Gray Computing: A Framework for Distributed Computing with Web Browsers

By

Yao Pan

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

December 16, 2017

Nashville, Tennessee

Approved:

Jules White, Ph.D.

Abhishek Dubey, Ph.D.

Aniruddha Gokhale, Ph.D.

Douglas Schmidt, Ph.D.

Caglar Oskay, Ph.D.

ACKNOWLEDGMENTS

First of all, I would like to express my sincerest gratitude to my advisor Professor Jules White for his consistent support and encouragement for me over the past years. Dr. White has offered tremendous help to me from giving research directions and advice, to revising publications and preparing academic presentations. The discussions and meetings with him have always been enlightening and inspiring. I have learned so much from his attitude toward work and his passion for research.

I would like to thank my committee members, Professor Douglas C. Schmidt, Professor Aniruddha Gokhale, Professor Abhishek Dubey, and Professor Caglar Oskay for serving as my committee members. I want to thank you for letting my defense be a valuable and enjoyable moment, and for your brilliant comments and suggestions.

I would like to thank Professor Jeff Gray. Thank you for giving me valuable feedback and suggestions for my research work. I am very thankful to Professor Yu Sun. I have benefited so much from your advice and guidance in research as well as in life.

I would also like to thank all my friends especially Fangzhou Sun, Siwei He, who supported me and motivated me to strive towards my goal. You made my time at Vanderbilt enjoyable and memorable.

A special thanks to my mother and father for all their love, support and sacrifice. They have always been my strongest support and their help is more than I could possibly describe in a short acknowledgment. This last word of acknowledgment I have saved for my fiancee Chao. You are like an angel enlighten my world. Thank you for all your support during my Ph.D. studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The Internet is much richer and more sophisticated than it was 20 years ago. Instead of only displaying static text and images, dynamic and interactive content are everywhere. Displaying richer content requires the use of greater computational resources on the client browser's host machine. As a result of the increasing richness of Internet content, visitors are performing increasingly complex computational work on websites' behalf: from validating syntax and requirements of input values of a web form before submission to the server, to animating page elements. Website visitors implicitly perform computational work for the website owner without knowing exactly what type of work their browsers will perform. The line between what computational tasks should or should not be offloaded to the visitor's browser is not clear cut and creates a blurred boundary, which we term *gray computing*.

Previously, browser-based JavaScript applications were single-threaded. Complex computations would directly impact user interaction with the web page, preventing the offloading of heavy computational tasks to website visitors. However, the emergence of new standards, such as Web Workers introduced in HTML5 [3], allow background JavaScript threads on web pages, offering the potential for much more significant background usage of a visitor's computing resources.

Popular websites receive millions of unique visitors everyday. While most people have very powerful computers relative to the demands of ordinary website rendering. How much computing power is currently untapped? We take YouTube as an example. There are 6 billion hours of video watched on YouTube each month [4]. Assume that each client computer has an average processing ability of 30 GFLOPS, which is estimated from the average client computing power of the famous volunteer computing project BOINC [5]. Further,

conservatively assume that each client computer is only 25% utilized by computing tasks offloaded by YouTube. In this case, the combined processing power of the client computers would be 67 PFLOPS. Sunway TaihuLight, the fastest super computer on record through November 2016, has a computing speed of 93 PFLOPS [6]. In this example, the combined processing power of the web visitors to YouTube is $\approx$2/3 of the largest supercomputer in the world. Even if the true processing power is only 1/10 of this estimate, the processing power would still make it among the TOP50 fastest super computers in the world [6].

The question arises whether we could harness this huge idle computing power and build a large-scale distributed computing engine using website visitors' browsers. In the past, this was not possible because browser-based JavaScript applications were single-threaded, complex calculations would freeze foreground web page and directly impact user interaction with the web page. The emergence of new standards, such as Web Workers introduced in HTML5, which allow background JavaScript threads on web pages, offer the potential for much more significant background usage of visitor's computing resources.

We propose gray computing: a framework for distributed computing with web browsers [7]. Although with great potential, there are important research questions need to be answered before we can conclude gray computing is a viable solution:

1. How to address clients' volatility? Website visitors may leave the web pages they are visiting anytime. The computation may be terminated before completion, adding load to the server without any return on investment. The unknown page view duration adds significant complexity in devising an algorithm for efficient task distribution.

2. How to prevent malicious clients from sabotaging computational integrity? The clients of a gray computing engine are arbitrary website visitors and there are no authentication or credentials to guarantee that they are trustworthy. It is possible that attackers can interfere with a gray computing application by sending falsified results to servers. Mechanisms are needed to ensure that the accuracy of gray computing applications will not be negatively affected by falsified results.

3. Is gray computing cost-effective for website owners? For gray computing, extra costs are incurred on the server-side due to extra data transfer and CPU load from distributing computational tasks and their associated data. It is possible that paying for a set of cloud servers to complete the task is more cost effective than distributing the task using gray computing. Evaluations are needed to determine whether gray computing is more cost effective than commodity cloud computing resources.

4. How to bridge the performance gap between JavaScript in gray computing and native codes? Gray computing computation is performed in web browsers using JavaScript. Poor JavaScript computational performance could undermine the cost-effectiveness and competitiveness of gray computing versus traditional distributed computing platforms.

5. Will gray computing impact user experience in a user perceptible manner? Website visitors do not want to be interrupted or have a degraded experience while they are viewing web pages. It is important to determine whether or not the background tasks of gray computing will be intrusive to users and impact the responsiveness of a visited web page.

6. What are the incentives for users to participate gray computing?

This dissertation provides a comprehensive analysis of the architecture, performance, security, cost effectiveness, user experience, and other issues of gray computing. Several real-world applications are examined and gray computing is shown to be cost effective for a number of complex tasks ranging from computer vision to bioinformatics to cryptology.

With great power comes great risks. The huge potential of gray computing unveiled also arouses our concern of whether attackers could misuse it as well [8]. The execution of Web Workers does not need any explicit permission from users. It is possible for attackers to embed JavaScript codes in the web pages delivered to users in order to gain access to the processing power on their machines. While users are surfing the Internet, these

background resource-stealing JavaScript tasks allow the attackers to use users' browsers to execute computational tasks. We term this new kind of attack **browser resource stealing attack**.

"Great Canon", a large-scale DDoS attack against Github in 2015 [9] has shown the immense computational potential that browser resource stealing attack can tap. The JavaScript codes in the analytics services of Baidu, China's largest search engine, were hijacked to include JavaScript codes that instructed visitors' browsers to send HTTP requests to a series of victim Github pages. The attack was able to send 2.6 billion requests per hour at its peak and caused Github to be intermittently unavailable for 5 days [10], making it among the top 5 largest DDoS attacks in the past decade [11]. The attack drew massive public attention as it only leveraged the browsers of unsuspecting website visitors instead of relying on compromised machines or unpatched vulnerable application versions.

Browser resource stealing attack is very similar to botnets. They both try to steal computational resources from victims. However, the scale of a botnet is limited by the number of Internet connected computers multiplied by the percentage of computers with vulnerabilities that can be exploited. Attackers have developed various means to recruit bots by infecting computers. Some explore the inner vulnerabilities of existing systems, some apply social engineering with fishing or drive-by downloads. However, still only a small percentage of the computers are easily infected. While for browser resource stealing attack, every computer with browsers is vulnerable because the attack does not exploit any vulnerability. Is browser resource stealing attack a significant threat? Several key research questions have been proposed:

1. What types of attacks can browser resource stealing attack launch?

2. How does browser resource stealing attack compare to non-browser-based attacks such as Botnets?

3. Is browser resource stealing attack economically attractive to attackers?

4. How can browser resource stealing attack be detected and prevented?

In this dissertation, we show that current ad networks, such as Google AdWords, allow attackers to perform browser resource stealing attack by displaying ads that launch Web Worker tasks. Our analysis shows that attackers can use a number of strategies to display ads that generate large numbers of impressions but cost very little due to their low click-through rate. Each ad impression allows an attacker to steal computational resources from the browser that the ad is displayed in. We show that browser resource stealing attack can be attractive to an adversary for DDoS attack and distributed password cracking compared to cloud computing or rented botnets. We also discuss the limitations of browser resource stealing attack and suggest potential countermeasures.

Web applications are popular targets for a variety of cyber-attacks because they are easy to access and often contain vulnerabilities that are exploitable. An intrusion detection system monitors the web applications and issues alert when an attack attempt has been detected. Existing implementations of intrusion detection systems usually require extracting features from network packets information or string characteristics of input that are considered relevant to attack analysis. These features need to be manually selected which is time-consuming and require in-depth security domain knowledge. Also, a large amount of labeled legitimate and attack request data is needed for supervised learning algorithms to learn to classify normal and abnormal behaviors. The labeled data can be expensive or impractical to obtain for many real-world web applications.

Deep learning has achieved great success recent year in computer vision [12], speech recognition [13], natural language processing [14], etc. Especially, deep learning has shown not only capable of classification but also automatically extracting features from high dimensional raw input. End-to-end deep learning [15] refers to the approaches where deep learning is responsible for the entire process from feature engineering to prediction. Raw input is fed into the network and high-level output is generated directly. For example, in speech recognition, traditional approaches try to solve the problem by layer-to-layer trans-

formation from audio to phonemes, to words and finally to transcripts. While end-to-end deep learning systems erase the intermediate steps and use audio input to directly learn the output. In this dissertation, we are trying to explore the potential of end-to-end deep learning in intrusion detection systems.

Many deep learning libraries and framework in JavaScript emerge such as Keras.js [16], WebDNN [17], deeplearn.js [18]. These libraries implemented low-level linear algebra and matrix calculation which are essential for training and applying deep learning models. This enables us to build deep learning application through web browsers and offload expensive computation to the client browsers.

In our research, we evaluate the feasibility of an unsupervised/semi-supervised approach for web attack detection. The proposed approach utilizes Robust Software Modeling Tool (RSMT) to derive the call graph from web application runtime when a request has been processed. We then train a stacked denoising autoencoder to encode and reconstruct the call graph. The encoding process can learn a low-dimensional representation of the raw features with unlabeled request data. The reconstruction error of the request data is used to recognize anomalies. We empirically test our approach on both synthetic datasets and real-world applications with intentionally perturbed vulnerabilities. The results show that the proposed approach can efficiently and accurately detect attack requests ranging from SQL injection, cross-site scripting, deserialization attack with minimal domain knowledge and labeled training data. We also explore the possibility of leveraging Blockchain technology to make attack disclosure automatic and public.

1.1 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 surveys the related work of browser-based computing and attacks. Chapter 3 describes the architecture of gray computing and how we address each research question. Chapter 4 presents browser resource stealing attack. Chapter 5 presents the application of gray computing in deep

learning and web attack detection. Finally, the dissertation concludes by summarizing contributions and identifying possible future work in Chapter 6.

