocked legitimate requests and maximizing blocked attacks.

t.id	vector	label
1	$\langle s_1, s_2, s_3 \rangle$	B
2	$\langle s_1, s_2, s_4 \rangle$	B
3	$\langle s_4, s_5 \rangle$	P

t.id	s_1	s_2	s_3	s_4	s_5	clz
1	1	1	1	0	0	B
2	1	1	0	1	0	B
3	0	0	0	1	1	P

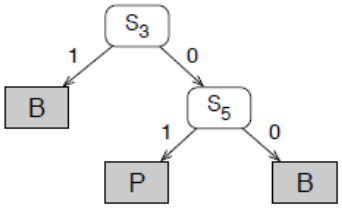

Figure 3.2: A figure from [32] that depicts attack decompositions, their encodings, and a derived decision tree. **clz** refers to the pass/block classification.

3.4 Machine Learning and AI-driven WAFs

Appelt et al. [32, 83] used machine learning techniques such as random forest to generate the original attack strings. Thang [84] also uses the random forest method to detect code injection attacks. Liu et al. [85] and Kar et al. [86] train support vector machines for improving attack detection, specifically against SQLi. Betarte et al. [87] propose a machine learning model based on one-class classifcation and n-gram analysis that outperforms CRS.

3.4.1 Improving Detection of Attacks

Previous studies have proposed anomaly detectors based on characteristics of HTTP requests such as character distribution, parameter length, and parameter value. TokDoc is a WAF that analyzes HTTP requests and replaces suspicious parts with benign pieces learned from the past [82]. One recent study uses session patterns [88]. Others use such approaches as the Random Forest Method [84] and feature analysis and SVM optimization [85].

It has been shown that it often takes more time to manually configure a WAF than it takes for a tester or hacker to bypass it [89], [90]. Thus, researchers have started to develop ways to automatically configure and repair them. [32] proposes an approach to automatically repair vulnerable WAFs by augmenting existing rulesets based on an analysis of test results, using machine learning and metaheuristics. The focus was SQL injection attacks since they sit atop OWASP Top 10.

3.5 Automatic Fixing of Vulnerabilities

As opposed to relying on a WAF which can be bypassed as mentioned in previous sections, security vulnerabilities within the code can be fixed. [91] was a survey of 20 papers that proposed some solution for automatically detecting and fixing vulnerabilities classified by OWASP Top 10. The target languages were either PHP or Java, and the three most popular vulnerabilities to address were A1 (injection), A7 (XSS), and A3 (sensitive data exposure). Of particular interest is

Chapter 4

Methodology

In Chapter 4, the overall research design, methodology, and implementation for this thesis will be described. Section 4.1 describes the motivation behind this research, followed by the research questions in Section 4.2. Subsequently, Section 4.3 describes the research methodology and design. Next, Section 3.1 describes the design a multi-vocal literature review that was performed to aid in determining which attacks to focus on. Finally, Section 4.4 presents the implementation of the research.

4.1 Research Motivation

Section 2.2.5 presents the background on what virtual patching is and why it can be useful. To summarize, virtual patching provides a way for an organization to quickly patch a known software vulnerability in an application without having to touch the application itself. This can be accomplished via a WAF like ModSecurity, which has a flexible rule language that is well-suited to virtual patching [27]. Virtual patching can use whitelist and/or blacklist approaches.

Manual configuration of a WAF can be difficult. It generally requires someone with technical knowledge of the web application under protection, and that someone may also need to be aware of potentially conflicting security policies of other web applications configured on the web server [92]. This can lead to misconfiguration that can be as devastating as not having any protection [92]. Palo Alto Networks discovered "that 65% of publicly disclosed security incidents in the cloud were the result of customer misconfigurations" [5]. In 2019, a CloudFlare outage was caused by a the additional of a new rule that added "a regular expression that backtracked enormously and exhausted CPU used for HTTP/HTTPS serving" [4]. Also in 2019, a former Amazon employee was able to steal more than 100 million credit application made with Capitol One by exploiting a misconfigured ModSecurity installation [93]. This signifies the need for robust automatic configuration of security controls.

As mentioned in Chapter 3, although WAFs are commercially popular [28], there is not a particularly large body of academic research concerning WAFs and

virtual patching, nor are there many open-source solutions. The research that does exist tends to focus on creating or preventing common types of injection attacks like SQLi and XSS, and many involve advanced machine learning techniques [32]. Others perform an evaluation of one or more existing WAFs.

Finally, according to OWASP [49] there are two main tenants with regards to virtual patching, where order indicates priority:

1. No false positives - Do not block benign traffic.
2. No false negatives - Do not allow attacks.

To see why the first tenant should have precedence over the second, consider the adverse effects false positives can have on an organization's business. If users and customers are often unable to complete transactions due to their legitimate requests being blocked, they will become frustrated and as a result the organization may have some unhappy customers, potentially leading to loss of business and thus revenue. To assuage this, the organization might configure the WAF to operate in a detection-only mode, but this only serves to undermine the second tenant since attacks can no longer be proactively blocked.

Given the lack of open-source implementations and academic study of automated virtual patching and WAF configuration, this research aims to investigate the feasibility and effectiveness of such solutions utilizing tools that are often already a part of the application development and testing process. The ranked tenants above will serve as guidelines for what to implement and evaluate.

4.2 Research Questions

The research motivation described in 4.1 yielded three research questions. They are as follows:

RQ1. How can dynamic analysis tools be used to automatically generate virtual patches for a vulnerable application?

RQ2. How can static analysis tools be used to automatically generate virtual patches for a vulnerable application?

RQ3. How effective are automatically generated virtual patches compared to a standard set of WAF rules?

4.3 Research Method and Design

This section will describe the overall research method and design for this thesis. This includes the research strategy, data generation and analysis methods.

4.3.1 Research Strategy

To address the research questions, a series of experiments will be performed that pit selected open-source vulnerability scanners against selected vulnerable web

applications with a rule-based WAF standing in-between. These experiments will be conducted for several different configurations of the WAF:

- Rule engine disabled
- Rule engine enabled with a standard rule set in place
- Rule engine enabled with generated virtual patches in place

In order to generate the virtual patches, output from an application security testing tool (DAST or SAST) must be translated into rules and directives that can be understood by the WAF. A prototypical tool called VPgen will be developed for this purpose. VPgen will process the report generated by an application security testing tool for a vulnerable application and use it to generate rules and directives that can be understood by a WAF. This strategy can be considered an experimental strategy [94], since it will be characterized by the observation, measurement, and comparison of results obtained with and without the implemented prototype.

High-level design of VPgen

The high-level design of VPgen can be seen in Figure 4.1. The steps are as follows:

1. **Input vulnerability report** Take the path to a vulnerability report generated by a security analysis tool as input.
2. **Is format supported?** Determines whether the report type is supported or not based on vulnerability scanner and file format.
3. **Process report** The type, location, and parameter for each vulnerability is scraped from the report and stored.
4. **Generate rules** Directives and rules specific to the scraped vulnerability type, location, and parameter are created.
5. **Output rules file** The directives and rules are output to a file that the WAF should reference.

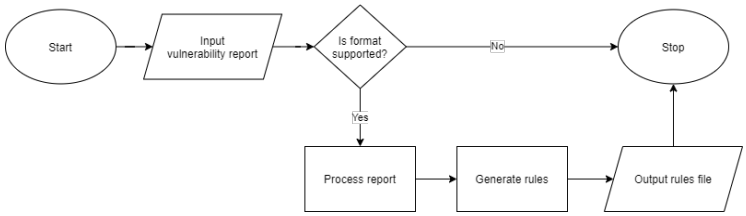

Figure 4.1: High-level design of VPgen

4.3.2 Data Generation and Analysis

Data will be generated by using vulnerability scanners to attack vulnerable web application. The data itself will depend on how the vulnerability scanner is being used. It will be operated in two different modes described below.

Active Scanning

Vulnerability scanners operating in active scanning mode will automatically crawl an application in order to find any and all data entry points. They will then automatically fuzz these data entry points using payloads of known attacks. In this mode, the data generated will be the number of vulnerabilities discovered for each specific type of attack/vulnerability.

Targeted Fuzzing

In this mode, specific pages and parameters of the vulnerable web application will be manually selected via the vulnerability scanner and fuzzed. The vulnerability scanner will provide a nice interface to facilitate this, but the user must specify the location and the payload(s). The data generated from this mode will be the number of payloads that were blocked by or passed through the WAF.

