Attacking Computer Security

from the Perspective of Educators, Users, and Analysts

by

Erik Trickel

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2023 by the
Graduate Supervisory Committee:

Adam Doupé, Co-Chair
Yan Shoshitaishvili, Co-Chair
Tiffany Bao
Ruoyu Wang

ARIZONA STATE UNIVERSITY

May 2023

ABSTRACT

As computers and the Internet have become integral to daily life, the potential gains from exploiting these resources have increased significantly. The global landscape is now rife with highly skilled wrongdoers seeking to steal from and disrupt society. In order to safeguard society and its infrastructure, a comprehensive approach to research is essential.

This work aims to enhance security from three unique viewpoints by expanding the resources available to educators, users, and analysts. For educators, a capture the flag as-a-service was developed to support cybersecurity education. This service minimizes the skill and time needed to establish the infrastructure for hands-on hacking experiences for cybersecurity students.

For users, a tool called CloakX was created to improve online anonymity. CloakX prevents the identification of browser extensions by employing both static and dynamic rewriting techniques, thwarting contemporary methods of detecting installed extensions and thus protecting user identity.

Lastly, for cybersecurity analysts, a tool named Witcher was developed to automate the process of crawling and exercising web applications while identifying web injection vulnerabilities. Overall, these contributions serve to strengthen security education, bolster privacy protection for users, and facilitate vulnerability discovery for cybersecurity analysts.

DEDICATION

I dedicate this dissertation to Robin Trickel, my best friend and wife, with gratitude and love. Without your unwavering support, encouragement, and strength, this dissertation would not have been possible. Your patience, understanding, and sacrifices throughout this journey have been my driving force, and I cannot express how grateful I am to have you by my side.

Thank you for being my best friend and for always being there for me.

I love you more than words can express.

ACKNOWLEDGMENTS

Completing this dissertation has been an enriching and challenging experience, and I am humbled and grateful for the support and guidance of so many individuals.

To start, none of this would have been possible without Adam Doupé and Yan Shoshitaishvili. Their invaluable guidance has been instrumental in shaping my research and my life over the past seven years. I would also like to thank the other members of my dissertation committee Fish Wang and Tiffany Bao for their contributions to this work and for being an essential part of my journey.

I am grateful to my colleagues and peers who offered their friendship, support, and expertise, which has provided me an amazing place to learn and grow.

To my family and friends, thank you for your encouragement and support throughout this journey. Your belief in me has been my driving force and I am thankful for your presence in my life.

Finally, I would like to express my gratitude to Robin Trickel for supporting and encouraging me through this process. Your strength and understanding have been invaluable. Without you, I never could have reached this milestone.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Over the last forty years, the world of hacking has morphed from an idealistic place populated by young and curious explorers who only wanted to hack the planet and keep the world of electrons free for all[1] into a world-wide battlefield in which nation-states and criminal organizations launch attacks to access and control the flow of information. As a result, modern attackers have gone from a chaotic group seeking to explore their digital world to highly sophisticated and coordinated actors that patiently wait for the optimal moment to engage their enemies. Modern hackers leverage vulnerabilities on one system to compromise another, and then, as stealthily as possible, they slowly exfiltrate the data they seek and eliminate any evidence of their activity. This metamorphosis of the hacking world demands an equal rise in the skills and tools available to educators, users, and analysts. To meet these threats, I assert that we can simplify the incorporation of problem-based learning and gamification into an educator's cybersecurity courses, we can cloak browser extensions from website detection without modifying the web browser, and we can use randomness to improve the automated exploration and vulnerability detection in web applications.

[1]See Loyd Blankenship's essay *The Conscience of a Hacker*, aka *The Hacker's Manifesto*, which he said he wrote to describe the essence of "what [hackers] were doing and why we were doing it" [27], [135].

1.1 Educators

Society faces a computer security crisis because the demand for security analysts is outpacing the creation of qualified professionals [2]. Despite the global computer security workforce growing to 4.7 million this year, an additional 3.4 million computer security workers are still required worldwide [80]. The shortage of qualified cybersecurity professionals in the workforce contributes to the occurrence of high-profile security incidents, such as T-Mobile's data breach, where hackers accessed the personal data of 37 million current customers [60]. In addition, attacks against the nation's critical infrastructure could have devastating effects that go well beyond the financial losses we are witnessing today [81].

Teaching computer security is a challenging problem because the knowledge and skills are a complex and constantly moving target. Solving security problems requires strong objective critical thinking skills [39], [102], [146]. In other words, analysts must learn to think like the attackers and then learn to defend against those attacks and exploits. Although listening to lectures and studying vulnerabilities provides a theoretical start, it is not enough—hands-on practice is crucial for mastering the highly-complex theoretical concepts involved in cybersecurity [143].

Incorporating live cybersecurity competitions (CTFs) in the curricula of cybersecurity courses improves the student's learning outcomes. First, students participating in CTFs receive hands-on experience applying security principles in an active manner [120]. Moreover, CTFs, played in teams, offer the highest level of student engagement because it facilitates an interactive dialog between the participants where they synthesize the security concepts [32]. As a result, CTFs offer a much higher rate of learning than simply reading materials or listening to a lecture, which are passive forms of

engagement. Second, as a form of problem based learning, CTFs offer a more effective method of building critical thinking skills as opposed to a lecture based format [138]. Third, using CTFs gamifies the learning experience. In contrast to the lecture format, gamification has been shown to increase student performance and participation in voluntary activities and attempting challenging assignments [71], [79]. Thus, it is not surprising that live cybersecurity competitions, which take advantage of those ideas, are on the rise.

Unfortunately, not all educators have the time or skills necessary to run a CTF competition. To address this need, I describe a CTF-as-a-service solution in Chapter 2 that quickly creates an infrastructure for running custom CTF events. The system is capable of scaling from a few competitors to 100s of teams.

1.2 Users

As the web expands and continues being the platform of choice for delivering applications to users, the browser has become a core component of a user's interactions with the web. Modern browsers advertise a wide range of features, from cloud-syncing and notifications to password management and peer-to-peer video and audio communications. An important feature of modern browsers is their ability to be extended by users, as they see fit, by installing *browser extensions*. Namely, Google Chrome and Mozilla Firefox, the browsers with the largest market share, offer dedicated browser extension stores that house hundreds of thousands of extensions [64]. In turn, these extensions advertise a wide range of additional features, such as enabling the browser to store passwords with online password managers, blocking ads, and saving articles for later reading.

From a security perspective, the ability to load third-party code into the browser comes at a cost, even though extensions rely on web technologies such as HTML, JavaScript, and CSS. Browsers afford extensions significantly more privileges than they do to a webpage. For example, the same origin policy restricts webpages from accessing content, such as a cookie, that does not originate from the same domain. For a webpage to bypass this restriction, it must implement cross-origin resource sharing, whereas extensions may not only access resources of any domain but may also alter the content. Historically, malicious extensions abuse these privileges to perform advertising fraud and to steal private and financial user data [83], [92], [137], [147].

Next to security issues, using browser extensions can also lead to the loss of privacy. Given that users choose the extensions to install, it is possible to make inferences about a user's thoughts and beliefs *based solely on the extensions she keeps.* For example, the detection of a coupon-finding extension [61] reveals information about the user's income-level. Additionally, an extension that hides articles about certain political figures [62], [63] reveals the user's political leanings. Lastly, the use of browser extensions may provide a means for websites to persistently identify a user over the course of distinct browser sessions.

Although browser vendors do not offer any programmatic methods for a webpage's JavaScript to detect the extensions currently installed in a user's browser, researchers recently discovered side-channel techniques for fingerprinting many extensions. Sjösten et al. were the first to demonstrate a new method for detecting browser extensions that exploited the public nature of *web-accessible resources* (WARs) [128]. A WAR is any resource (e.g., JavaScript or image) within an extension that the extension identifies as externally accessible. As a result, a webpage can determine whether a visitor uses an extension by requesting one of the exposed WARs. Sjösten et al.

showed that more than 50% of the top 1,000 browser extensions use WARs, which any webpage might use to detect extensions. Later, Starov and Nikiforakis demonstrated another technique for fingerprinting extensions that uses an extension's modifications to the document-object-model (DOM) to detect their presence [131]. The authors developed XHound, a system that automatically discovers the DOM side-effects of extensions. Through their experiments, they showed that more than 10% of the top 50K extensions were fingerprintable.

Users need a tool that will protect their anonymity from websites even though they have fingerprintable extensions installed. The Internet needs a system will put privacy protection back into the hands of the users, so that, if they desire, then they have tools available to enhance their privacy.

In Chapter 3, I introduce CloakX as a means for users to safeguard their privacy by countering the current state-of-the-art in extension fingerprinting. CloakX modifies, randomizes, and supplements the fingerprintable attributes of extensions without requiring any alterations to the web browser or any intervention from the extension's creator.

1.3 Developers

Web application vulnerabilities are showing no signs of waning as the number of web application keeps increasing and the supported frameworks keep diversifying. These web vulnerabilities, such as SQL injections, can be catastrophic to the developers of the web application, the companies running the web application, and the end-users who visit and store their data on the website application.

Although improving the education of developers has helped curb the number

of web application vulnerabilities, historically, developers—and web developers in particular—ignore even well-known security concerns. For example, even after nearly 20 years, SQL injection vulnerabilities are still the 3rd most common web vulnerability with 1162 SQL injection vulnerabilities found in 2022 [17], [133]. Therefore, it is unlikely that developer education alone will eliminate web vulnerabilities.

Due to the number and diversity of web applications, it is critical to create automatic techniques that discover web vulnerabilities. Prior work has proposed different crawling and detection techniques, which utilizes one of the following approaches: white-box [52], [77], [91], [98], black-box [3], [78], [110], [119], and grey-box [56], [124]. However, these approaches are limited in their applicability to web application language, vulnerability type, or application inputs.

In Chapter 4, I introduce Witcher, an innovative framework for discovering web vulnerabilities that takes inspiration from grey-box coverage-guided fuzzing. Witcher's approach involves exploring the input space of web applications using execution coverage data to direct the creation of random inputs, rather than relying solely on fixed heuristics. This allows for efficient and effective vulnerability detection that exceeds the current state-of-the-art in web vulnerability scanning.

Chapter 2

CTFS AS A SERVICE

Abstract

Although we are facing a shortage of cybersecurity professionals, the shortage can be reduced by using technology to empower all security educators to efficiently and effectively educate the professionals of tomorrow. One powerful tool in some educators' toolboxes are Capture the Flag (CTF) competitions. Although participants in all the different types of CTF competitions learn and grow their security skills, Attack/Defense CTF competitions offer a more engaging and interactive environment where participants learn both offensive and defensive skills, and, as a result, they develop their skills even faster. However, the substantial time and skills required to host a CTF, especially an Attack/Defense CTF, is a huge barrier for anyone wanting to organize one. Therefore, we created an on-demand Attack/Defense tool via an easy-to-use website that makes the creation of an Attack/Defense CTF as simple as clicking a few buttons. In this paper, we describe the design and implementation of our system, along with lessons learned from using the system to host a 24-hour 317 team Attack/Defense CTF.

2.1 Introduction

Attack/Defense Capture the Flag events (ADCTFs) are a type of live cybersecurity competition that attempts to maximize the learning for the competitors. In an ADCTF, the participants practice finding vulnerabilities, developing exploits, and

defending against exploits. Additionally, the competitors are often in teams, which further increases their learning [84], [120]. Beyond the learning that takes place during the competition, many competitors also experience significant learning in preparing for the competition and creating write-ups after it concludes [33].

Although organizing and running challenge-based competitions is relatively simple[2], organizing and running an ADCTF competition requires a significant amount of time and a broad range of skills. An ADCTF organizer must spend a large amount of time to meticulously build a secure infrastructure for hosting the game. Moreover, the organizer must develop some type of application that securely controls the game and scores the participants' activities. All of this means that an organizer must be an expert in operating systems, networking, application development, and server administration to successfully organize and host an ADCTF.

To address this pressing need, we relied on our experience gained over the last fourteen years hosting ADCTFs and created a CTF-as-a-Service platform, which is now available at https://ShellWePlayAGame.org (SWPAG) and the source code is available on GitHub [136]. On SWPAG, anyone can organize and host their own ADCTF. After filling out the proper information, a completely configured and stable ADCTF is created in a cloud environment, thus relieving the organizer of the operating system, networking, and server administration burden. Our goal is that SWPAG will empower all security educators, even those with limited network or administrative skills, to easily host their own ADCTF for educational purposes.

[2]In fact, there are only a handful of ADCTFs, while most available competitions are challenge-based [37].

2.2 Background and Motivation

Live cybersecurity exercises benefit the security community in several ways. First, the exercises allow the participants to practice the theory and concepts they have acquired from books and articles [141]. Second, the real-time aspect of a finite event that occurs for a limited amount of time and the competitive-drive of the participants improves learning [72]. Third, live cybersecurity exercises provide a deeper engagement and increase academic learning time, which results in faster learning and mastery of the concepts [55], [57], [139]. Fourth, the participants learn how to operate in a dynamic setting, having to react to attacks by developing, on the spot, defenses and countermeasures. Last, the events allow participants to showcase their skills.

Collectively, we have been organizing, running, and competing in cybersecurity competitions for many years. From this unique vantage point, we have seen first-hand the effect that live cybersecurity competitions have on the participants, who are driven to invest a substantial amount of resources in preparing, executing, and post-evaluating. Preparation includes classroom learning, peer teaching, independent study, and the creation of novel tools. Execution requires them to think critically and generalize their theoretical knowledge while having to react, in real time, to unexpected circumstances. The post-evaluating entails objectively evaluating their performance, discussing the effectiveness of their attack and defense mechanisms, and studying solutions to the problems they could not solve. Many participants learn even more by taking the time to write blog posts that discuss their lessons learned, the details of how they found and eventually exploited the vulnerability, and their strategy and approach to the competition. As a result, these participants have a tendency to grow and improve after every event.

The first cybersecurity competition was held in 1996 at DEF CON [73]. The early DEF CON competitions were in what is now considered a challenge style using a single host with custom-written vulnerable services. The participants would discover vulnerabilities in each service and then prove it by crafting an exploit. Even though this style of event focuses purely on offensive skills, it is still an excellent way for participants to practice and refine their security skills.

The next iteration of CTF competitions allowed participants to refine *both* their offensive and defensive skills: Attack/Defense Capture the Flag events. This type of event is an interactive competition in which each team receives an identical machine that is running vulnerable services. The competitors then use their security skills to protect their own services while simultaneously trying to break into the same services on their opponents' machine. Once successful, the competitors must obtain proof that they succeeded at exploiting an opponent's service by gaining access to a unique piece of data referred to as a *flag*. With this flag in their virtual hand, they must then turn it in to a scorekeeper for points. ADCTFs are a fun and exciting way for security researchers to showcase, enhance, and refine their security skills while also competing with one another for fame and glory.

Since 2003, we have hosted the international Capture the Flag (iCTF) competition, which was not only one of the first ADCTFs but is now one of the largest [143]. We have continued to host the iCTF every year since then (the most recent edition was in March of 2017). Each year, we experiment with various designs and approaches to the game [33], [47], [126], [140], [142].

After running the competition for fourteen years, we recognized that many of the game infrastructure components were reused year after year [143]. Therefore, in August 2014, the UCSB SecLab released an open-source framework for hosting interactive

CTF competitions with the hopes of easing the burden on other ADCTF organizers and to give educators access to an ADCTF competition for their classroom [136]. By abstracting the common infrastructure (starting services, scoring, service checking, VM creation) and by defining a common interface to create services, the authors enabled anyone, with significant manual effort, to create and host an ADCTF-like competition. Even though the iCTF framework provides the components necessary to run an ADCTF event, their setup and configuration is far from trivial. An organizer must still spend a significant amount of time understanding how the components work. After that, she must create the network, take the time to deploy the machines, and create and install vulnerable services. In addition, an organizer must debug any components that fail to work properly, which can involve investigating the database, finding the various log files on each of the machines, and even patching bugs. Thus, the technical barrier to adoption is still substantial because the would-be organizers must understand a great deal about networking, server administration, network security, application development, and application security.

This led us to realize that the community would benefit greatly from a turn-key solution. To validate this conclusion, we surveyed the teams that participated in our competition in 2015. We asked them "If you could press a button on a website to automatically host your own CTF competition, with no technical setup on your part, would you or your group use it?" 31 out of the 36 responders answered that they would.

All of this pushed us toward taking the open source platform to the next level and offer it as an easy-to-use service; thus, now we are proud to present our CTF-as-a-Service solution, which is available at https://ShellWePlayAGame.org (SWPAG). SWPAG offers the capability to launch an ADCTF that leverages the computing

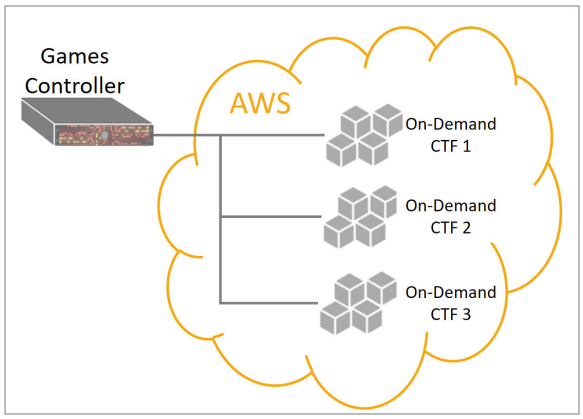

Figure 1. CTF-as-a-Service Overview.

resources of the cloud. After entering some information and clicking a few buttons, an organizer launches a CTF instance, which is created and configured within a few minutes on Amazon's Web Services platform (AWS).

2.3 Design of the CTF-as-a-Service

While the SWPAG website acts as the front-end for users to configure an ADCTF, behind the SWPAG website the CTF-as-a-Service platform has one master controller called the Games Controller (GC), which is responsible for managing all of the CTF instances (see Figure 1). A CTF instance is the logical space containing all the virtual machines (VMs) necessary to run a single ADCTF event. In this section, we describe the GC and the CTF instance components.

2.3.1 The Games Controller

The SWPAG website is the web front-end that organizers use to manage events. An organizer can be anyone—including students, educators, CTF teams, or organizations. Initially, an organizer must create an account. After creating an account, she may create a new CTF instance and modify any of the settings. The organizer may choose to select intentionally-vulnerable services from a library of existing services and eventually may write and upload her own vulnerable services, which will then become a part of the library and available to other educators[3]. In addition, the organizer chooses various parameters for the competition, such as the number of teams, the members of the teams, the game start time and duration, and so on. Next, the organizer provides credentials for a valid account on Amazon Web Services (AWS), which is currently the only cloud service provider that is compatible with the platform. Once setup is complete, the teams wishing to participate in an event can register for it. When the organizer is ready, she clicks a button to launch the CTF instance and the GC takes care of the rest.

The GC is responsible for creating, managing, and terminating all the CTF instances. Using the supplied credentials, the GC accesses the AWS account and creates a Virtual Private Cloud (VPC) for each CTF instance. VPCs are a networking feature of AWS that enable the provisioning of a logically isolated section of AWS's cloud [10]. Within the VPC, the GC has full control of the IP addresses of the servers and the network routing between them. The VPC is configured to keep all network

[3]Much like a riddle, the difficulty of exploiting a vulnerable service is in its novelty, as such, the services stored in the library will be easier because the prior contestants will often post their analysis and solutions. However, even with leaked solutions, it is often still difficult for less experienced students to successfully implement an exploit.

traffic within it, and, as a result, the attacks launched during the competition cannot affect external hosts. After the GC configures the VPC, it creates the Game Master (GM).

2.3.2 The CTF Instance Components

The GM automates many of the difficult and time consuming tasks. It is responsible for communicating with the GC and for orchestrating the creation and management of the game. After the GM's creation is complete, it starts by configuring the CTF instance and then creating the Database, Router, Gamebot, Scriptbot, Team Interface, Scoreboard and the team VMs.

The first component the GM instantiates is the Database. The Database is the central component of the game's operation—stores all the information associated with the competition (e.g., the flags submitted, the status of the services for each round, and the team's information). Being the central component of the game, all the other components access the database, except for the vulnerable team VMs. The components access the Database over a private subnet that is different from the one used by the team VMs. As a result, the Database is inaccessible from the team VMs or from the Internet. We limited access to the Database to reduce the attack surface area of the game infrastructure. We also designed it this way so that the unencrypted database communications were protected because they only travel over the private subnet.

The second component created by the GM is the Router. The main purpose of the Router is to masquerade transmissions to the team VMs, to capture the traffic, and to act as a single entry point for the teams. First, it forwards all team-to-team

transmissions and official service verification transmissions that verify each team's service is running properly. While forwarding these transmissions, it anonymizes the packets by masquerading all the traffic as itself. Thus, when a team receives a packet it has the source IP of the Router. We designed it this way to prevent teams from dropping traffic from their competitors while allowing the service checks to get through. One interesting development note is that this was not as straight forward to setup as we thought because the AWS network prevents masquerading by default [22]. To bypass this restriction, the source/destination checking must be disabled.

The Router captures, stores, and potentially limits all the traffic that it forwards (i.e., team-to-team and Scriptbot-to-team traffic). Even limiting to only team-to-team and Scriptbot-to-team traffic, the logs (in raw pcap format) grow rather quickly. For example, for our most recent competition the compressed traffic logs were more than 100GB. However, this competition was for 24 hours with 317 teams, so we expect SWPAG CTFs to have much less traffic. Next, it limits the number of connections per second each team may initiate to another team; however, it does not limit the maximum number of connections a team may have open concurrently. Once the Router creation completes, the GM instantiates the Gamebot, Scriptbot, Scoreboard, Team Interface, and the Teams' VMs in parallel.

The Router is created with a static external IP, and it serves as a single entry point for teams to access their VMs. It does this by forwarding ports 1337 and higher to each team's SSH port. We chose to design the team's access this way because we found that stopped and started, recreated, or upgraded team VMs would receive a different public IP address [6]. We could have designed it so that every team received a static IP address, however (1) Amazon limits the number that can be used per account and (2) we were unsure of how many needed to be requested. The chosen

port forwarding method allows us to have 1,000 teams without needing to request an increase in the number of static IP addresses.

The Gamebot is the heartbeat of the game. The game duration is divided in ticks. A tick does not occur after a fixed and constant amount of time, instead, it occurs after a fixed amount of time plus a random adjustment. After each tick, the prior round ends and a new round begins. At end of a round, the Gamebot calculates the score for each team based on their performance during the prior round.

The next component is Scriptbot. Prior to a game starting, Scriptbot sits and waits for Gamebot to create the first tick of the game. Once Scriptbot sees the first tick, it will tirelessly test the teams' services and update the flags on each team's VM every round. The test and update processes execute in parallel, but to obfuscate itself and to spread out the load the Scriptbot generates a randomized delay for every process it must execute in a round. The maximum delay is set so that Scriptbot will complete all the processes before the end of the current round. In addition to the randomized delay, Scriptbot accesses the team's services via the Router, which masquerades all the traffic, so that Scriptbot's requests look the same as the team-to-team traffic. After it executes each process, Scriptbot updates the database with the results.

The Team Interface is both the keymaster and gatekeeper. Using the Team Interface, the teams retrieve their private SSH keys so that they can access their team's VMs. The Team Interface also allows them to retrieve a flag identifier for the round. The flag identifier is a value that will help them find the flag on their opponent's machine. For example, it might be name of the file they must look inside once they exploit the associated service. In addition, the Team Interface will provide each team with a unique flag token so that they can submit a flag without needing to use their username and password. Next, the Team Interface accepts any flags

that teams submit. To access the Team Interface, the teams must retrieve a login access token and a flag submission token from SWPAG. For more information on the flag mechanisms, see the article *Ten Years of iCTF: The Good, The Bad, and The Ugly* [143].

The last system component is the Scoreboard. The Scoreboard provides feedback to the teams on their performance. On the leaderboard section of the Scoreboard, it displays each team's score and a graph showing the historical performance of the top teams. On the service list of the Scoreboard, the status of the team services are shown, so that the teams can evaluate if their security mechanisms are affecting the functionality and availability of their services.

Next, GM creates a virtual machine on AWS for each team. Although GM creates the instances in parallel, it limits the number of concurrent requests to avoid receiving a request rate limit exceeded error from AWS [5]. To keep their system available for all their users, AWS has a fluctuating request rate limit and if an account exceeds the limit they receive the error[4].

During the team VM creation process, the GM installs and starts the vulnerable services chosen by the organizer. It also configures the VMs with a static route that forces the team-to-team traffic through the Router. If a team decides to change or remove this static route (which they can do because each team has root access to their own VM), then they will be unable to communicate to the other teams because direct team-to-team traffic is blocked by an AWS security group (see Section 2.3.3 for additional details). Once the GM completes instantiating and configuring a team's VM, GM tests the VM's vulnerable services.

[4]Through trial and error we have found it is unlikely we will receive the error if we limit the number of concurrent requests to ten.

2.3.3 Network Configuration

Each CTF instance must have several network configuration steps completed before a game can start. As mentioned previously, the GC creates a VPC. The new VPC is assigned an IP address range of 172.31.0.0/16. Within the VPC, the GC creates two subnets. The first subnet is the Game Components subnet, which is limited to 172.31.64.0/20. The Game Components subnet contains all the game servers, GM, Database, Gamebot, Scriptbot, Team Interface, and Scoreboard. The second subnet created by GC is called the War Range subnet. The GC defines the War Range subnet as 172.31.128.0/17. The War Range contains the teams' VMs. However, the GM limits team IP addresses to 172.31.129.0/19, which means it can currently only handle 8,190 machines, however the largest ADCTF ever held had only 317 machines it should be sufficient for the foreseeable future.

The Scriptbot is the only machine in the network that is dual homed to both the Game Components and War Range private subnets. The Scriptbot is dual homed so that it can create a static route to the Router, which obfuscates its origin while running its service tests on the team VMs.

For a virtual machine to be accessible, it must be associated with an AWS security group. An AWS security group is a virtual firewall that permits inbound and outbound traffic based on the rules assigned to it [7]. The security groups reside on the network and are inaccessible to the VMs—in fact, unlike a firewall running on a VM, packets not permitted by a security group are dropped before ever reaching the VM.

The GM associates every virtual machine in a CTF instance with one of four AWS security groups. The first group protects the servers that are only internal. This group only permits connections on ports 80 and 22 so long as the connections originate

from the Game Components subnet. Similarly, the web security group has the same restrictions except that it allows Internet traffic to connect using ports 80 and 443. The next security group protects the Router. The first rule permits access to all addresses connecting to ports 1024-2352, which are the ports used for the SSH port forwarding (see the connection from the user to her VM in Figure 2). The Router security group also allows connections to the ports between 1024 and 65535 if the connection originates from the War Range, which is represented by an arc between the teams in Figure 2. The fourth security group is for the teams. The only rule in this group permits access to ports 1024-65535 if the connection originates from the Router, which means the only way to connect to another team is by sending packets through the Router. Referring to Figure 2, notice the arcing connection through the Router and the connection from Scriptbot are permitted on port 20000, whereas, the direct connection from team three is prevented by the AWS security group. This was designed like this because we wanted to give each team root access to their VM, and, as a result, each team's VM must be considered hostile and outside of our control. However, the VM-independent nature of the security groups provides a simple mechanism to achieve the desired effect.