Chapter 2

Related Work

2.1 Volunteer Computing

2.1.1 Overview

Gray computing is similar to the concept of volunteer computing [19], which is a distributed computing paradigm in which computer owners donate their idle computing resources to scientific research projects, to reduce costs and speed up research progress. Both gray computing and volunteer computing rely on heterogeneous and untrustworthy clients' computing resources.

In the mid-1990s, consumer PCs have became more and more powerful and increasingly connected to the Internet. At the same time, researchers from the areas of biology, astronomy and physics face the challenge of handling increasingly growing data. For example, the human genome project which started 1990s and aimed to determine the sequence of chemical base pairs making up human DNA took more than 10 years to compute [20]. The idea of using consumer PCs for distributed computing arose. The first volunteer computing project was Great Internet Mersenne Prime search (GIMPS) [21] which was started in Jan 1996 aiming to search for Mersenne prime numbers. In 1997, another project Distributed.net [22] was launched to break keys by brute-force search of the key space. They are the earliest prototypes demonstrating the feasibility of volunteer computing.

The term "volunteer computing" was first introduced by Luis F. G. Sarmenta [19]. He developed a Java applet-based volunteer computing system called Bayanihan [23] in 1998. In his paper, several challenges and corresponding solutions for volunteer computing paradigms were discussed such as adaptive parallelism, fault-tolerance, and performance on scalability.

In 1999, SETI@home [24] was released by University of California, Berkeley as a project for analyzing vast amounts of telescope signals to search for extraterrestrial intelligence. In 2000, Folding@home [25] was launched by Stanford University. The volunteer computing project aims to simulate protein folding to help disease research.

All these projects developed their own middleware: they required volunteers to install a specialized client and they had to create their own server-side infrastructure for distributing jobs to volunteer computers and for collecting computational results. However, few scientists had the resources and skills to develop such software.

Berkeley Open Infrastructure for Network Computing (BOINC) [5] was founded in 2002 at University of Califronia, Berkeley. BOINC provides a general-purpose middleware for volunteer computing, including a client, client GUI, application runtime system, server software. BOINC makes it easier for scientists to use volunteer computing. Now BOINC hosts more than 60 projects in a wide range of scientific areas including SETI@home, Einstein@home (searches for gravitational wave), ClimatePrediction.net (studies long-term climate change), World Community Grid, etc.

We have witnessed a rapid growth in the computing power of volunteer computing over years. Back in 2002, SETI@home, the largest volunteer computing project at that time had a processing ability of around 60 TFLOPS [24]. While as of Oct 2015, BOINC stably maintains more than 390,000 active volunteer computers and more than 9 PFLOPS computing power, which makes it comparable to the top 5 fastest supercomputers in the world [6].

Besides the success in attracting public participation, volunteer computing also produced fruitful scientific achievements. Over hundreds of publications including in high-prestige journals such as Nature and Science were supported by BOINC-based projects [26]. A brief history of volunteer computing is shown in Figure 2.1.

The concept of volunteer computing is closely related with parallel computing, grid computing and cluster computing. Parallel computing is a type of computation in which

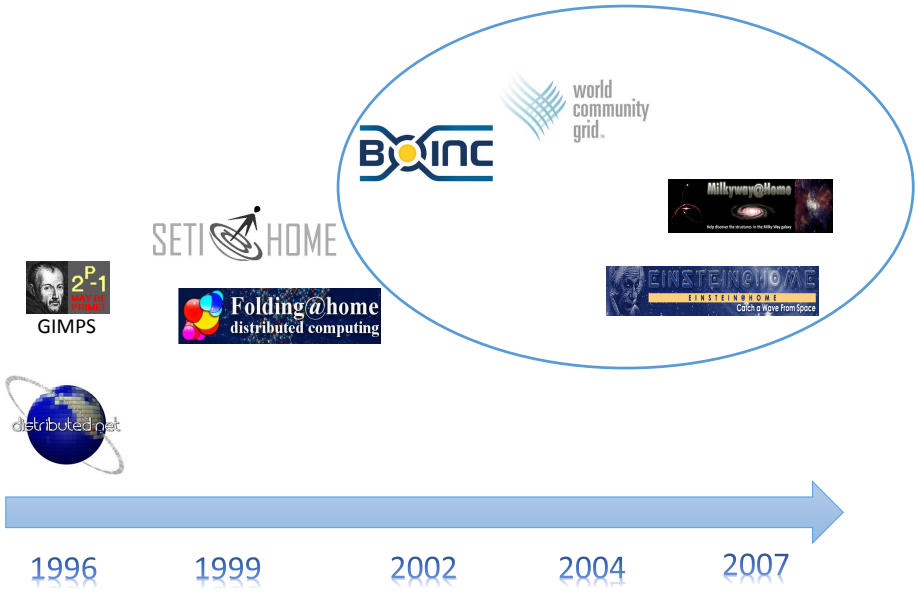

Figure 2.1: Milestones in volunteer computing history.

many calculations are carried out simultaneously. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism.

Volunteer computing is a special kind of distributed computing. Distributed computing is a system in which components located on networked computers communicated and coordinate their actions in order to achieve a common goal.

Grid computing is a form of distributed computing where a super virtual computer is composed of many networked loosely coupled computers acting together to perform very large tasks. Originally, grid computing is proposed to let organizations such as companies, research labs and universities to share computational resources. The organization contribute their idle resources and borrow resources from others when in need. The relationship between participant and projects are more symmetry. Some projects includes Globus, Condor [27], Open Science Grid [28].

Volunteer computing is a type of distributed computing in which computer owner donate their idle computing resources to help run computationally expensive projects. Volunteer computing share many commonalities with grid computing: they are both geographic

Table 2.1: Comparison of different distributed computing types.

	Cluster computing	Grid computing	Volunteer computing
Task distribution	Each node performs the same task	Each node performs different tasks/applications	Each node performs different tasks/applications
Geographic distribution	Single location	Geographically dispersed	Geographically dispersed
Connectivity	LAN, tightly coupled	Internet, loosely coupled	Internet, loosely coupled
Hardware configuration	Homogeneous	Heterogeneous, usually high-performance computers	Heterogeneous, usually ordinary commodity computers
Business model	Paid and own	Symmetry, organizations share resources with each other	Asymmetry, volunteers supply resources to projects for free
Example	Tianhe-2	Open Science Grid, ACCRE	BOINC

dispersed, composed of highly heterogeneous computers. The biggest difference is the business model. The principle of grid computing

A computer cluster consists of connected computers that work together that they can be viewed as a single system. Cluster differs from Cloud and Grid in that a cluster is a group of computers connected by a local area network (LAN), whereas cloud and grid are more wide scale and can be geographically distributed. Another way to put it is to say that a cluster is tightly coupled, whereas a Grid or a cloud is loosely coupled. Tightly coupled refers to the cases where components have high bandwidth connection and low latency. While loosely coupled means low bandwidth communication and high latency. Also, the failure of one component in loosely coupled system usually do not affect others. Also, clusters are made up of machines with similar hardware, whereas clouds and grids are made up of machines with possibly very different hardware configurations.

A comparison of characteristics of cluster computing, grid computing, volunteer computing and cloud computing is shown in Table 2.1.

2.1.2 Gaps in Existing Research

Gray computing can also be seen as a special form of volunteer computing. It shares the same characteristics with volunteer computing such as client heterogeneity, loosely coupled structure. However, gray computing also faces some new challenges that were not presented in traditional volunteer computing:

1. Volatility. Unlike cloud computing or cluster computing where the computation happens on reliable computers, a gray computing engine has no control over its clients and clients can leave and terminate the computation at will. The client may leave before the computation is complete, adding load to the server without any return. The unknown page view duration adds significant complexity to devising an algorithm for efficient task distribution.

2. Performance. Gray computing uses browser as the computing medium. Therefore, JavaScript is the core language for computation. However, JavaScript was considered slow and not suitable for high performance computing. Code porting from other languages to JavaScript may be time-consuming and error-prone.

Another problem with volunteer computing is it appeals to only a narrow scope of users: the volunteer population has remained static around 500,000 for several years [29]. That is less than 0.02% of the current number of devices connected to the Internet [30]. It seems the growth of volunteer computing power has reached a bottleneck. Gray computing does not require users to install any client-side software. Users only need to open a webpage to contribute their computing resources. Therefore, gray computing has access to a much larger potential pool of clients: popular websites usually have millions of daily visitors.

2.2 Browser-based Computing

2.2.1 Overview

A number of researchers have investigated volunteer computing with browsers. The primary advantage of a browser-based approach is that the user only needs to open a web page to take part in the computation. Krupa et al. [31] proposed a browser-based volunteer computing architecture for web crawling. Konishi et al. [32] evaluated browser-based computing with Ajax and the comparative performance of JavaScript and legacy computer languages. In our work, we add a comprehensive analysis of: 1) the architectural changes

to optimize this paradigm for websites served from cloud environments; 2) analysis of the impact of page-view time on scheduler efficiency and data transfer; 3) a cost model for assessing the cost-effectiveness of distributing a given task to browser-based clients as opposed to running the computation in the cloud; and 4) present a number of practical examples of data processing tasks to support website operators, particularly social networks, e-commerce, as well as life science.

MapReduce [33] is a programming model for large parallel data processing proposed by Google, which has been adapted to various distributed computing environments such as volunteer computing [34]. There are some early prototypes implementing MapReduce with JavaScript [35]. Lin et al. [34] observed that the traditional MapReduce is proposed for homogeneous cluster environments and performs poorly on volunteer computing systems where computing nodes are volatile and with a high rate of unavailability. They propose MOON (MapReduce On Opportunistic eNvironments) which extends Hadoop with adaptive task and data scheduling algorithms and achieves a 3-fold performance improvement. However, MOON targets institutional intranet environments like student labs where computer nodes are connected with a local network with high bandwidth and low latency. We focus on cloud and Internet environments with highly heterogeneous computational ability, widely varying client participation times, and non-fixed costs for data transfer and task distribution resources.

2.2.2 Gaps in Existing Research

Some researchers have noticed the potential of browser-based distributed computing and applied the infrastructure to volunteer computing. QMachine [36] is a web service incorporating volunteer web browsers worldwide together for bioinformatics computing tasks. It requires the explicit permissions from participating users. None of the existing work has discussed the cost model of browser-based distributed computing and whether there are really monetary incentives to perform such computing. Additionally, if there is

monetary incentives, what are the factors that affect the profit of the computation?

2.3 Botnet

2.3.1 Overview

Botnets or zombie networks, are a major threat in network security. Botnets refer to a network of computers infected with a malicious program that allows criminals to control the infected machines remotely without the users' knowledge [37]. Various attacks can be made through botnets such as DDoS attacks [38], spam [39], phishing, or theft of confidential information [40].

The browser-based distributed attacking can be seen as a new form of botnet where the bots are not infected by malicious programs but simply are visiting webpages. One characteristic of the browser-based botnets is ephemeral since the visiting time of webpage visitors are usually short and unpredictable. However, a major advantage of browser-based botnets is its massive scale since it is possible to compromise a website of millions of users to turn them into a big botnet. Besides, the browser-based botnets are dynamic: the visitors come and leave. It is hard to trace and locate these bots.