Results from attacking and generating virtual patches via DAST tools will be used to address RQ1. For RQ2, a SAST tool will be used. Finally, to address RQ3, the results from attacking a vulnerable web application protected by a standard, application-wide WAF rule set will be used. This data will be analyzed quantitatively via recall and precision measures that will be explained below.

4.4 Research Implementation

This section will describe the in-depth implementation of the research method and design explained above in Section 4.3. The first several subsections will describe the selection criteria for the WAF, base rule set 4.4.2, DAST tools 4.4.3, SAST tool 4.4.4, vulnerable applications 4.4.5, and vulnerability/attack types 4.4.6. Afterwards, in Section 4.4.7, VPgen will be described in depth.

4.4.1 Selection of the WAF

The main criterion for the selection of the WAF was that it needed to be open-source. Although there are many popular commercial solutions [28], not being able to dig into their implementations in order to definitively determine how they operate is a negative for this project. Few have been openly evaluated, so we would have to take the vendor at their word in terms of their effectiveness and capabilities.

The other criteria were:

- Rule-based
- Through documentation
- Actively maintained
- Active discussion board or mailing list
- Prevalence in academic literature

Although this methodology is not specific to any one WAF, ModSecurity is a natural candidate. It fulfills all of the criteria, as mentioned in Sections 2.2.3 and 2.2.4. Other options include WebKnight [36] and ShadowDaemon [39], with the former also occasionally appearing in the literature [35, 37].

Due to bugs and security issues with ModSecurity 3 [44, 45], ModSecurity version 2.9.4 is selected.

4.4.2 Selection of Attack Detection WAF Rules

This thesis is not about improving defenses against specific types of attacks, so it was decided that an existing rule set should be leveraged by the virtual patching tool (VPgen). This rule set could also be used as a baseline to compare against. The main criterion for this rule set is was that its rules should be able to be easily categorized and grouped by attack/vulnerability type. Since ModSecurity was chosen as the WAF, that makes the OWASP Core Rule Set (CRS) [29] is an obvious candidate as the rule set.

The CRS groups rules by attack type such that different groups of rules reside in different configuration files. These rules will also have specific ID ranges and tags that further help communicate their type. This will enable a tool like VPgen to only include rules that are relevant to the identified vulnerability. Listing 4.1 shows the rule configuration files from CRS's `rules/` directory.

Code listing 4.1: CRS Rule Files

```
REQUEST-900-EXCLUSION-RULES-BEFORE-CRS.conf
REQUEST-901-INITIALIZATION.conf
REQUEST-903.9001-DRUPAL-EXCLUSION-RULES.conf
REQUEST-903.9002-WORDPRESS-EXCLUSION-RULES.conf
REQUEST-903.9003-NEXTCLOUD-EXCLUSION-RULES.conf
REQUEST-903.9004-DOKUWIKI-EXCLUSION-RULES.conf
REQUEST-903.9005-CPANEL-EXCLUSION-RULES.conf
REQUEST-903.9006-XENFORO-EXCLUSION-RULES.conf
REQUEST-905-COMMON-EXCEPTIONS.conf
REQUEST-910-IP-REPUTATION.conf
REQUEST-911-METHOD-ENFORCEMENT.conf
REQUEST-912-DOS-PROTECTION.conf
REQUEST-913-SCANNER-DETECTION.conf
REQUEST-920-PROTOCOL-ENFORCEMENT.conf
REQUEST-921-PROTOCOL-ATTACK.conf
REQUEST-930-APPLICATION-ATTACK-LFI.conf
REQUEST-931-APPLICATION-ATTACK-RFI.conf
REQUEST-932-APPLICATION-ATTACK-RCE.conf
REQUEST-933-APPLICATION-ATTACK-PHP.conf
REQUEST-934-APPLICATION-ATTACK-NODEJS.conf
REQUEST-941-APPLICATION-ATTACK-XSS.conf
REQUEST-942-APPLICATION-ATTACK-SQLI.conf
REQUEST-943-APPLICATION-ATTACK-SESSION-FIXATION.conf
REQUEST-944-APPLICATION-ATTACK-JAVA.conf
REQUEST-949-BLOCKING-EVALUATION.conf
RESPONSE-950-DATA-LEAKAGES.conf
RESPONSE-951-DATA-LEAKAGES-SQL.conf
RESPONSE-952-DATA-LEAKAGES-JAVA.conf
RESPONSE-953-DATA-LEAKAGES-PHP.conf
```

```
RESPONSE-954-DATA-LEAKAGES-IIS.conf
RESPONSE-959-BLOCKING-EVALUATION.conf
RESPONSE-980-CORRELATION.conf
RESPONSE-999-EXCLUSION-RULES-AFTER-CRS.conf
```

Note that rule configuration files are of the format:

```
<REQUEST|RESPONSE>-<ID_PREFIX>-<ATTACK_TYPE>.conf
```

This way, if a rule begins with the ID 942, that indicates that it is supposed to protect against SQLi attacks. More explicitly, rules have tags (such as `"attack-xss"`) that indicate the attack type. See listing 4.2 for an example.

Code listing 4.2: Example of an XSS Rule

```
SecRule REQUEST_COOKIES|!REQUEST_COOKIES:/__utm/|REQUEST_COOKIES_NAMES|
    ↪ REQUEST_HEADERS:User-Agent|ARGS_NAMES|ARGS|XML:/* "@detectXSS" \
    "id:941100,\
    phase:2,\
    block,\
    t:none,t:utf8toUnicode,t:urlDecodeUni,t:htmlEntityDecode,t:jsDecode,t:
        ↪ cssDecode,t:removeNulls,\
    msg:'XSS Attack Detected via libinjection',\
    logdata:'Matched Data: XSS data found within %{MATCHED_VAR_NAME}: %{
        ↪ MATCHED_VAR}',\
    tag:'application-multi',\
    tag:'language-multi',\
    tag:'platform-multi',\
    tag:'attack-xss',\
    tag:'paranoia-level/1',\
    tag:'OWASP_CRS',\
    tag:'capec/1000/152/242',\
    ctl:auditLogParts=+E,\
    ver:'OWASP_CRS/3.3.0',\
    severity:'CRITICAL',\
    setvar:'tx.xss_score=+%{tx.critical_anomaly_score}',\
    setvar:'tx.anomaly_score_pl1=+%{tx.critical_anomaly_score}'"
```

Core Rule Set version 3.3.0 is selected. In this version, anomaly scoring mode is the default, and the paranoia level 1. These defaults will apply for the active scanning tests, but the paranoia level will be incremented up to its maximum level for the targeted fuzzing tests. In anomaly scoring mode, matching rules will contribute to a transactional score. If this score exceeds a specified threshold, the request will be blocked. Paranoia levels control how many CRS rules should apply. The higher the paranoia level, the more rules and thus the higher the level of security.

4.4.3 Selection of DAST Tools

There are many available DAST tools/vulnerability scanners as evidenced by OWASP's page on the topic [68]. There have also been several papers and tool that attempt to assess their effectiveness [69, 95]. Portswigger's Burp is a popular commercial vulnerability scanner, but its free community version is limited [96]. In addition, an open-source vulnerability scanner is more desirable since it allows us to see

how attack payloads are generated and categorized, which is important when it comes to mapping vulnerability reports to virtual patches in VPgen.

Based on the previously referenced literature, OWASP's large list, and previous virtual patch generation work [89], OWASP Zed Attack Proxy (ZAP) [97] and Arachni web scanner [98] were chosen. However, Arachni has not been maintained for several years, and relies on some old dependencies that make it difficult to build and work with on a newer system. It has also been removed from the rolling Kali Linux distribution [99], which contains a curated list of vulnerability analysis and web application attack tools [100]. Instead, wapiti, another actively maintained vulnerability scanner that is still a part of the rolling Kali Linux distribution, was selected.

4.4.4 Selection of SAST Tools

A difficulty with static analysis tools is that although they provide helpful information to the developer such as the line(s) of code causing the vulnerability, the source, and the sink, this is not necessarily actionable information. This is where automatic exploit generation can help (see Section 2.3.3). If an exploit for a given vulnerability can be generated, then it follows that at least an exploit-based patch can be created [72]. Selection was simple, since the only tool that appears to be publicy available is NAVEX [75, 76]. NAVEX is not a purely static tool, though it can be split into two parts, the first of which generates the initial exploits via code property graphs, taint analysis, and string constraint solvers, and as such is completely static. Thus, the first part is selected for this thesis.