2.3.4 Intelligent Component Recreation

As anyone who has created a machine from scratch knows, it is not uncommon for some small part of the install process to fail, and, unfortunately, this happens when creating VMs in the cloud as well. Thus, GM has a robust and extensible set of tests that it runs to verify the VMs are operating correctly. For the game components, the tests verify that the machine is accessible, the proper ports are open and responding

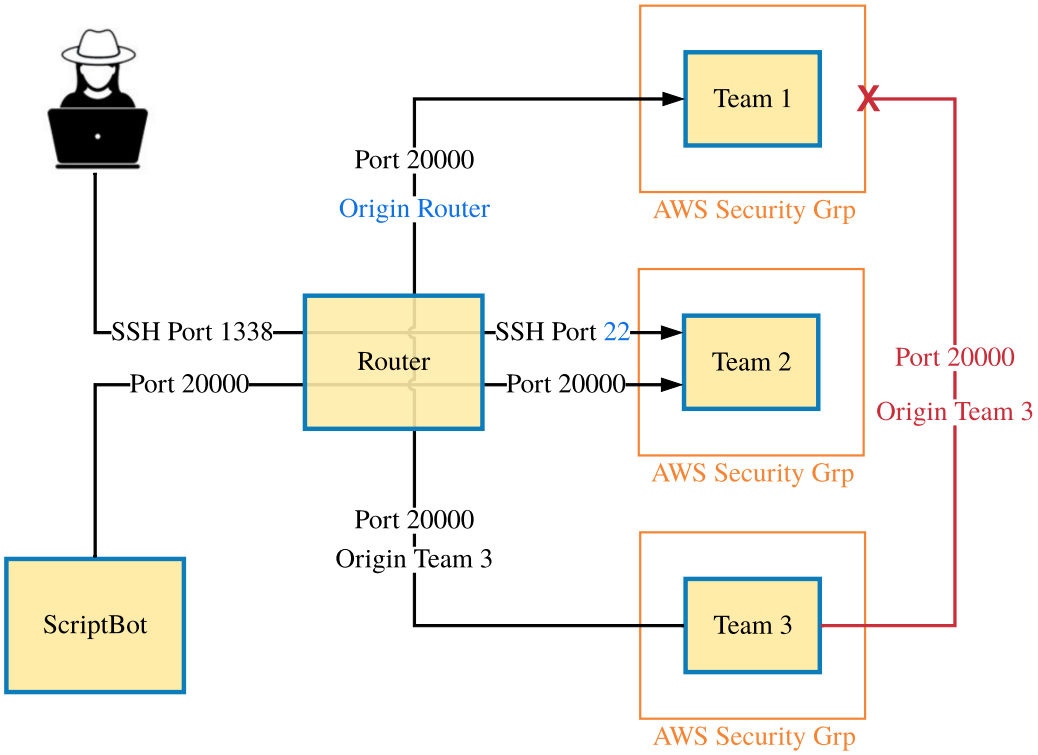

Figure 2. Scriptbot, Router, and Team VM Connectivity.

appropriately, and the internal services are up and running. For the team VMs, it checks each of the vulnerable services by running the same scripts that will be used to verify service operation during the competition. If any of the tests fail for a machine, the GM automatically destroys the machine and recreates a new one from scratch.

2.4 Validation

We expected that the CTF-as-a-Service platform would scale to handle large-scale events, however we would not know for certain unless we tested it. Thus, after

completing the development, we ran a load test with 250 teams to uncover any latent defects and evaluate its ability to manage a large-scale event. After fixing the issues, we torture tested it and the AWS network by hosting the 2017 international Capture the Flag event (iCTF) in which 317 teams competed for fame, glory, and an entry into the 2017 DEF CON CTF.

2.4.1 Load Testing, Round One

To understand the performance characteristics of the CTF-as-a-service framework in a large-scale environment, we ran a three phased load test. In phase one, we wanted to verify the components would work with a large number of teams. In the second phase, we tested the infrastructure with random team-to-team traffic and team flag submissions. In the third phase, we focused all the team-to-team traffic on one team VM. For the tests, we created 250 team VMs with ten vulnerable services running on each machine. Each team VM was configured with four processors and 16GB of RAM.

Once all the machines were running, the infrastructure reported that most of the team's services were down even though all of the machines passed the GM's verification tests. After some investigation, we realized that the Scriptbot was not able to execute all its tests within a single round. In a round, it needed to run about 8,800 scripts (each one running in a separate process), which required far more processing power and memory than we expected. After increasing it from 8-processors to 32-processors, it was completing most the time. However, we realized that the number of Scriptbots would need to be increased for large scale events to ensure that all the teams' services were properly validated each round.

Fortunately, the other infrastructure components operated as expected while the

game was idle and we were able to start the second phase. For this phase, we started simulating traffic between the teams and performing flag submissions to the Team Interface by each of the teams. All the components performed exceptionally well. We had zero issues while this was running. Moreover, we ran this test for several hours and during the entire test the Scriptbot successfully verified the services on all the machines.

For the third phase, we had 249 of the team VMs connect repeatedly to a single team VM to see how the victim would handle the directed attack. During the test, the victim VM, surprisingly was able to respond to requests, and we were also able to SSH to the box, however the response time for both were exceptionally slow. During all the phases, the Router handled the load well even though it was only using a two processor instance.

2.4.2 The Second Load Test—iCTF 2017

We successfully tested the scalability of our CTF-as-a-service framework by using it to host the iCTF event on March 3rd, 2017. This edition of the competition was different from previous years in that it (1) was open to the public, (2) lasted 24 hours, and (3) was a DEF CON CTF 2017 qualifier; whereas, in prior years we have limited it to academic teams, only eight hours, and was not a qualifier. For this competition, we developed ten vulnerable services, and 317 teams registered.

To reduce our risk of infrastructure failure, we decided to over-provision the CTF instance. First, based on the load tests discussed in Section 2.4.1, we choose to create four Scriptbots that were each responsible for testing a particular subset of the teams. Next, we choose to configure the Database and all four Scriptbots with a VM that

had 36 processors and 60GB of RAM. We configured the Gamebot, Scoreboard, Team Interface, and Router with a VM that had 16 processors with 64GB of RAM. Last, we configured the teams' VMs to use a 4-processor machine with 16GB of RAM.

When we started the competition, the infrastructure withstood 317 teams pounding each other and did not suffer from any infrastructure problems for the first 18 hours. However, just after the 18th hour of the competition, the infrastructure started to crumble. Specifically, the Router stopped allowing connections between the teams. During the competition, we tried to fix the issue, and, although we suspected it was a DOS attack, we could not convince the offending team to stop. So, unfortunately, we had to end the competition early.

After an in-depth forensic investigation [134], we discovered that a team cheated[5] and used their custom-developed in-game botnet (running on nearly all 317 teams VMs) to launch a DDoS attack against another team. In this attack, the bots opened a connection to the victim machine and then terminated it, however, it never sent a FIN packet. This caused the Router to hold each connection open until it timed out. As the number of connections grew, the Router reached a point where it was unable to accept new connections.

With the over-provisioned configuration, the cost of the 1,504 processor infrastructure for twenty-four hours was approximately 3,500 USD, which Amazon covered with a generous sponsorship. Fortunately, smaller and shorter competitions should cost a fraction of that amount. For example, a six-hour competition with one hour for setup and twenty teams should cost less than 50 USD[6].

[5]The iCTF, as a hacking competition, does not have many rules, however DoS attacks are explicitly against the iCTF rules.

[6]For example, creating a six-hour game that uses two t2.2xlarge instances for Database and

Despite the challenges, the 2017 iCTF load test proved that it is possible to leverage the cloud to support large-scale ADCTF competitions.

2.5 Lessons Learned

While developing and running the CTF-as-a-Service framework we ran into several issues that we will discuss in this section. Fortunately, most of the problems we uncovered are solvable, and while we have addressed many of them, we believe that they will serve as useful lessons learned to those developing complicated distributed systems.

While trying to create a large game, we found that we could not have more than ten components starting up at the same time (see Section 2.3.2). This limitation is from a request limit imposed by AWS, and it forced us to limit the number of simultaneous instantiations to ten. As a result, bringing up 326 VMs takes over three hours. However, we found that we could cut the process by one-third of the time by using a custom Amazon machine image [8]. To create the image, we first create a team VM with all the services installed and running properly and then have AWS make a private image of the VM.

Another interesting issue related to instantiating a large game, is that sometimes AWS may not have enough available resources to create the machines as fast as we are requesting. For example, while trying to instantiate the 317 team VMs, with four processors each, AWS stopped allowing new instances and reported that it had run out of resources for the configuration we were using within the availability zone being used.

Scriptbot, four t2.large instances for the remaining components, and twenty t2.medium instances for the teams, is estimated to cost 25 USD.

AWS has several geographically dispersed regions that are designed to be completely isolated from each other [9], called availability zones. Within each region, a VM can be assigned to a particular availability zone. However, each subnet of the VPC must reside entirely within one availability zone. As a result, the obvious work-around failed because we could not simply start bringing up machines in a different availability zone and the design would not support spanning multiple subnets. So, to bring up the boxes, we would have to wait a few minutes after receiving the error and restart the process.

As discussed in Section 2.4.2, we experienced a DDoS attack during the 2017 iCTF competition where nearly all the teams bombarded a single victim. Even though we load tested an attack from 249 machines to a single machine, and found that it could withstand the attack, we did not test what would happen if the attacking machines did something closer to a SYN-flood DDoS attack. In the near future, we plan to research what happened by creating a game and recreating the attack. We will use this environment to understand exactly what went wrong and to devise a solution. In addition, we will use this test to explore the possibility of using a monitoring application to notify us when the network is starting to experience connectivity issues.

While running the iCTF competition, we found that the GM's component testing and recreation process is too smart and made debugging difficult. The retry logic is absolutely necessary for the automated CTF-as-a-Service environment. In a production environment, if something fails, it is probably an issue with the instantiation so often destroying and recreating the component will solve the issue. Therefore, it is no surprise that this works great in a production environment with a stable code base. However, this approach is not applicable in an environment with unstable code, like the iCTF event. For example, if a mistake is made while making changes to a vulnerable

service and it no longer works, the developer needs the machine to continue running so that she can debug the problems. However, the GM, not realizing this, will destroy the VM and attempt to recreate it until it runs out of retries and then it will simply destroy the component. As a result, with the component destroyed, the developer cannot view the logs or otherwise investigate what was actually causing the error.

Shortly after the competition began, we realized that many of the teams were given access credentials to two team VMs instead of one. To understand how this occurred it is necessary to explain how the Team Interface worked for the registration phase and the execution phase. The CTF-as-a-Service platform was designed to run as a single unit. However, for the iCTF, we needed to have a registration server up and running several weeks prior to the event starting (SWPAG will handle all registration for the CTF-as-a-Service). To do this, we created a CTF instance and left only the Database and Team Interface components running because team registration and verification was handled by those components. Once we closed registration a few day prior to the start of the competition, we exported all the teams from the Database and loaded them into the production iCTF instance. During the loading process, each team's identifier was regenerated by the process, so a team had an identifier in the old system that was different from the new one, but everything else was the same. Just before the competition started, we switched the DNS from the old Team Interface to the production instance. Within a short period of time, we realized that since the DNS was the same and the team's sessions did not expire they could see the login credentials of the team that received their old identifier (because the Team Interface stored the session information in a client-side cookie). For example, the team zero_cool with the identifier of 117 on the registration server could retrieve the private key for crash_override, which was team 117 on the production server.

The only true solution to this we could devise during the event was to create a completely new production CTF instance. Fortunately, we were able to bring up the second CTF instance in a different availability zone while the participants played on the first. So, with not much more than the push of a button, we had two CTF instances running concurrently using nearly 650 VMs. After the first eight hours of the competition, we instituted a break during the break we disabled the first production instance and pointed players to the new game, which ensured that each team only had access to a single machine unless they found some other way to compromise them[7].

The iCTF-specific issues highlight the requirement differences between a general CTF-as-a-Service versus what is required when hosting an iCTF event. First, the iCTF is often closer to a development environment because we are constantly trying to push the envelope and find new and interesting ways to execute an ADCTF. So, for the iCTF we need less of an automated black box and more direct access to the components and configuration. Second, the iCTF competition is also the largest ADCTF, and we expect most organizers using SWPAG will host events with less than 50 teams—in fact, we will limit the size of events that SWPAG will host automatically. Third, the iCTF has open registration for its events whereas for CTF-as-a-Service SWPAG will handle registration.

[7]For example, we do not advise participants to post the email address and password for their team to the public chat channel.

2.6 Related Work

Although they have been around for many years the difficulty and time constraints have resulted in only a few online ADCTFs being held each year. In the United States, the two largest being our iCTF event and RuCTF.

Buena Vista University's ADCTF is a cloud-based infrastructure that is geared towards giving participants a gentle introduction to an ADCTF competition [23]. The goal of the organizers is to keep the event small so that the complexity of successfully competing is reduced. As a result, the environment relies on a single administration VM that takes care of managing the services, flags, and scoring. While some similarities exist to our CTF-as-a-Service framework, the stated goal of their system and its subsequent design are significantly different.

Another group working in a similar area is the joint team working on the Build It, Break It, Fix It (BIBIFI) contests [125]. In these contests, the participants first build a system according to the specifications published by the organizers. The teams submit their solutions and are scored based on their conformance to the specifications. Next, the teams enter the break it phase. When the breakers believe they have found a defect, they submit the flaw with an explanation. Their submission is automatically scored and more points are awarded for security vulnerabilities. In the fix-it phase, the build teams receive the bug reports and must fix the discovered flaws. This type of contest is similar to an ADCTF and provides an exciting learning opportunity for the teams. However, it is not currently offered in a framework that could be easily implemented by those that might wish to host their own BIBIFI event.

The Cyber Range Instantiation System (CyRIS) enables educators to automatically deploy and manage cyber ranges for cybersecurity education [115]. Similar to ADCTFs,

a cyber range is a controlled virtual environment that is used to give participants hands-on security experience. While this work is interesting, this research takes a different approach to education and lacks the game aspect of ADCTFs, which pushes students to go beyond the call of duty. Moreover, the cyber range still requires the organizer to possess a certain level of sophistication, our expectation is that we will empower even less savvy organizers than CyRIS.

PicoCTF is designed to increase interest in computer science among high school students [28]. PicoCTF is an attack-focused style of competition. Participants interact with it using a web-based graphical user interface, which is designed as an interactive game. The game even features cut-scenes, sound effects, four levels, and 57 challenges. PicoCTF is different from our CTF-as-a-Service because its target audience is different, and it does not offer any defense exercises.

For the last ten years, the Zero Day Initiative has hosted the Pwn2Own event at CanSecWest [68]. In the Pwn2Own hacking challenge, participants try to compromise the security of various up-to-date computer devices and if they do, they win the device or money. This event differs from the CTF-as-a-Service because it is an attack-only style and its goal its goal is to help vendors find 0-day vulnerabilities and not helping to educate the participants.

Another style of competitions focus on network defense. In these competitions, participants protect their networks by reacting to intrusions from external attackers [35], [107]. This style of competition features only network defense, and, unlike ADCTFs, they do not have an attack component for the competitors.

2.7 Conclusion

SWPAG is a powerful educational tool that empowers anyone to launch their own ADCTF leveraging an easy-to-use interface. Although ADCTFs provide several benefits to teaching security professionals, until now, the creation of an event was a substantial undertaking that required a broad range of networking and administration skills. SWPAG leverages AWS and UCSB's open source iCTF framework to provide a secure environment for teaching the security professionals of tomorrow. While it is still in the early stages of its development, the platform has already survived a 317-competitor ADCTF event and is ready to support future online ADCTF events.

2.8 References

[5] *Amazon API Error Codes*, http://docs.aws.amazon.com/AWSEC2/latest/APIReference/errors-overview.html, 2017.

[6] *Amazon EC2 Instance IP Addressing*, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-instance-addressing.html, 2017.

[7] *Amazon EC2 Security Groups for Linux Instances*, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html, 2017.

[8] *Amazon Machine Images (AMI)*, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html, 2017.

[9] *Amazon Regions and Availability Zones*, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html, 2017.

[10] *Amazon VPC FAQs*, https://aws.amazon.com/vpc/faqs/, 2017.

[22] *AWS NAT Instances*, http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_NAT_Instance.html, 2017.

[23] N. Backman, "Facilitating a battle between hackers: Computer security outside of the classroom," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16, Memphis, Tennessee, USA: ACM, 2016, pp. 603–608, ISBN: 978-1-4503-3685-7. DOI: 10.1145/2839509.2844648. [Online]. Available: http://doi.acm.org/10.1145/2839509.2844648.

[28] P. Chapman, J. Burket, and D. Brumley, "Picoctf: A game-based computer security competition for high school students," in *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*, San Diego, CA: USENIX Association, 2014. [Online]. Available: https://www.usenix.org/conference/3gse14/summit-program/presentation/chapman.

[33] N. Childers, B. Boe, L. Cavallaro, L. Cavedon, M. Cova, M. Egele, and G. Vigna, "Organizing Large Scale Hacking Competitions," in *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Bonn, Germany, Jul. 2010.

[35] A. Conklin, "The use of a collegiate cyber defense competition in information security education," in *Proceedings of the 2Nd Annual Conference on Information Security Curriculum Development*, ser. InfoSecCD '05, Kennesaw, Georgia: ACM, 2005, pp. 16–18, ISBN: 1-59593-261-5. DOI: 10.1145/1107622.1107627. [Online]. Available: http://doi.acm.org/10.1145/1107622.1107627.

[37] *CTF Time*, https://ctftime.org, 2017.

[47] A. Doupé, M. Egele, B. Caillat, G. Stringhini, G. Yakin, A. Zand, L. Cavedon, and G. Vigna, "Hit 'em Where it Hurts: A Live Security Exercise on Cyber Situational Awareness," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Orlando, FL, Dec. 2011.

[55] J. A. Fredricks, P. C. Blumenfeld, and A. H. Paris, "School engagement: Potential of the concept, state of the evidence," *Review of educational research*, vol. 74, no. 1, pp. 59–109, 2004.

[57] M. Gettinger and J. K. Seibert, "Best practices in increasing academic learning time," *Best practices in school psychology IV*, vol. 1, pp. 773–787, 2002.

[68] B. Gorenc, *Pwn2own 2017 at cansecwest*, https://www.zerodayinitiative.com/blog/2017/3/23/pwn2own-2017-an-event-for-the-ages, Mar. 2017.

[72] J. Hamari, J. Koivisto, and H. Sarsa, "Does gamification work?–a literature review of empirical studies on gamification.," in *47th Hawaii International Conference on System Sciences (HICSS)*, Hawaii, 2014.

[73] T. Harmon, *Cyber Security Capture The Flag (CTF): What Is It?* https://blogs.cisco.com/perspectives/cyber-security-capture-the-flag-ctf-what-is-it, 2016.

[84] S. Jariwala, M. Champion, P. Rajivan, and N. J. Cooke, "Influence of Team Communication and Coordination on the Performance of Teams at the iCTF Competition," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 2012.

[107] B. E. Mullins, T. H. Lacey, R. F. Mills, J. E. Trechter, and S. D. Bass, "How the cyber defense exercise shaped an information-assurance curriculum," *IEEE Security Privacy*, vol. 5, no. 5, pp. 40–49, Sep. 2007, ISSN: 1540-7993. DOI: 10.1109/MSP.2007.111.

[115] C. Pham, D. Tang, K.-i. Chinen, and R. Beuran, "Cyris: A cyber range instantiation system for facilitating security training," in *Proceedings of the Seventh Symposium on Information and Communication Technology*, ser. SoICT '16, Ho Chi Minh City, Viet Nam: ACM, 2016, pp. 251–258, ISBN: 978-1-4503-4815-7. DOI: 10.1145/3011077.3011087. [Online]. Available: http://doi.acm.org/10.1145/3011077.3011087.

[120] M. Prince, "Does active learning work? a review of the research," *Journal of engineering education*, vol. 93, no. 3, pp. 223–231, 2004.

[125] A. Ruef, M. W. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel, "Build It, Break It, Fix It: Contesting Secure Development," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016. [Online]. Available: http://arxiv.org/abs/1606.01881.

[126] Y. Shoshitaishvili, L. Invernizzi, A. Doupé, and G. Vigna, "Do You Feel Lucky? A Large-Scale Analysis of Risk-Rewards Trade-Offs in Cyber Security," *ACM Symposium on Applied Computing*, Mar. 2014.

[134] *The 2016-2017 iCTF DDoS*, https://ictf.cs.ucsb.edu/pages/the-2016-2017-ictf-ddos.html.

[136] *The iCTF Framework*, https://github.com/ucsb-seclab/ictf-framework.

[139] L. VaBlasco-Arcas, I. Buil, B. Hernandez-Orteg, and F. J. Sese, "Using Clickers in Class. the Role of Interactivity, Active Collaborative Learning and Engagement in Learning Performance," in *Computers and Education*, vol. 62, Pergamon Press, Mar. 2013, pp. 102–110.

[140] K. Vamvoudakis, J. Hespanha, R. Kemmerer, and G. Vigna, "Formulating Cyber-Security as Convex Optimization Problems," in *Control of Cyber-Physical Systems*, ser. Lecture Notes in Control and Information Sciences, vol. 449, Springer, Jul. 2013, pp. 85–100.

[141] G. Vigna, "Teaching Hands-On Network Security: Testbeds and Live Exercises," *Journal of Information Warfare*, vol. 3, no. 2, pp. 8–25, Feb. 2003.

[142] ——, "Teaching Network Security Through Live Exercises," in *Proceedings of the Third Annual World Conference on Information Security Education (WISE)*, C. Irvine and H. Armstrong, Eds., Monterey, CA: Kluwer Academic Publishers, Jun. 2003, pp. 3–18.

[143] G. Vigna, K. Borgolte, J. Corbetta, A. Doupé, Y. Fratantonio, L. Invernizzi, D. Kirat, and Y. Shoshitaishvili, "Ten Years of iCTF: The Good, The Bad, and The Ugly," in *Proceedings of the USENIX Summit on Gaming, Games and Gamification in Security Education (3GSE)*, San Diego, CA, Aug. 2014.

Chapter 3

CLOAKING EXTENSIONS

Abstract

Browser fingerprinting refers to the extraction of attributes from a user's browser which can be combined into a near-unique fingerprint. These fingerprints can be used to re-identify users without requiring the use of cookies or other stateful identifiers. Browser extensions enhance the client-side browser experience; however, prior work has shown that their website modifications are fingerprintable and can be used to infer sensitive information about users.

In this paper we present CloakX, the first client-side anti-fingerprinting countermeasure that works without requiring browser modification or requiring extension developers to modify their code. CloakX uses client-side diversification to prevent extension detection using anchorprints (fingerprints comprised of artifacts directly accessible to any webpage) and to reduce the accuracy of extension detection using structureprints (fingerprints built from an extension's behavior). Despite the complexity of browser extensions, CloakX automatically incorporates client-side diversification into the extensions and maintains equivalent functionality through the use of static and dynamic program analysis. We evaluate the efficacy of CloakX on 18,937 extensions using large-scale automated analysis and in-depth manual testing. We conducted experiments to test the functionality equivalence, the detectability, and the performance of CloakX-enabled extensions. Beyond extension detection, we demonstrate that client-side modification of extensions is a viable method for the late-stage customization of browser extensions.

3.1 Introduction

To empower users to protect their own privacy, in this chapter we propose CloakX, a client-side countermeasure against extension fingerprinting. Instead of trying to remove the fingerprintable attributes of extensions, our approach is to automatically alter, randomize, and add to these attributes without requiring web browser modifications or any involvement from the extension's developer. Through these modifications, CloakX diversifies the extension's anchorprints, which are fingerprints consisting of items that can be accessed directly from a webpage, and structureprints, which are fingerprints that embody the structural changes an extension makes to a webpage (for more details refer to Section 3.2.2). On the surface, client-side diversification of the fingerprintable attributes seems straightforward; however, the dynamic nature of JavaScript and the complexity of the browser extension's architecture necessitated a complex approach that relies on both static and dynamic program analysis.

CloakX uses static and dynamic analysis techniques to automatically diversify the extension's fingerprint without modifying the browser, without requiring any changes by the extension's author, and without altering the extension's functionality. To diversify the extension's anchorprint, CloakX automatically renames WARs, IDs, and class names and corrects any references to them in the extension's code, which severs the link between the published extension and the currently installed version. In addition to static changes, the diversification is also performed by our dynamic DOM proxy (Droxy), which intercepts DOM modifications from the extension's code and makes the changes on-the-fly. To diversify the extension's structureprint, Droxy also injects random tags, attributes, and custom attributes into each webpage, which obfuscates the extension's structureprint. As a result, an extension cloaked by CloakX

is undetectable by a webpage using anchorprints and is obfuscated from a webpage using structureprints; however, from the user's point of view, the extension operates the same.

In summary, we make the following contributions:

- We present the design of a novel system that automatically identifies and randomizes browser extension fingerprints to defend against existing extension fingerprinting techniques without requiring any browser changes or any involvement from the extension's developer.
- We describe the implementation of our design into a prototype, CloakX, that uses a combination of: (1) static rewriting of extension JavaScript code and (2) a dynamic DOM proxy, Droxy, that intercepts and rewrites extension requests on-the-fly.
- We use a combination of high-fidelity testing (extensive manual testing) and low-fidelity testing (broad automated testing) on the extensions rewritten by CloakX to quantify the breakage caused by our system, demonstrating that client-side modification of extensions introduces minimal defects.
- We also evaluate the detectability of cloaked extensions and show that some cloaked extensions are undetectable while others are more difficult to detect.

3.2 Background

In this section, we provide insights into the complexity of modern browser extension frameworks that must be taken into account when designing a client-side countermeasure against extension fingerprinting. We start by describing the architecture of browser extensions, focusing on the details that pertain to their fingerprintability. Next, we discuss fingerprinting and detecting extensions using anchorprints (fingerprints

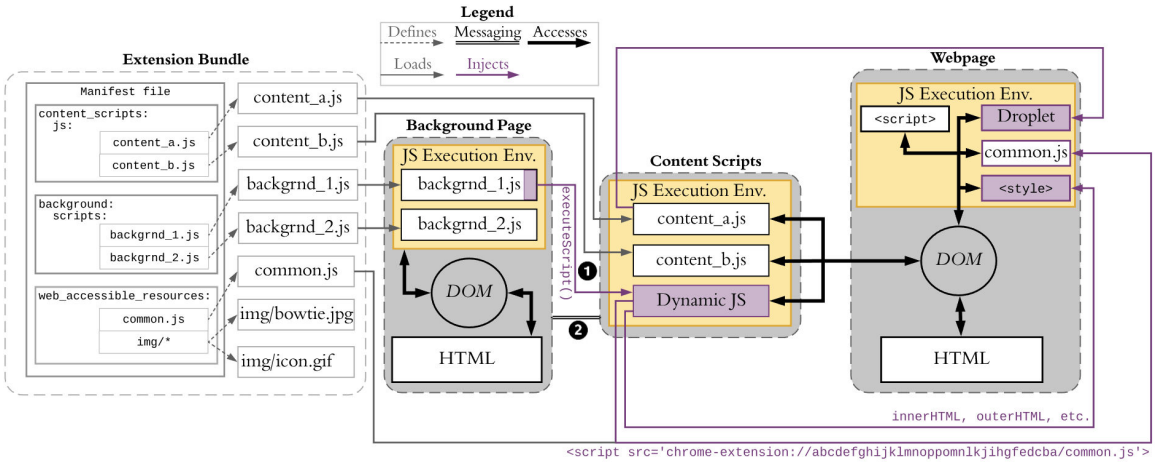

Figure 3. Extension architecture. A high-level overview of Chrome's extension architecture with the static content of the extension on the left side and the multiple execution environments on the right. Background pages can 1) inject content scripts dynamically using the executeScript() method in Chrome's extension API and 2) send and receive messages from the content scripts.

that are comprised of items directly accessible from a tracking webpage's JavaScript) and structureprints (fingerprints built from the extension's behavior). Last, we finish this section by presenting the threat model that CloakX can defend against.

3.2.1 Browser Extensions Explained

While modern web browsers provide an ever-increasing range of functionality to users and webpages, an off-the-shelf browser cannot possibly provide a sufficiently large set of features to satisfy every user's browsing needs. To improve the user's browsing experience, browsers enable users to enhance their functionality through extensions. Users add extensions to their browsers to change the browser's look, to add helpful toolbars, to block ads, and to enhance popular webpages [51].