Various methodologies and tools have been developed to measure and detect attacks initiated with botnets. Most detection techniques rely on horizontal correlation which is observing the behavioral similarities of multiple bots of the same botnet. Botsniffer [41] uses statistical algorithms to detect botnets based on crowd-like behaviors. BotMiner [42] proposes a detection framework to perform clustering on Command and Control (C&C) communication traffic. Disclosure [43] detects botnet through large-scale netflow analysis.

Traditional botnet detection methods generally do not work well on browser-based botnets. First of all, it is difficult to get access to the raw network data on a large scale that would allow traffic analysis. In addition, if the detection target is the compromised website, simply blocking these websites is not a good idea since that may affect users daily activi-

ties. If the detection target is the clients, the clients in the browser botnet only become bots when they visit the webpage, they are normal users when they leave. We cannot simply block them.

2.3.2 Gaps in Existing Research

People have shown interest in misusing browsers for web attacks. Lam et. al [44] propose Puppetnets which is a distributed attack infrastructure misusing web browsers. However, the attacks assessed were limited to single-threaded web pages performing tasks that were not computationally intensive, such as worm propagation, click fraud, etc. In contrast, Web Worker-based attacks run in the background and can do substantial computational work without impacting foreground JavaScript performance and tipping off users to their presence.

The concept of a browser botnet was first proposed by Kuppan [45]. They introduced the idea of using Web Workers as a potential attack vector for a DDoS attack. However, they did not consider the cost to launch such attacks and whether the economics and available computing power would be attractive to attackers. Pellegrino et. al [46] present a preliminary cost analysis on browser-based DDoS attacks. Their analysis is limited to DDoS attacks while we analyze a broader set of computational tasks, such as distributed password cracking, and provide concrete comparisons with cloud computing providers.

Chapter 3

Gray Computing

3.1 Overview

Websites routinely distribute small amounts of work to visitors browsers in order to validate forms, render animations, and perform other computations. These requires the use of greater computational resources on the client browser's host machine. As a result of the increasing richness of Internet content, visitors are performing increasingly complex computational work on websites' behalf: from validating syntax and requirements of input values of a web form before submission to the server, to animating page elements. Website visitors implicitly perform computational work for the website owner without knowing exactly what type of work their browsers will perform. The line between what computational tasks should or should not be offloaded to the visitor's browser is not clear cut and creates a blurred boundary, which we term *gray computing*.

Previously, browser-based JavaScript applications were single-threaded. Complex computations would directly impact user interaction with the web page, preventing the offloading of heavy computational tasks to website visitors. However, the emergence of new standards, such as Web Workers introduced in HTML5, allow background JavaScript threads on web pages, offering the potential for much more significant background usage of a visitor's computing resources.

Offloading computational tasks to browsers can be useful in many circumstances. For example, large websites routinely do big data processing [33] to process website visitor logs, user photos, social network connections, and other large datasets for meaningful information. The question this article investigates is whether or not organizations could cost-effectively offload these big data processing tasks to client web browsers in background JavaScript threads. How much computational power could be harnessed in this type of

model, both for computations directly beneficial to the web visitor, such as product recommendations, as well as for attackers that compromise a website?

There are clearly significant legal and ethical questions around this concept of gray computing, but before deeply exploring them, it is important to ensure that there is actually significant potential computational value in gray computing. For example, if the costs, such as added load to the webserver or reduced website responsiveness, reduce web traffic, then clearly gray computing will not be exploited. Further, if the computational power of browsers is insignificant and can not perform sufficient work to outweigh the outgoing bandwidth costs of the data transferred to clients for processing, then no user incentives or other models to entice users into opting into computationally demanding tasks will be feasible.

Gray computing is similar to the concept of volunteer computing [19], which is a distributed computing paradigm in which computer owners donate their idle computing resources to scientific research projects, to reduce costs and speed up research progress. Both gray computing and volunteer computing rely on heterogeneous and untrustworthy clients' computing resources. The problem with volunteer computing is it appeals to only a narrow scope of users: the volunteer population has remained static around 500,000 for several years [29]. That is less than 0.02% of the current number of devices connected to the Internet [30]. It seems the growth of volunteer computing power has reached a bottleneck. Gray computing does not require users to install any client-side software. Users only need to open a webpage to contribute their computing resources. Therefore, gray computing has access to a much larger potential pool of clients: popular websites usually have millions of daily visitors. For example, YouTube has about 990 million daily unique visitors and the average daily time for each visitor on site is about 9 minutes, as of April 2017 [47]. Assume that each client computer has an average processing ability of 30 GFLOPS, which is estimated from the average client computing power of the famous volunteer computing project BOINC [5]. Further, conservatively assume that each client computer is only 25%

utilized by gray computing tasks offloaded by YouTube. In this case, the combined processing power of the client computers would be 46.4 PFLOPS. For comparison, SETI@home, a volunteer computing project to search for extraterrestrial intelligence, has an average computing power of 0.78 PFLOPS [48]. Sunway TaihuLight, the fastest super computer on record through April 2017, has a computing speed of 93 PFLOPS [6]. In this example, the combined processing power of the web visitors to YouTube is half of the largest supercomputer in the world.

However, gray computing also faces some challenges that are not presented in traditional volunteer computing:

1. *Performance:* Gray computing uses web browsers as the computing medium. Therefore, JavaScript is the core language for computation. However, JavaScript is generally considered slow and not suitable for high performance computing. Further, most computationally heavy tasks are not implemented in JavaScript and porting existing code from other languages to JavaScript can be time-consuming and error-prone.

2. *User experience:* Unlike traditional volunteer computing where users may leave the computers overnight or do computation when a screensaver is running, the users of gray computing are browsing webpages at the same time. The computational work should not interfere with user experience or slow down webpage rendering or interaction in a perceptible manner.

3. *Cost effectiveness:* Although gray computing utilizes the computational resources of each individual website visitor for free, extra costs are still incurred on the server-side due to extra data transfer and CPU load from distributing computational tasks and their associated data. There has not been deep research into the cost-effectiveness of volunteer computing. With cloud computing providers offering elastic computing resources with prices as low as a few cents per hour [49], it is possible that paying for a set of cloud servers to perform a computationally heavy task is cheaper than the

outgoing bandwidth costs of a gray computing system.

4. *Client volatility:* Although volunteer computing also suffers from unpredictable availability of clients, the situation is worse in gray computing. The duration of a website visit can last from a few minutes to several seconds, which is far shorter than the average duration in volunteer computing. It is also more likely for visitors to close a browser tab before a computation is complete than to shut down a volunteer computing client, adding load to the server without any return. The unknown page view duration adds significant complexity for devising an algorithm for efficient task distribution.

Open Question $\Rightarrow$ Is it feasible and cost-effective to build a gray computing data processing infrastructure using website visitors' browsers? Although prior research on volunteer computing has investigated browser-based volunteer computing engines, past research has focused on scenarios where website visitors keep specialized web pages open for long periods of time [23]. There has not been a deep investigation into the cost effectiveness of building browser-based distributed data processing engines with respect to the impact on the user experience while visiting a website. Each of these topics is considered in this article as an evaluation of gray computing. A number of key research challenges exist that stem from the volatility, security, user experience considerations, and cost of performing distributed data processing with gray computing. Our research aims to determine if gray computing is an appealing enough target to warrant further research into these issues.

This chapter presents an architectural solution that addresses these research challenges and proves the feasibility of performing distributed data processing tasks with gray computing. To prove the feasibility of the gray computing concept, we built, empirically benchmarked, and analyzed a variety of browser-based distributed data processing engines for different types of gray computing tasks, ranging from facial recognition to rainbow table generation. We analyzed the computational power that could be harnessed both for valid user-oriented concerns, as well as by cyber-attackers. The same experiments were per-

formed using Amazon's cloud services for distributed data processing in order to compare the performance and cost of gray computing to cloud computing.

To address potential legal and ethical concerns with gray computing, we discuss deployment strategies for gray computing. Gray computing can be deployed so that computing serves as an exchange medium. Users enjoy free services of a website at the cost of performing background computations while browsing. Gray computing can be applied to general business websites with reward incentives to motivate visitors to do computational tasks. For instance, users visiting online retail sites can be asked if they are willing to participate in the distributed computational tasks and shopping points could be rewarded in return. Gray computing can also be deployed as a form of volunteer computing so that volunteers can contribute to scientific projects by simply opening a website instead of installing any software.

3.2 Research Questions

In this section, we present a number of key research questions need to be addressed in order to determine if it is feasible to tap into gray computing.

3.2.1 Research Question 1: How to Effectively Allocate Tasks to Clients With Unknown Page View Times?

Unlike a centralized data processing engine (e.g., cloud computing servers) where the computation happens on a reliable cluster, a gray computing engine has no control over its clients, who can leave and terminate the computation at will. Page viewing habits differ significantly across users. If a gray computing engine simply assigns tasks to clients randomly, the client may leave before the computation is complete, adding load to the server without any return on investment. The unknown page view duration adds significant complexity to devising an algorithm for efficient task distribution.

3.2.2 Research Question 2: How to Prevent Malicious Clients from Sabotaging Computational Integrity?

The nodes of a gray computing engine are arbitrary website visitors and there are no authentication or credentials to guarantee that they are trustworthy. It is possible that attackers can interfere with a gray computing application by sending falsified results to servers. Mechanisms are needed to ensure that the accuracy of gray computing applications will not be negatively impacted by falsified results.

3.2.3 Research Question 3: Is Gray Computing Cost-effective for Website Owners?

Although gray computing utilizes the computational resources of each individual website visitor, which are free, extra costs are still incurred on the server-side due to extra data transfer and CPU load from distributing computational tasks and their associated data. If the amount of data to be transferred is too large, it is possible that the extra cost added to the server-side is more than the value of the computed result from the client. In other words, it is possible that paying for a set of cloud servers to complete the task is more cost effective than distributing the task using gray computing, particularly when cloud computing providers offer elastic computing resources with prices as low as a few cents per hour [49]. Evaluation is needed to determine whether gray computing is more cost effective than commodity cloud computing resources.

3.2.4 Research Question 4: How to Bridge the Performance Gap between JavaScript in Gray Computing and Native Codes?

Gray computing computation is performed in web browsers using JavaScript. However, JavaScript is an interpreted language that is usually not considered for high-performance computation and is generally believed to be several orders of magnitudes slower than C or C++ [50]. Poor JavaScript computational performance could undermine the cost-

effectiveness and competitiveness of distributed data processing with gray computing versus traditional platforms, such as Map Reduce clusters.

3.2.5 Research Question 5: Does Gray Computing Work for Mobile Devices?

With the popularity of mobile devices, an increasing portion of website traffic comes from devices such as smartphones and tablets. Of course, mobile devices have very different hardware and software environments compared to desktop computers. Mobile devices do not necessarily access the Internet through browsers, but often by native apps. It is an interesting question to see how gray computing performs in mobile systems.