4.4.5 Selection of Vulnerable Web Applications

The criteria for the selection of vulnerable web applications is the following:

- Open-source
- Written in PHP
- Appears in academic literature
- Can be built on a Linux VM without issues caused by legacy dependencies or PHP versions
- Can be easily scanned by an automated vulnerability scanner

PHP is a requirement since NAVEX only works on PHP applications.

OWASP maintains a list of vulnerable web applications [101]. Of these, DVWA and WackoPicko seem to appear in the literature [95, 102, 103]. For the genuinely vulnerable web applications, the NAVEX provides a large list of applications it was evaluated against [75]. However, many of these tools were more than a decade old and relied on outdated version of MySQL and PHP. In addition, they were difficult to automatically scan with vulnerability scanner.

4.4.6 Selection of Security Vulnerabilities

The criteria for selecting the security vulnerabilities is as follows:

- Supported by NAVEX
- Supported by both Wapiti and OWASP ZAP
- Can be detected by OWASP CRS
- Present in one or more of the vulnerable web applications

This resulted in the following list of attacks/vulnerabilities:

- SQL Injection (SQLi) (both blind and normal)
- Cross Site Scripting (XSS) (both reflected and persistent)
- File Inclusion (FI) (path traversal, local, and remote inclusion)
- Remote Command Execution (RCE) (both command and code injection)

4.4.7 Implementation of VPgen

VPgen is a Python script that takes input in the form of an OWASP ZAP or Wapiti vulnerability report, processes it, generates rules, and outputs a file containing a list of Apache directives and ModSecurity tools. It does this by defining mappings from OWASP ZAP and Wapiti alerts to OWASP CRS ruleset files and tags.

OWASP ZAP report processing

An example of an OWASP ZAP alert is shown in Listing 4.3. Each alert item contains one or more instances. Each instance contains the location (uri), method, vulnerable parameter, and attack string used.

Code listing 4.3: Excerpt from an OWASP ZAP report where DVWA was the target application.

```
<alertitem>
    <pluginid>40018</pluginid>
    <alertRef>40018</alertRef>
    <alert>SQL Injection</alert>
    <name>SQL Injection</name>
    <riskcode>3</riskcode>
    <confidence>2</confidence>
    <riskdesc>High (Medium)</riskdesc>
    <desc>&lt;p&gt;SQL injection may be possible.&lt;/p&gt;</desc>
    <instances>
    <instance>
    <uri>http://waf-virtualbox/DVWA/vulnerabilities/sqli/?Submit=Submit&amp;id=ZAP
        ↪ %27+AND+%271%27%3D%271%27+--+</uri>
    <method>GET</method>
    <param>id</param>
    <attack>ZAP&apos; OR &apos;1&apos;=&apos;1&apos; -- </attack>
    </instance>
    <instance>
    <uri>http://waf-virtualbox/DVWA/vulnerabilities/sqli_blind/?Submit=Submit&amp;
        ↪ id=ZAP%27+AND+%271%27%3D%271%27+--+</uri>
    <method>GET</method>
    <param>id</param>
```

```
    <attack>ZAP&apos; OR &apos;1&apos;=&apos;1&apos; -- </attack>
    </instance>
    </instances>
    <count>2</count>
   ...
</alertitem>
```

Only released, high risk alerts resulting from an Active Scan Rule are considered. In addition, these alerts must normally be associated with a parameter. Each of these will be mapped to one or more OWASP CRS rule set files. See Table 4.1 for the mapping.

Table 4.1: A mapping of OWASP ZAP alert types to OWASP CRS rule set files.

Alert	**Rule Set**
Path Traversal	REQUEST-930-APPLICATION-ATTACK-LFI.conf REQUEST-933-APPLICATION-ATTACK-PHP.conf
Remote File Inclusion	REQUEST-931-APPLICATION-ATTACK-RFI.conf REQUEST-933-APPLICATION-ATTACK-PHP.conf
Cross Site Scripting (Reflected)	REQUEST-941-APPLICATION-ATTACK-XSS.conf
Cross Site Scripting (Persistent)	REQUEST-941-APPLICATION-ATTACK-XSS.conf
SQL Injection	REQUEST-942-APPLICATION-ATTACK-SQLI.conf
Server Side Code Injection - PHP Code Injection	REQUEST-933-APPLICATION-ATTACK-PHP.conf
Remote OS Command Injection	REQUEST-932-APPLICATION-ATTACK-RCE.conf

Wapiti report processing

An example of a Wapiti alert is shown in Listing 4.4. Each alert contains an array of successful attacks, with each element containing the location (path), parameter, and full HTTP request with the attack payload.

Code listing 4.4: Excerpt from a Wapiti report where DVWA was the target application.

```
    "SQL Injection": [
      {
        "method": "GET",
        "path": "/DVWA/vulnerabilities/brute/",
        "info": "SQL Injection (DMBS: MySQL) via injection in the parameter
            ↪ username",
        "level": 4,
        "parameter": "username",
        "referer": "http://waf-virtualbox/DVWA/vulnerabilities/brute/",
        "module": "sql",
        "auth": null,
        "http_request": "GET /DVWA/vulnerabilities/brute/?username=alice%C2%BF
            ↪ %27%22%28&password=Letm3in_&Login=Login HTTP/1.1\nHost: waf-
            ↪ virtualbox\nReferer: http://waf-virtualbox/DVWA/vulnerabilities/
            ↪ brute/",
```

```
      "curl_command": "curl \"http://waf-virtualbox/DVWA/vulnerabilities/brute/?
          ↪ username=alice%C2%BF%27%22%28&password=Letm3in_&Login=Login\" -e \"
          ↪ http://waf-virtualbox/DVWA/vulnerabilities/brute/\""
    },
    {
      "method": "GET",
      "path": "/DVWA/vulnerabilities/sqli/",
      "info": "SQL Injection (DMBS: MySQL) via injection in the parameter id",
      "level": 4,
      "parameter": "id",
      "referer": "http://waf-virtualbox/DVWA/vulnerabilities/sqli/",
      "module": "sql",
      "auth": null,
      "http_request": "GET /DVWA/vulnerabilities/sqli/?id=default%C2%BF%27%22%28&
          ↪ Submit=Submit HTTP/1.1\nHost: waf-virtualbox\nReferer: http://waf-
          ↪ virtualbox/DVWA/vulnerabilities/sqli/",
      "curl_command": "curl \"http://waf-virtualbox/DVWA/vulnerabilities/sqli/?id
          ↪ =default%C2%BF%27%22%28&Submit=Submit\" -e \"http://waf-virtualbox/
          ↪ DVWA/vulnerabilities/sqli/\""
    }
  ],
```

Mapping Wapiti alerts to OWASP CRS rule set files is not as straightforward as it is with ZAP. This is because the alert types correspond to Wapiti attack modules, and each Wapiti attack module can contains several types of attacks. For example, the `sql` module also contains LDAP and XPATH injection attacks. The `file` module contains both LFI and RFI attacks in addition to some code injection attacks. This was determined by a combination of trial-and-error and viewing the Wapiti source code. See Table 4.2 for the complete mapping.

Table 4.2: A mapping of OWASP ZAP alert types to OWASP CRS rule set files.

Alert	Rule Set
Path Traversal	REQUEST-930-APPLICATION-ATTACK-LFI.conf REQUEST-931-APPLICATION-ATTACK-RFI.conf REQUEST-933-APPLICATION-ATTACK-PHP.conf
Cross Site Scripting	REQUEST-941-APPLICATION-ATTACK-XSS.conf
SQL Injection	REQUEST-942-APPLICATION-ATTACK-SQLI.conf
Blind SQL Injection	REQUEST-942-APPLICATION-ATTACK-SQLI.conf
Command Execution	REQUEST-932-APPLICATION-ATTACK-RCE.conf REQUEST-933-APPLICATION-ATTACK-PHP.conf

Rule generation

The goal with virtual patching is to have rules that apply to vulnerable locations and parameters, and for those rules to be specific to the vulnerability type. Location-specific virtual patches can be achieved by creating Apache configuration contexts using container directives. Configuration contexts provide a mechanism for the selective application of a configuration [27]. They come in pairs called tags, accept parameters, and enclose various other directives such as `Include`

and `SecRule`. Specifically, VPgen uses the `<LocationMatch>` directive to create a location-specific configuration context.

`<LocationMatch>` takes a regular expression as an argument. If a URL's path matches that argument, then directives enclosed within the `LocationMatch` tags will apply. As a running example, recall the OWASP ZAP alert from Listing 4.3. VPgen will create the following configuration context for the first instance:

Code listing 4.5: A location-specific context created by VPgen for the running example.

```
<LocationMatch "^/DVWA/vulnerabilities/sqli/$">
    ...
</LocationMatch>
```

Note that this approach could also work with nginx, or any other web server that has the notion of configuration contexts.