Although extensions utilize web technologies such as HTML, CSS, and JavaScript, they also have access to powerful extension-only APIs that enable them to, among

others, access and modify cross-origin content and a browser's client-side storage. However, before an extension can access broader privileges or interact with a webpage, it must request this access from the browser. As Figure 3 depicts, the modern extension architecture implements a layered security approach within the browser that creates multiple execution environments with varying levels of persistence and privileges for each extension and webpage.

The left-hand side of Figure 3 depicts the static parts of an extension, including items such as the manifest, JavaScript, HTML, and image files. For the browser to parse and install an extension, it must have a *manifest* file which defines the extension's properties. Similar to the manifest shown in Figure 3, extensions commonly rely on three properties, which describe background pages, content scripts, and web accessible resources [51].

When the *background* property is included in the extension's manifest, the browser automatically constructs a hidden background page for the extension. The background page contains HTML, a DOM, and a separate JavaScript execution environment (labeled as "Background Page" in Figure 3). The JavaScript executed in the background page often contains the main logic of the extension, maintains long-term state, and operates independently from the life-cycle of the webpages [92].

Content scripts bridge the gap between the background page and the current webpage. An extension uses content scripts to modify the current webpage and communicate with the background page. These content scripts are either statically declared by an extension in the manifest file or programmatically injected into the current webpage. For example, on the left-hand side of Figure 3, the manifest declares the two content scripts `content_a.js` and `content_b.js`. To programmatically inject

a content script, an extension must call `executeScript()` from a background page (see ❶ in Figure 3).

To modify a webpage, a content script uses the webpage's DOM [144]. DOM APIs provide a systematic way for interacting with a webpage. In this chapter, we call the content script's interaction with the DOM APIs *DOM requests*.

Notice in Figure 3 that the background page, content scripts, and webpage each run their own JavaScript execution environment. The separate execution environments prevent the JavaScript variables and functions from directly interacting. Google Chrome's documentation states that content scripts "live in an isolated world, allowing a content script to make changes to its JavaScript environment without conflicting with the page or additional *content scripts*" [36] (emphasis added). This statement, however, is misleading because we experimentally discovered that content scripts loaded from the same extension share variables and can call functions from other content scripts. Thus, an extension's content scripts share a single execution environment; however, they do not share an environment with the background page, webpage, or other extensions (depicted in Figure 3).

Using DOM requests, a content script has significant control over the rendered webpage. Content scripts can inject HTML into the webpage (using DOM element properties such as `innerHTML` or DOM methods such as `appendChild()`). We call this injected HTML *droplets* (the extension drops them onto the webpage). Among other elements, droplets may contain `<script>` tags where the extension includes either inline or remote JavaScript. By injecting JavaScript, the content script purposefully bypasses the isolation between the content scripts and the webpage's execution environments.

The Chrome Extension API provides privileged functionality available only to

extensions. Chrome grants background scripts broad access to the API's capabilities. However, Chrome grants content scripts limited access to the API while making the API inaccessible to webpages. For example, only an extension's background page can access network resources, view platform information, and communicate with native applications. However, both content scripts and background scripts may use the API to initiate and listen for communications from one another via the appropriate Chrome APIs (as shown by the double lines towards the bottom of Figure 3). Background scripts cannot directly interact with a webpage, however they can *indirectly* send messages to it via the extension API using the method `chrome.runtime.sendMessage()` [34]. Part of the reason for this layered security model, including the separate execution environments, is to isolate the components and prevent webpages from unauthorized access to the extension API's more sensitive functions.

Another important property in the manifest is the *web-accessible-resources* property [99]. Prior to January 2014, Chrome permitted external access to all of an extension's resources, i.e., a webpage could reference resources belonging to installed extensions. In more modern versions of Google Chrome, an extension must explicitly whitelist a resource before a webpage may retrieve it [100]. An extension whitelists its resources by adding them to the *web-accessible-resources* property in the manifest. Once added, a resource becomes accessible to any webpage or any installed extension.

To access a web accessible resource (WAR) from the context of a web page, a webpage developer uses a URL of the format:

```
chrome-extension://[extId]/[path-to-resource]
```

The `extId` in the URL is a unique identifier generated by the Google Web Store upon publication of an extension which does not change when extensions are updated.

3.2.2 Extension Fingerprinting and Detection

In 2017, Sjösten et al. demonstrated that, with WAR fingerprinting, any extension using WARs is trivially detectable by a webpage [128] by creating a database of which WARs are utilized by each extension available in the Google Store. Given that an extension's ID is globally unique and permanent, a tracker can detect an extension by requesting any one of its previously identified WARs. If the request is successful, then the corresponding extension is installed on the user's browser. Next to its simplicity and the 16,479 (28%) of extensions that utilize WARs (and are thus fingerprintable), WAR fingerprinting works in the browser's private mode.

Orthogonally to WAR fingerprinting, Starov et al.'s Extension Hound (XHound) [131] creates a DOM fingerprint based on the extension's DOM modifications. XHound uses dynamic analysis to exercise extensions and detect changes introduced to the DOM through the extension's operation. By loading a set of webpages with and without a given extension, XHound can compare the two resulting DOMs and isolate the DOM changes that were performed by the given extension. These changes can straightforwardly be converted into fingerprints which trackers can use to detect the presence of any DOM-modifying extension.

When using WAR and DOM fingerprints for detection of extensions, we reclassify all such fingerprints into *anchorprints* and *structureprints* to describe the method and accuracy of the detection techniques. Anchorprints rely on an *anchor* between the webpage's JavaScript and the extension. An anchor is a unique identifier formed to facilitate access and communication between webpages and extensions. An anchor provides a way to directly access elements and resources available to the webpage. Some examples of anchors include WARs, IDs, class names, and custom attributes. For exam-

ple, the Chrome extension Grammarly adds a unique class to the root `<html>` element on each webpage. Thus, if a webpage uses `document.getElementsByClassName()` and receives the `<html>` element, it is likely the user has Grammarly installed.

An anchorprint is comprised of all the WARs, IDs, class names, and custom attributes made available by an extension. With the items in an anchorprint, a webpage need only to query the DOM or send an `XMLHttpRequest` to detect an extension. WARs are the most powerful of the anchorprint elements because, due to the unique extension identifier, an anchorprint with even one WAR is always 100% accurate. Although IDs, class names, and custom attributes might be 100% accurate, they often have a much lower per element accuracy than WARs because webpages and extensions alike often use some of the same names. Despite this limitation, the accuracy of the anchorprint improves dramatically with each additional element included in it.

Structureprints are less precise (in terms of fingerprinting) but are formed based on the *structure* of the changes the extension makes to the underlying webpage. Structureprints effectively create a DOM fingerprint that uses the extension's unique and intended behavior to identify the extension. The idea of a structureprint is that it can be used to detect a specific extension because the extension always behaves in a predictable manner and alters a webpage consistently, thus creating a *structure* that is unique among extensions. For instance, consider a popular Google Calendar extension that is the *only* extension with a structureprint that contains the tags `a` and `img` with the following attribute names `href, location, target, blank, width, height, src, alt` and `style` Surprisingly, we found during our experiments that a tracking webpage can reliably detect 28.93% (1,511) of extensions using only the `tagName` of the DOM elements added or deleted from a webpage by an extension.

Adding attribute names, attribute values, and the text of the DOM elements to the structureprint increases the number of detectable extensions to 73.65% (3,847).

An important *subset* of structureprints that target an extension's behavior are called *behaviorprints*. For example, Grammarly creates a green button inside a `text area`. With manual analysis, it is possible to identify whether the green button has been added to the webpage without relying on the IDs or class names injected by Grammarly. Another example of using behaviorprints are in the detection of ad-blocking extensions, such as Detect AdBlock [42]. However, no recent research has shown how to create a behaviorprint in an automated way at scale. As a result, current behaviorprints are limited to targeted attacks against specific extensions or narrowly constrained categories of extensions (e.g., ad-blocking extensions).

Beyond the obvious implementation differences between anchorprints and structureprints, the fingerprint classes differ in their accuracy and their destructibility. For most anchorprints, matching the WAR, ID, class name, and custom attributes of a published extension often provides a (unique) one-to-one match. However, for structureprints, finding a match is often less certain because many extensions have similar behavior, which results in the same structureprint. Another key difference between anchorprints and structureprints is the permanence of their link between the published extension and the user's installed version. For anchorprints using WARs, IDs, and class names, CloakX completely renames the values. By renaming the values, CloakX completely destroys the link between the published extension and the user's installed version. Without that link, it is impossible for a tracking webpage to use the anchorprint to identify the installed extension because the anchorprint no longer matches the published extension. Whereas with structureprints, the destruction of the link between the published extension and the user's installed version is difficult. This

difficulty occurs because of the requirement that a cloaked extension retain the same behavior (i.e., user experience). By maintaining the same behavior, the structureprint of a cloaked extension is only being obfuscated, which means that with enough effort a tracker can eventually deobfuscate the cloaked structureprint and, thus, detect the cloaked extension.

3.2.3 Threat Model

In our threat model, attackers use a database of fingerprints to detect the extensions installed by a visitor to the site. However, we limit the attackers to the information and privileges afforded to the webpage's JavaScript execution environment. In essence, we assume that there are no zero-day vulnerabilities that would allow webpages to bypass the layered-security architecture depicted in Figure 3. Therefore, the attackers cannot access the content of an extension installed on a visitor's device.

In this chapter, we explore two different types of attackers. The automated attacker uses automated extension detection techniques. Specifically, we limit the automated attacker to anchorprints and structureprints. To detect an extension, the automated attacker must find either an exact or fuzzy match to an entry in their fingerprint database. The targeted attacker is permitted to manually generate targeted structureprints using portions of the structureprint (i.e., behaviorprints) for extension detection. While we focus on defending against the automated attacker because automated large-scale detection is a feasible attack, we also include the targeted attacker to explore how CloakX can defend against the targeted attacks.

3.3 CloakX

The core idea behind CloakX is to diversify each extension's fingerprint from the client-side while maintaining equivalent functionality *without making any changes to the browser and without requiring the developers to alter their extensions*. Client-side diversification of the anchorprints (fingerprints comprised of items directly accessible from a tracking webpage's JavaScript) and structureprints (fingerprints built from the extension's behavior) reduces the extension's detectability by breaking a webpage's ability to link together a published extension and the one installed on the user's machine. CloakX defeats detection using an anchorprint by randomizing the names of the WARs, IDs, and classes. However, CloakX does not completely defeat anchorprint detection using custom attributes. CloakX's approaches combat custom attribute-based detection by randomly injecting more unique custom attributes into each webpage. CloakX reduces the efficacy of structureprints by introducing random attributes and tags into the webpage. Although CloakX does not completely prevent detection using custom attributes or structureprints, it is a step beyond current solutions and CloakX achieves these protections without any changes to the browser and without requiring the intervention of extension developers.

Figure 4 shows the overall process of CloakX, a multiphase tool that leverages static- and dynamic-analysis techniques to achieve extension diversification while maintaining functional equivalence. In the first phase, CloakX analyzes the extension for the DOM fingerprints and CloakX identifies the droplets that must be statically analyzed. In the second phase, CloakX renames each WAR within the extension to a unique random value, finds all the references to the original name, and replaces them with their randomized counterpart. In the third phase, CloakX adds a dynamic

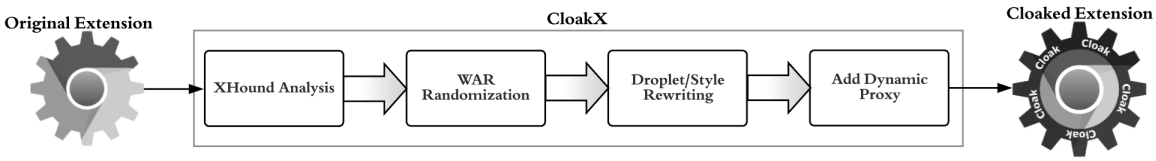

Figure 4. Overview of the CloakX process.

proxy (Droxy) to the extension's content and background scripts. Droxy dynamically intercepts DOM and WAR requests and substitutes the original ID, class names, and WAR names with their random counterparts. In the last phase, CloakX statically analyzes and rewrites the DOM IDs and class names inside droplets that cannot be dynamically intercepted by Droxy.

3.3.1 XHound Analysis

CloakX uses XHound (we obtained a copy of the XHound prototype by contacting the paper's authors [131]) to generate a DOM fingerprint for the extension and to identify the droplets injected into the webpage. Each DOM fingerprint consists of four types of artifacts: (1) adding a new DOM element, (2) deleting a DOM element, (3) setting or altering an element's attribute, and (4) changing text on the page.

Of the four types, DOM additions are the most common type of detectable artifacts according to XHound [131]. This is because DOM additions are generic operations that often rely on loose coupling with a webpage for them to be triggered. Whereas most DOM modifications or deletions require a tighter coupling between the extension and the webpage, which limits their applicability to the problems often solved by developers. For instance, consider a password manager extension that injects a stylized element into every password form field (so that the user can invoke the password manager interface). The extension adds the element to the webpage and gives it a

unique ID and a custom class name. It requires the ID to communicate with the element once it's placed on the webpage. Using the added ID and class name (i.e., the extension's anchorprint), a webpage can detect the extension by checking for the presence of either the unique ID or class name on the webpage.

Next, CloakX uses XHound to identify any droplets the extension injects into the webpage's execution environment so that CloakX can preprocess the droplets to identify the ID and class names within them. As discussed in Section 3.2.1, droplets (purple-colored boxes in Figure 3) are JavaScript strings that an extension injects directly into a rendered webpage. Droplets can include any text literal such as HTML, JavaScript, or base64-encoded images; however, the preprocessing is only performed on droplets containing inline JavaScript and those `<script>` elements that reference WARs.

Finally, during this phase, CloakX creates a map from the original ID and class names used to fingerprint the extension to the new randomized values.

3.3.2 Diversification of Web-Accessible Resources (WARs)

The principle behind the diversification of Web-Accessible Resources (WARs) is straightforward: if each installation of an extension has different filenames for the same WARs, then a tracker can no longer create a global database of WARs and, therefore, can no longer detect the presence or absence of any given extension based on its WAR anchorprint.

In the first stage of the WAR diversification process, CloakX identifies all the resources declared as WARs in the manifest file of each extension. Although many extensions explicitly list the resources they wish to make accessible, it is also possible

to use a * wildcard [108]. With wildcards, an entire folder, its contents, and all its subfolders can be designated as web-accessible—this includes using a single *, which designates every file in the extension as web-accessible. Even though making every file in the extension web-accessible is likely an implementation error, we discovered 419 extensions that made all of their resources web-accessible, out of 59K analyzed extensions. In the second stage, CloakX computes the shortest unique file path to facilitate the search-and-replace in the final stage. Specifically, CloakX reduces the full path of each WAR to the minimum length necessary to uniquely identify the resource (compared to all the other resources in the extension). This operation reduces the number of resource references missed (i.e., false negatives) associated with dynamic string concatenation (often a directory path).

In the final stage of the WAR diversification process, CloakX uses the shortest unique path to find every use of the WAR within the extension's files and to replace that with the appropriate random value, maintaining the correctness of WAR references for each extension.

In addition to the static alterations described above, CloakX relies on Droxy, discussed in the Section 3.3.3, to dynamically translates any WAR requests missed by the static replacement method.

3.3.3 Droxy

The next step in the CloakX process adds Droxy to the extension. Droxy is a content script that injects random attributes and tags into the DOM to further obfuscate the extension's DOM fingerprint while also translating any uncloaked WAR requests and the IDs and class names used in DOM requests into their cloaked versions.

CloakX patches Droxy into the extension and configures Droxy to execute before any of the extension's content scripts.

Droxy adds random attributes and tags to the DOM to reduce the accuracy of detection using structureprints. As each webpage is loaded, Droxy adds a random number of randomly generated tags to the DOM to make extension detection less accurate. To further frustrate detection using structureprint matching, Droxy adds random attributes to the DOM elements added by each extension.

Droxy also uses cross injection of custom attributes to frustrate anchorprint detection. For trackers using custom attributes to detect extensions, cross injection allows the user to impersonate other extensions, which increases a tracker's false positives when using anchorprint detection. This is done by adding custom attributes that are randomly selected from a list of the 244 unique custom attributes used by other extensions with a DOM fingerprint.

Droxy also dynamically catches any WAR requests made using the resource's original filename, which serves as a backup for the static replacement method described in Section 3.3.2. Droxy achieves this by watching for changes to the DOM using a `MutationObserver()` that checks for uncloaked WAR requests inside the DOM elements altered by the extension. In addition, Droxy overrides the `XMLHttpRequest.open()` method and adds functionality to translate any WAR requests for the original filename to the new, randomized filename.

Droxy translates the ID and class names used to create a DOM anchorprint. As the first content script to load, Droxy overrides DOM accessor and mutator methods before the extension uses them to interact with the DOM, which effectively wraps all DOM requests in a translation layer (blue area in Figure 5). Each of the overridden methods are augmented to intercept and translate ID and class names used to create

the DOM fingerprint. Droxy determines which ID and class names to translate by checking the ID and class names against the cloaking map, created in Section 3.3.1. The cloaking map contains name–value pairs where each XHound-discovered ID and class name is paired with a randomized version. If it finds a match in the cloaking map, it translates the original value on-the-fly into the randomized version. By intercepting and translating the fingerprintable ID and class names to randomized values, Droxy alters the extension's DOM fingerprint from the perspective of a tracker's execution context breaking the link between the user's installed extension and the publicly available version.

To prevent the use of anchorprint detection, Droxy translates IDs and class names into random values according to the map created in Section 3.3.1. For ID and class name translation, Droxy also tracks DOM queries and DOM mutations. Droxy intercepts and inspects the extension's queries that use IDs, element names, class names, and query selectors, which include the methods `getElementById()`, `getElementsByName()`, `getElementsByTagName()`, `getElementsByClassName()`, `querySelector()`, and `querySelectorAll()`. To handle more complex query selectors, Droxy parses the selectors using the open-source Sizzle engine to accurately identify the ID and class names [127].

For DOM mutations performed via JavaScript, Droxy intercepts all the ways in which an ID or class name can be introduced to the DOM. This dynamic interception of ID and class names is done by overriding `setAttribute()` and `getAttribute()` methods and redefining `id` and `className` properties to use the overridden `setAttribute()` and `getAttribute()`. In addition, Droxy overrides the `classList` property. Because `classList` is an object, Droxy overrides the `add()`, `contains()`, and `remove()` meth-

ods of the `classList`. As a result, Droxy translates the extension's use of IDs and class names whether it is done when a DOM element is created or modified.

For DOM mutations performed via the injection of raw HTML, Droxy uses static and dynamic analysis to make the translation of ID and class names straight-forward and precise. Droxy overrides the methods used to inject raw HTML, such as the `innerHTML` property and `insertAdjacentHTML()` method. Droxy uses the browser to parse the HTML by creating a mock container and adding the HTML to it without attaching the mocked container to the DOM. Droxy queries the mock container to identify and transform the ID and class names into their randomized versions. Droxy then exports the string representation from the mock container's DOM and then calls the original method to apply the modified string to the webpage's DOM.

In addition to DOM queries and mutations, Droxy intercepts styles and translates on-the-fly. An extension can include styles via text content inside `<style>` or `CSSStyleSheet`'s methods such as `addRule()` or `insertRule()`. Once intercepted, Droxy uses CSS parsing to locate the IDs and class names. If found, Droxy replaces the ID or class name with its randomized counterpart.

Droxy replaces an extension's droplets with the statically rewritten version (`content_a.js` and `Dynamic JS` in Figure 5). As a part of the droplet rewriting process described in Section 3.3.4, Droxy receives a hash value of the original droplet and modified version of the code for each droplet used by the extension. Droxy then matches the current droplet's hash to the ones provided and replaces it with its cloaked counterpart. Droxy performs the matching and replacement by customizing the properties `textContent`, `innerText`, and `HTMLScriptElement`'s and the methods `append()` and `appendChild()` This process is depicted by the dashed arrow near

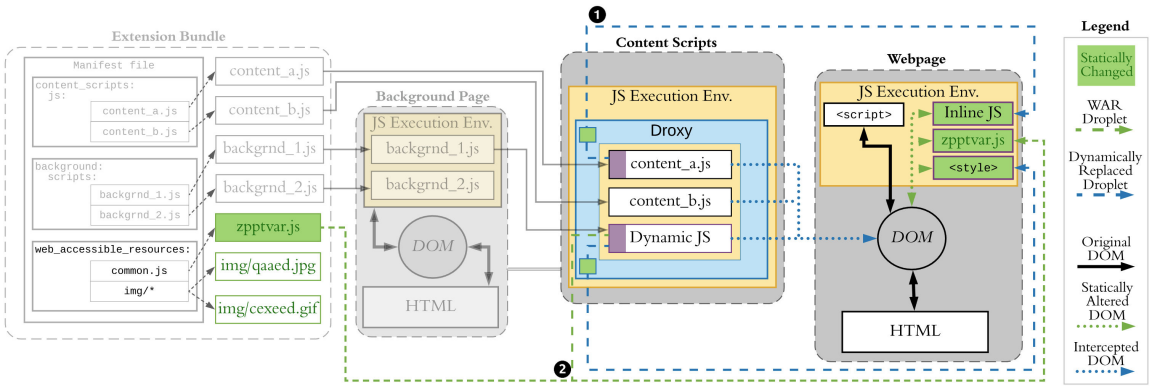

Figure 5. Diversified CloakX rewritten extension.CloakX hides fingerprints by rewriting the droplets, content styles, and renaming of web-accessible resources (WARs) and through Droxy's on-the-fly substitution. As a result, a tracking webpage cannot access the original identifiers; however, the internal logic of the extension still can because Droxy translates those requests.

❶ in Figure 5. Droxy relies on the preprocessed JavaScript because rewriting the code on-the-fly in the browser efficiently is currently infeasible.

3.3.4 Static Droplet Rewriting

As discussed previously and shown in Figure 3, Droxy cannot intercept a droplet with dynamically inserted JavaScript because when the inserted code is executed in the webpage's JavaScript execution environment. Unfortunately, Droxy is also unable to cloak the droplet before inserting it because the heavy-weight static analysis necessary would significantly degrade the extension's performance. Therefore, CloakX statically analyzes the droplets offline, identifies where the extension adds the fingerprintable ID and class names to the DOM, rewrites the JavaScript code, and Droxy dynamically substitutes the original code with its rewritten counterpart.

Extensions commonly use *generic* values for IDs and class names, which often overlap with JavaScript keywords or JavaScript code constructs that refer to the

class names and IDs dynamically. In addition, the expressiveness of JavaScript means that the ID and class names usage are context-sensitive. For example, if the fingerprintable class name is `content`, CloakX should only replace the instance of `#content` and ignore `element.content`, `content.maximizer`, and `content-shaper`, as each have a different semantic meaning. Developers often construct ID and class names dynamically in the code, which necessitates a more sophisticated form of static analysis. For example, an extension might attempt to access an element with the ID `content` by using `getElementById(‘‘con’’ + ‘‘tent’’)`, which would be missed by a regular expression searching for the full word.

CloakX statically rewrites droplets offline (i.e., before an extension is installed) using static analysis to identify the appropriate locations in the JavaScript. CloakX limits its rewrites to the ID and class names that occur in the JavaScript and are added to the webpage via the DOM. By identifying and only altering these DOM altering instances, CloakX limits the possibility of breaking the extension with the alterations. In essence, the static rewriting requires a tool that performs taint analysis where it labels DOM interactions as sinks and then analyzes the backward slices of the control flow graph (CFG) until it finds the fingerprintable IDs and class names as sources.

We decided to use TAJS—a state-of-the-art and feature rich JavaScript analyzer—as the program analysis core of the CloakX static rewriting. We chose TAJS because it (1) performs type analysis on JavaScript, (2) supports most of the ECMAScript 5 standard and DOM functionality, (3) is under active development, (4) is open source [59], and (5) is the product of recent research [12]–[14], [85]–[87], [89].

TAJS performs dataflow analysis by using techniques that examine the flow of data along program execution paths. As TAJS iterates over the CFG, it creates a

semilattice of program states that are unique for each basic block in the CFG [88]. For each variable represented in the lattice at a given basic block, TAJS assigns a set of possible values. The dataflow analysis completes when the values inside the lattice reach a fixed point and no longer change with each iteration. Using these values, it is possible to follow data both forwards and backwards through the CFG [88].

3.3.4.1 TAJS for Extensions

We enhanced TAJS to support static rewriting of the droplets by adding support for Chrome extensions, adding DOM taint analysis, and maximizing its exploration of the CFG. In addition, we plan to make our changes to TAJS publicly available because there are currently no other program analysis tools for browser extensions.

We added extension support to TAJS by creating stubs for Chrome's extension API and implementing support for necessary methods such as `sendMessage(), getURL(), executeScript(), onMessage.addListener()` .

We implemented taint analysis within TAJS that tracks data through an application until it reaches a sink, where a sink is a location of interest within the CFG [18]. For the purposes of this analysis, TAJS tracks string literals matching the fingerprintable IDs and class names through the CFG until they are used to interact with the DOM. As a part of the taint tracking, we added functionality that maintains an *audit trail* of the changes to each variable while traversing the CFG so that upon reaching a sink CloakX can trace the values of interest to their origins.

We increased TAJS's code coverage by adding edges to the end of the CFG that force a call to every named and anonymous function defined within the code. For the purposes of extension rewriting, it is necessary that TAJS analyzes all the JavaScript

within a droplet because some functions appear unreachable without complete semantic understanding of Chrome's extension execution environment. However, the dynamic aspects used by TAJS itself to strike a balance between soundness and precision came at the cost of code coverage [14]. For example, TAJS does not analyze functions unless they are called by the JavaScript and the call is also reachable from the beginning of the CFG. Because extension rewriting requires TAJS to analyze all of the JavaScript within a droplet, we added edges to the end of the CFG that simulates a call to every named and anonymous function in the droplet. The potential downside to adding the edges is the decreased precision of our analysis (i.e., we are adding behavior to the application that does not exist at run-time), however for the purposes of identifying DOM fingerprints the trade-off is acceptable.

3.3.4.2 Static Analysis Results

Automated analysis of real-world JavaScript code is a difficult problem and despite all the advances made by TAJS, it, as well as similar tools, cannot analyze some JavaScript programs. As a JavaScript program increases in complexity and size, it becomes increasingly less likely TAJS will complete the analysis due to the explosion of dataflows (i.e., the classic state space explosion problem). As acknowledged by the authors, TAJS initially targeted hand-written JavaScript applications of a "few thousand lines of code" [88]. Plus, the addition of the fake edges dramatically increased the complexity of the CFG and the number of states, which decreased the code TAJS could successfully analyze to about 1,000 lines of code.

Fortunately, CloakX only needs TAJS analysis for the 197 extensions using droplets, which is only 3.2% of the extensions identifiable by XHound, because Droxy handles

the rest of the extensions. Out of those 197 extensions, TAJS analyzed 212 total scripts of which 94 were JavaScript files that were designated as a WAR (and thus accessed via a `src` attribute, see Figure 5) and 118 were inline JavaScript. TAJS successfully completed analysis of 134 scripts (63.2%) finding 19,380 basic blocks and analyzing 18,497 (95.44%). However, TAJS was unable to analyze 78 (36.8%) of the inline JavaScript and WARs because the analysis for 34 scripts timed out, 6 scripts failed with an analysis exception, 6 scripts failed due to syntax errors in the JavaScript, and 32 scripts failed when TAJS crashed.

After manually analyzing the results we found the following reasons for why TAJS failed.

Exceeded timeout threshold. Most of the JavaScript code that caused TAJS to timeout were large JavaScript files that varied in size from 75 kilobytes to over a megabyte. In other cases, TAJS failed to finish analyzing smaller JavaScript code because of a bug in the forced path exploration code.

Analysis exceptions. TAJS failed to complete the analysis because it was missing support for the ECMAScript standard.

Syntax errors. TAJS was unable to analyze scripts with error in the JavaScript syntax.

Crashed. Some of the scripts triggered a bug in TAJS, causing it to crash with null pointer, stack overflow, or other miscellaneous exceptions.