3.2.6 Research Question 6: Will Gray Computing Impact User Experience?

In traditional specialized volunteer computing clients, such as BOINC [51], users are devoted to the computational task and it is acceptable to assume the system resources to be fully utilized. However, in the context of gray computing, website visitors do not want to be interrupted or have a degraded experience while they are viewing web pages. It is important to determine whether or not the background tasks of gray computing will be intrusive to users and impact the responsiveness of a visited web page.

3.2.7 Research Question 7: What are the Incentives for Users to Participate Gray Computing?

The offloading of computational tasks to website visitors in gray computing is invisible and does not necessarily need to get users' approval. This immediately raises the legal and ethical issues of violating users' right to know and misusing their computational resources. Some key questions to be considered are: What are the potential legitimate ways of deploying gray computing? What are the possible incentive mechanisms to attract visitors and to prevent misuse?

3.3 Architecture of Gray Computing

We developed a novel architecture for browser-based data processing engines hosted out of cloud computing environments, such as Amazon EC2.

To reduce the workload of the task distribution server, we utilize the cloud provider's storage service to serve data for computing directly out of the storage service. The task distribution server is only responsible for determining what tasks should be given to a client. The task distribution server handles HTTP GET requests and responds with a JSON-based string containing the task ID and data location URL for each task. Each client calls the API once before working on an individual task. Our empirical analysis shows a relatively small workload is added to the task distribution server and the cost is negligible.

Besides reducing the workload of the task distribution server, serving data directly out of the cloud-based storage server also exploits the pricing asymmetry in cloud storage services. In Amazon S3 and Microsoft Azure, only outbound data transfer is charged. Data transferred into the storage service is free. This means clients can report the results of computation directly to the storage service for free. This setup allows the data processing engine to reduce bandwidth costs.

Since the clients are highly volatile and typically only have short page view times, improving data transfer speeds is critical to the overall performance. It is essential to maximize the time that clients spend executing computational tasks and minimize the time spent waiting for input data. One optimization that has been applied in our architecture is the use of a Content Delivery Network (CDN). Instead of serving the input data through a single storage server, a CDN works by serving the content with a large distributed system of servers deployed in multiple data centers in multiple geographic locations. Requests for content are directed to the nodes that are closest and have the lowest latency connection to the client. Take Amazon's CDN service CloudFront as an example. The original version of the data is stored in an origin server, such as S3. Amazon copies the data and produces a CDN domain name for the data. The clients request the data using the CloudFront do-

main name and CloudFront will determine the best edge location to serve the contents. An overview of this proposed architecture is shown in Figure 3.1.

In order to manage and coordinate the computational tasks, we built a cloud-based task distribution server using Node.js and the Express framework. The task distribution server partitions the data into small chunks and assigns them to clients for processing. We use Amazon S3 as our storage server to store input data for clients and receive results from clients. An EC2 server is used to subdivide the tasks and maintain a task queue to distribute tasks to clients. In our proposed architecture, tasks are distributed to clients as follows:

1. The client requests an HTML webpage from the server.
2. The server injects an additional JavaScript file into the webpage that includes the data processing task.
3. A JavaScript Web Worker, which executes in a background processing thread after the page is fully loaded, is used to perform the heavy data processing computation without impacting the foreground JavaScript rendering.
4. The client sends AJAX HTTP requests to retrieve the input data from the CDN. Once it receives the input data, it runs a map or/and reduce function on the data.
5. The client issues an HTTP PUT or POST of the results directly back to the cloud storage server. After submitting the results, the Web Worker messages the main thread to indicate completion. Upon receipt of this message, the main thread sends a new HTTP request to fetch another data processing task from the server.

3.4 Answering Research Question 1: Adaptive Scheduling Algorithm for Volatile Clients

The clients in gray computing are website visitors' browsers, which are not static and reliable. The clients may join and leave the available computational resource pool frequently. The result is that a client may leave before its assigned computational task is

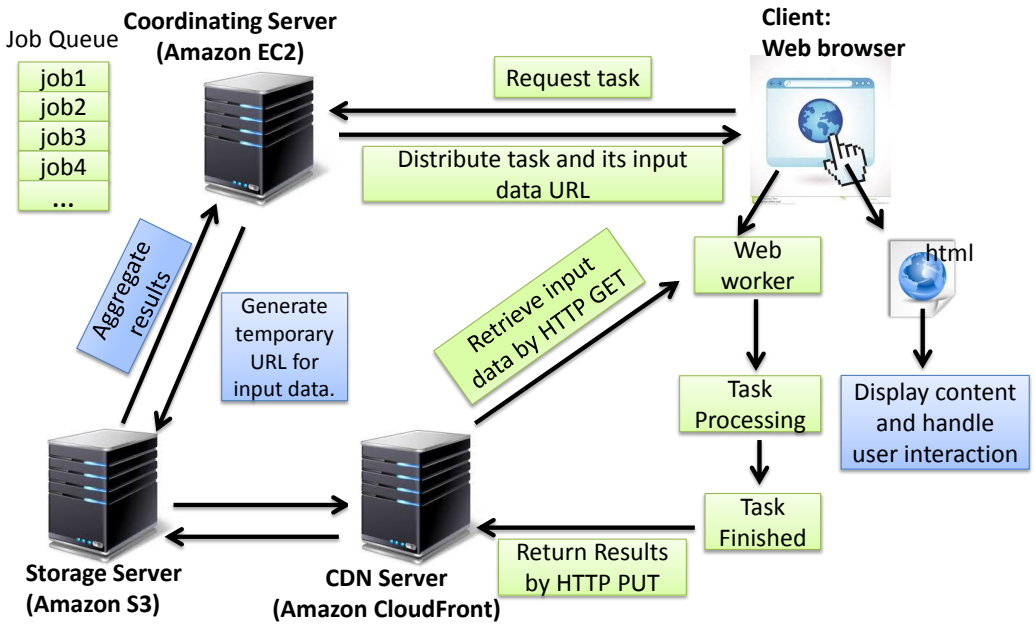

Figure 3.1: Architecture of the proposed distributed data processing engine for gray computing.

finished, adding extra data transfer and producing no computational value. We define μ as the successful task completion ratio. The value of μ is important and directly influences the cost of gray computing. The higher value of μ, the less data transferred is wasted, and thus more cost-effective gray computing will be.

There are two factors affecting the successful task completion ratio μ: the page view duration of the client and the computing time of the assigned task. The relationship between μ, average page view duration and task sizes is depicted in Figure 3.2. Assume assigned task size is proportional to the computing time needed. For a fixed task chunk size, the longer the page view duration, the fewer chances the client will leave before the computation is completed. The distribution of the page view duration of a website's clients is determined by the website's own characteristic. However, the computing time of the assigned tasks is what we can control. Assume the whole application can be divided into smaller chunks of arbitrary size. Reducing the single task size assigned to the clients will increase the task completion ratio, but result in more task units. More task units means more cost on the requests to fetch data and return results. Thus, there is a tradeoff between

the size of tasks and the number of tasks.

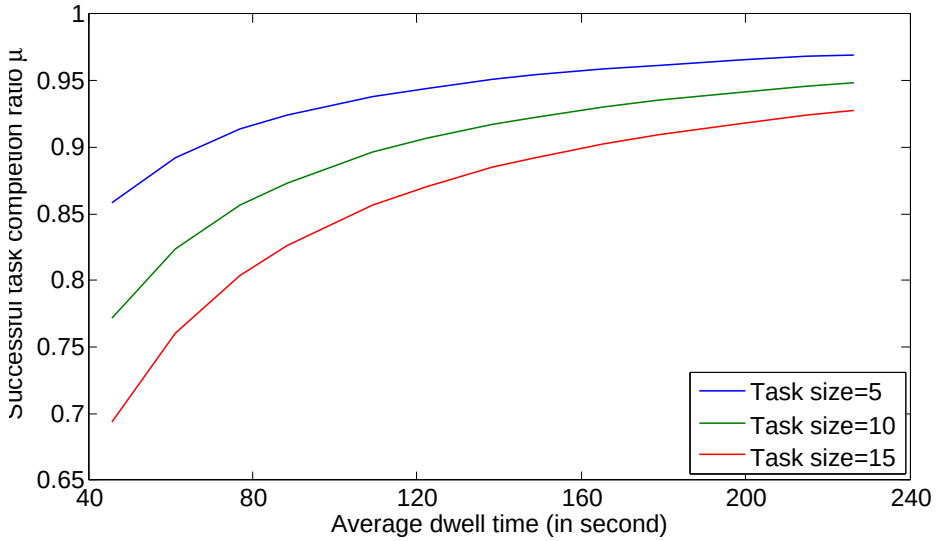

Figure 3.2: The relationship between average page view duration and μ for different task sizes.

Instead of treating all the clients as the same and assigning them tasks of the same size, we developed an adaptive scheduling algorithm. We utilize the fact that website visitors' dwell time follows a Weibull distribution [52, 53]. The probability density function of the Weibull distribution is given by:

$$f(t|k,\lambda) = \frac{k}{\lambda}(\frac{t}{\lambda})^{k-1}e^{-(\frac{t}{\lambda})^k} \quad t \geq 0 \tag{3.1}$$

In the probability density function, if $k > 1$, the Weibull distribution has the property of positive aging, which means the longer the object has survived, the more likely it is to fail. If $0 < k < 1$, the Weibull distribution has the property of negative aging, which means the longer the object has survived, the less likely it is to fail. Researchers have discovered from real-world data that 99% of web pages have a negative aging effect [52]. This implies the probability that a visitor leaves a web page decreases as time passes. This can be explained by the fact that visitors usually have a quick glance of a web page to see whether they are interested in the content or not. During this stage, the probability of leaving the page is high, but after the visitor passes this stage, they may find the content interesting and are

willing to spend time dwelling on the page. Thus, the probability of leaving becomes lower.

Inspired by the Weibull distribution of page dwell time, we propose an adaptive scheduling algorithm which dynamically adjusts the subsequent task size assigned to the same client according to the dwell time of the client. Instead of partitioning the task into equally sized chunks, we assign task size of m_i to the i-th request from the same client where $m_i < m_{i+1}$ when $1 \leq i < n$. Our adaptive scheduling algorithm works by assigning tasks of smaller size to clients first and increasing the subsequent task sizes until a threshold is reached. In this way, we achieve higher successful time completion ratio than fixed task size while also reducing the number of task requests.

For comparison, we also implemented a Hadoop [54] scheduler, which works as follows: The entire job is partitioned into task units of the same size. The server maintains a queue of tasks to assign. When a client connects to the server, the server randomly selects a task from the queue and assigns it to the client. If a task fails, the failed task will be pushed onto the back of the queue and wait to be reassigned to other clients.