Now that a location-specific configuration has been achieved, the next step is to make it parameter-specific. VPgen is concerned with parameters that appear in the query string (GET requests) and body (POST request). Unfortunately, `LocationMatch` only matches against a URL's path. Apache does have a `QUERY_STRING` variable, so that combined with an `If` directive should accomplish the goal for GET requests. However, this would not cover parameters that appear in the body of a POST request.

Fortunately, the ModSecurity rule language contains a special type of `SecRule` that allows the update a of rule targets at configure-time. This can be done by ID, message, or tag. Recall that rules in OWASP CRS can also be categorized by tag, e.g., "attack-xss". The mappings between the vulnerability scanners and the tags are identical to the ones in Tables 4.1 and 4.2, except replace the configuration files with tags. Thus, VPgen will generate a `SecRule` that removes all targets via negation for a certain tag, followed by another `SecRule` that adds a single parameter via the `ARGS` target:

Code listing 4.6: Configure-time updates of rule targets created by VPgen for the running example.

```
SecRuleUpdateTargetByTag "attack-sqli" "!ARGS,\
                          !ARGS_NAMES,\
                          !REQUEST_COOKIES,\
                          !REQUEST_COOKIES_NAMES,\
                          !REQUEST_HEADERS,\
                          !FILES,\
                          !FILES_NAMES,\
                          !PATH_INFO,\
                          !QUERY_STRING,\
                          !REQUEST_BODY,\
                          !REQUEST_BASENAME,\
                          !REQUEST_FILENAME,\
                          !XML,\
                          !REQUEST_LINE,\
                          !REQUEST_URI,\
                          !REQUEST_URI_RAW"

SecRuleUpdateTargetByTag "attack-sqli" "ARGS:id"
```

Note that the list of negated targets is a hard-coded list. It comes from a manual review of all the relevant rules in OWASP CRS, and is meant to be exhaustive.

Finally, using the mapping from Table 4.1 to include the relevant rule(s), the configuration context from 4.5 to create a location-specific virtual patch, the special configuration update rules from 4.6 to make it parameter-specfic, and adding some additional directives that are required as per CRS documentation, VPgen creates the following virtual patch for the running example:

Code listing 4.7: Complete virtual patch generated by VPgen for a SQLi vulnerability in DVWA

```
<LocationMatch "^/DVWA/vulnerabilities/sqli/$">
    SecDefaultAction "phase:1,log,auditlog,pass"
    SecDefaultAction "phase:2,log,auditlog,pass"

    Include /coreruleset/rules/REQUEST-942-APPLICATION-ATTACK-SQLI.conf

    Include /coreruleset/rules/REQUEST-949-BLOCKING-EVALUATION.conf
    Include /coreruleset/rules/RESPONSE-980-CORRELATION.conf

    SecRuleUpdateTargetByTag "attack-sqli" "!ARGS,\
                              !ARGS_NAMES,\
                              !REQUEST_COOKIES,\
                              !REQUEST_COOKIES_NAMES,\
                              !REQUEST_HEADERS,\
                              !FILES,\
                              !FILES_NAMES,\
                              !PATH_INFO,\
                              !QUERY_STRING,\
                              !REQUEST_BODY,\
                              !REQUEST_BASENAME,\
                              !REQUEST_FILENAME,\
                              !XML,\
                              !REQUEST_LINE,\
                              !REQUEST_URI,\
                              !REQUEST_URI_RAW"

    SecRuleUpdateTargetByTag "attack-sqli" "ARGS:id"
</LocationMatch>
```

4.5 Evaluation Design

4.5.1 Setup

Hardware and virtualizations

As opposed to setting up physical machines, a virtualization strategy was pursued. Three VirtualBox (version 6.1.22) Virtual Machine (VM)s were created and deployed to perform the experiments. Their specifications, along with the host machine's specifications, will be summarized below.

- **Host machine**
    - OS: Windows 10 Education (64-bit)

- CPU: i7-8550U @ 1.80GHz
- RAM: 16 GB

- **Web server**
  - OS: Xubuntu 20.04.2
  - CPUs: 1
  - RAM: 4 GB
- **WAF machine**
  - OS: Xubuntu 20.04.2
  - CPUs: 1
  - RAM: 4 GB
- **Client/attacker machine**
  - OS: Kali 2021.2
  - CPUs: 2
  - RAM: 4 GB

Network configuration

The network configuration consisted of two Network Address Translation (NAT) networks. The WAF machine and client machine are connected via NAT Network 2, while the web server and WAF machine are connected via NAT Network 1. This means that the client machine cannot directly communicate with the web server. Instead, it needs to go through the WAF machine, which is setup as a reverse proxy. See Figure 4.2 for a depiction of this setup.

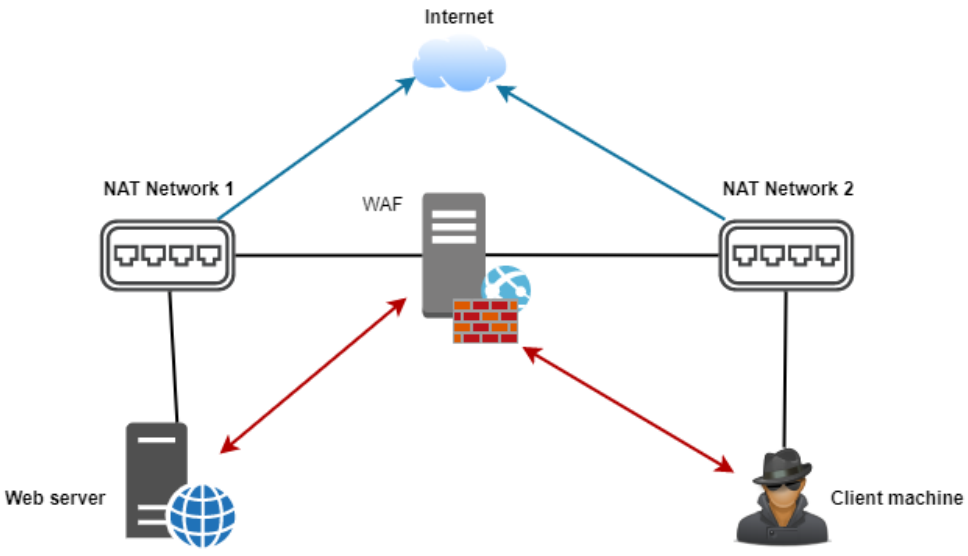

Figure 4.2: VM configuration for experiments. The headless black lines represent connections. The colored directional lines represent HTTP requests/responses.

A NAT network allows connected VMs to communicate with each other as well as with other hosts in the physical network and even external networks like the internet. However, these other hosts and external networks cannot access the VMs. This is important since we are deploying vulnerable web applications, which should not be exposed to other machines or external networks. See Figure 4.3 for a more in-depth depiction of how a NAT network operates.

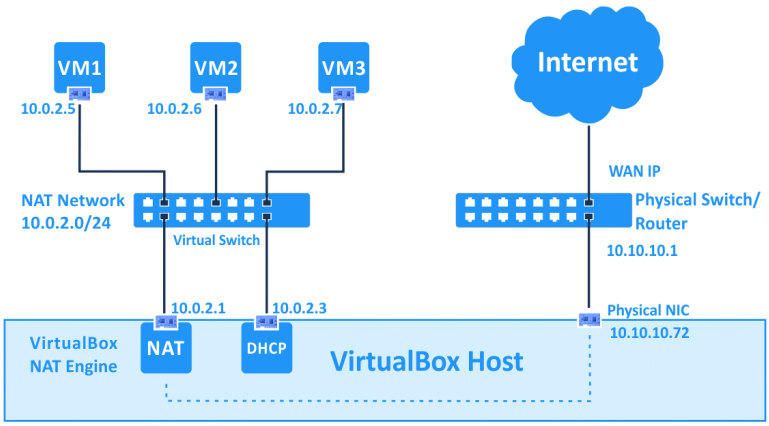

Figure 4.3: An in-depth depiction of how a VirtualBox NAT network operates. Image created by Nakivo [104].

WAF setup and deployment

ModSecurity 2.9.4 is configured as an Apache module for Apache web server 2.4.41 on an Xubuntu VM. It is set up as a reverse proxy, so all requests it receives are routed to the web server. Xubuntu was chosen because it is a more lightweight but still user-friendly version of Ubuntu.