3.3.5 Cloaked Extension

Once CloakX completes its modifications to the extension, the extension is cloaked and it appears to a webpage using anchorprint or structureprint detection techniques

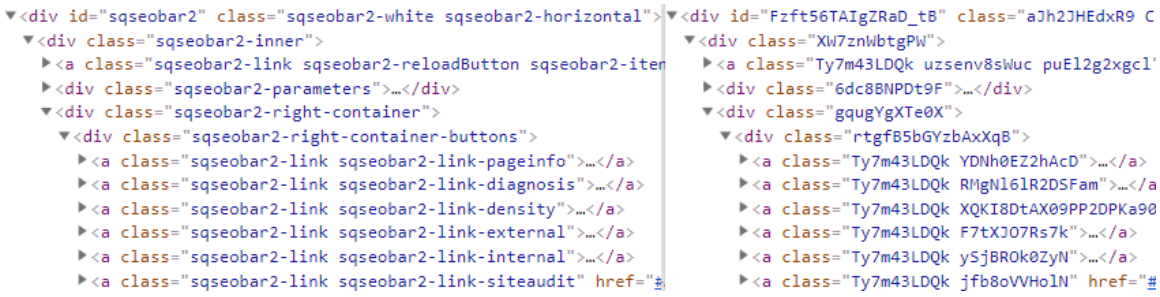

Figure 6. Original code of SEOquake extension (left) and SEOQuake extension when patched by CloakX (right).

as though the user no longer has that particular extension installed. Architecturally, the resulting extension is similar to Figure 5 with Droxy surrounding the content scripts and translating the extension's DOM requests and droplet injections. To a webpage, the results look similar to the HTML source shown in Figure 6.

The permanence of the cloaked anchorprint and structureprint depends on whether the extension is subject to static rewriting. For cloaked extensions that rely on purely dynamic mutations, the structureprint changes each time the cloaked extension is loaded. Droxy alters the structureprint by injecting new randomly generated noise into the DOM and re-randomizing the cloaked ID and class names. However, CloakX must statically alter extensions with WARs or droplets. As a result, the cloaked fingerprint of extensions requiring static rewriting remains the same until a new version of the extension is reprocessed by CloakX. Although guessing the name of a cloaked WAR is unlikely because CloakX generates a random alphanumeric value that is at least ten characters in length for each WAR; even if an adversary guesses the name of a WAR, the detectability would cease when a new version of the extension was released.

3.3.6 Deployment

Although we describe CloakX as a client-side mechanism (as this is where the fingerprint rewriting is done), to reduce end-user friction, we envision CloakX as the final step in an extension's release and update process, all of which can be performed by the extension store and would require no intervention by the users. Prior to releasing the extension to users, the store sends the extension to CloakX for preprocessing. During CloakX's preprocessing, CloakX installs Droxy and generates a cloaking-template for the extension. The cloaking-template contains a configuration file that identifies the static variable replacements necessary for WARs, IDs, and class names. When a user requests a preprocessed extension, CloakX uses the cloaking-template to quickly generate and implement random WAR, IDs, and class names for the current user.

3.4 Evaluation

Altering extensions without modifying the browser or relying on extension developers to make changes is a complex process, and while CloakX is a prototype and does not cover every possible scenario, we wanted to evaluate its current effectiveness. Thus, in this section we evaluate the efficacy of CloakX by (1) testing the breakage introduced by its use (2) the detectability of the cloaked extensions and (3) the performance of the cloaked extensions.

In November 2017, we extracted 59,255 extensions from the Chrome Store. Of those, we identified 13,693 extensions with only WAR fingerprints; however, 67 of the extensions had errors that prevented them from loading. Next, we identified 2,537

extensions having only DOM fingerprints, but Chrome could not load nine of the extensions. The last set of 2,786 extensions had both WAR and DOM fingerprints, one of which would not load in Chrome.

3.4.1 Functionality Experiments

Testing the functionality of a large set of applications is subject to two problems. First, the tests must explore all the relevant execution paths in the application. Second, the tests should test the entire set of applications. Furthermore, any testing approach will leave code unexplored and applications uncovered, and thus the results form an estimation of functionality breakage. In this work, we perform two different experiments to address both of these challenges: a low-fidelity and a high-fidelity experiment.

The low-fidelity experiment tested the entire population and the high-fidelity experiment randomly sampled from the population. The low-fidelity experiment automatically exercised the original and cloaked extensions and compared the error messages generated by each. The low-fidelity experiment provides a lower bound on the breakage across the entire population. The high-fidelity experiment involved manually—and extensively—exercising the extension, which provided deeper coverage of the extension's functionality. Due to the time-consuming nature of each high-fidelity run, we used a random sample of the extensions from each population.

3.4.1.1 Low-fidelity Functionality Experiments

To measure functionality breakage introduced by CloakX broadly across all extensions, we performed automated experiments that measured the change in errors from the original extension to the cloaked extension. To execute the experiment, we created a headless browser session using Selenium's ChromeDriver with full logging enabled, which includes errors from the extension's content scripts. Next, we visited a triggering web page, which is similar to the webpage used by XHound to activate the extension's functionality. In addition, for those extensions with DOM fingerprints identified by XHound, the triggering webpage also included dynamically generated triggers. After the page loaded, the browser waited 30 seconds for any delayed actions to execute. Other than the static and dynamic triggers, the automated experiments do not simulate additional user actions, which might be necessary to execute all the extension's functionality. These steps comprise a *run*, which is completed once for the original extension and once for the cloaked extension.

After both runs finish, we compared the severe JavaScript error messages between the two runs. If the cloaked extension generated the same errors, then the extension passed. Otherwise, if the cloaked extension generated any new or different errors, then the extension failed. Because the automated tests exercise limited functionality and only compare errors, this experiment represents the best case scenario (i.e., the lower bound) on the errors introduced by CloakX. However, the automation allowed us to run the experiment across the entire population.

Table 1 shows the results for WAR and DOM cloaking separately. Note that at the time we ran the experiments, which took place several months after collecting the extensions, some of the original versions stopped working because of Chrome browser

Table 1. Automated Test Results.

Extension set	**Total**	**Tested**	**Passed**	**Results**	
				Pass	***Fail***
WAR Fingerprintable	13,693	13,626	13,493	99.02%	.98%
DOM Fingerprintable	2,537	2,526	2,493	98.69%	1.31%
WAR & DOM Fingerprintable	2,786	2,785	2,727	97.92%	2.08%
Totals	**19,016**	**18,937**	**18,713**	**98.82%**	1.18%

updates, obsolete back-end servers, etc. As a result, we only tested working extensions and, therefore, the results only contain errors introduced by CloakX.

In the low-fidelity experiments, CloakX retained equivalent functionality for 99.02% (13,493) of the WAR fingerprintable extensions, 98.69% (2,493) of DOM fingerprintable extensions, and 97.92% (2,727) of WAR and DOM fingerprintable extensions. For the WAR fingerprintable extensions, we found that the most frequent cause of the failures was the loading of WARs from remote websites. For the DOM fingerprintable extensions, most of the new error messages generated by the cloaked extensions were severe JavaScript errors caused by (1) extensions loading remote content or (2) missing functionality in Droxy. For the WAR and DOM fingerprintable extensions, we found the same errors as seen in the WAR and DOM only tests. To verify the WAR and DOM cloaking did not interfere with one another, we also ran this group using only one of the modifications at a time. The total number of errors was the same for the joint run as it was for the two additional runs with the single modifications, which indicates the modifications did not interfere with one another.

3.4.1.2 High-fidelity Functionality Experiments

The high-fidelity experiments consisted of manually exercising and evaluating the operation of the cloaked extensions. The high-fidelity evaluation was inspired by the methodology used by Snyder et al. [130]. This methodology focuses on the extension's operation from the perspective of the user. If the cloaking process introduces an error, but the user does not perceive a difference in the extension's operations, then we deem the extension passes. This method of evaluation exercises much more of the extension's code than the automated tests and it provides an additional metric that evaluates the actual operation of each extension. The high-fidelity experiments were performed by the authors using the testing framework detailed next.

We built a custom framework to methodically follow a four-phase evaluation of each extension and advise the tester on the current step in the process. In phase one, the framework loads the original extension and gives the user five minutes to understand its basic operation (including the time necessary to read the extension's description in the Chrome Store). In phase two, the framework reloads the original extension and the user exercises its functionality for five minutes. In phase three, the framework loads the modified extension and the user spends five minutes completing operations similar to the ones completed in phase two to verify it is still operational. In the last phase, the user records any notes on the evaluation and chooses whether the extension passed or failed.

Similar to the automated tests, we divided the extensions into three groups based on the type of fingerprints they emitted. As a result, the populations for each of the high-fidelity tests were as follows: 13,626 WAR fingerprintable extensions, 2,526 DOM fingerprintable extensions, and 2,727 WAR and DOM fingerprintable extensions.

Table 2. Manual Test Results.

Extension set	**Random**	**Top 25**	**Overall**
	Pass/Fail	***Pass/Fail***	***Pass/Fail***
WAR Fingerprintable	25 / 0	25 / 0	50 / 0
DOM Fingerprintable	24 / 1	24 / 1	48 / 2
WAR & DOM Fingerprintable	24 / 1	24 / 2	47 / 3

To create samples for these groups, we created both random and systematic samples containing 25 extensions each. We created the first sample by randomly selecting 25 extensions from the population. We formed the systematic sample by selecting the top 25 most popular extensions based on the number of downloads listed on the Chrome Web Store. Throughout the manual tests, if we could not test an extension because the original version was broken or it was only available in a foreign language, then it was discarded and another one was selected according to the associated sampling method. The resulting samples contained quite a bit of diversity between the extensions. Although we found a few instances of overlapping functionality, we kept these extensions in the samples. However, when we found a duplicate extension, we discarded the duplicate and tested a different extension. Some example extensions included in the test samples included a utility for those who are color blind, a search bar tool, a product search by image, a data extraction tool, and a gesture utility for navigation.

Out of all 150 experiments, 145 of the cloaked extensions retained equivalent functionality (see Table 2). All of the WAR fingerprintable extensions retained their functionality. 96% (48 out of 50 extensions) of the DOM fingerprintable extensions and 94% (47 out of 50 extensions) of the WAR and DOM fingerprintable extensions retained their functionality.

After analyzing the broken extensions, we found three different causes for the broken extensions.

Remote source code using original resource name. The extension loads remote Facebook SDK, which looks for obfuscated ID and class values.

Extension relies on hardcoded values that Droxy alters. An extension relies on hardcoded logic that expects its content scripts to appear in a specific order. However, Droxy must be the first content script, which changes the position of all of the extension's original content scripts, and in one case, it broke the extension.

Droxy implementation limitation. Droxy does not currently support recursive iframe sourcing, *cloneNode*, and some advanced CSS rules that the `cssutils` Python library fails to properly parse.

With engineering improvements to Droxy, we can remediate each of the errors listed above and increase the success rate. For the remote source code, Droxy could intercept the remote source code request and parse it before it is executed. This, of course, would add additional performance overhead. The hardcoded logic could be rectified by overriding the methods that accesses the content scripts. The implementation limitations can be addressed by adding logic to support them into Droxy.

3.4.2 Detectability Experiments

The detectability experiments evaluated the efficacy of the cloaking against an extension tracking webpage. In the first experiment, the tracker used anchorprints to detect extensions with either WAR or DOM fingerprints. In the second experiment, the tracker used structureprints to detect the extensions with DOM fingerprints. In the third experiment, we investigated the use of behaviorprints to detect cloaked

extensions. Last, we explored different methods for detecting the use of CloakX on an extension.

For the first three experiments, we set the fingerprint matching threshold to three. To meet the matching threshold, the tracker must be able to match the extension's fingerprint to three or fewer extensions in its repository. When the tracker meets the matching threshold, it has successfully detected the extension.

We chose a threshold of three because thresholds higher than three showed a sharp decrease in the tracking benefit gained from an extension detection. The matching threshold represents the number of extensions that match a structureprint. The best threshold depends on the requirements of the web tracker and the resources available. The main purpose of the threshold for our experiments was to balance the search time complexity of the fuzzy searches with the increase in the matching of cloaked extensions. For example, by raising the threshold to 20, the web tracker matches three additional cloaked structureprints (one of which matches 18 extensions).

3.4.2.1 Detectability Experiment Using Anchorprints

The anchorprint detectability experiments focused on detection using WARs, IDs, and class names. In the first phase of the experiment, we harvested the anchorprints of the extensions. Next, we loaded each of the original extensions and used a tracking webpage to verify that the extensions were detectable using the anchorprint. Finally, we loaded each of the cloaked extensions and used a tracking webpage to evaluate the detectability of the cloaked extensions using its anchorprint. For a successful detection, the tracker must meet the matching threshold.

In our experiment, we found that none of the cloaked extensions were detectable

using their WARs, IDs, and class names after cloaking. In the first phase, we harvested 17,833 anchorprints, which includes 16,411 extensions with WAR fingerprints and 1,422 that have DOM fingerprints with IDs and classes. However, we chose to limit the testing to the 17,678 extensions that could be executed after being cloaked and assumed that the 155 broken extensions were detectable (thus providing a lower bound on detectability).

In the second phase, we matched 17,534 of the 17,678 original extensions. The ID and class name functionality of the tracker failed to match 144 extensions because it either failed to trigger the extension's anchorprint or it found too many matching extensions. The ID and class name tracker did not find matches for 26 extensions because those extensions required dynamic triggering and the tracker could not use dynamic triggering and still extract the anchorprint; thus, the extensions did not inject their anchorprint into the webpage. The remaining 118 extensions did not count as a detection because the IDs and class names matched more than three other extensions, which exceeded our threshold for a detection.

Initially, the WAR functionality of the tracker failed to find 956 of the WAR fingerprinted extensions using `XMLHttpRequest` because none of the WAR declarations in the manifest file existed in the extension. However, we discovered we could reliably match these extensions by timing how long it took for three WAR requests to return. The first request is for the declared but missing resources of the extension. The second request was for the extension's `manifest.json`, which was not declared as a WAR. The third request was for a randomly generated resource that does not exist in the extension and is not a WAR. If the missing request (i.e., the first) takes the longest to return, then the extension has the resource defined as a WAR but the resource does not exist in the extension. Thus, we improved the tracker such that if the tracker

failed to match an extension using any of the WARs, then it performs these three requests for each of the WARs in the 956 extensions and if the first request takes the longest it has detected the extension.

In the third phase, we were able to detect 96 of the cloaked extensions using their anchorprints. After investigating several extensions that were detected, we found that matches occurred because CloakX was not translating the ID and classes for the extensions due to errors introduced through the cloaking process. In other words, the experiment found 96 additional cloaked extensions that did not maintain functionality equivalent to their original versions. Thus, with the additional errors but no actual matches, we found that 98.55% (17,582) of the extensions were undetectable using anchorprints.

3.4.2.2 Detectability Experiment Using Structureprints

The structureprint experiment tested the detectability of cloaked extensions using exact and fuzzy matching to detect the extensions. In the first phase, we ran each of the 5,311 DOM fingerprintable and WAR and DOM fingerprintable extensions through XHound to gather the structureprints. In the next phase, we ran each of the 5,223 cloaked extensions through XHound to gather cloaked fingerprints. We considered the extensions that failed the automated tests as detectable. Similar to the WAR detection experiments, we did not test the broken extensions, but we assume that they were detectable. In the last phase, we used the structureprints generated in phase one to match the cloaked fingerprints.

The accuracy and precision of detecting structureprints varies depending on both (1) the DOM elements used to build the structureprint and (2) the matching technique

Table 3. Structureprint Detection Test Results.

Structureprint Key Type	**Exact Matching**		**Fuzzy Matching**
	Original	***Cloaked***	***Cloaked***
Tags, Attributes, Text	3,756 (71.91%)	91 (1.74%)	217 (4.15%)
Tags and Attribute Values	2,092 (40.05%)	91 (1.74%)	95 (1.82%)
Tags	1,420 (27.19%)	91 (1.74%)	91 (1.74%)

used to identify the extension. Therefore, to explore how CloakX can prevent the detection of various types of structureprints, we ran the last phase several times using three different structureprints (each one representing less information used in the structureprint) and two different matching techniques (one on exact matching and one on fuzzy matching) to ensure CloakX reduced detection for each of them.

The structureprints varied based on the contents used to build the fingerprint. The first type used all the XHound data, in other words, each fingerprint included added and changed tags, attribute names, attribute values, and text data. While these are the most accurate, they are also the most brittle; as a result, it is likely that the accuracy will degrade considerably in a real-world environment with dynamic HTML content and visitors that have several extensions installed. The second type of structureprint used only the tags and attribute names, which means the fingerprint did not use the attributes values or text. The third type of structureprint used only the tags.

For detection, the experiment extracted an extension's structureprint and then used exact and fuzzy matching against the structureprint database to identify the extension. Exact matching worked well for detecting uncloaked extensions; however, due to the preciseness required for an exact match, cloaked extensions evaded exact matching. Thus, we also tested using fuzzy matching with a 90% level of confidence. Fuzzy matching was successful when the match was made with a 90% level of confidence.

Using either matching technique, if the tracker met the matching threshold (three or fewer matches) using the extension's structureprint then we counted the extension as detected.

Overall, we found that cloaking significantly limited the number of extensions detectable using structureprints. With the full structureprints (tags, attribute names, attribute values, and text) and exact matching, we were able to detect 3,756 of the 5,311 original extensions. The reason that 1,555 extensions were undetectable is because the number of matches made using the extension's structureprint exceeded the matching threshold for a detection (a structureprint must match three or fewer extensions for a successful detection). Using the full structureprints on cloaked extensions, none of the cloaked extensions were detected using exact matching and only 126 extensions were detected using fuzzy matching. Using partial structureprints (attributes and tags), we were able to detect 2,092 of the original extensions; however, the cloaked extensions were undetectable using exact matching and only four were detectable using fuzzy matching. Using the tag only structureprints, we detected 1,420 of the original extensions; however, we were unable to detect any of the cloaked extensions using either matching technique.

3.4.2.3 Detectability Experiment Using Behaviorprints

To understand the limitations of CloakX, we performed an experiment to test the detectability of cloaked extensions using behaviorprints. We chose ten of the most popular extensions with structureprints and to avoid duplication we excluded all ad-blocking extensions except AdBlock. In addition, we examined ten extensions that we randomly selected from those with structureprints. By analyzing their structureprints,

we manually created their behaviorprints from portions of the structureprint that remain constant after cloaking.

For the popular extension sample, six of the extensions added elements to the DOM that made them uniquely identifiable. The extensions LastPass, Pinterest Save Button, and Grammarly all add a base64 encoded image to the DOM that makes them uniquely identifiable. The extensions Ghostery, Evernote, and Skype add a `style` tag to the `head` element with features that made them uniquely identifiable. The extension Turn Off the Lights adds a `data-video` attribute. Although the `data-video` attribute is detectable when the extension is cloaked, CloakX randomly includes this attribute even when the extension is not installed, which increases the attacker's false positive rate and makes it more difficult to correctly detect when the extension is truly installed. Even though the cloaked version of AdBlock was detectable, its behaviorprint was not distinguishable from other popular ad-blocking extensions (e.g., AdBlock Plus, uBlock Origin, and AdGuard AdBlocker) because they all perform the same behavior by deleting ads from the DOM and not injecting any other elements into the DOM. Thus, the detection of ad-blocking extensions exceeds the matching threshold for the identification of a user. Ace Script and Honey added `div` tags with an ID, which means CloakX obfuscated the behaviorprint, and the extensions were not detectable.

For the random sample of ten extensions, five extensions were detectable using behaviorprints and five were undetectable. Similar to popular extensions, five of the ten extensions added elements to the DOM that made them uniquely identifiable. For example, two of them added custom text to the web page. Two of the undetectable extensions performed actions on the DOM, which were duplicated by a number of other extensions. Thus, those extensions exceeded the matching threshold and were

undetectable. Finally, the three remaining undetectable extensions only added class names, IDs, and common tags to the DOM, which are obfuscated by CloakX.

3.4.3 Detectability of CloakX

For our last set of experiments, we evaluated three different techniques meant to determine whether an extension was cloaked by CloakX, thus detecting CloakX. These detection experiments were limited to the 2,447 extensions with structureprints that contained at least one ID or class name.

In the first experiment, we created a method for detecting CloakX after analyzing the lengths of the IDs and class names in cloaked and uncloaked extensions. The IDs and class names generated by CloakX were initially six characters in length and comprised of random alphanumeric characters. However, the IDs and class names in uncloaked extensions averaged 15.1 characters for IDs and 15.9 characters for class names. In addition, we discovered that only 62 uncloaked extensions met the criteria of having all their IDs and class names with a length of six (most of which had only one ID or class name). As a result, we created a method for detecting CloakX that marked an extension as CloakX-enabled if all the IDs and class names injected by the extension were six characters in length. Next, we ran the CloakX detector on 2,447 uncloaked extensions and cloaked extensions, thus evaluating 4,894 extensions. The CloakX detector reported 2,509 cloaked extensions, which means we had 62 false positives, 2,447 true positives, and 2,385 true negatives with an accuracy of 98.7%. As a result, we modified CloakX to randomize the length of the IDs and class names it renames. After making this change, we were no longer able to accurately detect the existence of CloakX based on the length of IDs and class names.

As a follow up, the second experiment attempted to identify cloaked extensions by measuring the entropy of the injected ID and class names. Our hypothesis was that the entropy of randomly generated IDs and class names would be measurably different from those chosen by extension developers. We found that the uncloaked ID and class names exhibited nearly the same amount of entropy as their randomized counterparts. As a result, we could not accurately identify the existence of CloakX using the entropy of ID and class names even though the cloaked values contained randomly generated characters.

In the last CloakX detection experiment, we identified the use of CloakX by exploiting popular extensions that both exhibited a behaviorprint and injected an ID or class name into the webpage. In particular, we found Evernote and Grammarly offered a strong behaviorprint and a related ID. Once we identified the existence of the extension's behaviorprint we looked for the ID or class name, if it did not exist then we determined CloakX was likely installed. For instance, Evernote injects a `style` tag with unique elements and it uses an ID for the same `style` tag. When a `style` tag is found that contains Evernote's elements and the `style's` ID is not `style-1-cropbar-clipper`, then the tracker records that it found a cloaked version of Evernote. Similarly, when Grammarly's green icon is detected and the top level `html` tag does not contain a class starting with `gr`, the tracker records that it found a cloaked version of Grammarly. We tested this by running the tracker against all 2,447 uncloaked extensions and the two cloaked versions of Evernote and Grammarly. The tracker accurately identified the cloaked versions of both extensions with zero false positives.

3.4.4 Performance Experiments

CloakX minimally impacts the performance of Chrome in our automated tests. We tested CloakX's performance by randomly selecting 500 extensions that contain structureprints because their cloaking requires more resources. Each individual test loaded Chrome, loaded the extension, and ran a triggering webpage from the local machine, which either triggered a page load event or timed out. We executed the tests ten times on both the original and modified extensions. The tests were performed across 16 cores with each core running at 2.2 Ghz. On average, the original extensions took 12.3128 seconds and used 66,790 KB of memory whereas the modified extensions took 12.3221 seconds and used 67,123 KB of memory. Thus, the average increase in overhead for the cloaked extensions was a .07% increase in execution time (0.0093 seconds per extension) and a .49% increase in memory use (333 KB per extension).

3.5 References

[12] E. Andreasen, A. Feldthaus, S. H. Jensen, C. S. Jensen, P. A. Jonsson, M. Madsen, and A. Moller, "Improving tools for javascript programmers," in *Proc. of International Workshop on Scripts to Programs. Beijing, China:[sn]*, 2012, pp. 67–82.

[13] E. Andreasen and A. Moller, "Determinacy in static analysis for jQuery," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 17–31, 2014, ISSN: 03621340. DOI: 10.1145/2714064.2660214. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2714064.2660214.

[14] E. S. Andreasen, A. Moller, and B. B. Nielsen, "Systematic Approaches for Increasing Soundness and Precision of Static Analyzers," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, no. June, 2017. DOI: 10.1145/3088515.3088521.

[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[34] *Chrome.runtime - getbackgroundpage()*. [Online]. Available: https://developer.chrome.com/extensions/runtime#method-getBackgroundPage.

[36] *Content scripts*. [Online]. Available: https://developer.chrome.com/extensions/content_scripts.

[42] *Detect adblock – most effective way to detect ad blockers*. [Online]. Available: https://www.detectadblock.com/.

[51] *Extension overview*. [Online]. Available: https://developer.chrome.com/extensions/overview.

[59] *Github - tajs*, http://nicolas.golubovic.net/thesis/master.pdf. [Online]. Available: \url{https://github.com/cs-au-dk/TAJSh}.

[85] S. H. Jensen, P. A. Jonsson, and A. Moller, "Remedying the eval that men do," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ACM, 2012, pp. 34–44.

[86] S. H. Jensen, P. a. Jonsson, and A. Moller, "Remedying the Eval That Men Do," *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 34–44, 2012. DOI: 10.1145/2338965.2336758. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336758.

[87] S. H. Jensen, M. Madsen, and A. Moller, "Modeling the html dom and browser api in static analysis of javascript web applications," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ACM, 2011, pp. 59–69.

[88] S. H. Jensen, A. Moller, and P. Thiemann, "Type analysis for javascript," in *International Static Analysis Symposium*, Springer, 2009, pp. 238–255.

[89] ——, "Interprocedural analysis with lazy propagation," in *International Static Analysis Symposium*, Springer, 2010, pp. 320–339.

[92] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, Aug. 2014, pp. 641–654, ISBN: 978-1-931971-15-7.

[99] *Manifest - web accessible resources.* [Online]. Available: https://developer.chrome.com/extensions/manifest/web_accessible_resources.

[100] *Manifest version.* [Online]. Available: https://developer.chrome.com/extensions/manifestVersion.

[108] Nicolas Golubovic, *Attacking Browser Extensions, MS Thesis, Ruhr-University Bochum*, http://nicolas.golubovic.net/thesis/master.pdf, 2016.

[127] *Sizzle javascript selector*. [Online]. Available: https://sizzlejs.com/.

[128] A. Sjösten, S. Van Acker, and A. Sabelfeld, "Discovering browser extensions via web accessible resources," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ACM, 2017, pp. 329–336.

[130] P. Snyder, C. Taylor, and C. Kanich, "Most websites don't need to vibrate: A cost-benefit approach to improving browser security," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 179–194.

[131] O. Starov and N. Nikiforakis, "XHOUND: Quantifying the fingerprintability of browser extensions," in *Security and Privacy (SP), 2017 IEEE Symposium on*, IEEE, 2017, pp. 941–956.

[144] *W3 dom overview*. [Online]. Available: https://www.w3.org/TR/DOM-Level-2-Core/introduction.html.

Chapter 4

FAULT ESCALATION AND FUZZING WEB APPLICATIONS

Abstract

Black-box web application vulnerability scanners attempt to automatically identify vulnerabilities in web applications without access to the source code. However, they do so by using a manually curated list of vulnerability-inducing inputs, which significantly reduces the ability of a black-box scanner to explore the web application's input space and which can cause false negatives. In addition, black-box scanners must attempt to infer that a vulnerability was triggered, which causes false positives.

To overcome these limitations, we propose Witcher, a novel web vulnerability discovery framework that is inspired by grey-box coverage-guided fuzzing. Witcher implements the concept of *fault escalation* to detect both SQL and command injection vulnerabilities. Additionally, Witcher captures coverage information and creates output-derived input guidance to focus the input generation and, therefore, to increase the state-space exploration of the web application. On a dataset of 18 web applications written in PHP, Python, Node.js, Java, Ruby, and C, 13 of which had known vulnerabilities, Witcher was able to find 23 of the 36 known vulnerabilities (64%), and additionally found 67 previously unknown vulnerabilities, 4 of which received CVE numbers. In our experiments, Witcher outperformed state of the art scanners both in terms of number of vulnerabilities found, but also in terms of coverage of web applications.