To derive μ values for the schedulers, we ran simulations in Matlab with varying average page view times and task durations. We assumed the page view duration follows a Weibull distribution with average page view duration equal to T_{apv}. We used a duplication factor of $d = 2$ for a voting scheme that required 2 matching results. The results are shown in Table 3.1. We can see that 1) μ increases as the average page view time increases (*i.e.*, the successful task completion rate goes up), and 2) The adaptive scheduler achieves higher μ and faster task completion times compared to the Hadoop scheduler.

Table 3.1: Comparison of different schedulers in terms of task completion success ratio μ.

Scheduler	μ			
Average page view duration	30s	1min	2min	3min
Hadoop	0.80	0.85	0.88	0.90
Adaptive	0.84	0.88	0.92	0.93

3.5 Answering Research Question 2: Task Duplication and Majority Voting to Ensure Computational Integrity

Because clients in a gray computing distributed data processing engine are not guaranteed to be trustworthy, it is possible that the results returned from them can be fraudulent.

Malicious client attacks can be classified as follows:

- **Independent malicious attacks:** In this case, malicious clients act independently. Given the same input, they will return arbitrary incorrect results. Unintentional errors such as I/O failure or CPU miscalculations can also be modeled by this category [55].
- **Collaborative malicious attacks:** A stronger type of attack is when attackers have control over a group of clients. When a client from the coordinated group of malicious clients is assigned a task from the server, it checks to see whether another malicious client has received a task with the same input. If so, the malicious client will return the same falsified result in order to fool the server. It is possible that multiple collaborative malicious parties exist simultaneously. However, in the following analysis, we will consider the worst case in which all malicious clients collaborate with each other.

If the server does not verify results, falsified results will be accepted by the server and taint the overall accuracy of data processing tasks. Therefore, verification mechanisms need to be implemented in order to guarantee that the overall result will not be rendered useless by a small subset of malicious clients. One simple and basic approach to handle malicious results is to duplicate tasks, where identical task units are assigned to different clients to compute and clients' results are compared for consistency. Duplication is widely used in a variety of distributed computing system designs, such as volunteer computing [56] and MapReduce [57].

There are many more complex techniques for verification of volunteer computing results. For example, sampling approaches [58, 59, 60] verify each client by sending a task

unit with a known correct result. If the result from a client doesn't match the correct result, the client will be blacklisted. Credibility approaches [56, 61] maintain reputation scores for each client by observing the behavior of the clients. These approaches overcome the drawbacks of the extra computational costs of task duplication, but are vulnerable to some specific attacks: a client can behave well at first to gain credibility and send falsified results after that. Distributed agreement and consensus algorithms, such as Byzantine agreement [62], are also possible but not ideally suited to an environment with a centralized server and no communication between clients.

A simple but effective approach to handle malicious clients is to use a task duplication and majority voting scheme. More complex strategies exist, such as credibility-based approaches, but are not necessarily a good fit for the highly-volatile browser-based data processing environment. Many visitors may access a website only once and the website may have no history statistics to derive their visitor reputation scores. Suppose the overall application consists of n divisible task units. There are C concurrent clients and among them, M are malicious clients, which will send falsified results. f is the fraction of malicious clients among the total clients. Our duplication approach works by assigning identical task units to k different clients and verifying the results returned from different clients. If the computation results from different clients are the same, then the result will be accepted. If not, a majority voting strategy is applied and the result from the majority of clients is accepted. The server randomly distributes tasks to clients, while ensuring no client receives both a task and its duplicate. Thus, clients have no control over which task units they will receive. The duplication approach fails only when a majority of a task's duplications are sent to a collaborative group of malicious clients.

Error Rate: The error rate ε is defined as the fraction of the final accepted results that are malicious. As described by Sarmenta et al. [56], the error rate can be given by:

$$\varepsilon = \sum_{j=d}^{2d-1} \binom{2d-1}{j} f^j (1-f)^{(2d-1-j)} \tag{3.2}$$

where d is the minimum number of matching results needed for the majority voting scheme. For example, for $d = 2$, the server accepts a result when it gets two identical results. When the server gets two different results, it keeps reassigning the task until 2 identical results are received. Redundancy, R, is defined as the ratio of the number of assigned task units to the number of total task units. It can be proved that $R = d/(1-f)$.

Although simple, the duplication approach is robust even with small d. For large websites with millions of visitors, it is difficult for attackers to gain control over a large portion of the clients. Suppose a hacker has a botnet of a thousand machines, $f = 0.001$ for a website with a million total clients. As shown in Figure 3.3, even with $d = 2$, we can ensure 99.9999% of all the accepted results are correct. The choice of d is also dependent on the type of application. Some applications are generally less sensitive and even no duplication is acceptable since the falsified results do little harm.

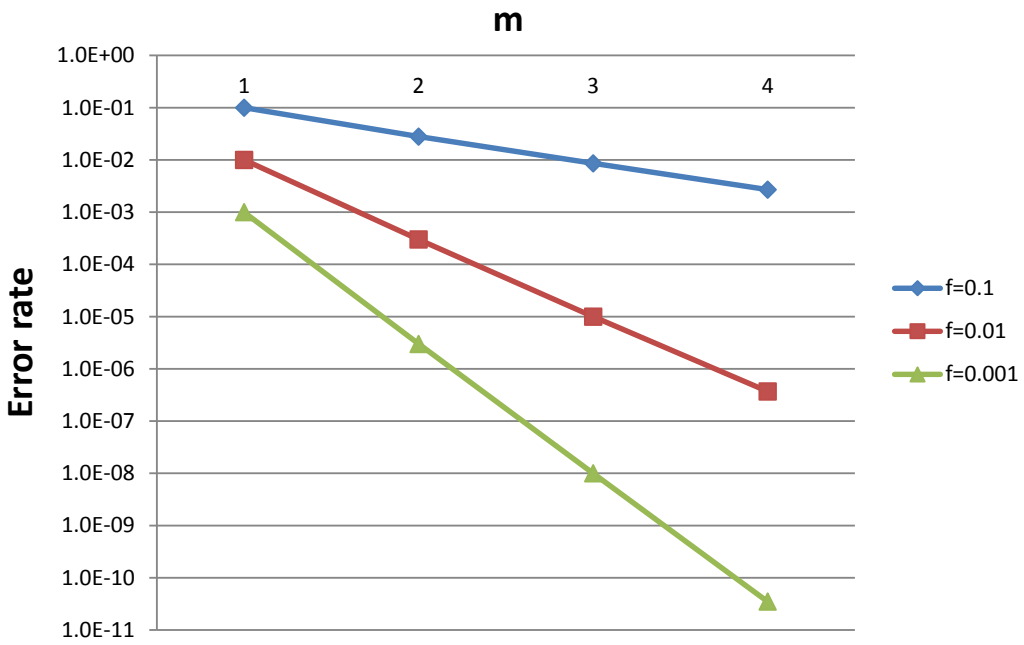

Figure 3.3: Error rates for different parameter values of d and f.

3.6 Answering Research Question 3: Cost Model to Determine Cost Effectiveness of Application

Another challenge that must be dealt with is the cost-effectiveness of gray computing. To assign tasks to a large number of browser clients, a coordinating server is needed to send the input data to clients and collect the computing results from them. The extra data transferred, in addition to the original web page, will consume additional bandwidth and processing power of the server. It is possible that this extra cost can be more than the value of the results computed by the clients for some types of applications. Therefore, we need a cost model to help developers to decide whether it is cost-effective to deploy an application through gray computing and estimate the cost savings.

In our cost model, we use the Amazon EC2 cloud computing service as our benchmark. We chose Amazon's EC2 because it is a cost-effective and highly scalable cloud platform to process big data. Additionally, cloud computing has a quantifiable computing cost that we can compare against. For example, Amazon provides Map/Reduce services whose cost is based on the type and time of the computing resources that are consumed. Note that our cost model is generic and could be parameterized with costs from any cloud provider, but currently we focus on Amazon EC2.

For our gray computing distributed data processing engine, the cost can be broken down as:

$$\begin{aligned} C_{browser} &= C_{transfer} + C_{request} + C_{distribute} \\ &= C_{toclient} + C_{toserver} + C_{request} + C_{distribute} \\ &\approx C_{toclient} + C_{toserver} + C_{request} \end{aligned}$$

$C_{transfer}$ is the data transfer cost calculated on volume of data; $C_{request}$ is the HTTP request cost incurred based on the number of GET or POST requests; $C_{distribute}$ is the virtual compute units consumed by task distribution. We have shown in our empirical analysis that

Table 3.2: Peak task distribution throughput and cost of different EC2 instance types.

EC2 Inst. Type	Throughput	Cost	Price/Throughput
t1.micro	450 tasks/sec	$0.02/hr	$1.23E-8
m1.small	1560 tasks/sec	$0.06/hr	$1.06E-8
m1.medium	3300 tasks/sec	$0.12/hr	$1.01E-8
m1.large	8100 tasks/sec	$0.24/hr	$8.23E-9

$C_{distribute}$ is nearly zero for our architecture because the data is served directly out of the cloud storage service and the task distribution server only needs to do a small amount of work.

As illustrated in Figure 3.1, a task distribution server is needed to dispatch tasks to all the web clients. This server accepts requests from clients and returns the brief task information (e.g., the task ID and task input data location in S3), so that clients can then fetch the correct data in S3 separately. Obviously, the more concurrent clients there are, the higher the throughput the server needs to support.

Our reference implementation architecture, described in Section 3.3, reduces the load on the task distribution server by serving data for processing directly out of the cloud-based storage system. The task distribution server is not involved in serving the large volumes of data that need to be processed or storing the results. To assess the load produced by this architecture, we conducted an experiment to estimate the cost of hosting a task distribution server.

The task distribution server that we implemented exposes an HTTP API to handle HTTP GET requests and responds with a JSON-based string containing the task ID and data location URL for each task. Each client calls the API once before working on an individual task. In order to estimate the number of servers needed to support a given task distribution throughput, we performed a number of load stress tests against different types of general-purpose EC2 servers and calculated the peak throughput each server can support.

As shown in Table 3.2, m1.large has the best performance/cost ratio. Facebook received

about 100 billion hits everyday [63], which is about 4.167 billion hits per hour. Every page hit triggers a task request, so the task dispatching server fleet to handle this traffic load needs to support a throughput of 4.167 billion requests / hour, or 1.16 million requests / second. Using the m1.large server type, a total of 144 servers are required to support such a high throughput, which costs about $34.56 / hour.

The key question is the relative expense of the task distribution compared to the cost of distributing the task data. With such a large volume of page visits, we assume that each client requests 100KB input data from S3 (100KB is a small and conservative estimate considering a lot of web page resources such as images are over 100KB). The larger the amount of data per client, the more insignificant the task distribution cost is relative to the data transfer costs. The total amount of data to be transferred out every hour will be 4.167 billion * 100KB = 388 TB. Based on current S3 pricing [64], this will cost about $37,752 / hour. It can be seen that the cost of hosting the dispatching server only accounts for about 0.09% of the total cost to handle the clients. Because of the small task distribution cost, we ignore it in the use case analyses presented later.