Web application server setup and deployment

All web applications are deployed to /var/www/html. All were built or deployed from source. Apache web server 2.4.41 is used, along with PHP 5.6.40 and MySQL 5.1.73. These old versions were required for some compatibility reasons with NAVEX and the old applications it tested. The database for each application was reset between attempts. DVWA provides a button for accomplishing this, but for WackoPicko it has to be done manually.

Vulnerability scanner setup and deployment

Both OWASP ZAP 2.10.0 and Wapiti 3.0.4 come pre-installed as Kali tools on the Kali Linux distribution. Kali Linux was chosen because it was designed for penetration testing and security auditing. Wapiti 3.0.4 was uninstalled and replaced with Wapiti 3.0.5, which was built from source.

For OWASP ZAP, a context was created for each vulnerable web application. Each context would include all pages except for those disruptive to the current user's session, such as a logout action or a database reset. Form-based authentication was configured, where the URL to login is identified along with the username and password parameters. User records are created for each role in the application (such as admin and a regular user). Strings corresponding to a user being logged out or logged in were also identified.

For Wapiti, it was not possible to set this configuration up ahead of time. The links to be excluded were specified as command-line options, and each user was logged in ahead of time via a separate Wapiti utility. The generated cookie file is also specified as an option, as are the types of attacks to perform against the vulnerable web application. See Listing 4.8 for an example.

Code listing 4.8: Wapiti commands for attacking DVWA.

```
wapiti-getcookie -u http://waf-virtualbox/DVWA/login.php -c dvwa_cookies.json
wapiti -u http://waf-virtualbox/DVWA/
    -x http://waf-virtualbox/DVWA/logout.php
    -x http://waf-virtualbox/DVWA/vulnerabilities/csrf/
    -x http://waf-virtualbox/DVWA/setup.php
    -x http://waf-virtualbox/DVWA/security.php
    -x http://waf-virtualbox/DVWA/vulnerabilities/captcha/
    -x http://waf-virtualbox/DVWA/login.php
    -c dvwa_cookies.json
    -m xss,sql,permanentxss,file,exec
    -f json
```

Targeted fuzzing

A benefit to our virtual patching approach is that locations that do no correspond to any known vulnerabilities will not have any issues with false positives. For example, the login page to DVWA does not have a security vulnerability. Similarly, locations and parameters that only have a specific type of vulnerability should have relatively fewer false positives. For example, the login age to WackoPicko has a SQLi vulnerability.

OWASP ZAP's Fuzzer will be used to fuzz select requests and parameters. Specifically, the login request with username and password parameter will be targeted for both applications. The payloads will come from a leaked phpBB database of passwords that is available via FuzzDB [105]. ModSecurity will be configured with CRS set at each paranoia level (1–4), with and without virtual patches.

NAVEX setup

NAVEX has a complex setup. It relies on old versions of many other open-source projects that are no longer maintained and dependencies that must be downloaded and built from source. For example, it uses Gremlin 2.5, whereas the latest stable version is 3.5. It also relies on Neo4j 2.1 as a graph database, where the

latest stable release is 4.3.1. The complete instructions are available at NAVEX's GitHub repository [76], although they were insufficient on their own.

Chapter 5

Results

In Chapter 5, the results from applying the methods of Chapter 4 will be presented. The first section will test for vulnerabilities. The second section will test for false positives.

5.1 Vulnerability Testing

In this stage of testing, DASTs tools are used to attack web applications and report on any vulnerabilities that are found. The attacks are categorized as:

- SQLi - SQL injection
- XSS - Both reflected and stored Cross Site Scripting
- RCE - Both command injection and code injection
- FI - Path Traversal along with Local/Remote File Inclusion.

5.1.1 Rule Engine Disabled

In this test, the ModSecurity rule engine was disabled. The web applications are completely unprotected. See Table 5.1 for results.

	ZAP				Wapiti			
Application	SQLi	XSS	RCE	FI	SQLi	XSS	RCE	FI
DVWA	3	6	1	2	2	7	2	1
WackoPicko	1	4	2	2	2	2	2	2

Table 5.1: The number of vulnerabilities discovered when scanning an application with a disabled ModSecury rule engine.

5.1.2 Only Core Rule Set Enabled

In this testing, the ModSecurity rule engine is turned on, and the CRS is setup with default settings (e.g., in anomaly detection mode with default thresholds).

See Table 5.2 for the results.

	ZAP				Wapiti			
Application	SQLi	XSS	RCE	FI	SQLi	XSS	RCE	FI
DVWA	1	0	1	1	2	0	1	1
WackoPicko	1	0	0	2	1	0	1	2

Table 5.2: Number of vulnerabilities discovered when scanning an application with just the CRS enabled

5.1.3 DAST-driven Virtual Patching

In these tests, the ModSecurity rule engine is turned on, and virtual patches generated from the results of Section 5.1.1 are used to protect the application. Since there are two DAST tools being used, that means there will also be two sets of virtual patches.

ZAP Results

This tests how well the virtual patches generated from ZAP's vulnerability report protect the vulnerable web application. See Table 5.3 for the results.

	ZAP				Wapiti			
Application	SQLi	XSS	RCE	FI	SQLi	XSS	RCE	FI
DVWA	0	0	1	1	2	0	2	1
WackoPicko	0	0	0	2	2	0	2	0

Table 5.3: Number of vulnerabilities discovered when scanning an application that has been virtually patched via a previous run of ZAP

Wapiti Results

This tests how well the virtual patches generated from Wapiti's vulnerability report protect the vulnerable web application. See Table 5.4 for the results.

5.1.4 SAST-driven Virtual Patching

In these tests, a SAST called NAVEX was used to generate virtual patches. In fact, it seems that the answer to RQ2 is the same as to RQ1: if an SAST tool is able to produce a report with the URLs, parameters, and types of the attacks, then it can be used to generate virtual patches. NAVEX seems to be the only tool capable of this.

	ZAP				Wapiti			
Application	SQLi	XSS	RCE	FI	SQLi	XSS	RCE	FI
DVWA	1	0	1	1	2	0	2	1
WackoPicko	0	0	2	2	20	0	2	0

Table 5.4: Number of vulnerabilities discovered when scanning an application that has been virtually patched via a previous run of Wapiti

NAVEX results

This tests how well the virtual patches generated from NAVEX's exploit report protect the vulnerable web application. Unfortunately, NAVEX was unable to find any vulnerabilities for attack types beyond SQLi. However, the results for SQLi are consistent with the virtual patches generated by the DAST tools.

	ZAP				Wapiti			
Application	SQLi	XSS	RCE	FI	SQLi	XSS	RCE	FI
DVWA	0	-	-	-	2	-	-	-
WackoPicko	0	-	-	-	2	-	-	-

Table 5.5: Number of vulnerabilities discovered when scanning an application that has been virtually patched via Navex

5.2 Targeted Fuzzing

This tests how well VPgen rules (generated via OWASP ZAP) reduce false positives. FuzzDB's phpbb.txt dataset is used, which consists of 20 495 passwords. VPgen relies on the CRS rulesets, so adjusting the paranoia level does make a difference.

5.2.1 DVWA Results

This was tested on the login page for the password parameter. Since it is a location without a vulnerability, the false positive rate for VPgen is always 0, which can never be worse than the CRS's false positive rate.

5.2.2 WackoPicko Results

This is tested on the login page for the username parameter. It has at least a SQLi vulnerability, though OWASP ZAP also detected an XSS vulnerability. At the highest paranoia level (PL4), VPgen rules have a false positive rate that is 17% of the CRS's rate, which is much better. Otherwise, it is no worse.

Table 5.6: Number of legitimate requests blocked (false positives) per paranoia level for DVWA

Paranoia	CRS	VPgen
PL1	1	0
PL2	2	0
PL3	6	0
PL4	76	0

Table 5.7: Number of legitimate requests blocked (false positives) per paranoia level for WackoPicko

Paranoia	CRS	VPgen
PL1	1	1
PL2	2	2
PL3	6	6
PL4	76	13

Chapter 6

Discussion

In this Chapter, the thesis will discuss the methodology from Chapter 5 and the results from Chapter 5. First, in Section 6.1, this thesis will be compared to related works from Chapter 3 and others. Next, the implications to academia and industry will be discussed in Sections 6.2 and 6.3. Finally, the limitations of the approach will be discussed in Section 6.4.