4.1 Introduction

In this Chapter, we propose Witcher, a novel web vulnerability discovery framework that is inspired by grey-box coverage-guided fuzzing. Our idea is to explore the web application's input space (without solely relying on hard-coded heuristics) by using execution coverage information to efficiently guide the generation of random inputs.

White-box static analysis tools [52], [77], [91], [98] rely on analyzing the web application's source code which is not always available. Moreover, white-box tools typically model the semantics of the specific language, which makes them language-specific, and thus applying those tools to new languages or frameworks require significant effort.

Black-box web application vulnerability scanners [3], [78], [110], [119] do not require source code and can analyze any web application—regardless of the web application's programming language. These tools generate legitimate web application inputs to explore the application and then attempt to infer the existence of vulnerabilities by sending input designed to trigger a vulnerability to the web application. The vulnerability-inducing inputs, however, are significantly constrained as they originate from manually curated strings or templates based on expert heuristics for vulnerability types [25]. As a consequence, black-box scanners will miss vulnerabilities triggered by inputs that are outside the pre-configured strings and templates. In essence, using hard-code vulnerability inducing inputs significantly reduces the ability of a black-box scanner to explore the web application's input space, thus introducing false negatives.

Even worse, black-box scanners can only infer vulnerabilities based on the output of the web application. Such inference can be error-prone. For example, consider a web application that returns an HTTP `500` status code (which denotes an internal

server error). Existing black-box scanners such as Burp [119] use this error code to decide if their vulnerability-inducing input successfully triggered a vulnerability—in the case of a black-box scanner looking for a SQL Injection vulnerability, an HTTP `500` error can indicate that the input caused an SQL error. However, such an error can be caused by other, unrelated issues, such as an implementation bug rather than a security vulnerability. Therefore, inferring a vulnerability from the outside introduces false positives.

Some recent work has introduced the concept of grey-box fuzzers for automatically testing web applications [56], [124]. These tools use coverage information to guide the generation of inputs. These tools have had some success; however, the approaches target only a single language, do not detect SQL or command injection vulnerabilities, and are closed source [56], and are relatively slow [124].

The application of grey-box coverage-guided fuzzing to web vulnerabilities faces a number of challenges. On a high level, the challenges to web fuzzing arise because the web application code—the *target under test* that contains the vulnerabilities of interest—is not the entire *execution object* but is instead a small subcomponent. When fuzzing a binary, the entire binary is both the *execution object* and the *target under test* (i.e., the security analyst is analyzing whether a vulnerability exists anywhere in binary). However, when fuzzing web applications the *execution object* contains three components: the web server, which parses the HTTP request; the web application runtime environment, which uses the input from the web server to generate a response; and the data storage and local executor, which the web application's logic uses to complete the request. For most web applications, the web application runtime environment breaks down into two subcomponents: the web application code and the code's execution environment (e.g., the interpreter or virtual environment). The web

application code, a subcomponent of the web application runtime environment, is the *target under test* that contains the web application's logic and the vulnerabilities. The multi-component aspect and the other non-target components create several of the challenges that impede the use of grey-box coverage guided fuzzing to discover web vulnerabilities.

Detecting the input that triggers a web application vulnerability. Detecting whether an input triggers a vulnerability requires a tool to reason about the system being in a vulnerable program state. When detecting memory corruption vulnerabilities, traditional binary fuzzing uses a segmentation fault as an indication that input sent to the binary transitioned the system into a vulnerable program state. Current black-box scanner approaches use heuristics to infer that a given input triggers a vulnerable program state. Therefore, a key challenge is to create an approach that can identify when input to a web application leads the web application in a vulnerable program state.

Generating feasible inputs for end-to-end execution. As the execution object is composed of a web server and web application code, a successful input must satisfy both components (i.e., be a valid HTTP request for the web server and also include the necessary input parameters for the web application logic). While, in theory, a random input generation scheme will eventually produce feasible inputs, it is critical to design an approach that generates inputs that are both syntactically and semantically valid for the target web application, thus fuzzing effectively.

Collecting effective web application coverage. A strength of grey-box coverage-guided fuzzing for binary applications is that the fuzzer only keeps randomly generated inputs that exercise *new* code of the application and collecting this *coverage* information is a critical part of modern fuzzing [96], [101]. One possible approach for collecting

coverage for web applications is to insert instrumentation into the web application. However, such an approach is not generally applicable to all web application, does not scale, and requires source code, which is not always available. A scalable and web application-independent approach is necessary to address web application coverage accounting.

Mutating inputs effectively. Similar to binary fuzzing, the mutation strategy of a grey-box coverage-guided fuzzer is also important to fuzz web applications effectively. However, little research has been done to study the mutation strategy for web applications. Therefore, we need to create mutation strategies that can generate high-quality new inputs and increase the fuzzing effectiveness.

Witcher is designed to tackle the prior four challenges. It does not require the source code of individual web applications, and we show that effective code coverage is possible with only 1–5 lines of changes to the language's interpreter. This change can then be used for *any* web application that runs on the interpreter. To demonstrate our approach, we implement Witcher support for web applications written in PHP, Python, Node.js, Java, Ruby, and C. For each of these languages, Witcher is able to detect both SQL injection and command injection vulnerabilities.

To demonstrate Witcher's advantages over the current state-of-the-art, we perform a multi-faceted evaluation. We compare different configurations of Witcher, enabling and disabling different features to demonstrate the features' efficacy. We evaluate Witcher on 13 web application with known vulnerabilities and five modern web applications with no known vulnerabilities. Overall, Witcher found 90 vulnerabilities in total, 67 of which were previously unknown. We then compare Witcher's code coverage and vulnerability discovery to Burp [119], a commercial black-box web vulnerability scanner, on nine of the PHP web applications. Last, we compare Witcher's code

coverage to the recently published black-box vulnerabilty scanner Black Widow [49] and the grey-box scanner WebFuzz [124].

In summary, we make the following contributions:

- We create a set of techniques that address the challenges of applying grey-box coverage-guided fuzzing to web applications, and we propose a new framework that enables coverage-guided fuzzing on web applications.
- We develop Witcher, a grey-box web application vulnerability *fuzzer* that can discover multiple types of vulnerabilities from different web applications. Witcher automatically analyzes server-side binary and interpreted web applications written in PHP, JavaScript, Python, Java, Ruby, and C and detects SQL injection, command injection, and memory corruption vulnerabilities (only in C-based CGI binaries).
- We evaluate Witcher to understand the specific impacts achieved by our various techniques, the effectiveness of the approach on real-world web applications, and its applicability to the analysis of non-traditional targets such as IoT devices. In our evaluation, Witcher identified 23 out of 36 known vulnerabilities, which outperforms the state-of-the-art web vulnerability discovery tool. Moreover, in all but one web application, Witcher reached more lines of code than the state-of-the-art scanners Black Widow and WebFuzz. Witcher also identified 67 previously unknown vulnerabilities, which we are in the process of disclosing to the relevant parties.

To support open science and future researchers in the field, we have open sourced our Witcher prototype, our dataset of web applications, and the results of our experiments.

4.2 Background

Before we discuss the details of Witcher, we first introduce web application and injection vulnerabilities, and then provide a high-level overview of automated application testing and coverage-guided fuzzing.

4.2.1 Web Applications and Vulnerabilities

Typically, a web application runs on a web server and interacts with its clients over a network. A client accesses the web application by sending an HTTP request to the web server, which parses and routes the request to the web application. The web application takes the input, performs the appropriate actions, and responds to the request. In this architecture, the web server acts as a gateway to the web application and a web application can be written in any language. Witcher accesses web application resources using either HTTP requests or direct Custom Gateway Interface (CGI) requests.

HTTP Requests. HTTP is a stateless client–server protocol used by web servers [76]. An HTTP request consists of a request line, zero or more header fields, and an optional message body.

Although they are not limited to it[8], web applications typically accept user input through the `Cookie` header (used for establishing stateful-requests), URL query parameters (ampersand-delimited list of `name=value` pairs), and the HTTP body (ampersand-delimited list of `name=value` pairs). For simplicity, we refer to all of the

[8]A web application may parse any aspect of the HTTP request for user input.

methods for transmitting user input (the headers, the URL query string, and the HTTP body) as HTTP parameters or simply *parameters*.

CGI Requests. The Custom Gateway Interface (CGI) enables a web server to directly invoke executable programs by translating an HTTP request into a CGI request (where aspects of the HTTP request are accessible via environment variables and standard input) [75]. Although many web applications replaced CGI with FastCGI, Apache Modules, and NSAPI plugins [1], CGI applications are still extensively used in embedded devices, such as routers and web cameras [29].

Injection Vulnerabilities. Injection vulnerabilities are an instance of code and data mixing [46], and they occur when a web application sends unsanitized user data to an external parser, such as the shell to execute commands or a database to execute a SQL query. A malicious adversary can exploit such a vulnerability by supplying user input that tricks the external parser into mis-interpreting the user-supplied data as code, thus altering the semantics of the parsing.

In a SQL injection vulnerability, an attacker sends a properly formatted payload with SQL code in their input, which is sent to the database as an SQL query. When the database executes the query, it also executes the attacker's injected SQL code. Similarly, in a command injection vulnerability, an attacker creates a payload that causes additional shell commands to execute.

4.2.2 Motivating Example

Consider the PHP web application in Listing 4.1 in the Appendix, which we created based on patterns that exist in real-world web applications and CVEs (described in § 4.5.2). Depending on the page's purpose, it offers the user different form fields. For

Listing 4.1. Example PHP code with three SQL injections that the commercial black-box scanner Burp does not find.

```
1           ...
2           <? $pid = $_GET['pid']; $act = $_GET["act"]; ?>
3           <input name="pid" value="<?= $pid ?>">
4           <input name="pname" value="<?=get_name($pid)?>">
5           <select name="ptype">
6           <option value="dog_red">Red Dog</option>
7           <option value="dog_grey">Grey Dog</option>
8           </select>
9           <input type="hidden" name="act" value="a"/>
10          <?php
11          $pname = $_GET["pname"];
12          $inp = explode('_', $_GET["ptype"]);
13          $tab=$inp[0]; $c = $inp[1];
14          $pid = isset($pid) ? $pid : uniqid();
15          if (count($inp) >= 2 && $act == "a") {
16                  $pid   = $conn->real_escape_string($pid);
17                  $pname = $conn->real_escape_string($pname);
18                  $c = $conn->real_escape_string($c);
19                  $sql  = "INSERT into {$tab} (id, pname, color)";
20                  $sql .= " VALUES ('{$pid}','{$pname}','{$c}')";
21                  $ret   = mysqli_query($conn, $sql);
22          } else if (count($inp) >= 2 && $act == "u"){ //TBD
23                  if (get_name($pid) != null){
24                          $sql = "UPDATE dog SET color= '{$c}' ";
25                          $sql .= "WHERE id = '{$pid}'";
26                          $ret = mysqli_query($conn, $sql);
27                          ...
```

additions to the database, it includes `pname` and `ptype`. For updates to the database, it includes `pid` and `ptype`. However, in this example, the update functionality was removed from the web application front-end but was left in the server side PHP. As a result, the client interface does not give any hint about the update functionality, which makes it unlikely for a black-box vulnerability scanner to trigger the latent PHP update code.

The code in Listing 4.1 contains three SQL injection vulnerabilities that the commercial black-box vulnerability scanner, Burp, does not detect. The first vulnerability exists in the add functionality. A successful attack requires a change to occur in the first half of the `ptype` field, which is used in the SQL statement without being sanitized. The second vulnerability occurs in the latent PHP update code. For an

attacker to exercise the vulnerability, they must discover the update action and use the `color` portion of the `ptype` field to exploit the vulnerability. The last vulnerability requires the use of the add and update functionality because the update code requires the `pid` to exist in the database but does not enforce any limitations on the format of the `pid`. Thus, an attacker inserts the payload into the `pid` field in the database and then triggers an update to exploit the third vulnerability.

A black-box vulnerability scanner will most likely not find any of the three vulnerabilities. It is unlikely to find the first vulnerability because to reach it the `ptype` variable must contain an underscore, which does not exist in the scanner's predefined list of payloads. Next, black-box scanners are unlikely to find the other vulnerabilities because the client interface does not include the value necessary to trigger the update.

Nevertheless, Witcher finds all three of the vulnerabilities automatically. Witcher finds the first vulnerability by mutating valid input to include the underscore and values that will result in a malformed SQL statement. On parsing the malformed SQL, Witcher detects the vulnerability. Witcher finds the other two vulnerabilities because during the fuzzing process it will mutate `act`'s value to 'u' and mark the input as interesting because `act=u` resulted in a previously unseen program state. Witcher then concentrates on the interesting input, which will cause Witcher to trigger the second vulnerability by adding a malformed version of `ptype` and the third vulnerability by using a malformed `pid` value that was stored into the `pid` column using the 'a' action. Although the third vulnerability requires the application to enter a particular

state, Witcher does not analyze the application state; instead, Witcher triggers the vulnerability because the database maintains the proper state between requests.[9]

4.2.3 Automated Application Testing

Automated application testing falls into one of three categories, which vary depending on how much access the testing technique has to the application: black-box, white-box, and grey-box. In black-box testing, the testing runs without access to the internals of the target application [101]. As a result, black-box testing focuses only on the inputs and outputs of the application [43]. For example, a black-box web vulnerability scanner, such as Burp or Skipfish, works from outside a web application to find new inputs [45].

On the other end of the spectrum, white-box tools generate inputs by analyzing the source code of the application with the goal of better understanding the application's semantics [101]. Some examples of white-box analysis include symbolic execution and taint tracking [26], [38], [132]. White-box tools have access to the target application's source. Thus, white-box tools can reason about the internal structure as well as the operation of the application and can evaluate operation without being limited to paths that can be reached during execution; however, they are focused on a particular programming language and often suffer from false positives.

Grey-box testing blurs the line between white- and black-box testing as it runs with limited access to the application. The testing application uses a less-intensive form of either static or dynamic analysis. For example, coverage-guided mutational

[9]Unlike most binary fuzzing targets where each execution is a blank slate, the database preserves state between executions.

fuzzing uses either static or dynamic instrumentation to gather coverage information, which is used to identify input that exercises new execution paths in a program (thus breaking the purely black-box approach).

4.2.4 Coverage-Guided Fuzzing

Fuzzers automatically test applications by inputting test cases and causing the target application to enter different program states. When the fuzzer starts, it receives a set of input *seeds* that it places into a test case queue. The fuzzer then derives new test cases from those in the queue.

To derive a test case from those in the queue, the fuzzer mutates the test case using a variety of mutation strategies. For example, American Fuzzy Lop (AFL) uses deterministic mutation strategies such as bit flipping, integer arithmetic, and dictionary insertion [11]. In addition, AFL uses random strategies such as random splicing and insertion of data from a user-supplied dictionary. After mutating the input, the fuzzer sends the altered input to the target application.

For coverage-guided fuzzing, the fuzzer captures coverage data that approximates the program states to guide test case selection. The fuzzer captures coverage data that approximates the program states that is far less complete than an execution trace. The instrumentation approximates the program states because it is too processing intensive for a fuzzer to capture and analyze a complete execution trace for each execution. The fuzzer obtains coverage data through either static or dynamic instrumentation. For static instrumentation, an analyst compiles the target application's source code with a modified compiler. For dynamic instrumentation, a dynamic instrumentation

tool (e.g., Pin) or a emulator modified to provide coverage data (e.g., QEMU-user) produces coverage information during execution [74], [118].

A coverage guided fuzzer saves a test case when it deems the test case as interesting. The fuzzer tags a test case as interesting when it causes the program to reach a new location or causes the application to emit a fatal signal, such as a segmentation fault, which often means the application entered a vulnerable state.

4.3 Challenges

Inherent challenges exist in creating a grey-box coverage-guided web application vulnerability fuzzer. We group these challenges into those that *enable* automated analysis and those that *augment* the exploration of the input space.

4.3.1 Enabling Fuzzing of Web Applications

Enabling the automated analysis of web applications requires the fuzzer to generate input that will reach the target application and to detect the existence of a vulnerability.

1. **How to detect web injection vulnerabilities?** A fuzzer's goal is to identify when a test case causes the program to enter a vulnerable program state. Typically, the types of faults generated by SQL and command injection vulnerabilities do not culminate in an error signal that a fuzzer can detect and they often occur in a separate process (e.g., the data storage layer). Therefore, we must develop a new approach that will enable the fuzzer to detect SQL and command injection vulnerabilities.
2. **How does the system generate a test case that will exercise an end-to-end**

execution of the web application? Web applications require the test cases to match a semi-structured format to pass both the syntax checks of the web server *and* the semantics of the web application. In contrast, mutational fuzzers generate high-entropy random data that does not effectively explore the input space of applications.

Without enforcing some structure on the test cases, the fuzzer will not be able to explore the state space of the web application. First, if the test case fails to meet the HTTP request format, then the test case will not reach the target web application because the web server will reject it. (see § 4.2.1). Second, the test case must include the parameters expected by the target application. Without the parameter variable names, a reasonable exploration of the target's input space is impossible because a fuzzer would generate billions of test cases to randomly guess a single variable name of only a few characters.

4.3.2 Augmenting Fuzzing for Web Injection Vulnerabilities

Even if a fuzzer meets the prior challenges to enable fuzzing for web applications, adding those features is not sufficient to efficiently explore the target's input space and discover the vulnerabilities. Analysis of applications using a coverage-guided mutational fuzzer is a computationally intensive task and despite numerous resources its use often results in only a portion of the input space being explored. For web applications, this problem is even worse because fuzzers do not receive execution trace information from the targeted web application code and the fuzzer does not effectively mutate the parameter and values.

1. **How to effectively collect coverage of the web application?** Coverage-

guided fuzzers require instrumentation of the target application to gather coverage information. However, in the case of fuzzing applications written in interpreted languages, limited tools exist that allow instrumentation of the target web application. Instead, the fuzzer instruments the interpreter's runtime binary—not the target web application code. As a result, the coverage information reflects the interpreter's code and not the target web application, thus causing the fuzzer to focus on exploring the runtime interpreter's code instead of the web application's code. Although coverage of the interpreter will change with alterations to the web application's execution, a large portion of the coverage data is irrelevant noise that obfuscates the target web application's coverage information. Therefore, to facilitate exploration of a web application the instrumentation must only report the target web application's execution.

2. **How to effectively mutate test cases?** Although efficient for generating test cases for binary application input, the mutation strategies used by fuzzers focus on the creation of test cases with high-entropy that require no context. However, these high-entropy test cases are less effective for the exploration of web application's input state space. Even if the high-entropy test cases are properly formed HTTP requests, the test cases lack efficacy in testing web applications because they fail to take advantage of the contextual information available to the client (e.g., the variable names found in the HTML form fields). Therefore, it is necessary to create new mutation strategies that incorporate the proper format and exploits the context offered by the web application's client interface.

4.4 Witcher's Design

Witcher is a grey-box web vulnerability scanner that uses a coverage-guided

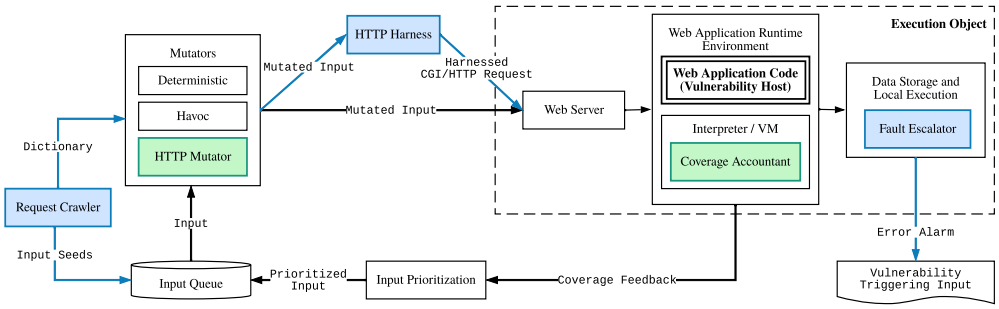

Figure 7. Overview of Witcher. Witcher's components with a blue border are enabling (i.e., necessary to fuzz a web application). Witcher's components with a green border are augmenting and enhance the fuzzer's performance on web applications.

mutation fuzzer to drive the automated exploration of web applications. Witcher is categorized as a grey-box fuzzer because, similar to traditional coverage-guided fuzzing, it relies on coverage data to identify interesting test cases. Other than instrumenting an interpreter for coverage data, Witcher does not perform any analysis on the source code; thus, Witcher can operate without any access to the source because it can run a byte-code version of a web application. As much of the binary fuzzing research uses AFL as a starting point, we chose to use AFL as the base for demonstrating the efficacy of the Witcher framework.

Witcher solves the challenges impeding the use of coverage-guided mutation fuzzing (described in § 4.3) using five additional components. To enable fuzzing for web injection vulnerabilities, Witcher implements the Fault Escalator, the HTTP Harness, and the Request Crawler (the blue components in Figure 7). To augment fuzzing for web injection vulnerabilities, Witcher implements the Coverage Accountant and the HTTP Mutator (the green components in Figure 7).

4.4.1 Enabling Fuzzing for SQL and Command Injection Vulnerabilities

4.4.1.1 Fault Escalator

For a program to be free of vulnerabilities it must be impossible for user-supplied input to transition the program to a vulnerable program state, thus by identifying when this vulnerable program state occurs a scanner can detect a vulnerability in the target application. In traditional binary fuzzing, the vulnerable state results from a memory corruption vulnerability and binary fuzzers detect the transition to a vulnerable state by detecting a segmentation fault signal [101].

We leverage this insight and expand the concept to allow the fuzzer to detect when a program transitions to a vulnerable program state resulting from a SQL or command injection vulnerability. SQL and command injection vulnerabilities occur when user input causes an external parser (shell command parsing for command injection and SQL parsing for SQL injection) to interpret the user input data as code. For example, a SQL injection vulnerability occurs when attacker-controlled input alters the syntax of a SQL query. In a well-formed SQL query, user-controlled input cannot alter the syntax of a SQL query. As a result, we can view a syntax error thrown by an external parser as analogous to the segmentation fault signal that results from a memory corruption vulnerability. This correlation forms the basis behind Fault Escalator: if attacker controlled input causes a syntax error in the external parser, then an attacker can alter the command, and it is more likely than not that an exploitable vulnerability exists. Thus, when the parsing error occurs, Fault Escalator escalates the error to a segmentation fault, which notifies the fuzzer that the current test case caused a vulnerable program state. For example, imagine a PHP application that executes

`mysqli_query($con,‘‘SELECT ID from tbl where ID=’’. $_GET[’id’])` and the fuzzer sets `id=1’`, which results in a malformed SQL statement due to the single quote. When the page executes the SQL statement, the SQL parser will return a parsing error, which is intercepted by Fault Escalator and escalated to a segmentation fault that is detected by the fuzzer.

If an application uses unsanitized input to create a SQL statement or a shell command, then the stochastic input generated by the fuzzer is likely to result in a parsing error. 2Although not every input generated by a fuzzer will cause a SQL syntax error in a vulnerable query, given the stochastic nature of the fuzzer, it is unlikely that an vulnerable query will fail to result in a SQL parsing error during a fuzzing session. This is also confirmed by our experiments: none of the vulnerabilities that Witcher missed are related to false negatives in Fault Escalator.

Command Injection Escalation. For command injection, Witcher implements fault escalation using `dash`'s command parser. The program `dash` is the Debian Almquist shell, which is designed to be POSIX-compliant and as small as possible. `Dash` replaces `/bin/sh` on most Linux systems [58]. Linux uses `/bin/sh`, and its smaller replacement `dash`, when an application executes a shell command. For example, a PHP script using `exec()`, `system()`, or `passthru()`, or a Node.js script using `exec()`[10] send their command to `/bin/sh`, which means that `dash` parses and runs the command. Witcher's version of `dash` (3 lines of code difference from the original) escalates a parsing error to a segmentation fault. Thus, if the application uses unsanitized user input to create a SQL or shell command, then the random data input by the fuzzer into the web application will result in a parsing error and trigger a segmentation fault.

SQL Injection Escalation. Witcher's Fault Escalator implements SQL injection

[10]Node.js's `spawn()` method does not use `/bin/sh`.

escalation for MySQL and PostgreSQL using a technique similar to command injection escalation. To catch the syntax error, Witcher uses `LD_PRELOAD` to hook the `libc` function `recv()`, which is used to communicate with the database. Whenever any response from the database contains a SQL syntax error message, Witcher triggers a segmentation fault.

Fault Escalation is not Limited to Syntax Errors. Although the fault escalation techniques for SQL and command injection detection rely on the existence of a syntax error, the concept of fault escalation applies any type of warning, error, or pattern. For example, Witcher might handle file inclusion by overriding libc's `open` function and escalating an error when the filename parameter contains non-ascii values.

Memory Corruption Vulnerabilities. Witcher detects memory corruption vulnerabilities in CGI binary applications without the aid of fault escalation. This occurs because the fuzzer inherently detects memory corruption vulnerabilities when executing a binary application due to the segmentation fault triggered by the input.

Bugs and Vulnerabilities. Similar to a segmentation fault, the occurrence of a syntax error in a SQL statement or shell command resulting from user input signifies a bug that should be fixed and is highly likely to be vulnerable. In our evaluations, the occurrence of a syntax error signified a problem with the validation or sanitization of user input, which often meant the existence of SQL or command injection vulnerability.

However, it is possible that, due to constraints on user input, an attacker is unable to leverage the syntax error to exploit the SQL injection or command injection. For example, a web application that restricts an unsanitized parameter to be only one character, might not represent an exploitable vulnerability, but rather a bug. For this reason, we will label any escalated fault as either a vulnerability or a bug, depending on whether we confirmed the vulnerability was *exploitable* or *not*.

Cross-site Scripting Vulnerabilities. The fault escalation technique leverages the randomness generated by the fuzzer to identify critical vulnerabilities in the *server environment*. Unfortunately, cross-site scripting vulnerabilities do not readily fall into this category (browsers are very forgiving in their parsing of HTML and therefore it can be difficult to reliably and quickly detect an cross-site scripting in HTML). As a result, we choose to focus on command injection and SQL injection. Moreover, SQL and command injection vulnerabilities represent a class of vulnerabilities that mutation-based fuzzers could not readily detect prior to our work.

4.4.1.2 Request Crawler

The Request Crawler (Reqr) operates as a black-box crawler that automatically discovers HTTP requests and parameters. Reqr extracts HTTP requests from all types of web applications including web applications that rely heavily on client-side JavaScript to render the web application's interface, links, forms, submissions, and requests (e.g., Rconfig, Juice Shop, and WebGoat in § 4.5).

Reqr operates similar to black-box vulnerability scanners: it is given an entry point URL and optionally valid login credentials and the login URL. Reqr uses the Node.js library Puppeteer (an API used to control Chromium) to simulate user actions and capture requests [121]. After Reqr starts, it will login to the web application (if required) and load the entry point. Once a page is loaded, Reqr statically analyzes the rendered HTML to identify the HTML elements that create HTTP requests or HTTP parameters, such as `a`, `form`, `input`, `select`, and `textarea`. Next, Reqr listens for HTTP requests while simulating user events (e.g., mouse clicks, entering values into form fields, and scrolling the page) both systematically and randomly. Reqr

systematically fires the events by targeting every HTML element that accepts user events. In addition, Reqr randomly fires user input events (e.g., clicks, form fills, scrolling, and typing) using the Gremlins testing tool [69].

When Reqr completes, it creates a file containing all the request information. Witcher uses the request information to create the fuzzer's seeds and to build the fuzzer's dictionary.

4.4.1.3 Request Harnesses

Witcher's HTTP harnesses translates fuzzer generated inputs into valid requests. Due to the different execution models, Witcher has a different harness design for PHP and CGI binaries than it does for Python, Node.js, Java, and QEMU-based binaries. For PHP and CGI web applications, Witcher translates from the fuzzer input format into a CGI request. For Python, Node.js, Java, and QEMU-based binaries Witcher translates fuzzer's input into an HTTP request (see § 4.2.1).