$C_{transfer}$ can be further broken down into $C_{toclient}$, $C_{toserver}$ and C_{code}.

$$
\begin{aligned}
C_{toclient} &= k * I * P_{transferout} \\
&= k * I * (P_{OriginToCDN} + P_{CDNout}) \\
C_{toserver} &= O * P_{transferin} \\
&= O * (P_{toCDN} + P_{toOrigin}) \\
C_{code} &= n * I_{code} * P_{transferout}
\end{aligned}
$$

$C_{toclient}$ is the cost to fetch input data for computing from the S3 storage server; I is the original size of input data to be transferred to clients for processing; $P_{OriginToCDN}$ is the unit price to transfer data from the origin server (S3 in our case) to CDN servers (CloudFront in our case); P_{CDNout} is the unit price to transfer data from CDN servers to the clients. $C_{toserver}$

is the cost to return computation results from clients to the storage server; O is the size of data to be returned; P_{toCDN} is the unit price to transfer data from clients to CDN servers and $P_{toOrigin}$ is the unit price to transfer data from CDN servers to the origin server. C_{code} is the bandwidth cost to distribute the processing logic code, such as hashing algorithms in rainbow table generation, to the browser clients. n is the total number of user visits to the website. During each visit, the client may request and complete multiple tasks, but only one piece of processing code needs to be transferred. I_{code} is the size of the processing code in JavaScript. $P_{transferout}$ is the unit price of bandwidth cost to transfer data from server to client.

The actual data transferred from server to client will be more than the original size of input data I. One reason is described in Section 3.5, the same task needs to be sent to d different clients to be robust to malicious clients. Another reason is the clients may leave before the computation finishes. The data transferred to these clients is wasted. The actual volume of data transferred out will be k times the data needed for one copy of the task, where

$$\begin{aligned} k =& d + \sum_{n=1}^{\infty} d(1-\mu)^n \\ =& d + d*(1-\mu)/\mu = d/\mu \end{aligned}$$

d is the duplication factor. The variable μ is the successful task completion ratio of browser clients (*i.e.*, the percentage of distributed tasks that are processed successfully before a browser client leaves the site).

In Section 3.4, we discuss the estimation of value μ and its relationship with average page view duration, task granularity, and task distribution algorithms.

$$C_{request} = (k+d)*n*P_{request} \tag{3.3}$$

n is the number of tasks that need to be processed. $P_{request}$ is the unit price of HTTP requests to the CDN server. For each task distribution session, the client needs one HTTP request

to fetch the data and one HTTP request to return the results. Since each task needs to be duplicated d times and not all the tasks are completed successfully, more fetch requests are needed than return requests. That is, kn requests to fetch input data and dn requests to return the results.

The cost to run the tasks in the cloud is given by C_{cloud}:

$$C_{cloud} = T_{cloud} * I * P_{unit} \tag{3.4}$$

where P_{unit} is the cost per hour of a virtual compute unit. For example, Amazon provides an "ECU" virtual computing unit measure that is used to rate the processing power of each virtual machine type in Amazon EC2. Virtual compute units are an abstraction of computing ability. T_{cloud} is the computing time to process 1 unit of data with one virtual compute unit in the cloud.

The distributed data processing engine built on gray computing is only cost effective when:

$$C_{cloud} > C_{browser} \tag{3.5}$$

That is, the cost to distribute and process the data in the browsers must be cheaper than the cost to process the data in the cloud.

$$\begin{aligned} T_{cloud} * I * P_{unit} > & k * I * (P_{OriginToCDN} + P_{CDNout}) \\ & + O * (P_{toCDN} + P_{toOrigin}) \\ & + n * I_{code} * P_{transferout} \\ & + (k+d) * n * P_{request} \end{aligned} \tag{3.6}$$

Because prices are all constant for a given cloud service provider, the key parameters here are T_{cloud}, which can be computed by a benchmarking process, and k, which can be computed based on the voting scheme for accepting results and average page view time of clients.

The cost saving U is defined as:

$$U = C_{cloud} - C_{browser} \tag{3.7}$$

A positive U indicates an application is suitable for gray computing.

The benchmarking process to examine whether an application is cost-effective is as follows:

1. Start a cloud computing instance.
2. Put I GB of input data on the node.
3. Launch the data processing task.
4. Measure the time, t, to process the I input data.
5. If $t/I > k * P_{transferout}/P_{unit}$, the computation is worth distributing to web clients.

There is also extra cost for the visitor who is running these tasks, such as CPU resources, power, Internet data costs, bandwidth, etc. The cost model we described above is from the website owners view. It is used to determine whether offloading specific task is cost effective for the website owners. We think this cost-return is the most fundamental thing to analyze because otherwise websites will not be even interested in deploying gray computing.

It is difficult to build a unified cost model for users because users may have different bandwidth costs and preferences. We agree users should have the right to determine whether they are willing to contribute their computing power. Generally we would not encourage users to participate in computation while using battery power and cellular data (as we have mentioned, JavaScript can be configured to run while charging and with WiFi access).

For the power consumption cost, although conditions vary, we can produce a rough estimate. An Intel i5 3570 has an idle power consumption of 67W (http://cpuboss.com/cpu/Intel-

Core-i5-3570) and peak power consumption of 121W. Since the CPU is quad-core, running a Web Worker task will increase the CPU utilization level by 25%, which is roughly an increase of (121-67)*0.25 = 13.5W in power consumption. The electricity cost in our part of the U.S. is around $0.12 per KWh. So a 13.5W increase results in only $0.00162 electricity fee per hour. While the computing power of an i5 3570 CPU is at least equivalent to an m4.large Amazon EC2 instance, which is priced at $0.1 per hour. The electricity cost will only account for 1% of the value of the computational result. Therefore, the electricity cost on the user is minimal.

The key insight here is the computational resources of most commodity computers are wasted and the idle power consumption is high relative to the idle computation that is performed. People provision their computers for peak usage but the actual average utilization rate is typically extremely low. Cloud computing is one approach to increase the utilization rate by centralizing the resources. Our gray computing approach is just another way to increase the utilization rate on commodity computers, but does not add significant marginal cost to the electricity consumption.

3.7 Answering Research Question 4: Improving JavaScript Performance with Code Porting

Moving the computation from a general OS to web browsers, JavaScript is the key language to carry out the computational tasks, so we first set out to assess the performance of JavaScript.

JavaScript was previously not a high-performance language, as early experiments and benchmarks showed [50]. However, as the Internet evolved and web pages became increasingly complex and interactive, there has been a growing demand for high-performance JavaScript engines to run tasks, such as 3D rendering, machine learning, or image processing. For example, the entire Unreal gaming engine has been ported to JavaScript and runs at acceptable frame rates [65].

Because the performance of JavaScript is critical to the user experience on complex sites and directly affects page load and response time, popular browsers such as Chrome, Firefox, and Safari have made great efforts to optimize their JavaScript computing engines and have produced significant increases in JavaScript computational speed.

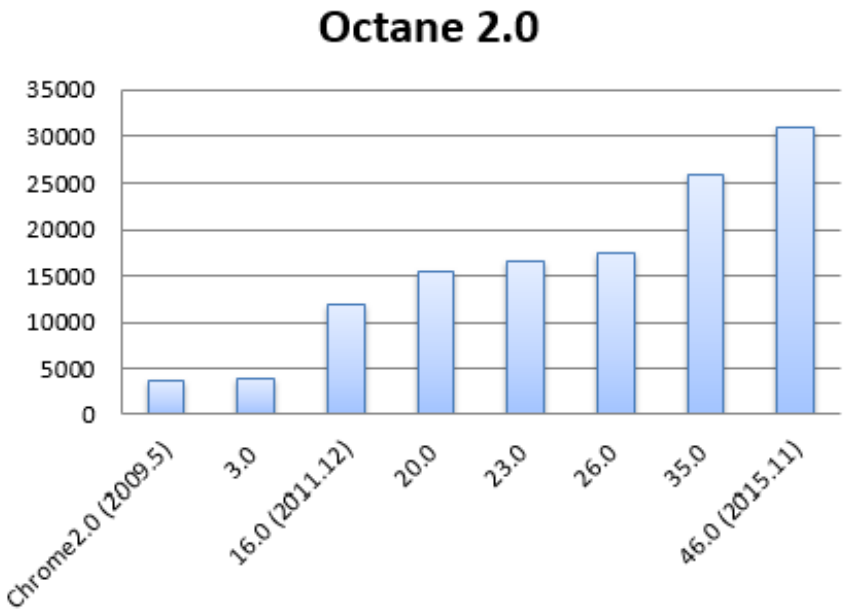

Figure 3.4: JavaScript benchmark scores for different versions of browsers over time.

Figure 3.4 shows the scores of Octane [66], which is a JavaScript benchmark, on different versions of Chrome browsers. These benchmarks include various representative real-world problems. The earliest version of Google Chrome (2.0) dates back to 2009 and the latest version is from 2015. As we can see, the JavaScript Engine performance has improved dramatically in recent years.

Table 3.3 shows performance results obtained from several benchmarks implemented in different languages [1]. The computer language benchmarks game is a site that includes implementations of the same algorithm in different programming languages. We selected JavaScript, Java, PHP, and C for comparison. The performance of JavaScript was tested on the Google V8 JavaScript engine. The following computational tasks were selected for

Table 3.3: Computing Time Comparison of Programming Languages For Representative Computing Tasks [1].

Benchmark	JavaScript	Java	PHP	C
Regex-dna	**3.55s**	20.80s	28.52s	5.46s
NBody	27.90s	21.54s	5min	**9.96s**
k-nucleotide	120.54s	46.38s	45.69s	**34.26s**
fasta	16.5s	5.66s	59.37s	**5.28s**
Binary-Trees	34.3s	**7.50s**	9min	9.64s
SpectralNorm	15.72s	16.37s	7min	**7.85s**
Smith-Waterman	18.3s	12.9s	N/A	**3.17s**
RONN	34.9s	22.5s	N/A	**8.1s**

the benchmarking. Different types of computations that we expected to be similar to useful gray computing tasks were covered.

- **Regex-dna** is designed to test the performance of a regex pattern match on a DNA sequence. Regex pattern match is a common operation for sequence alignment.
- **k-nucleotide** repeatedly updates hashtables and k-nucleotide strings. It is used to test integer operations.
- **SpectralNorm** calculates an eigenvalue using the power method. It tests floating-point operations.
- **Smith-Waterman** is a sequence alignment algorithm implementation that we will discuss in Section 3.11.2.
- **RONN** is a protein disorder prediction program that we will discuss in Section 3.11.2.

As can be seen in Table 3.3, C performs the best, as expected. However, the speed difference between JavaScript and C on the benchmarked computing tasks is at most 6X. Moreover, JavaScript shows absolute advantages over popular server scripting languages, such as PHP. To further examine JavaScript efficiency, we also benchmarked a number of complex computational tasks in the evaluations performed in Section 3.11.2 with real-

world use cases. The results confirm the conclusion above that JavaScript has comparable performance with native C/C++ code.