6.1 Comparison to Related Works

Appelt et al. [32] generated WAF rules to supplement an existing ruleset, which is somewhat similar to what this this thesis attempts. VPgen creates rules that leverage the CRS for attack detection. However, Appelt et al. only focused on SQLi, whereas this paper selected several different vulnerabilities. Betarte et al. [33] generated virtual patches based on a representation of the vulnerable web application, which is similar to our approach, especially with regards to SAST-generated virtual patches. However, Betarte et al. do not leverage another rule set; they start from scratch. Their results saw much improved performance over CRS in terms of performance and blocking attacks, whereas our generated virtual patches only had a better false positive rate.

Salemi [80] used a type of application security testing tool, called RASP, to automatically generate rules, which is very similar to the approach this paper used. However, they do not utilize an existing ruleset, and instead try to create rules from scratch that either block requests coming from certain IP addresses or use a primitive type of SQLi detection. OWASP's CRS has been widely studied [6] and has an active community that is constantly working to improve its performance [29]. Thus, unless the focus of the work is a novel attack detection technique, it makes the most sense to leverage a stable and effective ruleset like CRS.

In most papers that try to test the effectiveness of a given WAF or rule set, the curated lists of payloads are sent to the WAF with little regard to no regard for web application under protection. For example, the closest that the WAF world has to a standard test set of data is HTTP CSIC 2010 [106]. It was developed specifically for testing WAFs, and yet it is not useful for testing the virtual patches

generated by VPgen because the paths and parameters in HTTP CSIC 2010 need to correspond to vulnerable places in the web application that's under protection. Therefore, coming up with a sensible and comprehensive way to test and evaluate VPgen and the virtual patches it creates was a major struggle.

Other papers have used custom approaches, such as selecting arbitrary specific services of the vulnerable web application to test, and using test suites to generate data [32]. Betarte et al. [33] used OWASP ZAP and simply counted the number of vulnerabilities that were found with and without the virtual patches. This was influential to this paper's approach. However, they did not look for or take into account false positives.

6.2 Implications to academia

This was a relatively practical effort that involved the novel integration of existing tools in order to virtually patch web applications in a novel way. This is something that has been mentioned in technical blog [78, 79, 89] posts, but nothing similar has been evaluated in academia until now. There is one particular topic that was explored that could have a more pronounced effect on academia: WAFs powered by string constraint or SMT solvers. However, this was not successfully implemented due to the complexities with running NAVEX as well as time constraints.

6.2.1 String Constraint Solvers in WAFs

The main application domain for string constraint solvers is software verification and testing, especially with the automated detection of security vulnerabilities [107]. There are many static analysis tools that make use of string constraint and SMT solvers across a variety of programming languages such as Java, PHP, and JavaScript [107]. However, no one appears to have employed these in conjunction with a WAF. In this thesis we get close. The solver is used by NAVEX to determine whether an attack is feasible on a supposedly vulnerable code path.

6.2.2 NAVEX

Considerable time and effort was put into making NAVEX function. In addition to having some very old dependencies (such as Gremlin 2.5 and Neo4j 2.1.8), critical pieces of functionality were missing, such as the taint analysis, formula generation, inclusion map generation, and database sanitizations. The researchers that designed and developed NAVEX were contacted, but they were unable to provide assistance. This has been noted in the literature as well [108]. Although we were able to get NAVEX to run from start to finish, we were never able to exactly replicate NAVEX's results from its paper. Part of this is due to not implementing the database sanitizations that NAVEX uses to reduce false positives. Although mentioned as a central feature of NAVEX [75] and its predecessor Chainsaw [74], it was not described in enough detail for us to replicate.

Significant portions of NAVEX's code was re-written, modified, and even improved for this thesis. Support for PHP string interpolation was added, increasing the number of modeled PHP built-in functions from 35 [75] to 36.

6.3 Implications to Industry

Since similar types of virtual patching approaches have been mentioned on companies' technical blogs before [78, 79, 89], it seems likely that this effort will have positive implications to industry. This can reduce the time-to-fix metric [89] down to however long it takes the security testing tool to run, which can be just minutes.

6.3.1 Rule Generation Based on Commonly Used Tools

OWASP ZAP is a very popular open-source vulnerability scanner, as is ModSecurity as a firewall and the CRS as a standard ruleset. Thus, VPgen is a fairly accessible tool for those organizations who need to get up and running quickly with a virtual patch for some recently acquired legacy code that has glaring security issues.

6.4 Limitations

In this section, various limitations to the approach will be discussed.

6.4.1 Negated SecRule Targets are Hardcoded

The list of negated targets in the first `SecRuleUpdateTargetByTag` directive was hard-coded. This is heavy-handed, since not every target that was negated appeared in every rule. In addition, the `ARGS` target was added to some rules that did not previously look at any arguments.

6.4.2 Application Security Testing Tools Miss Vulnerabilities

Neither DAST tools nor SAST can always find 100% of the security vulnerabilities within an application. Without additional (usually manual) testing, it is possible that the patches generated do not cover certain vulnerabilities. This is where the approaches described in [89] and [78] may be superior, since all CRS rules still apply.

Chapter 7

Conclusion

The final chapter of this thesis is split between the Conclusion in Section 7.1 and the discussion of Future Work in Section 7.2.

7.1 Conclusion

This master's thesis aimed to determine whether application security testing tools such as Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) tools could be used to automate the creation of virtual patches for a vulnerable web application. Virtual patch creation is often a highly technical and manual process that could lead to serious security issues if performed improperly. To facilitate automation, a tool called VPgen was developed that can transform the output from several different application security testing tools into rules and directives that can be understood by a web application firewall. The results are promising, but not all is clear. Using DAST tools such as ZAP and Wapiti can reduce the number of false positives when the ruleset being leveraged is very strict. When the rules are not as strict, the false positive benefit is negligible, and the organization risks an attack slipping past.

Conclusions are tough to draw for SAST tools and virtual patching. The tool used in this paper, NAVEX, was only effective in detecting SQLi vulnerabilities, and its associated virtual patches were not any better than the DAST tools' results.

7.2 Future Work

There are several different directions for future research in automated virtual patching. VPsec could be transformed from a simple Python script into a full-fledged virtual patching framework that processes many different types of vulnerabilty reports and leverage several different types of rulesets. Ideally it would be extensible so that any developer or security tester could easily add a report type or ruleset of their choice. In addition, a tool like msc_pyparser could be used to more carefully leverage a ModSecurity rule language rule set instead of hard coding a

number of targets. It could also prevent VPsec from configuring rule set rules that should never have a vulnerable parameter as their target. As opposed to relying on the output from other tools, perhaps a static analyzer could be built into a WAF itself. [79] gives some insight into this. One idea is to leverage a string constraint solver, like Z3-str2 used by NAVEX [75], from the rules of the WAF. ModSecurity has the ability to call out to Lua scripts, which in turn can make system calls and run other types of scripts.

Bibliography

[1] K. Parker, J. Horowitz, and R. Minkin. (2020). "How the coronavirus outbreak has – and hasn't – changed the way americans work," [Online]. Available: https://www.pewresearch.org/social-trends/2020/12/09/how-the-coronavirus-outbreak-has-and-hasnt-changed-the-way-americans-work/ (visited on 07/03/2021).

[2] Synergy Research Group. (2020). "Covid-19 boosts cloud service spending by $1.5 billion in the third quarter," [Online]. Available: https://www.srgresearch.com/articles/covid-19-boosts-cloud-service-spending-15-billion-third-quarter (visited on 07/02/2021).

[3] Palo Alto Networks. (2021). "Cloud threat report 1H 2021," [Online]. Available: https://federalnewsnetwork.com/wp-content/uploads/2021/05/1H21-CTR-full.pdf (visited on 06/29/2021).

[4] J. Graham-Cumming. (2019). "Details of the cloudflare outage on july 2, 2019," [Online]. Available: https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/ (visited on 07/02/2021).

[5] Palo Alto Networks. (2019). "Cloudy with a chance of entropy," [Online]. Available: https://unit42.paloaltonetworks.com/cloudy-with-a-chance-of-entropy/ (visited on 07/04/2021).

[6] J. J. Singh, H. Samuel, and P. Zavarsky, "Impact of paranoia levels on the effectiveness of the ModSecurity web application firewall," in *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, Apr. 2018, pp. 141–144. DOI: 10.1109/ICDIS.2018.00030.

[7] J. Oberoi, H. Samuel, and P. Zavarsky, "How much web security is just enough? analysis of granulated web application firewall rules on web server performance," Aug. 2018.