CGI Harness. PHP (via `php-cgi`) and CGI binaries use the same harness because both rely on a CGI request and the invoked endpoint runs to completion once invoked. For PHP and CGI binaries, the HTTP harness uses `LD_PRELOAD` to create a fork server that starts the interpreter or the binary just before it processes the input. The harness receives each new input from the fuzzer, translates the input into a CGI request, and then transmits the request into a newly forked process.

HTTP Request Harness. Witcher fuzzes the other interpreted languages and the QEMU-based web applications through their associated web server by leveraging an HTTP request harness. The HTTP request harness decouples the fuzzer from the targeted platform and enables the fuzzer to work on applications that it does not

automatically support. For example, AFL cannot fuzz a Node.js web application that uses `Express` because the application runs indefinitely waiting for new requests and is multi-threaded.

The HTTP request harness creates a bridge between AFL and the web server to leverage the web server's interface to the web application. The HTTP harness includes its own fork server that increases the request submission throughput. The harness receives input from the fuzzer, it translates this input into a well-formed HTTP request, and sends the HTTP request to the web server. Last, when Fault Escalator detects a SQL statement or shell command that causes a syntax error, Fault Escalator sends a segmentation fault to the HTTP harness process, which the fuzzer automatically detects.

Translating Fuzzing Input into a Request. Both the CGI Harness and the HTTP Request Harness act as translators between the fuzzer and the web application. Witcher automatically creates seeds for the fuzzer that follow a null-terminator delimited format. The seeds include fields for cookies, query parameters, post variables, and other header values. As a result, the fuzzer creates test cases based on the format, which the harness then translates into the appropriate request type.

In addition to handling the fuzzer's input, the harness sets a few other parameters for the output request. The harness keeps the request path static for each instance of the fuzzer, which means Witcher fuzzes a single URL at a time. In addition, the harness adds any session cookies, query variables, or post variables that are necessary for the web application to operate correctly. For example, most of the endpoints in the OpenEMR web application require a valid login session. As a result, prior to invoking the fuzzer, Witcher inputs valid login credentials to generate a valid session cookie, which the harness includes in every request.

4.4.2 Augmenting Fuzzing for Web Injection Vulnerabilities

4.4.2.1 Coverage Accountant

Witcher's Coverage Accountant (inside the Interpreter block on the right-side of Figure 7) provides byte-code execution coverage information to the fuzzer for the interpreted languages PHP, JavaScript, Python, and Java as well as web applications that can be executed using QEMU-user or QEMU-system. Witcher uses the Coverage Account because trying to fuzz a web application by instrumenting the interpreter results in a significant amount of noise. For example, when we used AFL's standard approach of instrumenting the interpreter for a simple web page that had six unique paths the fuzzer reported that it found over a thousand unique paths. The discrepancy occurs because by instrumenting the interpreter the fuzzer focuses on test cases that alter the interpreter's execution paths; however, changing the interpreter's execution path does not usually translate to the target web application. Even though many of the paths identified by the fuzzer do not provide additional coverage of the web application code, the fuzzer stores and attempts to mutate each of the test cases because they changed the execution of the interpreter. The increased number of *equivalent* test cases prevents the fuzzer from making meaningful progress exploring the target web application. Therefore, Witcher created the Coverage Accountant to more accurately capture the web application's execution paths.

Interpreter Instrumentation. Despite the different interpreter architectures, the instrumentation of the byte-code is similar between them. The interpreter reads the source file and translates the code into byte-code instructions. Next, the interpreter executes the instruction.

During the execution of an instruction, the augmented interpreter calls Witcher's code coverage library function. Witcher's library function receives the line number, opcode, and parameters of the current byte-code instruction. Witcher then updates the fuzzer's coverage information using the line number and opcode of the current and prior instructions.

Witcher's interpreter instrumentation targets the web application. In Listing 4.1, the code has six visible paths plus several latent paths that occur within the functions `$_GET()`, `mysqli_query()`, and `uniqid()`. Thus, with Witcher's PHP instrumentation the fuzzer will find six paths it deems unique.

CGI Binaries. In addition to interpreted languages, Witcher supports fuzzing CGI binaries. For CGI binaries, Witcher uses AFL's instrumentation when the binary's source code is available. When its source code is unavailable, Witcher's fuzzer uses dynamic instrumentation via QEMU [123]. Although the QEMU-user modifications for instrumentation are already included with AFL, Witcher makes additional modifications to QEMU-user to enable fault escalation. For QEMU-system, Witcher's modifications target the data structures used to store QEMU's intermediate language, which is processed similarly to the byte-code used by the interpreted languages.

Beyond AFL. Witcher uses AFL as the coverage guided mutational fuzzer; however, the Witcher framework can incorporate more advanced fuzzers. If a new fuzzer uses an improved technique for instrumentation, such as PTrix [30], or mutation, such as Tfuzz [114] or AFL++ [53], then Witcher can incorporate the fuzzing tool while still employing the web crawling and fault escalation to detect a wider set of vulnerabilities than either of those tools could do alone.

4.4.2.2 HTTP-specific Input Mutations

We modified AFL by adding two new mutation stages that focus on manipulating HTTP parameters. The purpose of these mutations is to inject parameters into the inputs more quickly than standard AFL and to share/swap values at the variable level instead of treating the parameters as a mere sequence of unstructured bytes. In effect, the mutators reduce and modulate AFL's entropy in a way that is more consistent with the syntax and semantics of web applications.

HTTP Parameter Mutator. The HTTP Parameter Mutator cross-pollinates unique parameter name and values between the interesting test cases stored in the fuzzer's queue. Witcher fuzzes one URL endpoint at a time; however, an interdependency often exists between the variables of different test cases. By cross-pollinating the parameters, the fuzzer provides targeted test cases that are more likely to trigger new execution paths than random byte mutations. For example, in Listing 4.1 if a test case contains `act=a` and another contains `ptype=dog_red`, then by combining them, the fuzzer would reach the vulnerable code.

HTTP Dictionary Mutator. The HTTP dictionary mutator decreases the number of executions necessary to pair the current input with the variables in the dictionary. Many endpoints serve multiple purposes, as a result, an endpoint may have several requests that use different HTTP variables. For a given endpoint, Witcher places all the HTTP variables discovered by Reqr into the fuzzing dictionary. The HTTP dictionary mutator takes advantage of the contextually similar variables by mixing and matching them with the current request. The HTTP dictionary mutator does this by randomly selecting one to ten variables from the dictionary and adds them to the current test case.

4.5 Evaluation

In this section, we aim to answer the following research questions through the evaluation of Witcher:

RQ1. How effective are Witcher's augmentation techniques at exploring the web application and identifying vulnerabilities? Do both augmentation techniques contribute to fuzzing (§ 4.5.1)?

RQ2. How effective is Witcher at identifying vulnerabilities in web applications (§ 4.5.2)?

RQ3. How does Witcher's code coverage and vulnerability discovery compare to a commercial black-box vulnerability scanner and cutting-edge vulnerability scanners (§ 4.5.3)?

4.5.1 Witcher Augmentation Techniques Evaluation

To better understand the impact of Witcher's augmentation features on web application fuzzing, we evaluate Witcher with different configurations and test them on two data samples. The first is a microtest using 10 self-created PHP scripts, and the second is OpenEMR, a real-world web application.

Recall that we designed two fuzzing augmentation techniques: coverage accountant and HTTP mutator. In this experiment, we used Witcher with four different configurations:

AFLR does not have coverage accountant or HTTP mutator. This configuration is meant to be a baseline against Witcher with fuzzing augmentation.

AFLHR has HTTP mutation yet does not have coverage accountant.

WiCR has coverage accountant yet does not have HTTP mutator.

WiCHR has both coverage accountant and HTTP mutator.

4.5.1.1 Microtest Evaluations

In the microtest evaluation, we ran each configuration on a set of ten PHP scripts designed to test the capabilities of Witcher. Each of the scripts includes a single path that reaches an injection vulnerability. The evaluation of a script with a particular configuration ran until either the target injection was reached or four hours elapsed, whichever occurred first.

The dictionary simulated the output generated by Reqr and it included the parameters used by each of the scripts, plus 100 unrelated parameters to simulate unused variables. Each script and configuration were run five times to stabilize the results.

Each of the microtests targeted the functionality of Witcher's components or added additional difficulty.

Listing 4.2. Excerpt from Post-2 in the microtest. The code is simliar for Post-5, Post-10, Cookie-5, and Get-5 scripts.

```
1    ...
2    if(isset($_POST['nv1'])) {
3            if(isset($_POST['nv2'])) {
4                    $ret=mysqli_query($con,"SELECT * FROM tbl
5                    WHERE ID='$_GET['vul']'");
6                    ...
```

Listing 4.3. Excerpt from Equals-1 microtest.

```
1    ...
2    if($_GET['nv1'] == "YYYY") {
3            $ret=mysqli_query($con,"SELECT * FROM tbl
4            WHERE ID='$_GET['vul']'");
5            ...
```

The first set of scripts (`post-2`, `post-5`, `post-10`, `get-5`, and `cookie-5`) follow the same general format that tests Witcher's ability to input the type of variable under test. For example, post-2 (Listing 4.2 in the Appendix) executes a SQL statement that directly concatenates the value returned by `$_GET['vul']` (i.e., an unsanitized value) when the functions `isset($_POST['nv1'])` and `isset($_POST['nv2'])` both return true. As a result to pass the test, the fuzzer must provide the post variables `nv1` and `nv2` and the URL parameter `vul` that contains a value that will trigger a SQL parsing error.

Listing 4.4. Excerpt from Loop microtest.

```
...
for($i=0; $i < strlen($teststr); $i++){
        if ($i < $nv1_len){
                if ($teststr[$i] == $nv1[$i]){
                } else {
                        $all_match = FALSE;
                        break;
                }
        } else {
                $all_match = FALSE;
                break;
        }
}
if ($all_match){
        $ret=mysqli_query($con,"SELECT * FROM tbl
        WHERE ID='$_GET['vul']'");
} ...
```

Listing 4.5. Excerpt from FindVar microtest.

```
...
if(isset($_POST['ao3'])) {
        if($_POST['ao3'] == "add") {
                $ret=mysqli_query($con,"SELECT * FROM tbl
                WHERE ID='$_GET['vul']'");
                ...
```

The next set of scripts test Witcher's ability to provide specific variable and values. To reach the vulnerable SQL statement in `select-3`, the variables and values were provided in the dictionary (as though they were harvested by the crawler) because they were provided in the user interface via the `<select>` tags. `equals-1`, `equals-3`, and `loop-10` tests, the values necessary to reach the vulnerable SQL are not provided in the user interface; thus, the fuzzer, must discover the values. `equals-1` (Listing 4.3 in the

Appendix) executes the vulnerable SQL statement that concatenates the unsanitized input variable `$_GET['vul']` when `$_GET['nv1'] == ''YYYY''`, the necessary value `YYYY` was not provided in the dictionary. Similarly, in `equals-3`, the fuzzer must discover three unknown values to reach the vulnerable statement. `loop-10` (Listing 4.4 in the Appendix) evaluates the input using a `for` loop to perform a byte-by-byte comparison instead of using `==` to compare the entire string, which provides the fuzzer some breadcrumbs to discover the unknown value and reach the vulnerable statement. The last test is similar to `equals-1` except the fuzzer is provided the necessary value but not the variable name. `findvar-1` (Listing 4.5 in the Appendix) executes the vulnerable statement when `isset($_POST['ao3'])`; however, `a03` is not provided in the seeds or dictionary. Excerpts from some of the microtest scripts are available in the appendix.

Table 4 shows the overall results for the microtests. Based on the result, we see that AFLR failed to find any of vulnerabilities. It performed poorly because the additional noise from placing the instrumentation in the interpreter greatly reduced the number of cycles through all inputs, which limited the number of dictionary values it explored.

On the other hand, WiCHR performed the best. WiCHR reached the vulnerability a total of 34 times. However, WiCHR was unable to find the vulnerability in 3 of the looping tests because AFL gives less precedence to coverage that contains repeated instructions. Therefore, both augmentation techniques are helpful to increase the effectiveness of web vulnerability discovery, and thus Witcher will include the two techniques in subsequent evaluations.

We used the Mann Whitney U-test to verify that the differences between the configurations were statistically significant [94]. Because we opted to run until first

Table 4. Microtest Comparative Evaluation Results. The values represent the number of crashes reached after five trials that were up to four-hours in duration.

Microtest	AFLR	AFLHR	WiCR	WiCHR
post-2	0	**5**	**5**	**5**
post-5	0	**5**	**5**	**5**
post-10	0	**2**	**5**	**5**
get-5	0	**4**	**5**	**5**
cookie-5	0	**4**	**5**	**5**
equal-1	0	0	**1**	**2**
equal-3	0	0	0	0
findvar-1	0	0	0	0
loop-10	0	0	0	**2**
select-3	0	**4**	**5**	**5**

crash or timeout, we used the sum of elapsed time per trial to calculate the differences between the configurations. The WiCHR configuration took the least amount of time to run on every trial and the improvement versus the other configurations was statistically significant under the Mann Whitney U-test.

4.5.1.2 OpenEMR Evaluations

To evaluate the performance of Witcher's configurations on a real-world web application, we performed a second comparative evaluation using OpenEMR version 5.0.1.7. We used Reqr to identify the application's URLs and input variables.

Next, Witcher fuzzed each of the URLs in five independent trials using each configuration. We excluded AFLR because of its poor performance in the microtest evaluation; thus, we evaluated the remaining 3 configurations AFLHR, WiCR, and WiCHR. We initialized the database and sessions at the start of each trial to aid consistency from run-to-run.

To perform the evaluation, we gathered PHP code coverage data to use in the Mann-Whitney test. We used Xdebug, a PHP extension, to extract PHP code coverage information [41]. Next, we calculated the total lines visited for all scripts using a particular configuration and trial. With the total lines visited, we then compared the configurations using the results from each trial.

Table 5. OpenEMR Results. This table shows the lines of code covered and the vulnerabilities discovered for each of the five trials.

	AFLHR		WiCR		WiCHR	
	Lines	Vulns	Lines	Vulns	Lines	Vulns
Trial 1	23,113	2	29,723	7	30,714	8
Trial 2	23,142	2	29,082	5	30,777	8
Trial 3	23,011	1	29,543	6	30,935	9
Trial 4	23,111	2	29,105	6	30,833	8
Trial 5	23,220	3	29,160	6	30,800	8

Table 5 in the Appendix shows the results: the total lines of code reached using each of the different configurations in each trial. WiCHR consistently executed the most lines of code followed by WiCR and then AFLHR. The differences in performance between the feature sets was statistically significant: the Mann-Whitney U-Test resulted in a `p-value` of 0.01208.

Table 5 also shows the vulnerabilities discovered for each trial and configuration. All the feature sets found vulnerabilities on each trial; however, both WiCR and WiCHR performed significantly better than AFLHR. WiCHR identified the most vulnerabilities on each trial.

4.5.2 Witcher Evaluation

Based on the results of the feature comparison shown in § 4.5.1, we selected the WiCHR configuration to compare Witcher with other web scanning tools. We used as an evaluation dataset a diverse set of web applications written in different languages and running on different platforms: some that have known vulnerabilities and some that were up-to-date with no known injection vulnerabilities. In this evaluation, we manually confirmed each vulnerability by verifying whether the vulnerability was exploitable or not. Excluding the interesting bugs, all the remaining command and SQL injection vulnerabilities were severe because they give an attacker the capability to destroy, alter, and exfiltrate data [109]. For the command injection vulnerabilities, we verified the application executed an arbitrary shell command. For SQL injections, we automatically exploited the vulnerabilities by providing the crash information from Witcher to SQLMap, which gained full control over the database or could execute arbitrary SQL functions.

For the known vulnerable applications we used a set of eight PHP applications, five firmware images (binaries where the source is likely written in C, and the platform is ARM, MIPSEL, and MIPSEB), one Java, one Python, and one Node.js application with a combined total of 36 known vulnerabilities. We searched for public CVEs of SQL injection and command injection vulnerabilities that had working exploits (so that we could verify the existence of the vulnerability), and this resulted in: Doctor Appointment, Login Management, Hospital Management, and rConfig. We selected WackoPicko, OpenEMR, and Juice Shop because of their known vulnerabilities and

Table 6. Web applications used in the evaluation.

Application	Lang. or Platform	Release Date	Ver.	GitHub Stars	Google Results	Lines of Code	Prior Research
OpenEMR	PHP 7	2018	5.0.1.7	1.6k ★		9,443	[4], [103]
WackoPicko	PHP 5	2018	1.0	265 ★		2,510	[40], [44], [49], [50], [93], [97]
Doctor Appt. Sys.	PHP 7	2020	1.0	n/a	≈10 [66]	3,981	-
User Login Mgmt. Sys.	PHP 7	2020	2.1	n/a	≈3 [67]	1,490	-
rConfig	PHP 7	2018	3.9.2	80 ★		48,405	-
Hospital Mgmt. Sys.	PHP 7	2019	4.0	n/a	≈100 [65]	9,443	-
D-Link DIR-823G	C/MIPSEL	2018	1.0.3.B3	n/a		1,585,157	-
D-Link DIR-823G	C/MIPSEL	2018	1.0.2.B5	n/a		1,569,829	-
D-Link DIR-645	C/ARM	2014	1.0.4.B12	n/a		465,324	-
D-Link DIR-825	C/MIPSEB	2015	1.2.10.B1	n/a		542,992	-
Tenda AC9	C/ARM	2018	15.03.05.19	n/a		982,880	-
WebGoat	Java	2020	8.10	3.9k ★		14,761	[48]
FlaskBB	Python	2018	2.0.2	2.0k ★		14,534	[105]
Juice Shop	Node.js	2020	8.1.0	242 ★		26,221	[50], [82]
Thredded	Ruby/Rails	2021	16.16	1.3k ★		4,426	-
phpBB	PHP 7	2021	3.3.3	1.4k ★		318,104	[44], [48], [49], [113]
osCommerce	PHP 7	2017	2.3.4.1	272 ★		44,355	[40], [49], [70], [97], [112]
Wordpress	PHP 7	2021	5.7.1	15k ★		253,183	[24], [44], [48], [49], [113]

use in prior research (see Table 6 in the appendix for prior work that used the same web applications for their evaluation).

We also selected five firmware targets to demonstrate Witcher's ability to fuzz on non-interpreted web applications. We chose D-Link's 825, 823G version 1.0.2B03, 823G version 1.0.2B05, and 645 as well as the Tenda AC9 because the firmware's web server runs in the QEMU emulator, they each have known CVEs, and their CVEs included working exploit scripts. Table 8 shows the known vulnerabilities in all the applications, along with the CVE number (if known) and the vulnerability type.

We also selected up to date versions of web applications used in the evaluation of prior work to ensure that Witcher would fuzz the latest versions of web applications. In particular, we choose phpBB, osCommerce, and Wordpress, each of which were

Table 7. The known vulnerabilities in each web application, the amount that Witcher found, missed, and previously unknown vulnerabilities that Witcher discovered. *Witcher found one input where the user controls a parameter to execve, however we could not determine if it was exploitable so we consider this a bug rather than a vulnerability.

Application Description	Known Vulnerabilities			Unknown Vulnerabilities
	Existing	Found	Missed	Found
Doctor Appt. Sys.	1	1	0	3
Hosp. Mgmt.	5	5	0	43
Login Mgmt.	1	1	0	5
OpenEMR	5	1	4	5
rConfig	2	0	2	11
WackoPicko	3	2	1	0
D-Link 645	1	0	1	0*
D-Link 823G	1	1	0	0
D-Link 823G	1	1	0	0
D-Link 825	1	0	1	0
Tenda AC9	1	0	1	0
FlaskBB	0	0	0	0
Juice Shop	2	2	0	0
osCommerce	0	0	0	0
phpBB	0	0	0	0
Threadded	0	0	0	0
WebGoat	12	9	3	0
Total	36	23	13	67

evaluated in four or more prior publications, and we also added Thredded, a Ruby on Rails web application.

The name of the 18 web applications used in this evaluation are summarized in Table 6 in the Appendix, along with the language or platform of the web application, the release date of the version of the web application tested if known (the oldest was released in 2014), the version, the number of stars on GitHub for the web application (as an estimate of the popularity of the web application), the number of Google results for a custom Google Dork (link to dork given in reference) if the web application's

Table 8. Known vulnerabilities and results from Witcher's evaluation.

Application Description	Name	Type	Results	Reason Missed
OpenEMR	CVE-2019-17197	SQL	Missed	Crawler Missed
	CVE-2019-14529	SQL	Missed	Crawler Missed
	CVE-2019-16404	SQL	Missed	Crawler Missed
	CVE-2018-17181	SQL	Missed	Crawler Missed
	CVE-2018-17179	SQL	Missed	Crawler Missed
WackoPicko	login.php	SQL	Found	
	passcheck.php	Command	Found	
	similar.php	Stored SQL	Missed	Failed to Recall
Doctor Appt.	CVE-2020-29283	SQL	Found	
Login Mgmt.	CVE-2020-25952	SQL	Found	
rConfig	CVE-2019-16662	Command	Missed	
	CVE-2019-16663	Command	Missed	Crawler Missed
Hosp. Mgmt.	CVE-2020-5192	SQL	Found	
D-Link 825	CVE-2020-10213	Command	Missed	Crawler Missed
D-Link 823G	CVE-2018-17787	Command	Found	
D-Link 823G	CVE-2019-15530	Command	Missed	Did not trigger
D-Link 645	CVE-2015-2051	Command	Missed	Did not trigger
Tenda AC9	CVE-2018-16334	Command	Missed	Crawler Missed
Juice Shop	login	SQL	Found	
	search	SQL	Found	
WebGoat	attack2	SQL	Found	
	attack3	SQL	Found	
	attack4	SQL	Found	
	attack5a	SQL	Found	
	attack5b	SQL	Missed	Java Inst Bug
	attack8	SQL	Found	
	attack9	SQL	Found	
	attack10	SQL	Missed	Java Inst Bug
	Adv/attack6a	SQL	Found	
	Adv/challenge	SQL	Missed	PUT not Supported

source is not on GitHub (as another way to estimate real-world usage), the lines of code of the web application, and if this web application was used in prior research.

To run this evaluation, we created Docker containers for each of the web applications, started the web application, and ran Witcher. Witcher's configuration included the entry URL, identification of the login page, the associated credentials, and a selector for the form field. We limited Witcher's crawler to run for four hours, while we fuzzed each URL with two or more input variables for 20 minutes, As a result, the total run time varied depending on the number of endpoints identified by the crawler.

The overview of the results for this evaluation are shown in Table 7. Witcher successfully crawled and fuzzed all of the web applications, ultimately finding a total of 90 unique vulnerabilities of which 67 were previously unknown. All discovered vulnerabilities were from web applications that had known vulnerabilities (i.e., Witcher did not discover previously unknown vulnerabilities in the latest versions of Thredded, phpBB, osCommerce, or Wordpress).

Witcher discovered 23 of the 36 (63.9%) known vulnerabilities; however, Witcher missed 13 (36.1%) vulnerabilities. Table 8 shows the detailed results of exactly which known vulnerabilities were found or missed, along with a brief description of why. In particular, eight vulnerabilities were missed because the crawler was unable to find the URL. Some URLs were not discovered by the crawler because the application required a specific series of steps, such as selecting a patient in OpenEMR (this is the known problem of exploring stateful web application [43], [49]). The crawler also missed URLs when the URL was not included in the web application's user interface, such as in the case of a backdoor URL in the Tenda AC9 firmware. In the WebGoat application, the crawler missed two vulnerabilities due to a bug in the implementation that caused the webserver to unexpectedly crash and another because the HTTP

harness does not currently support the HTTP `PUT` method. It is common for dynamic analysis tools to have a higher false negative rate; nevertheless Witcher's false negative rate of 36.1% is lower than the rates reported in other publications with 47% [106] and 60% [45]. Although Witcher did not find any memory corruption vulnerabilities during the evaluation, Witcher can detect them because a memory corruption vulnerability will often result in a segmentation fault.

As shown in Table 7, Witcher found 67 previously unknown vulnerabilities (65 SQL and 2 command injections). While we plan to responsibly disclosing the unique vulnerabilities that are still relevant and undiscovered, we have already received unpublished CVEs for the OpenEMR SQL vulnerabilities: CVE-2020-11754, CVE-2020-11755, CVE-2020-11756, and CVE-2020-11757.

In addition to the vulnerabilities that Witcher reported, Witcher reported two false positives and three bugs. However, the bugs were interesting because they demonstrate the potential of using high-entropy input for testing web applications.

4.5.3 Grey-box and black-box comparison

Now that we evaluated the effectiveness of Witcher at identifying vulnerabilities in web application in § 4.5.2, we compare Witcher, a grey-box web application vulnerability fuzzer, against the state-of-the-art commercial black-box web application vulnerability scanner Burp [119], the data-driven web application crawler Black Widow [49], and the recently published grey-box crawler and fuzzer WebFuzz [124]. We choose the Black Widow and WebFuzz scanners because of their recency and performance. For example, Black Widow outperformed six other open-source web-vulnerability scanners (Arachni [16], Enemy of the State [44], Skipfish [129], jÄk [113],

Table 9. Results of vulnerabilities discovered Burp and Witcher: the number of vulnerabilities found by Burp (solo), BurpPlus Witcher, and Witcher. Number in () indicates the unique vulnerabilities found by this configuration.

Application	Burp (solo)	BurpPlus Witcher	Witcher
Doctor Appt. Sys.	2 (0)	3 (0)	**3** (0)
Hosp. Mgmt.	13 (0)	13 (0)	**43 (30)**
Login Mgmt.	1 (0)	1 (0)	**6 (5)**
OpenEMR	0 (0)	0 (0)	**5 (5)**
osCommerce	0 (0)	0 (0)	0 (0)
phpBB	0 (0)	0 (0)	0 (0)
rConfig	0 (0)	0 (0)	**11 (11)**
WackoPicko	1 (0)	1 (0)	**2 (1)**
Wordpress	0 (0)	0 (0)	0 (0)
	17 (0)	18 (0)	**70 (52)**

w3af [145], and ZAP [110]). Although we had hoped to compare Witcher's NodeJS fuzzing against BackREST, the authors were unable to share the tool due to proprietary concerns [56]. We limited the evaluations to nine of the applications (shown in Table 6) that were written in PHP, so that we could collect code coverage using the method described in § 4.5.1.2.

Burp Evaluation. To compare our approach with Burp, we evaluate how much code of the target web application is executed and how many vulnerabilities are discovered. Because Burp has its own crawling components, we compare against Burp in two different configurations: (1) Burp (solo) with no changes, where Burp crawls the web application itself, and (2) BurpPlus Witcher, where we provide Burp with the requests derived from Witcher's crawler. Therefore, the comparison of the results between BurpPlus Witcher and Witcher will not be related to the differences between the crawlers, but to the differences in input generated for the applications.

We configured Burp's scan in the same way for both Burp (solo) and BurpPlus

Witcher. When we configured each scan, we chose the built-in configuration for the most complete crawl and the maximum audit coverage, and the most complete crawl was limited to five hours by default. Burp's audit (i.e., finding vulnerabilities) did not have a timeout option and ran until completion.

One of the differences between Burp and Witcher is that Burp rotates URLs and does not focus on a single target URL at a time, instead, it moves through all the URLs multiple times, which increases the likelihood of discovering new parts of a web application due to the state changes caused by another page. However, Witcher focuses on a single page at a time, which means it is less likely to trigger multi-page states.

Table 10. This table shows the difference between code triggered by Witcher and WebFuzz + BurpPlus (with SQL auditing enabled). Although it did change the results by as much as 4,000 lines of code, the change did not change the outcome of the comparison.