Although we did not test their capabilities, newer proposed JavaScript libraries and frameworks, such as WebCL [67], enable client JavaScript programs to access client GPUs to further improve computational speed. WebCL is a standard for JavaScript implementation of the OpenCL API, which enables web applications to harness GPU and multi-core CPU parallel processing from within a Web browser. Currently, WebCL is still in draft version and no native support is provided by common web browsers. Therefore, we did not include WebCL in our experiments. However, there is research showing that WebCL can bring up to a 50% reduction in execution time by utilizing the GPU [68], which could be a superior technique for gray computing in the future.

Porting code to JavaScript: Besides the efforts from popular web browsers on improving the JavaScript interpreter, there is another direction of efforts that aims to port existing code from other programming languages to JavaScript. A variety of highly optimized tools have been developed to automatically port C/C++ programs to JavaScript.

Emscripten [69] is a compiler developed by the Mozilla team that compiles LLVM (Low Level Virtual Machine) assembly code into highly-optimizable JavaScript in the asm.js [70] format. It works by first compiling C/C++ code through Clang into an intermediary representation in LLVM. The ported JavaScript code is machine generated and thus different from the native hand-written version. A key question is the amount of performance overhead added by the machine-generated implementation compared to the original implementation. Prior research results [71] showed that optimized JavaScript code can achieve performance of roughly 2x to 8x slower than C++ code. We conducted some benchmark experiments of our own to validate this claim. The results were quite startling: the ported JavaScript code not only outperformed the hand-written JavaScript code, but *achieved performance near the native hand-written C++ code*. Table 3.4 shows the results from the comparison. The first column is the hand-written native C++ code. The

second column is the equivalent hand-written implementation of the JavaScript code. The third column is the ported JavaScript code produced by translating the hand-written native C++ code using Emscripten 1.2.0 with default parameters. The fourth column is the ported JavaScript code using Emscripten with the "-O2" optimization flag. The C++ code was compiled using LLVM gcc 5.0 with the "-O3" optimization parameter. The exact reason for the dramatic performance improvement of the Emscripten generated code is unclear. However, we hypothesize, and will investigate in future work, that the reason for the performance boost is that Emscripten targets a subset of JavaScript called asm.js that is designed to be highly optimizable by JavaScript interpreters. The reduced feature set of asm.js translates into much better performance. Asm.js eliminates JavaScript's object-oriented constructs and thus eliminates many hard-to-optimize code constructs. A more detailed description of asm.js can be found in [71].

Emscripten not only works on single C/C++ files but also complex projects containing hundreds or thousands of source files. The authors of Emscripten have successfully ported hundreds of C/C++ projects into JavaScript and a list of the ported projects is available on Emscripten's homepage [69]. Some of the ported projects include Unreal Engine 4 (3D gaming), Qt (UI), and SQLite (Database), and ffmpeg (video processing). In our own evaluation, we collected five widely used bioinformatics algorithms and successfully ported three of them into JavaScript with Emscripten. Emscripten does not work on all C/C++ projects. In our experiment, porting complex codebases often results in various compilation errors that must be resolved manually. One problem we encountered with Emscripten is that it will not work if the source files contain any file system API calls because JavaScript cannot access files in a Browser. Emscripten provides a workaround by preloading and embedding the files before hand. Emscripten also will not work if the source files contain any multi-threaded code. We will need to manually change the code to a single-threaded structure. Another issue is when there are inline functions declared in a header, which Emscripten cannot recognize and will report as an undefined function error.

We will need to make the function a static declaration. Also, the source file cannot contain any platform-dependent statement. The source code will need to be modified in order to be platform-independent. These types of errors have straightforward fixes, but do require manual intervention. The Emscripten FAQ page [72] describes additional situations where manual interventions are needed for porting code to JavaScript. However, Emscripten is a project is actively developed and maintained, so many of these limitations may be fixed in the future.

Table 3.4: Execution Time Comparison of Native C, Native JS, and Ported JS (from C).

Benchmark	C	native JS	ported JS
Nbody	6.6s	16.9s	6.2s
fasta	4.6s	15.9s	8.8s
SpectralNorm	6.7s	8.6s	7.8s

The Google Web Toolkit (GWT) [73] is an open source tool developed by Google that allows web developers to create JavaScript front-end applications in Java. The Java to JavaScript compiler of GWT, which originally was used to facilitate web development, can be used to port Java-based algorithms. We used GWT to port BioJava functions to JavaScript as part of our evaluation. There are also many successful examples using GWT to port complex Java projects to JavaScript such as Quake 2 [74] and the Eclipse Graphical Editing Framework [75]. The down-side of the GWT approach is that most high performance algorithms are implemented in C/C++. Thus, the application scope of GWT is limited. Another disadvantage is that GWT does not have as many optimizations as Emscripten, so the ported JavaScript code tends to be slower than JavaScript code ported by Emscripten [76].

There are other automated translation approaches that target JavaScript besides Emscripten and GWT: pyjs [77] which compiles Python into JavaScript, and Opal [78] which compiles Ruby into JavaScript. However, we focused on GWT and Emscripten in our experiments. A more extensive evaluation of the various approaches will be one direction of

our future work.

The process of porting code to JavaScript using GWT and Emscripten is straightforward. For GWT, we add the target Java source code to the GWT project and write a wrapper class in Java to call the function in the target source code. The JavaScript front-end is generated automatically. To port a project with Emscripten, the "emmake" command is called instead of "make" to build the project first and the "emcc" command is called on the generated LLVM code to produce a JavaScript file. The situation can be complex when the project has external library references and the porting may require some extra effort. However, the advantage is the porting process can be done by someone with experience only once and the resulting program can serve the general community.

Another issue with the ported code is the code size. We observed that the Emscripten-generated JavaScript code is much larger than equivalent hand-written JavaScript code. This is understandable since the ported JavaScript is more low-level. But it can be an issue since the JavaScript file needs to be transferred to clients for computation and these files would take extra network bandwidth. Two methods could be applied to optimize the JavaScript code size. The first is the Closure compiler [79], which compiles JavaScript to optimized JavaScript by removing dead code and renaming variables. The second way is to compress the file with gzip. This makes sense because gzip is supported by HTTP request headers and the compressed JavaScript files can be easily decompressed on the client side. Table 3.5 shows different implementations of rainbow table generation described in Section 3.11.2. The file size of JavaScript can be significantly reduced after applying the Closure compiler and gzip.

Table 3.5: File Size Comparison for Different Versions of Rainbow Table Generation.

Version	File size
C++	6kb
hand-written JS	4kb
ported JS	574kb
ported JS with closure compiler	464kb
further gzipped	113kb

3.8 Answering Research Question 5: Assessing the Feasibility of Gray Computing on Mobile Devices

With the popularity of mobile devices, mobile visitors are consuming a majority of a websites overall traffic. Gray computing can theoretically be applied to mobile visitors as well. However, the hardware and software environments of mobile devices are different from a typical desktop configuration. For example, mobile devices not only use browsers, but also apps to access web content. Also, mobile devices usually have limited processing power, battery resources and network bandwidth compared to desktops. These differences raise questions on whether gray computing can still work in a mobile environment. To assess the feasibility of applying gray computing to mobile visitors, we conducted experiments to evaluate the performance of running gray computing tasks on mobile devices.

Currently, mobile apps can be categorized into three types: native apps, web apps and hybrid apps. Native apps are apps created with native Android or iOS SDKs. Under the hood, they use Java or Objective-C/Swift. Native apps have access to low-level hardware APIs and are distributed in app stores. Web apps are actually web pages but with the option to be installed on your home screen by creating a bookmark. Web apps rely on HTML/CSS/JavaScript. Web apps are interpreted and sandboxed, and hence have less access to the camera, GPS, accelerometer, etc.

Hybrid apps are web apps in native shells. There are two sub-types of hybrid apps. The first kind of hybrid apps are built with hybrid app frameworks such as Apache Cordova,

Ionic, React Native, Appcelerator, etc. These frameworks allow developers to focus on one codebase (usually in JavaScript), and automatically generate apps that can be run on different platforms such as Android, iOS, etc. These frameworks are becoming increasingly popular among developers because of their cross-platform capabilities. Only one codebase is needed instead of multiple codebases for different platforms. This can significantly speed up the development process and reduce development and maintenance costs. The second kind of hybrid apps are apps built by Native SDK with WebView components. A WebView is like an in-app browser for developers to embed web content inside the apps. It is sometimes difficult to tell them from the native apps. The WebView approach is very popular especially when a lot of apps mainly show static data and are generally mobile versions of the websites. Although we do not have an exact number of what percentage of apps are hybrid apps, many top apps are embracing hybrid design [80].

How does gray computing work in different mobile-app settings? For pure native apps, gray computing might not be directly applicable because JavaScript is not used in native SDKs. However, the architecture of gray computing, the adaptive scheduling algorithm, the cost analysis, and security issues we described are still useful and could be used to manage native tasks distributed to mobile clients. Because web apps use HTML/JavaScript and run in browsers, there is no difference with the desktop browser setting. For hybrid apps, WebViews are very similar to browsers and can be used to render HTML/JavaScript content inside apps. However, WebViews have different implementations under different platforms (UIWebView/WKWebView for iOS and WebView for Android). It is unclear if WebViews can achieve similar performance to desktop browsers and if WebViews enforce any additional limitations on JavaScript.

In order to understand JavaScript performance in hybrid apps, we conducted experiments to test the performance of computationally intensive tasks in different WebView implementations. The task we chose was computing 200,000 SHA1 hashes of sequentially generated strings of length 8. As we can see from Table 3.6 and Table 3.7, newer versions

Table 3.6: Performance comparison of computing 200000 SHA1 hashes on various iOS platforms. Experiments conducted on an iPhone 6 Plus with iOS 10.3.

Platforms	Computation time
Native iOS	3.9s
Safari browser	5.7s
UIWebView	32s
WKWebView	6.0s
React-Native	51s

Table 3.7: Performance comparison of computing 200000 SHA1 hashes on various Android platforms. Experiments conducted on an Android Tango tablet with Android 4.4.

Platforms	Computation time
Native Android (4.4)	8.5s
Chrome browser	6.3s
WebView Android (4.4)	6.7s
WebView Android (4.0)	19.1s
React-native	26.1s

of WebView (WKWebView for iOS and WebView in Android 4.4+) have similar JavaScript performance with the mobile browsers of the corresponding platforms. Legacy WebView versions (e.g., UIWebView for iOS and WebView in earlier versions of Android) can be significantly slower than their mobile browser couterparts primarily because of the lack of support for JIT and other optimizations.

Besides the software environment, gray computing also faces different hardware environments on mobile devices. The processing power of mobile devices are usually considered slower than desktops. However, the performance gap is being narrowed as the processing power of mobile devices has improved rapidly in recent years.