[8] OWASP. (2017). "The ten most critical web application security risks (2017)," [Online]. Available: https://www.%20owasp.%20org/index.%20php/Top%5C_10-2017%5C_Top%5C_10 (visited on 04/19/2021).

[9] The MITRE Corporation. (2021). "About CWE," [Online]. Available: https://cwe.mitre.org/about/index.html (visited on 06/27/2021).

[10] The MITRE Corporation. (2021). "CWE list version 4.4," [Online]. Available: https://web.archive.org/web/20210619113352/https://cwe.mitre.org/data/index.html (visited on 06/27/2021).

[11] The MITRE Corporation. (2020). "2020 cwe top 25 most dangerous software weaknesses," [Online]. Available: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html (visited on 07/06/2021).

[12] The MITRE Corporation. (2020). "About CWE - frequently asked questions," [Online]. Available: https://cwe.mitre.org/about/faq.html (visited on 07/07/2021).

[13] The MITRE Corporation. (2021). "Cwe view: Weaknesses in owasp top ten (2017)," [Online]. Available: https://cwe.mitre.org/data/definitions/1026.html (visited on 07/07/2021).

[14] PortSwigger. (2021). "SQL injection," [Online]. Available: https://portswigger.net/web-security/sql-injection (visited on 06/15/2021).

[15] OWASP and kingthorin. (2021). "SQL injection," [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection (visited on 07/09/2021).

[16] OWASP. (2021). "SQL injection prevention cheat sheet," [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html (visited on 06/17/2021).

[17] OWASP and KirstenS. (2021). "Cross site scripting (XSS)," [Online]. Available: https://owasp.org/www-community/attacks/xss/ (visited on 07/09/2021).

[18] OWASP. (2021). "Cross site scripting prevention cheat sheet," [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html (visited on 07/08/2021).

[19] OWASP. (2020). "Path traversal," [Online]. Available: https://owasp.org/www-community/attacks/Path_Traversal (visited on 07/09/2021).

[20] OWASP. (2020). "Testing for remote file inclusion," [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/11.2-Testing_for_Remote_File_Inclusion (visited on 07/02/2021).

[21] OWASP, W. Zhong, and Rezos. (2021). "Code injection," [Online]. Available: https://owasp.org/www-community/attacks/Code_Injection (visited on 07/09/2021).

[22] OWASP and W. Zhong. (2021). "Command injection," [Online]. Available: https://owasp.org/www-community/attacks/Command_Injection (visited on 07/08/2021).

[23] OWASP. (2020). "Web application firewall," [Online]. Available: https://owasp.org/www-community/Web_Application_Firewall (visited on 07/09/2021).

[24] B. Lutkevich. (2019). "Web application firewall (WAF)," [Online]. Available: https://searchsecurity.techtarget.com/definition/Web-application-firewall-WAF (visited on 07/08/2021).

[25] Cloudflare. (2021). "What is a WAF? | web application firewall explained," [Online]. Available: https://www.cloudflare.com/learning/ddos/glossary/web-application-firewall-waf/ (visited on 07/05/2021).

[26] Positive Technologies. (2019). "What is a web application firewall?" [Online]. Available: https://www.ptsecurity.com/ww-en/analytics/knowledge-base/waf-web-application-firewall/ (visited on 06/20/2021).

[27] I. Ristić and C. Folini, *ModSecurity Handbook: The Complete Guide to the Popular Open Source Web Application Firewall*, 2nd ed. Feisty Duck, 2017, ISBN: 978-1907117077.

[28] J. D'Hoinne, A. Hils, R. Kaur, and J. Watts, "Magic quadrant for web application firewalls," *Gartner Research*, 2020.

[29] OWASP. (). "OWASP modsecurity core rule set," [Online]. Available: https://coreruleset.org/ (visited on 06/01/2021).

[30] victorhora and zimmerletw. (2021). "Mailing list: Mod-security-users," [Online]. Available: https://sourceforge.net/projects/mod-security/lists/mod-security-users (visited on 07/09/2021).

[31] SpiderLabs. (2021). "Modsecurity," [Online]. Available: https://github.com/SpiderLabs/ModSecurity (visited on 07/01/2021).

[32] D. Appelt, A. Panichella, and L. Briand, "Automatically repairing web application firewalls based on successful sql injection attacks," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2017, pp. 339–350.

[33] G. Betarte, R. D. L. Fuente, R. Martínez, J. Pírez, and F. Zipitría, "Towards model-driven virtual patching for web applications," in *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, Oct. 2016, pp. 109–118. DOI: 10.1109/LADC.2016.24.

[34] G. Betarte, Á. Pardo, and R. Martınez, "Web application attacks detection using machine learning techniques," in *2018 17th ieee International Conference on Machine Learning and Applications (icmla)*, IEEE, 2018, pp. 1065–1072.

[35] S. Prandl, M. Lazarescu, and D.-S. Pham, "A study of web application firewall solutions," in *International Conference on Information Systems Security*, Springer, 2015, pp. 501–510.

[36] AQTRONiX. (2019). "AQTRONiX WebKnight - open source web application firewall (WAF) for IIS," [Online]. Available: http://www.aqtronix.com/?PageID=99 (visited on 07/08/2021).

[37] N. Agarwal and S. Z. Hussain, "A closer look at intrusion detection system for web applications," *Security and Communication Networks*, vol. 2018, 2018.

[38] A. Salih. (2020). "Guardian web application firewall," [Online]. Available: https://github.com/asalih/guardian (visited on 07/09/2021).

[39] H. Buchwald. (2021). "Shadow daemon," [Online]. Available: https://shadowd.zecure.org/overview/introduction/ (visited on 07/09/2021).

[40] N. System. (2021). "NAXSI," [Online]. Available: https://github.com/nbs-system/naxsi (visited on 07/09/2021).

[41] B. Garn, D. S. Lang, M. Leithner, D. R. Kuhn, R. Kacker, and D. E. Simos, "Combinatorially xssing web application firewalls," in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2021, pp. 85–94.

[42] p0pr0ck5. (2018). "Lua-resty-waf," [Online]. Available: https://github.com/p0pr0ck5/lua-resty-waf (visited on 07/09/2021).

[43] SpiderLabs. (2020). "Reference manual (v3.x)," [Online]. Available: https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual-%5C%28v3.x%5C%29 (visited on 07/01/2021).

[44] The MITRE Corporation. (2019). "CVE-2019-19886," [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-19886 (visited on 07/02/2021).

[45] C. Folini. (2021). "Disabling request body access in modsecurity 3 leads to complete bypass," [Online]. Available: https://coreruleset.org/20210302/disabling-request-body-access-in-modsecurity-3-leads-to-complete-bypass/ (visited on 07/07/2021).

[46] SpiderLabs. (2020). "Reference manual (v2.x)," [Online]. Available: https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual-(v2.x) (visited on 07/01/2021).

[47] C. Sanders. (2017). "Is modsecurity's secrules turing complete?" [Online]. Available: https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/is-modsecuritys-secrules-turing-complete/ (visited on 07/05/2021).

[48] R. Barnett, D. Cornell, A. Hoffman, and M. Knobloch. (). "Virtual patching best practices | OWASP," [Online]. Available: https://owasp.org/www-community/Virtual_Patching_Best_Practices (visited on 02/02/2021).

[49] OWASP. (). "Virtual patching - OWASP cheat sheet series," [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Virtual_Patching_Cheat_Sheet.html#example-public-vulnerability (visited on 02/02/2021).

[50] W. C. Vink. (2018). "Methods for the controlled deployment and operation of a virtual patching program," [Online]. Available: https://www.giac.org/paper/gcia/12447/methods-controlled-deployment-operation-virtual-patching-program/143808 (visited on 07/09/2021).

[51] OWASP, J. Manico, and R. R. Hansen. (2021). "XSS filter evasion cheat sheet," [Online]. Available: https://owasp.org/www-community/xss-filter-evasion-cheatsheet (visited on 07/09/2021).

[52] OWASP and D. Mishra. (2020). "SQL injection bypassing WAF," [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection_Bypassing_WAF (visited on 07/08/2021).

[53] 0xInfection. (2021). "Awesome-WAF," [Online]. Available: https://github.com/0xInfection/Awesome-WAF (visited on 07/09/2021).

[54] M. Aldoub. (2020). "Here's a command injection waf bypass that works," [Online]. Available: https://twitter.com/Voulnet/status/1229031352596672513 (visited on 06/23/2021).