Application	$W \setminus (WF+)$	$W \cap (WF+)$	$(WF+) \setminus W$	Inc.	% Inc.
Doctor Appt. Sys.	**74**	1067	10	0	0.0%
Hosp Mgmt.	**71**	3,255	60	6	0.2%
Login Mgmt.	**3**	516	0	0	0.0%
OpenEMR	**22,473**	109,511	19,915	2,063	1.4%
OSCommerce	**3,833**	22,608	3,622	0	0.0%
phpBB	**15,133**	35,730	7,508	964	1.7%
rConfig	**104**	2,456	62	32	1.2%
WackoPicko	50	2,415		15	0.5%
Wordpress	**37,401**	87,610	10,365	4,430	3.3%

We summarized the results of this experiment in in Table 11 and Figure 8. In the code coverage results, Witcher executed more lines of code in every application over Burp (solo) and BurpPlus Witcher. Witcher increased code coverage by more than 100% for four of the applications. One surprising result is the phpBB testcase, where

Table 11. Results of PHP lines of code coverage between Witcher, Burp, BurpPlus, Black Widow, and WebFuzz. Each scanner is compared against Witcher. The $W \setminus B$ column shows the unique lines discovered by Witcher. The $W \cap B$ shows the lines found by Witcher and the other scanner. The $B \setminus W$ column shows the unique lines found by the other tool. If Witcher has the most unique lines, the value is green. If the other tool has the most unique lines then the value is in orange.

Application	Burp (solo)			BurpPlus Witcher			Black Widow			WebFuzz		
	$W \setminus B$	$W \cap B$	$B \setminus W$	$W \setminus B$	$W \cap B$	$B \setminus W$	$W \setminus B$	$W \cap B$	$B \setminus W$	$W \setminus B$	$W \cap B$	$B \setminus W$
Doctor Appt. Sys.	**34**	386	6	**34**	386	13	**209**	211	43	**74**	1,067	10
Hospital Mgmt	**971**	471	8	**1,021**	421	8	**92**	1,350	24	**164**	3,162	54
Login Mgmt	**104**	64	0	**53**	115	3	**37**	131	4	**27**	492	0
OpenEMR	**32,878**	7,859	7	**31,428**	9,309	40	**25,273**	15,464	2583	**25,237**	107,917	17,852
osCommerce	**5,733**	4,024	90	**3,890**	5,867	277	**2,657**	7,100	798	**4,147**	22,635	3,622
phpBB	**3,148**	22,183	851	**14,482**	10,849	1001	**3,210**	22,121	879	**15,483**	35,480	6,544
rConfig	**2,960**	592	15	**2,263**	1,289	15	**458**	3,094	239	**301**	2,259	30
WackoPicko	**343**	399	10	**258**	484	10	72	670		50	2,415	
Wordpress	**37,308**	15,987	50	**27,823**	25,472	1723	**7,036**	46,259	6482	**41,239**	109,076	5,935

the code coverage for BurpPlus Witcher was 52.3% worse than Burp (solo). This was the only experiment where BurpPlus Witcher reached the crawling timeout threshold. As a result, BurpPlus Witcher had fewer end points to investigate, which resulted in fewer lines covered.

As shown in the vulnerability results in Table 9, Witcher discovered 70 vulnerabilities of which 52 were unique between the three configurations. Witcher found the most vulnerabilities for six of the nine applications.

Black Widow and WebFuzz Evaluation. For the Black Widow and WebFuzz evaluations, we only compare the unique lines of code executed because Black Widow and WebFuzz target XSS vulnerabilities while Witcher targets SQL and command injection vulnerabilities. We seeded WebFuzz with the HTTP requests discovered by Witcher in the crawling stage to focus the evaluation on the fuzzers. Due to the nature of Black Widow's crawler, we were unable to seed Black Widow with the same

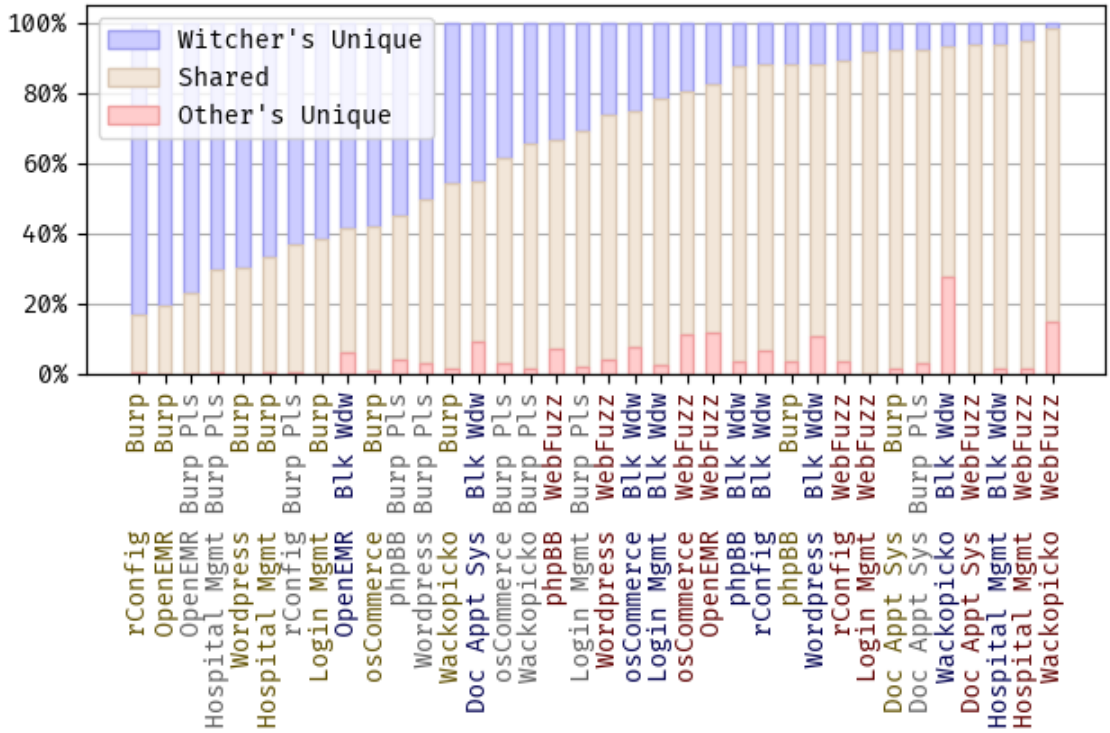

Figure 8. Each column in the stacked bar chart compares the lines found to another tool in an application. Each bar shows a percentage of the total lines found for the tool.

seeds. However, we did add a username and password parameter so that Black Widow would use an existing account.

Black Widow and WebFuzz interweave the execution of different pages while testing a web application. By interweaving execution, the tools may trigger and fuzz new application states that rely on the interdependence between two web pages. For example, the tool may add an item to a cart on the the product page and checkout on the shopping cart page. Currently, Witcher focuses on a single page at a time and is less likely trigger these interdependent states. In addition, Black Widow uses state monitoring to discover new application states.

Witcher's speed and mutation strategy outperformed the other scanners' URL

Table 12. This table compares the increase in code coverage introduced by combining BurpPlus's code coverage data to Black Widow's and WebFuzz's code coverage. It shows the unique lines found by Witcher and the scanner in the first two columns. In the third column, it shows the increase in the scanner's coverage over the results.

Application	Witcher v. Black Widow			Witcher v. WebFuzz		
	$W \setminus BW+$	$(BW+) \setminus W$	Inc.	$W \setminus WF+$	$(WF+) \setminus W$	Inc.
Doctor Appt.	**206**	43	0	**74**	10	0
Hosp. Mgmt.	**88**	24	0	**71**	60	6
Login	**19**	7	3	**3**	0	0
OpenEMR	**25,117**	2,606	23	**22,473**	19,915	2,063
OSCommerce	**2,497**	863	65	**3,833**	3,622	0
phpBB	**2,967**	1,159	280	**15,133**	7,508	964
rConfig	**426**	254	15	**104**	62	32
WackoPicko	66		2	50		15
Wordpress	**6,966**	6,733	251	**37,401**	10,365	4,430

interleaving and state monitoring. As shown in Table 11 and Figure 8, Witcher outperformed Black Widow and WebFuzz by finding more unique lines of code on all the applications except WackoPicko. On WackoPicko, the tools found additional lines of code in the shopping cart functionality. By interweaving the crawling and fuzzing, the tools were able to induce a new state in the shopping cart that exposed an otherwise hidden URL.

Vulnerability Target Bias. In the next evaluation, we tested whether different vulnerability targets may introduce result altering bias into the evaluation that unfairly benefited Witcher in the prior code coverage evaluation. Black Widow and WebFuzz target XSS vulnerabilities; whereas, Witcher targets SQL and command injection vulnerabilities. Black Widow and WebFuzz form valid pre-defined XSS payloads to detect an XSS vulnerability. Although Witcher does not generate an exploit payload—it inputs random bytes and swaps variables that will likely trigger a fault

Table 13. This table shows the difference between the code triggered by Witcher and Black Widow + BurpPlus (with SQL auditing enabled). Although it did change the results by as much as 280 lines of code, none of the additions changed the outcome of the comparison.

	Witcher v. Black Widow					Witcher v. WebFuzz				
Application	$W \setminus BW+$	$W \cap BW+$	$(BW+) \setminus W$	Inc.	% Inc.	$W \setminus WF+$	$W \cap BW+$	$(WF+) \setminus W$	Inc.	% Inc.
Doctor Appt.	**206**	258	43	0	0.00%	**74**	1,067	10	0	0.00%
Hosp. Mgmt.	**88**	1,386	24	0	0.00%	**71**	3,255	60	6	0.18%
Login	**19**	149	7	3	1.71%	**3**	516	0	0	0.00%
OpenEMR	**25,117**	77,499	2606	23	0.02%	**22,473**	109,511	19915	2063	1.36%
OSCommerce	**2,497**	8,947	863	65	0.53%	**3,833**	22,608	3622	0	0.00%
phpBB	**2,967**	38,529	1159	280	0.66%	**15,133**	35,730	7508	964	1.65%
rConfig	**426**	6,206	254	15	0.22%	**104**	2,456	62	32	1.22%
WackoPicko	66	676		2	0.19%	50	2,415		15	0.52%
Wordpress	**6,966**	71,927	6733	251	0.29%	**37,401**	87,610	10365	4430	3.27%

escalation—we cannot guarantee that implicit command and SQL injection driven assumptions did not influence Witcher. As a result, these different payloads and assumptions may introduce coverage bias, which makes the comparison between the tools less equivalent.

To evaluate the bias, we took an approach inspired by the comparison in Enemy of the State [43] where they added the w3af testing component to the state-aware-crawler to control for the vulnerability detection. In our evaluation, we simulated the same control by combining Black Widow's and WebFuzz's code coverage with the code coverage generated by BurpPlus (with only SQL auditing enabled and loading Burp with Witcher's URLs). Combining BurpPlus's SQL only results with Black Widow and WebFuzz did not alter the outcome of any comparison to Witcher making it less likely that the different vulnerability targets unfairly benefited Witcher. In Table 12, BurpPlus added lines covered to most of the web applications for both scanners. However, the additional lines did not change the outcome of the comparisons; moreover, the percent increase (amount of change / total lines covered) was less than 3.3% for all the applications (see Table 13). Thus, for the chosen web applications, it is unlikely that a vulnerability target bias impacted the results.

Performance. To understand the cost in terms of requests per second, we compared the number of requests per second made by Witcher, WebFuzz, Black Widow, and Burp. In the evaluation, we executed Witcher (one core), Burp (max of ten concurrent requests), WebFuzz (a single worker), and Black Widow (one core) for eight hours on each of the PHP web applications.

Table 14. The requests per second of Witcher and WebFuzz on the PHP applications used in the evaluation.

Applications	Witcher Req/s	WebFuzz Req/s	Black Widow Req/s	Burp Req/s
Doctor Appt. Sys.	**539.4**	43.4	3.4	7.3
Hosp. Mgmt. Sys.	**327.4**	26.6	3.0	5.0
Login Mgmt.	**180.9**	112.2	0.9	13.3
OpenEMR	**15.3**	1.5	1.7	5.1
osCommerce	**22.2**	4.7	0.7	2.4
phpBB	**14.7**	1.5	0.7	1.1
rConfig	**52.7**	10.1	3.3	3.3
WackoPicko	**101.9**	2.2	0.2	23.5
Wordpress	**24.4**	0.1	0.5	0.1
Average	**142.1**	22.5	1.6	6.8

Table 14 in the Appendix shows that Witcher sent the most requests per second for every web application. Witcher averaged 142.1 requests per second while WebFuzz averaged 22.5 req/s, Black Widow averaged 1.6 req/s, and Burp averaged 6.8 req/s. Although Witcher outperformed the other tools in code coverage, it issued six times the requests made by the next fastest tool; however, the coverage was not six times better. Thus, by applying a hybrid approach Witcher would likely improve coverage despite the potential cost to the requests per second.

4.6 Discussion

Witcher's use of fault escalation, dynamic request crawling, request harnessing, direct instrumentation, and HTTP-specific input mutations provides a framework for the effective application of coverage-guided mutational fuzzing to web applications. Our evaluation showed the effectiveness of the Witcher components, the ability to identify known and previously unknown vulnerabilities. In addition, we compared Witcher with Burp, Black Widow, and WebFuzz. Witcher outperformed the other tools, finding more vulnerabilities than Burp and covering more of the web applications than Burp, Black Widow, and WebFuzz.

Grey-box versus White-box versus Black-box Scanners. The implementation of grey-box fuzzing for web applications requires less effort to implement than its white-box counterparts. Witcher requires inserting a few lines of code into the target runtime for the language. However, a white-box tool models the semantics of the language. The semantics differ for each language; thus, significant effort is required to initially implement a semantic-driven white-box approach to a different language. For instance, Pixy [90] did not support object-oriented features of PHP, which limits its applicability to modern PHP. Moreover, white-box tools often fail when analyzing real-world code. However, we tested Witcher and grey-box fuzzing using multiple real-world targets, languages, and architectures. Although black-box vulnerability scanners are typically language agnostic, grey-box fuzzers out-performed a commercial black-box tool and two state-of-the-art vulnerability scanners.

4.6.1 Limitations

The current Witcher prototype is limited to discovering SQL injection and command injection vulnerabilities. While these two vulnerability classes represent high-severity vulnerabilities, there are other web vulnerabilities such as cross-site scripting, path traversal, local file inclusion, or remote code evaluation that Witcher does not currently detect.

Another limitation of Witcher is that it can only detect *reflected* injection vulnerabilities—that is, injection vulnerabilities where the untrusted user input flows unsanitized to a sensitive sink during one HTTP request. This is in contrast to second-order vulnerabilities, such as stored SQL injection, where the untrusted user input is *safely* stored by the web application on the initial HTTP request, where it finally flows unsanitized to a sensitive sink while processing a subsequent HTTP request. Although Witcher might be able to detect the vulnerability using Fault Escalator, it would not be able to reason about what input actually caused the vulnerability.

A related limitation is that Witcher does not reason about web application state. A key limitation of the Witcher prototype is that it fuzzes one URL at a time, which does not allow it to reason about or understand multi-state actions in the web application. However, Witcher is able to induce some application states between requests because the web application's database maintains state. Perhaps the techniques proposed in prior work to understand web application state [43], [49] could be applied to Witcher.

4.6.2 Future Work

While Witcher worked well in the evaluations, we see several potential improvements. Witcher could benefit more automation of the initial setup and configuration. Witcher would also benefit from simultaneous crawling and fuzzing that shares results and interweaves the execution of different URLs.

Witcher can be improved to detect other types of vulnerabilities. Witcher could be augmented to detect local file inclusion and path traversal vulnerabilities by (1) creating a honeypot directory (`witchers-honey/`) in each directory of the web application and (2) adding a detector that escalates when a new file is detected in the honeypot directory. Witcher could include XSS vulnerability detection likely at the cost of performance by using a JavaScript engine to render and detect the XSS using WebFuzz's technique.

4.7 Related Work

Recently, three grey-box fuzzing tools have emerged in the literature BackREST [56], WebFuzz [124], and Cefuzz [148]. Witcher is distinguishable from the tools because Witcher supports multiple languages, while BackREST only supports Node.js and WebFuzz and Cefuzz only support PHP. With respect to BackREST, Witcher is more robust because Witcher handles full web applications whereas BackREST focuses on exercising REST APIs. In addition, Witcher is open source; however, BackREST and Cefuzz are closed source and BackREST is unavailable for testing or evaluation. Witcher also differs from WebFuzz because Witcher uses a compiled fuzzer to generate inputs whereas WebFuzz's fuzzer is written in Python, which improves the requests

per second Witcher can make. Next, Witcher tracks execution by adding the instrumentation to the interpreter; whereas, WebFuzz directly modifies the web application's scripts and nearly doubles the size of the scripts. Lastly, Witcher targets command and SQL injection vulnerabilities whereas BackREST and WebFuzz target Cross-site Scripting vulnerabilities and Cefuzz targets remote code execution vulnerabilities.

Several black-box fuzzers exists for fuzzing web applications such as Burp [119], Acunetix Web Vulnerability Scanner [3], IBM AppScan [78], OWASP Zap [110], and Skipfish [129] Arachni [16], Enemy of the State [44], jÄk [113], w3af [145], and Black Widow [49]. Each the the scanners detect injection and other common web vulnerabilities [122]. Most of the tools "fuzz" using predefined heuristics or user-defined rules [21]. However, unlike the black-box tools, Witcher relies execution instrumentation to guide the input generation and fault escalation to detect an injection vulnerability.

AFL CGI Wrapper enables the fuzzing of CGI binaries by receiving input via standard input and translating it into a CGI request [54]. Although the initial inspiration of Witcher's CGI harness came from the AFL CGI Wrapper, it only detects memory corruption vulnerabilities does not identify injection vulnerabilities and it lacks the input generation capabilities of Witcher.

The tool μ4SQLi automatically produces inputs that lead to harmful SQL statements and bypasses application firewalls [15]. μ4SQLi starts with legitimate input and mutates the values using a predefined group of mutation operators that are meant to build new types of SQL injection payloads. Similar to Witcher, μ4SQLi uses a database proxy to monitor the network traffic between the database and the web server so that it detects whether the SQL statement is harmful. μ4SQLi differs from

Witcher because it does not rely on any execution instrumentation to guide input generation and it does not detect command injection vulnerabilities.

KameleonFuzz is a black-box web application fuzzer that detects XSS vulnerabilities [48]. KameleonFuzz attempts to uses an attack grammar and variable mutations to generate XSS payloads along with valid input, which it then submits to the site, and then detects whether the payload landed. It guides the mutations based on a fit score, which is calculated after the input is submitted. KamelonFuzz differs from Witcher because it does not use any execution information to guide the input generation phase and it does not use detect SQL or command injection vulnerabilities.

RESTler and Pythia automatically test REST APIs that have interfaces defined using Sparrow. RESTler uses grammar-based fuzzing and static analysis of API specifications to automatically test REST APIs [21]. Pythia, which builds on RESTler, adds coverage-guided fuzzing and learning-based mutations [20]. However, these tools focus bugs instead of vulnerabilities and only work on Sparrow documented REST apis, unlike Witcher, which works on web applications and detects SQL and command injection vulnerabilities.

Another recent tool, HYDRA, uses user weighting and output monitoring to guide the targeted generation of injection payloads [95]. HYDRA uses context changes to detect the injection vulnerabilities. However, Witcher uses fault escalation to detect injection vulnerabilities. In addition, HYDRA requires more interaction with the user for initialization of inputs and context weighting.

SRFuzzer fuzzes the web interface of router-based IoT devices to detect memory corruption and command injection vulnerabilities. SRFuzzer uses a browser-based crawler to gather the HTTP variables. SRFuzzer then fuzzes the variables and monitors

by testing the device's responsiveness and listening for reverse connections made by crafted command injection payloads.

Although not the primary contribution of Witcher, its device fuzzing offers features not included in SRFuzzer. SRFuzzer and Witcher use similar techniques to identify the HTTP variables. However, Witcher detects the vulnerability from inside the device (which means that the vulnerability trigger does not need to be a syntactically correct command injection). Moreover, SRFuzzer's scaling is limited by the use of the physical device; while Witcher uses an emulated version of the firmware and scales to fuzz in parallel. Finally, Witcher uses instrumentation of the binary to guide fuzzing, while SRFuzzer does not.

Rampart detects denial-of-service attacks using a PHP plugin to measure the execution time of user-created functions to detect anomalous execution performance, which indicates a denial-of-service attack [104]. Witcher's instruction instrumentation is more fine-grained than user-created functions because user-created functions often contain multiple lines of code. Thus, Witcher provides the fuzzer with additional information to guide its analysis than produced by Rampart.

Eriksson et al. created the vulnerability scanner Black Widow [49]. Black Widow uses navigation modeling, traversing, and inter-state dependencies to scan web applications. Witcher uses fault escalation to find SQL and command injection vulnerabilities, but Black Widow is limited to XSS vulnerabilities. Based on the evaluation results, the two approach while producing similar results also seem to activate different portions of the code. Thus, integrating the Witcher and Black Widow approaches will likely result in a more effective tool.

Several fuzzers exist that propose different methods for mutating context-free

grammars or other types of structured data [19], [31], [111], [116], [117]. We plan to investigate their performance on web applications in future work.

4.8 Conclusion

In this chapter, we propose Witcher, a novel web application vulnerability discovery platform that is generalizable to web languages without hard-coded heuristics for testing inputs. Witcher is inspired by coverage-guided mutational fuzzing. To bridge the gap between coverage-guided mutational fuzzing and web application vulnerabilities, we design multiple techniques in Witcher that generate both syntactically-valid and semantically-correct inputs and detect injection vulnerabilities. In our evaluation, we observed that Witcher is able to find both known and unknown web vulnerabilities effectively. Witcher is the first step toward the development of a web application *fuzzer*, as opposed to vulnerability scanners, and we believe this approach is a promising path forward to automatically identifying vulnerabilities in web applications.

4.9 References

[1] [Online]. Available: https://en.wikipedia.org/wiki/Common_Gateway_Interface.

[3] *Acunetix Web Vulnerability Scanner*, 2020. [Online]. Available: https://www.acunetix.com/.

[4] F. Akowuah, J. Lake, X. Yuan, E. Nuakoh, and H. Yu, "Testing the security vulnerabilities of openemr 4.1. 1: A case study," *Journal of Computing Sciences in Colleges*, vol. 30, no. 3, pp. 26–35, 2015.

[11] *American Fuzzy Lop*, 2020. [Online]. Available: https://github.com/google/AFL.

[15] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: An input mutation approach," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 259–269.

[16] *Arachni - Web Application Security Scanner Framework*, 2021. [Online]. Available: https://www.arachni-scanner.com/.

[19] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars.," in *NDSS*, 2019.

[20] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray, "Pythia: Grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations," *arXiv preprint arXiv:2005.11498*, 2020.

[21] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Rest-ler: Automatic intelligent rest api fuzzing," *arXiv preprint arXiv:1806.09739*, 2018.

[24] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *2010 IEEE symposium on security and privacy*, IEEE, 2010, pp. 332–345.

[25] E. Bazzoli, C. Criscione, F. Maggi, and S. Zanero, "XSS PEEKER: Dissecting the XSS exploitation techniques and fuzzing mechanisms of blackbox web application scanners," in *IFIP International Conference on ICT Systems Security and Privacy Protection*, Springer, 2016.

[26] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A taint based approach for smart fuzzing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE, 2012, pp. 818–825.

[29] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware.," in *NDSS*, vol. 16, 2016, pp. 1–16.

[30] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 633–645.

[31] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, "One engine to fuzz'em all: Generic language processor testing with semantic validation," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, 2021.

[38] B. Cui, F. Wang, Y. Hao, and X. Chen, "Whirlingfuzzwork: A taint-analysis-based api in-memory fuzzing framework," *Soft Computing*, vol. 21, no. 12, pp. 3401–3414, 2017.

[40] G Deepa, P. S. Thilagam, A. Praseed, and A. R. Pais, "Detlogic: A black-box approach for detecting logic vulnerabilities in web applications," *Journal of Network and Computer Applications*, vol. 109, pp. 89–109, 2018.

[41] Derick Rethans, *Xdebug: A Debugger and Profiling Tool for PHP*, 2020. [Online]. Available: https://xdebug.org.

[43] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Vulnerability Scanner," in *Proceedings of the 21st Symposium on USENIX Security*, Bellevue, WA, Aug. 2012.

[44] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 523–538.

[45] 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.

[46] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, "deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Nov. 2013.

[48] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: Evolutionary fuzzing for black-box xss detection," in *Proceedings of the 4th ACM conference on Data and application security and privacy*, 2014, pp. 37–48.

[49] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," *proceedings of IEEE SSP 2021*, 2021.

[50] D. Esposito, M. Rennhard, L. Ruf, and A. Wagner, "Exploiting the potential of web application vulnerability scanning," in *ICIMP 2018 the Thirteenth*

International Conference on Internet Monitoring and Protection, Barcelona, Spain, 22-26 July 2018, IARIA, 2018, pp. 22–29.

[52] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward Automated Detection of Logic Vulnerabilities in Web Applications," in *Proceedings of the 19th USENIX Security Symposium*, 2010, pp. 143–160, ISBN: 9781931971775.

[53] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, Aug. 2020.

[54] Floyd Fuh, *AFL CGI Wrapper*, 2020. [Online]. Available: https://github.com/floyd-fuh/afl-cgi-wrapper.

[56] F. Gauthier, B. Hassanshahi, B. Selwyn-Smith, T. N. Mai, M. Schlüter, and M. Williams, "Backrest: A model-based feedback-driven greybox fuzzer for web applications," *arXiv preprint arXiv:2108.08455*, 2021.

[58] V. Gite, *What is Dash (/bin/dash) Shell?* 2020. [Online]. Available: https://www.cyberciti.biz/faq/debian-ubuntu-linux-binbash-vs-bindash-vs-binshshell/.

[65] *Google Search for Uses of Hospital Management System*, 2021. [Online]. Available: https://tinyurl.com/hospitalmanagementsystemuses.

[66] *Google Search for Uses of Login System*, 2021. [Online]. Available: https://tinyurl.com/doctorappointmentsystem.

[67] *Google Search for Uses of Login System*, 2021. [Online]. Available: https://tinyurl.com/usermanagementsystem.

[69] *Gremlins - Monkey Testing Library for Web Apps and Node.js*, 2020. [Online]. Available: https://github.com/marmelab/gremlins.js.

[70] S. Gupta and B. B. Gupta, "Php-sensor: A prototype method to discover workflow violation and xss vulnerabilities in php web applications," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015, pp. 1–8.

[74] *High-performance binary-only instrumentation for afl-fuzz*, 2020. [Online]. Available: https://github.com/google/AFL/blob/master/qemu_mode/README.qemu.

[75] *HTTP State Management Mechanism*, 2020. [Online]. Available: https://tools.ietf.org/html/rfc3875.

[76] *HTTP/1.1 Message Syntax and Routing*, 2020. [Online]. Available: https://tools.ietf.org/html/rfc7230.

[77] Y. W. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S.-Y. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection," in *Thirteenth International World Wide Web Conference Proceedings*, 2004, pp. 40–52, ISBN: 158113844X. DOI: 10.1145/988672.988679.

[78] *IBM/HCL AppScan*, 2020. [Online]. Available: https://www.hcltechsw.com/appscan.

[82] B. Jabiyev, O. Mirzaei, A. Kharraz, and E. Kirda, "Preventing server-side request forgery attacks," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 1626–1635.

[90] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, IEEE, 2006, 6–pp.

[91] ——, "Static Analysis for Detecting Taint-style Vulnerabilities in Web Applications," *Journal of Computer Security*, vol. 18, no. 5, pp. 861–907, 2010, ISSN: 0926227X. DOI: 10.3233/JCS-2009-0385.

[93] N. Khoury, P. Zavarsky, D. Lindskog, and R. Ruhl, "Testing and assessing web vulnerability scanners for persistent sql injection attacks," in *proceedings of the first international workshop on security and privacy preserving in e-societies*, 2011, pp. 12–18.