[55] A. Khan. (2016). "Web application firewall, filter and bypass!" [Online]. Available: https://owasp.org/www-pdf-archive/OWASP-NL%5C_2016-04-21-Web%5C_Application%5C_Firewall,%5C_Filter%5C_and%5C_Bypass.pdf (visited on 11/13/2020).

[56] S. Chaliasos, G. Metaxopoulos, G. Argyros, and D. Mitropoulos, "Mime artist: Bypassing whitelisting for the web with javascript mimicry attacks," in *European Symposium on Research in Computer Security*, Springer, 2019, pp. 565–585.

[57] R. Lemos. (2021). "SAST, DAST, IAST, and RASP: Pros, cons and how to choose," [Online]. Available: https://techbeacon.com/sast-dast-iast-rasp-pros-cons-how-choose (visited on 07/09/2021).

[58] A. Gosain and G. Sharma, "Static analysis: A survey of techniques and tools," in *Intelligent Computing and Applications*, Springer, 2015, pp. 581–591.

[59] B. Chess and G. McGraw, "Static analysis for security," *IEEE security & privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[60] OWASP. (2020). "Source code analysis tools," [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools (visited on 07/09/2021).

[61] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, IEEE, 2014, pp. 590–604.

[62] M. Velez, *Foundations of software engineering: Taint analysis*, Oct. 2018. [Online]. Available: https://www.cs.cmu.edu/~ckaestne/15313/2018/20181023-taint-analysis.pdf.

[63] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1083–1094.

[64] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of model checking*, Springer, 2018, pp. 305–343.

[65] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[66] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang, "Z3str2: An efficient solver for strings, regular expressions, and length constraints," *Formal Methods in System Design*, vol. 50, no. 2-3, pp. 249–288, 2017.

[67] techopedia. (21). "Dynamic application security testing (DAST," [Online]. Available: `https://www.techopedia.com/definition/30958/dynamic-application-security-testing-dast` (visited on 07/09/2021).

[68] OWASP. (2020). "Vulnerability scanning tools," [Online]. Available: `https://owasp.org/www-community/Vulnerability_Scanning_Tools` (visited on 07/09/2021).

[69] S. Chen. (2017). "Evaluation of web application vulnerability scanners in modern pentest/ssdlc usage scenarios," [Online]. Available: `https://sectooladdict.blogspot.com/` (visited on 07/05/2021).

[70] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[71] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[72] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, IEEE, 2008, pp. 143–157.

[73] S. Huang, H. Lu, W. Leong, and H. Liu, "CRAXweb: Automatic web application testing and attack generation," in *2013 IEEE 7th International Conference on Software Security and Reliability*, Jun. 2013, pp. 208–217. DOI: `10.1109/SERE.2013.26`.

[74] A. Alhuzali, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Chainsaw: Chained automated workflow-based exploit generation," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, New York, NY, USA: Association for Computing Machinery, Oct. 24, 2016, pp. 641–652, ISBN: 978-1-4503-4139-4. DOI: `10.1145/2976749.2978380`. [Online]. Available: `https://doi.org/10.1145/2976749.2978380` (visited on 03/10/2021).

[75] A. Alhuzali, R. Gjomemo, B. Eshete, and V. N. Venkatakrishnan, "NAVEX: Precise and scalable exploit generation for dynamic web applications," p. 17, 2018.

[76] A. Alhuzali. (2019). "Navex," [Online]. Available: `https://github.com/aalhuz/navex` (visited on 07/09/2021).

[77] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, pp. 101–121, 2019.

[78] R. Barnett. (2012). "Modsecurity advanced topic of the week: Automated virtual patching using OWASP zed attack proxy," [Online]. Available: `https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/modsecurity-advanced-topic-of-the-week-automated-virtual-patching-using-owasp-zed-attack-proxy/` (visited on 07/09/2021).

[79] A. P. Research. (). "Do WAFs dream of static analyzers?" [Online]. Available: `http://blog.ptsecurity.com/2017/10/do-wafs-dream-of-static-analyzers.html` (visited on 02/02/2021).

[80] M. Salemi, "Automated rules generation into web application firewall using runtime application self-protection," 2020.

[81] R. Barnett, "WAF virtual patching challenge - securing WebGoat with ModSecurity," p. 26, 2009.

[82] 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.

[83] D. Appelt, C. D. Nguyen, A. Panichella, and L. C. Briand, "A machine-learning-driven evolutionary approach for testing web application firewalls," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 733–757, 2018.

[84] 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.

[85] C. Liu, J. Yang, and J. Wu, "Web intrusion detection system combined with feature analysis and svm optimization," *EURASIP Journal on Wireless Communications and Networking*, vol. 2020, no. 1, p. 33, 2020.

[86] D. Kar, S. Panigrahi, and S. Sundararajan, "Sqligot: Detecting sql injection attacks using graph of tokens and svm," *Computers & Security*, vol. 60, pp. 206–225, 2016.

[87] G. Betarte, E. Giménez, R. Martınez, and Á. Pardo, "Machine learning-assisted virtual patching of web applications," *arXiv preprint arXiv:1803.05529*, 2018.

[88] M. Tanrıverdi and A. Tekerek, "Implementation of blockchain based distributed web attack detection application," in *2019 1st International Informatics and Software Engineering Conference (UBMYK)*, IEEE, 2019, pp. 1–6.

[89] R. Barnett. (2012). "Dynamic DAST/WAF integration: Realtime virtual patching.," [Online]. Available: https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/dynamic-dastwaf-integration-realtime-virtual-patching/ (visited on 07/09/2021).

[90] R. Barnett. (2012). "OWASP virtual patching survey results," [Online]. Available: http://blog.spiderlabs.com/2012/03/owasp-virtual-patching-survey-results.html (visited on 09/27/2020).

[91] A. Marchand-Melsom and D. B. Nguyen Mai, "Automatic repair of owasp top 10 security vulnerabilities: A survey," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 23–30.

[92] V. Clincy and H. Shahriar, "Web application firewall: Network security models and configuration," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, vol. 1, 2018, pp. 835–836.

[93] B. Krebs. (2019). "What we can learn from the capital one hack," [Online]. Available: https://krebsonsecurity.com/2019/08/what-we-can-learn-from-the-capital-one-hack/ (visited on 07/09/2021).

[94] B. J. Oates, *Researching information systems and computing*. Sage, 2005.

[95] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2010, pp. 111–131.

[96] PortSwigger. (2021). "Burp suite," [Online]. Available: https://portswigger.net/burp (visited on 07/09/2021).

[97] OWASP. (2021). "OWASP zed attack proxy (ZAP)," [Online]. Available: https://www.zaproxy.org/ (visited on 07/09/2021).

[98] Sarosys LLC. (2021). "Arachni - web application security scanner network," [Online]. Available: https://www.arachni-scanner.com/ (visited on 07/09/2021).

[99] O. Security. (2020). "Kali linux package tracker - arachni," [Online]. Available: https://pkg.kali.org/pkg/arachni (visited on 07/09/2021).

[100] O. Security. (2021). "Kali linux tools listing," [Online]. Available: https://tools.kali.org/tools-listing (visited on 07/09/2021).

[101] OWASP. (2021). "OWASP vulnerable web applications directory," [Online]. Available: https://owasp.org/www-project-vulnerable-web-applications-directory/ (visited on 07/09/2021).

[102] M. Parvez, P. Zavarsky, and N. Khoury, "Analysis of effectiveness of black-box web application scanners in detection of stored sql injection and stored xss vulnerabilities," in *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, IEEE, 2015, pp. 186–191.

[103] Y. Makino and V. Klyuev, "Evaluation of web vulnerability scanners," in *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, IEEE, vol. 1, 2015, pp. 399–402.

[104] M. Bose. (2019). "Virtualbox network settings: Complete guide," [Online]. Available: `https://www.nakivo.com/blog/virtualbox-network-setting-guide/` (visited on 07/09/2021).

[105] F. Project. (2020). "FuzzDB," [Online]. Available: `https://github.com/fuzzdb-project/fuzzdb` (visited on 07/09/2021).

[106] C. T. Giménez, A. P. Villegas, and G. Á. Marañón, "Http data set csic 2010," *Information Security Institute of CSIC (Spanish Research National Council)*, 2010.

[107] R. Amadini, "A survey on string constraint solving," *arXiv preprint arXiv:2002.02376*, 2020.

[108] P. Li, W. Meng, K. Lu, and C. Luo, "On the feasibility of automated built-in function modeling for php symbolic execution," in *Proceedings of the Web Conference 2021*, 2021, pp. 58–69.