[94] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[95] M. Leithner, B. Garn, and D. E. Simos, "Hydra: Feedback-driven black-box exploitation of injection vulnerabilities," *Information and Software Technology*, p. 106 703, 2021.

[96] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[97] X. Li and Y. Xue, "Block: A black-box approach for detection of state violation attacks towards web applications," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 247–256.

[98] X. Li, W. Yan, and Y. Xue, "SENTINEL: Securing Database from Logic Flaws in Web Applications," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, 2012, pp. 25–36, ISBN: 9781450310918. DOI: 10.1145/2133601.2133605.

[101] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *arXiv preprint arXiv:1812.00140*, 2018.

[103] Z. McGee and S. Acharya, "Security analysis of openemr," in *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, IEEE, 2019, pp. 2655–2660.

[104] W. Meng, C. Qian, S. Hao, K. Borgolte, G. Vigna, C. Kruegel, and W. Lee, "Rampart: Protecting web applications from cpu-exhaustion denial-of-service attacks," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 393–410.

[105] S. Micheelsen and B. Thalmann, *A static analysis tool for detecting security vulnerabilities in python web applications*, 2016.

[106] R. Mohammed, "Assessment of web scanner tools," *International Journal of Computer Applications*, vol. 133, no. 5, pp. 1–4, 2016.

[109] *OWASP, SQL Injection*, 2022. [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection#:~:text=TheseverityofSQLInjection, Injectionahighimpactseverity.

[110] *OWASP Zed Attack Proxy*, 2020. [Online]. Available: https://www.zaproxy.org/.

[111] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329–340.

[112] G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications.," in *NDSS*, 2014.

[113] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, "Jäk: Using dynamic analysis to crawl and test modern web applications," in *International Symposium on Recent Advances in Intrusion Detection*, Springer, 2015, pp. 295–316.

[114] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 697–710.

[116] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019. DOI: 10.1109/TSE.2019.2941681.

[117] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, IEEE, 2020, pp. 460–465.

[118] *Pin - A Dynamic Binary Instrumentation Tool*, 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.

[119] PortSwigger, *Burp Suite. Application Security Testing*, 2020. [Online]. Available: https://portswigger.net/burp.

[121] *Puppeteer - Node.js library that provides high-level API access to Chrome and Chromium*, 2021. [Online]. Available: https://github.com/puppeteer/puppeteer.

[122] M. Qasaimeh, A. Shamlawi, and T. Khairallah, "Black box evaluation of web application scanners: Standards mapping approach," *Journal of Theoretical and Applied Information Technology*, vol. 22, Jul. 2018.

[123] *QEMU*, 2020. [Online]. Available: https://qemu.org.

[124] O. v. Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, "Webfuzz: Grey-box fuzzing for web applications," in *European Symposium on Research in Computer Security*, Springer, 2021, pp. 152–172.

[129] *Skipfish: Web Application Security Scanner*, 2020. [Online]. Available: https://github.com/spinkham/skipfish.

[132] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution.," in *NDSS*, vol. 16, 2016, pp. 1–16.

[145] *w3af - Open Source Web Application Security Scanner*, 2021. [Online]. Available: http://w3af.org/.

[148] J. Zhao, Y. Lu, K. Zhu, Z. Chen, and H. Huang, "Cefuzz: An directed fuzzing framework for php rce vulnerability," *Electronics*, vol. 11, no. 5, p. 758, 2022.

Chapter 5

CONCLUSION

In conclusion, the transformation of the hacking landscape from an idealistic pursuit to a global battlefield has necessitated an evolution in the skills and tools employed by educators, users, and developers. To address the shortage of qualified cybersecurity professionals, this dissertation proposes integrating live cybersecurity competitions into educational curricula, offering a CTF-as-a-service solution for educators who may lack the necessary resources. For users, CloakX is introduced as a tool to safeguard privacy by countering extension fingerprinting techniques, empowering individuals to enhance their privacy on the Internet. Finally, for analysts, Witcher is presented as a novel framework for discovering web vulnerabilities using grey-box coverage-guided fuzzing, enabling more efficient and effective vulnerability detection. By embracing these innovative approaches, we can better equip ourselves to navigate the complex and ever-evolving world of cybersecurity.

REFERENCES

[1] [Online]. Available: https://en.wikipedia.org/wiki/Common_Gateway_Interface.

[2] *2015 Global Cybersecurity Status Report*, https://www.isaca.org/cyber/documents/Cybersecurity-Status-Report_ifg_Eng_0115.pptx, 2017.

[3] *Acunetix Web Vulnerability Scanner*, 2020. [Online]. Available: https://www.acunetix.com/.

[4] F. Akowuah, J. Lake, X. Yuan, E. Nuakoh, and H. Yu, "Testing the security vulnerabilities of openemr 4.1. 1: A case study," *Journal of Computing Sciences in Colleges*, vol. 30, no. 3, pp. 26–35, 2015.

[5] *Amazon API Error Codes*, http://docs.aws.amazon.com/AWSEC2/latest/APIReference/errors-overview.html, 2017.

[6] *Amazon EC2 Instance IP Addressing*, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-instance-addressing.html, 2017.

[7] *Amazon EC2 Security Groups for Linux Instances*, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html, 2017.

[8] *Amazon Machine Images (AMI)*, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html, 2017.

[9] *Amazon Regions and Availability Zones*, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html, 2017.

[10] *Amazon VPC FAQs*, https://aws.amazon.com/vpc/faqs/, 2017.

[11] *American Fuzzy Lop*, 2020. [Online]. Available: https://github.com/google/AFL.

[12] E. Andreasen, A. Feldthaus, S. H. Jensen, C. S. Jensen, P. A. Jonsson, M. Madsen, and A. Moller, "Improving tools for javascript programmers," in *Proc. of International Workshop on Scripts to Programs. Beijing, China:[sn]*, 2012, pp. 67–82.

[13] E. Andreasen and A. Moller, "Determinacy in static analysis for jQuery," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 17–31, 2014, ISSN: 03621340. DOI:

10.1145/2714064.2660214. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2714064.2660214.

[14] E. S. Andreasen, A. Moller, and B. B. Nielsen, "Systematic Approaches for Increasing Soundness and Precision of Static Analyzers," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, no. June, 2017. DOI: 10.1145/3088515.3088521.

[15] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: An input mutation approach," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 259–269.

[16] *Arachni - Web Application Security Scanner Framework*, 2021. [Online]. Available: https://www.arachni-scanner.com/.

[17] *Are SQL Injections Still A Thing?* 2023. [Online]. Available: https://www.code-intelligence.com/blog/sql-injections.

[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[19] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars.," in *NDSS*, 2019.

[20] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray, "Pythia: Grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations," *arXiv preprint arXiv:2005.11498*, 2020.

[21] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Rest-ler: Automatic intelligent rest api fuzzing," *arXiv preprint arXiv:1806.09739*, 2018.

[22] *AWS NAT Instances*, http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_NAT_Instance.html, 2017.

[23] N. Backman, "Facilitating a battle between hackers: Computer security outside of the classroom," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16, Memphis, Tennessee, USA: ACM, 2016, pp. 603–608, ISBN: 978-1-4503-3685-7. DOI: 10.1145/2839509.2844648. [Online]. Available: http://doi.acm.org/10.1145/2839509.2844648.

[24] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *2010 IEEE symposium on security and privacy*, IEEE, 2010, pp. 332–345.

[25] E. Bazzoli, C. Criscione, F. Maggi, and S. Zanero, "XSS PEEKER: Dissecting the XSS exploitation techniques and fuzzing mechanisms of blackbox web application scanners," in *IFIP International Conference on ICT Systems Security and Privacy Protection*, Springer, 2016.

[26] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A taint based approach for smart fuzzing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE, 2012, pp. 818–825.

[27] L. Blankenship, *The Conscience of a Hacker*, http://phrack.org/issues/7/3.html, 1986.

[28] P. Chapman, J. Burket, and D. Brumley, "Picoctf: A game-based computer security competition for high school students," in *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*, San Diego, CA: USENIX Association, 2014. [Online]. Available: https://www.usenix.org/conference/3gse14/summit-program/presentation/chapman.

[29] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware.," in *NDSS*, vol. 16, 2016, pp. 1–16.

[30] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 633–645.

[31] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, "One engine to fuzz'em all: Generic language processor testing with semantic validation," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, 2021.

[32] M. T. Chi and R. Wylie, "The icap framework: Linking cognitive engagement to active learning outcomes," *Educational psychologist*, vol. 49, no. 4, pp. 219–243, 2014.

[33] N. Childers, B. Boe, L. Cavallaro, L. Cavedon, M. Cova, M. Egele, and G. Vigna, "Organizing Large Scale Hacking Competitions," in *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Bonn, Germany, Jul. 2010.

[34] *Chrome.runtime - getbackgroundpage()*. [Online]. Available: https://developer.chrome.com/extensions/runtime#method-getBackgroundPage.

[35] A. Conklin, "The use of a collegiate cyber defense competition in information security education," in *Proceedings of the 2Nd Annual Conference on Information Security Curriculum Development*, ser. InfoSecCD '05, Kennesaw, Georgia: ACM, 2005, pp. 16–18, ISBN: 1-59593-261-5. DOI: 10.1145/1107622.1107627. [Online]. Available: http://doi.acm.org/10.1145/1107622.1107627.

[36] *Content scripts*. [Online]. Available: https://developer.chrome.com/extensions/content_scripts.

[37] *CTF Time*, https://ctftime.org, 2017.

[38] B. Cui, F. Wang, Y. Hao, and X. Chen, "Whirlingfuzzwork: A taint-analysis-based api in-memory fuzzing framework," *Soft Computing*, vol. 21, no. 12, pp. 3401–3414, 2017.

[39] D. Dasgupta, D. M. Ferebee, and Z. Michalewicz, "Applying puzzle-based learning to cyber-security education," in *Proceedings of the 2013 on InfoSecCD '13: Information Security Curriculum Development Conference*, ser. InfoSecCD '13, Kennesaw GA, USA: ACM, 2013, 20:20–20:26, ISBN: 978-1-4503-2547-9. DOI: 10.1145/2528908.2528910. [Online]. Available: http://doi.acm.org/10.1145/2528908.2528910.

[40] G Deepa, P. S. Thilagam, A. Praseed, and A. R. Pais, "Detlogic: A black-box approach for detecting logic vulnerabilities in web applications," *Journal of Network and Computer Applications*, vol. 109, pp. 89–109, 2018.

[41] Derick Rethans, *Xdebug: A Debugger and Profiling Tool for PHP*, 2020. [Online]. Available: https://xdebug.org.

[42] *Detect adblock – most effective way to detect ad blockers*. [Online]. Available: https://www.detectadblock.com/.

[43] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Vulnerability Scanner," in *Proceedings of the 21st Symposium on USENIX Security*, Bellevue, WA, Aug. 2012.

[44] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 523–538.

[45] 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.

[46] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, "deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Nov. 2013.

[47] A. Doupé, M. Egele, B. Caillat, G. Stringhini, G. Yakin, A. Zand, L. Cavedon, and G. Vigna, "Hit 'em Where it Hurts: A Live Security Exercise on Cyber Situational Awareness," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Orlando, FL, Dec. 2011.

[48] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: Evolutionary fuzzing for black-box xss detection," in *Proceedings of the 4th ACM conference on Data and application security and privacy*, 2014, pp. 37–48.

[49] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," *proceedings of IEEE SSP 2021*, 2021.

[50] D. Esposito, M. Rennhard, L. Ruf, and A. Wagner, "Exploiting the potential of web application vulnerability scanning," in *ICIMP 2018 the Thirteenth International Conference on Internet Monitoring and Protection, Barcelona, Spain, 22-26 July 2018*, IARIA, 2018, pp. 22–29.

[51] *Extension overview*. [Online]. Available: https://developer.chrome.com/extensions/overview.

[52] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward Automated Detection of Logic Vulnerabilities in Web Applications," in *Proceedings of the 19th USENIX Security Symposium*, 2010, pp. 143–160, ISBN: 9781931971775.

[53] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, Aug. 2020.

[54] Floyd Fuh, *AFL CGI Wrapper*, 2020. [Online]. Available: https://github.com/floyd-fuh/afl-cgi-wrapper.

[55] J. A. Fredricks, P. C. Blumenfeld, and A. H. Paris, "School engagement: Potential of the concept, state of the evidence," *Review of educational research*, vol. 74, no. 1, pp. 59–109, 2004.

[56] F. Gauthier, B. Hassanshahi, B. Selwyn-Smith, T. N. Mai, M. Schlüter, and M. Williams, "Backrest: A model-based feedback-driven greybox fuzzer for web applications," *arXiv preprint arXiv:2108.08455*, 2021.

[57] M. Gettinger and J. K. Seibert, "Best practices in increasing academic learning time," *Best practices in school psychology IV*, vol. 1, pp. 773–787, 2002.

[58] V. Gite, *What is Dash (/bin/dash) Shell?* 2020. [Online]. Available: https://www.cyberciti.biz/faq/debian-ubuntu-linux-binbash-vs-bindash-vs-binshshell/.

[59] *Github - tajs*, http://nicolas.golubovic.net/thesis/master.pdf. [Online]. Available: \url{https://github.com/cs-au-dk/TAJSh}.

[60] D. Goldman, "37 million T-Mobile customers were hacked," *CNN*, Feb. 2023.

[61] Google Chrome Extension, *Automatically find and apply coupons*, https://chrome.google.com/webstore/detail/honey/bmnlcjabgnpnenekpadlanbbkooimhnj.

[62] ——, *Trump Filter*, https://chrome.google.com/webstore/detail/trump-filter/lhondapiaknegjpellpodegmeonigjic.

[63] Google Chrome Extensioon, *Hillary Blocker*, https://chrome.google.com/webstore/detail/hillary-blocker/kiblhkcoiojbdhhnjaekompfecgelfja.

[64] *Google Chrome Statistics*, 2023. [Online]. Available: https://truelist.co/blog/google-chrome-statistics.

[65] *Google Search for Uses of Hospital Management System*, 2021. [Online]. Available: https://tinyurl.com/hospitalmanagementsystemuses.

[66] *Google Search for Uses of Login System*, 2021. [Online]. Available: https://tinyurl.com/doctorappointmentsystem.

[67] *Google Search for Uses of Login System*, 2021. [Online]. Available: https://tinyurl.com/usermanagementsystem.

[68] B. Gorenc, *Pwn2own 2017 at cansecwest*, https://www.zerodayinitiative.com/blog/2017/3/23/pwn2own-2017-an-event-for-the-ages, Mar. 2017.

[69] *Gremlins - Monkey Testing Library for Web Apps and Node.js*, 2020. [Online]. Available: https://github.com/marmelab/gremlins.js.

[70] S. Gupta and B. B. Gupta, "Php-sensor: A prototype method to discover workflow violation and xss vulnerabilities in php web applications," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015, pp. 1–8.

[71] J. Hamari, J. Koivisto, and H. Sarsa, "Does gamification work?–a literature review of empirical studies on gamification," in *2014 47th Hawaii international conference on system sciences*, Ieee, 2014, pp. 3025–3034.

[72] ——, "Does gamification work?–a literature review of empirical studies on gamification.," in *47th Hawaii International Conference on System Sciences (HICSS)*, Hawaii, 2014.

[73] T. Harmon, *Cyber Security Capture The Flag (CTF): What Is It?* https://blogs.cisco.com/perspectives/cyber-security-capture-the-flag-ctf-what-is-it, 2016.

[74] *High-performance binary-only instrumentation for afl-fuzz*, 2020. [Online]. Available: https://github.com/google/AFL/blob/master/qemu_mode/README.qemu.

[75] *HTTP State Management Mechanism*, 2020. [Online]. Available: https://tools.ietf.org/html/rfc3875.

[76] *HTTP/1.1 Message Syntax and Routing*, 2020. [Online]. Available: https://tools.ietf.org/html/rfc7230.

[77] Y. W. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S.-Y. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection," in *Thirteenth International World Wide Web Conference Proceedings*, 2004, pp. 40–52, ISBN: 158113844X. DOI: 10.1145/988672.988679.

[78] *IBM/HCL AppScan*, 2020. [Online]. Available: https://www.hcltechsw.com/appscan.

[79] A. Iosup and D. Epema, "An experience report on using gamification in technical higher education," in *Proceedings of the 45th ACM technical symposium on Computer science education*, 2014, pp. 27–32.

[80] ISC, "The 2022 (isc) 2 global information security workforce study," 2022.

[81] J. Iwuozor, "The Biggest Threats to the US Critical National Infrastructure," *Itegriti*, May 2022.

[82] B. Jabiyev, O. Mirzaei, A. Kharraz, and E. Kirda, "Preventing server-side request forgery attacks," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 1626–1635.

[83] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas, "Trends and lessons from three years fighting malicious extensions," in *24th USENIX Security Symposium*, 2015.

[84] S. Jariwala, M. Champion, P. Rajivan, and N. J. Cooke, "Influence of Team Communication and Coordination on the Performance of Teams at the iCTF Competition," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 2012.

[85] S. H. Jensen, P. A. Jonsson, and A. Moller, "Remedying the eval that men do," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ACM, 2012, pp. 34–44.

[86] S. H. Jensen, P. a. Jonsson, and A. Moller, "Remedying the Eval That Men Do," *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 34–44, 2012. DOI: 10.1145/2338965.2336758. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336758.

[87] S. H. Jensen, M. Madsen, and A. Moller, "Modeling the html dom and browser api in static analysis of javascript web applications," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ACM, 2011, pp. 59–69.

[88] S. H. Jensen, A. Moller, and P. Thiemann, "Type analysis for javascript," in *International Static Analysis Symposium*, Springer, 2009, pp. 238–255.

[89] ——, "Interprocedural analysis with lazy propagation," in *International Static Analysis Symposium*, Springer, 2010, pp. 320–339.

[90] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, IEEE, 2006, 6–pp.

[91] ——, "Static Analysis for Detecting Taint-style Vulnerabilities in Web Applications," *Journal of Computer Security*, vol. 18, no. 5, pp. 861–907, 2010, ISSN: 0926227X. DOI: 10.3233/JCS-2009-0385.

[92] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, Aug. 2014, pp. 641–654, ISBN: 978-1-931971-15-7.

[93] N. Khoury, P. Zavarsky, D. Lindskog, and R. Ruhl, "Testing and assessing web vulnerability scanners for persistent sql injection attacks," in *proceedings of the first international workshop on security and privacy preserving in e-societies*, 2011, pp. 12–18.

[94] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[95] M. Leithner, B. Garn, and D. E. Simos, "Hydra: Feedback-driven black-box exploitation of injection vulnerabilities," *Information and Software Technology*, p. 106 703, 2021.

[96] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[97] X. Li and Y. Xue, "Block: A black-box approach for detection of state violation attacks towards web applications," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 247–256.

[98] X. Li, W. Yan, and Y. Xue, "SENTINEL: Securing Database from Logic Flaws in Web Applications," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, 2012, pp. 25–36, ISBN: 9781450310918. DOI: 10.1145/2133601.2133605.

[99] *Manifest - web accessible resources*. [Online]. Available: https://developer.chrome.com/extensions/manifest/web_accessible_resources.

[100] *Manifest version*. [Online]. Available: https://developer.chrome.com/extensions/manifestVersion.

[101] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *arXiv preprint arXiv:1812.00140*, 2018.

[102] B. Martini and K.-K. R. Choo, "Building the next generation of cyber security professionals," in *22nd European Conference on Information Systems (ECIS 2014)*, Tel Aviv, Israel, May 2014.

[103] Z. McGee and S. Acharya, "Security analysis of openemr," in *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, IEEE, 2019, pp. 2655–2660.

[104] W. Meng, C. Qian, S. Hao, K. Borgolte, G. Vigna, C. Kruegel, and W. Lee, "Rampart: Protecting web applications from cpu-exhaustion denial-of-service attacks," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 393–410.

[105] S. Micheelsen and B. Thalmann, *A static analysis tool for detecting security vulnerabilities in python web applications*, 2016.

[106] R. Mohammed, "Assessment of web scanner tools," *International Journal of Computer Applications*, vol. 133, no. 5, pp. 1–4, 2016.

[107] B. E. Mullins, T. H. Lacey, R. F. Mills, J. E. Trechter, and S. D. Bass, "How the cyber defense exercise shaped an information-assurance curriculum," *IEEE Security Privacy*, vol. 5, no. 5, pp. 40–49, Sep. 2007, ISSN: 1540-7993. DOI: 10.1109/MSP.2007.111.

[108] Nicolas Golubovic, *Attacking Browser Extensions, MS Thesis, Ruhr-University Bochum*, http://nicolas.golubovic.net/thesis/master.pdf, 2016.

[109] *OWASP, SQL Injection*, 2022. [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection#:~:text=TheseverityofSQLInjection, Injectionahighimpactseverity.

[110] *OWASP Zed Attack Proxy*, 2020. [Online]. Available: https://www.zaproxy.org/.

[111] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329–340.

[112] G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications.," in *NDSS*, 2014.

[113] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, "Jäk: Using dynamic analysis to crawl and test modern web applications," in *International Symposium on Recent Advances in Intrusion Detection*, Springer, 2015, pp. 295–316.

[114] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 697–710.

[115] C. Pham, D. Tang, K.-i. Chinen, and R. Beuran, "Cyris: A cyber range instantiation system for facilitating security training," in *Proceedings of the Seventh Symposium on Information and Communication Technology*, ser. SoICT '16, Ho Chi Minh City, Viet Nam: ACM, 2016, pp. 251–258, ISBN: 978-1-4503-4815-7. DOI: 10.1145/3011077.3011087. [Online]. Available: http://doi.acm.org/10.1145/3011077.3011087.

[116] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019. DOI: 10.1109/TSE.2019.2941681.

[117] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, IEEE, 2020, pp. 460–465.

[118] *Pin - A Dynamic Binary Instrumentation Tool*, 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.

[119] PortSwigger, *Burp Suite. Application Security Testing*, 2020. [Online]. Available: https://portswigger.net/burp.

[120] M. Prince, "Does active learning work? a review of the research," *Journal of engineering education*, vol. 93, no. 3, pp. 223–231, 2004.

[121] *Puppeteer - Node.js library that provides high-level API access to Chrome and Chromium*, 2021. [Online]. Available: https://github.com/puppeteer/puppeteer.

[122] M. Qasaimeh, A. Shamlawi, and T. Khairallah, "Black box evaluation of web application scanners: Standards mapping approach," *Journal of Theoretical and Applied Information Technology*, vol. 22, Jul. 2018.

[123] *QEMU*, 2020. [Online]. Available: https://qemu.org.

[124] O. v. Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, "Webfuzz: Grey-box fuzzing for web applications," in *European Symposium on Research in Computer Security*, Springer, 2021, pp. 152–172.

[125] A. Ruef, M. W. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel, "Build It, Break It, Fix It: Contesting Secure Development," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016. [Online]. Available: http://arxiv.org/abs/1606.01881.

[126] Y. Shoshitaishvili, L. Invernizzi, A. Doupé, and G. Vigna, "Do You Feel Lucky? A Large-Scale Analysis of Risk-Rewards Trade-Offs in Cyber Security," *ACM Symposium on Applied Computing*, Mar. 2014.

[127] *Sizzle javascript selector*. [Online]. Available: https://sizzlejs.com/.

[128] A. Sjösten, S. Van Acker, and A. Sabelfeld, "Discovering browser extensions via web accessible resources," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ACM, 2017, pp. 329–336.

[129] *Skipfish: Web Application Security Scanner*, 2020. [Online]. Available: https://github.com/spinkham/skipfish.

[130] P. Snyder, C. Taylor, and C. Kanich, "Most websites don't need to vibrate: A cost-benefit approach to improving browser security," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 179–194.

[131] O. Starov and N. Nikiforakis, "XHOUND: Quantifying the fingerprintability of browser extensions," in *Security and Privacy (SP), 2017 IEEE Symposium on*, IEEE, 2017, pp. 941–956.

[132] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution.," in *NDSS*, vol. 16, 2016, pp. 1–16.

[133] M. Stockley, *The web attacks that refuse to die*, https://nakedsecurity.sophos.com/2016/06/15/the-web-attacks-that-refuse-to-die/.

[134] *The 2016-2017 iCTF DDoS*, https://ictf.cs.ucsb.edu/pages/the-2016-2017-ictf-ddos.html.

[135] *The Conscience of a Hacker*, https://en.wikipedia.org/wiki/Hacker_Manifesto, 2017.

[136] *The iCTF Framework*, https://github.com/ucsb-seclab/ictf-framework.

[137] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, *et al.*, "Ad injection at scale: Assessing

deceptive advertisement modifications," in *IEEE Symposium on Security and Privacy (SP)*, 2015.

[138] A. Tiwari, P. Lai, M. So, and K. Yuen, "A comparison of the effects of problem-based learning and lecturing on the development of students' critical thinking," *Medical education*, vol. 40, no. 6, pp. 547–554, 2006.

[139] L. VaBlasco-Arcas, I. Buil, B. Hernandez-Orteg, and F. J. Sese, "Using Clickers in Class. the Role of Interactivity, Active Collaborative Learning and Engagement in Learning Performance," in *Computers and Education*, vol. 62, Pergamon Press, Mar. 2013, pp. 102–110.

[140] K. Vamvoudakis, J. Hespanha, R. Kemmerer, and G. Vigna, "Formulating Cyber-Security as Convex Optimization Problems," in *Control of Cyber-Physical Systems*, ser. Lecture Notes in Control and Information Sciences, vol. 449, Springer, Jul. 2013, pp. 85–100.

[141] G. Vigna, "Teaching Hands-On Network Security: Testbeds and Live Exercises," *Journal of Information Warfare*, vol. 3, no. 2, pp. 8–25, Feb. 2003.

[142] ——, "Teaching Network Security Through Live Exercises," in *Proceedings of the Third Annual World Conference on Information Security Education (WISE)*, C. Irvine and H. Armstrong, Eds., Monterey, CA: Kluwer Academic Publishers, Jun. 2003, pp. 3–18.

[143] G. Vigna, K. Borgolte, J. Corbetta, A. Doupé, Y. Fratantonio, L. Invernizzi, D. Kirat, and Y. Shoshitaishvili, "Ten Years of iCTF: The Good, The Bad, and The Ugly," in *Proceedings of the USENIX Summit on Gaming, Games and Gamification in Security Education (3GSE)*, San Diego, CA, Aug. 2014.

[144] *W3 dom overview*. [Online]. Available: https://www.w3.org/TR/DOM-Level-2-Core/introduction.html.

[145] *w3af - Open Source Web Application Security Scanner*, 2021. [Online]. Available: http://w3af.org/.

[146] D. T. Willingham, "Critical thinking: Why is it so hard to teach?" *Arts Education Policy Review*, vol. 109, no. 4, pp. 21–32, 2008.

[147] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee, "Understanding malvertising through ad-injecting browser extensions," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15, 2015, pp. 1286–1295.

[148] J. Zhao, Y. Lu, K. Zhu, Z. Chen, and H. Huang, "Cefuzz: An directed fuzzing framework for php rce vulnerability," *Electronics*, vol. 11, no. 5, p. 758, 2022.

APPENDIX A

CO-AUTHOR PERMISSION

I, Erik Trickel, affirm that all the co-authors of the following works

1. *Shell We Play A Game? CTF-as-a-service for Security Education.* The paper was co-written with Francesco Disperati, Eric Gustafson, Faezeh Kalantari, Mike Mabey, Naveen Tiwari, Yeganeh Safaei, Adam Doupé, and Giovanni Vigna.
2. *Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting.* The paper was co-written Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé.
3. *Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities.* The paper was co-written with Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, Adam Doupé

have granted their permission for the use of the works listed above in the dissertation, *Attacking Computer Security from the Perspective of Educators, Users, and Analysts.* This permission extends to all forms of reproduction, distribution, and display of the work.