Table 3.8 shows the JavaScript performance comparison of the desktop and mobile devices. A mid-range desktop and several representative mobile devices are chosen to run MD5/SHA1 hashing in JavaScript. Chrome is used for desktop and Android devices and Safari is used for iOS devices. The experiment shows iPhone 7 and iPad Pro have already achieved similar performance of a mid-range desktop in terms of MD5/SHA1 hashing. Of

Table 3.8: Performance comparison of various mobile devices with desktop computers. The table shows the time needed to compute same number of MD5/SHA1 hashes for different platforms.

Devices	MD5	SHA1
Desktop(i5 3570)	12.8s	16.3s
iPhone 7	11.8s	16.1s
iPhone 6	23.2s	31.0s
Samsung Galaxy S7	19.7s	27.4s
iPad Pro (2015 model)	13.2s	18.1s

course, the iPhone 7 is a very high-end smartphone and is not very representative of all mobile devices in use. But the trend is that the smartphones are becoming faster while the processing power of desktops has made limited progress.

Another obstacle with running gray computing tasks on mobile devices is the limited battery life and network data usage. Gray computing tasks may drain the batteries and increase the network data usage cost for devices using a cellular data plan. In general, we do not encourage users to use battery and cellular data plan to perform computation. The computational tasks can be configured to work only when the mobile devices are charged and when WiFi is available by checking the JavaScript Battery Status API [81] and Network Information API [82]. The Battery Status API provides a way for JavaScript to know whether a device is being charged or not and what is the battery charge level. The Network Information API [82] provides information about a mobile device' connection status (e.g., 'Wifi', 'cellular', etc.). The network information API is currently experimental and supported in Chrome and Firefox only.

Users should have the right to turn on or turn off the computation at any time, even when they are using battery and cellular network. Therefore, the gray computing tasks may incorporate a UI element, such as a button, which allows users to start or terminate a gray computing computation at anytime.

Moreover, gray computing is focused on short duration computations instead of long-running tasks. According to statistics from Alexa [47], the average time users spend on

most websites is 3-10 minutes. A few minutes of computation will consume some battery life on a mobile platform, but it is similar in computational demand to viewing a short video. The power consumption for the short duration computations that gray computing is designed for is unlikely to be significant. We are also looking forward to advances in wireless charging. After it becomes practical, the limitations of power consumption will be less impacted.

There are some existing projects that suggest mobile users are passionate with respect to contributing their processing power. Power to Give is an app developed by HTC for smartphone users to contribute their mobile devices' processing power to scientific research. The app has over a million installs so far. Therefore, we believe mobile users have the potential to participate in gray computing tasks regardless of battery and bandwidth consumption.

In summary, we believe gray computing still works for mobile device settings, although with some additional restrictions. Considering the portability and increasingly long time people spend on their smartphones and tablets, mobile devices can definitely contribute significant computing power to gray computing.

3.9 Answering Research Question 6: Examination of User Experience Impact

One concern with deploying gray computing to websites is whether the background computing tasks will affect the website's foreground user interaction tasks. Gray computing will be of little interest to website owners and visitors if the computing tasks affect the user experience heavily.

Web Workers is a new feature introduced in HTML5 [83], which can create a separate JavaScript thread in the background. Using Web Workers, background computing tasks and foreground rendering / user interaction tasks can run in separate threads, allowing the main JavaScript needed to render the page and validate forms to run unimpeded without affecting user experience. Some examples on the usage of Web Workers can be found here [3].

To test whether doing computationally intensive tasks in the background will affect the user experience, we conducted experiments with Tampermonkey [84] to inject data processing tasks into popular websites. Tampermonkey is a browser extension that allows users to install scripts that make on-the-fly changes to web page content before or after the web page is loaded in the browser.

The first concern regarding user experience is page load time. However, Web Workers spawn threads in the *onload()* function of a web page, which is executed only after the HTML page finishes loading. Therefore, background Web Worker threads can be run without affecting page load time.

The second concern is the page responsiveness, which is an important metric to ensure users do not experience any obvious lag while interacting with the page. Many metrics exist for page responsiveness. In our experiments, we mainly consider two metrics that simulate real user scenarios. The first metric is the frames per second (FPS) of the web pages while scrolling or playing animations. If the frame rate drops below a certain threshold, users may feel the web pages are not smooth while interacting. The second metric is the feedback lag after some interaction. We chose the search box item suggestion pop up time for evaluation because it is a pervasive and time-critical feature of many websites.

For the first metric, we used FPS to measure the impact of Web Worker tasks on responsiveness. We measured the FPS under two scenarios. The first scenario is while playing animations. We used an animation speed test [85] to record the FPS rate while the web page is playing various animations. We adjusted the complexity of the animation to make the frame rate around 30 FPS when no background tasks were running. The second scenario is while scrolling up and down the pages. The rendering burden increases dramatically while users scroll the web pages. FPS often drops while scrolling and therefore harms the user experience. We used Jank Meter [86] to record the FPS rate while scrolling up and down the web pages. We measured the FPS differences with a Web Worker tasks and without a Web Worker tasks under different background system loads.

Table 3.9: FPS Rate Under Different System Loads.

System Loads	Average FPS while playing animation
CPU 25%	32
CPU 50%	30
CPU 75%	16
Memory 30%	32
Memory 40%	33
Memory 60%	31
Memory 80%	31
Anti-virus (20% CPU)	30
Anti-virus (70% CPU)	18

We simulated different background system loads of CPU utilization, memory usage and disk access while running Web Worker tasks. For CPU utilization, we used CPUSTRES to simulate a background CPU utilization rate of 25%, 50%, 75%. For the memory usage, we used HeavyLoad to simulate memory usage of 30%, 40%, 60%, 80% on an 8GB RAM computer. For the disk access load, we simulated heavy disk access by running an anti-virus software scan. Notice the anti-virus scanning also contributes 20%-70% CPU utilization while running.

The experiment results in Table 3.9 show a Web Worker task only results in very small reduction in FPS rate when the background CPU utilization level is below 50%. However, the difference becomes larger as the CPU utilization level goes up and when reaches over 70%, there will be a significant drop of FPS rate. For the memory usage, there is no discernible difference after running a Web Worker task under different memory usage percentages. This is probably due to the relatively low memory footprint of most gray computing tasks as shown in Table 3.11. For the disk usage, pure disk access does not result in much difference in the FPS rate. However, the CPU utilization level while performing anti-virus scanning can reach more than 60% in peak, which results in temporary reduction of the FPS rate.

In summary, the CPU utilization level is the key factor affecting the web page respon-

siveness. An additional Web Worker task will not have user perceptive impact on the responsiveness of the web page when the background CPU utilization is below 50%. However, there can be significant slow down when the background CPU utilization level is more than 75%. The assumption we have is that users computers tend to stay idle for most of time while users are surfing the Internet. We did baseline CPU utilization monitoring on a desktop (with an Intel i5 3570 CPU) running Windows 10 using performance monitor tool [87] while the browser was open. The average CPU utilization level is only 10% - 20%. Of course, the situation varies from user to user. Therefore, the Web Worker tasks will have a visible user interaction element for users to terminate the computation at any time. After all, we are only trying to utilize the "idle" computing resources of users.

For the second metric, our goal was to test if the generation of search suggestions would be slowed down by a computationally intensive background task. Two scripts were written in Tampermonkey [84]. Script 1 was setup to inject a search term into the search bar and record how long it took for the web page's foreground client JavaScript to generate the search result HTML elements and produce a baseline time for comparison. Script 2 spawned a Web Worker to perform a computationally intensive task required by the applications we propose in Section 3.11.2, such as face recognition and image scaling, and then injected the search terms. The time to generate auto-completion suggestions while running the computationally intensive background tasks was measured. We compared the time to generate the search suggestions with and without the Web Worker task to see if the background computation would impact the user experience on a number of popular websites.

We used 50 different keywords as search inputs and for each keyword we ran 100 trials and averaged the load times. As the results shown in Table 3.10 illustrate, there was no discernible difference in the search suggestion load time with and without a Web Worker computational task running when the background CPU utilization is below 50%. This is consistent with the results in the previous FPS experiments. The results show that back-

ground gray computing tasks would not be easily discernible by the user – causing both alarm due to the potential for misuse and potential from a distributed data processing point of view. The experiment was conducted on Chrome 41.0. We also conducted experiments in Firefox with the similar browser extension Greasemonkey [88]. However, Firefox is currently a single-threaded architecture. Therefore, computation in a Web Worker of one tab would completely freeze the browser. The Firefox team is working on a project called Electrolysis [89] which adds multiple process ability to Firefox. Thus, the user experience should not be a problem for Firefox users in the near future. For IE, Web Workers are supported since 10.0. Computational tasks cannot be run in the background in earlier versions of IE.

Table 3.10: Comparison of Average Search Box Hint Results Generation Time with and without Web Worker Tasks.

System Load	Search Box Generation Time		
	Twitter	Wikipedia	Gmail
Without Web Worker	0.59s	0.46s	0.73s
With Web Worker (25% CPU util)	0.58s	0.48s	0.75s
With Web Worker (50% CPU util)	0.62s	0.50s	0.75s
With Web Worker (75% CPU util)	0.75s	0.71s	1.13s

To measure the added CPU load on the system, we stopped all other non-essential processes on the system that could consume CPU time. When we loaded the page with a Web Worker's computational task and injected the search term, the utilization rate of Core 1 was increased to 10-15%. The CPU utilization rate of Core 2 increased to 100%. After the search results were loaded, utilization of Core 1 remained around 3% and utilization of Core 2 remained at 100%. When we loaded the page without a Web Worker, only the utilization rate of Core 1 changed and Core 2 remained 0. The results show that the browser could efficiently leverage multiple cores and isolate the CPU load of the background task to prevent it from impacting foreground JavaScript performance.

We noticed that our results are different from some results observed in previous work [90] where a serious performance degradation can be seen due to the volunteer computing

Table 3.11: Memory Consumption of Some Gray Computing Tasks.

Tasks	Peak Memory Usage
MD5 hashing	12MB
Face detection	90MB
Image scaling	67MB
Rainbow table	20MB

clients. When users are working on their computers and take a break to leave it to run some volunteer computing applications, the applications can pollute the system's memory and lead to cache contention and paging. When users come back to work, the performance of the application they use is impacted. The reason gray computing does not suffer from the same issue is that the memory demands of gray computing applications are typically low because the task distribution system pessimistically divides work into small units that can be processed fully and have their results submitted within the possibly short page view durations, which generally produce very small memory footprint.

3.9.1 User Experience Impact Comparison with Online Ads

In our previous experiments, we have shown that gray computing tasks have little impact on user experience when the background system load is not heavy. However, there is an impact when the system load is high. Is it worthwhile for websites to deploy gray computing, and will website visitors accept gray computing?

We can consider this question from another perspective. Gray computing provides a way for website owners to reduce cost or make money. There are some other ways for website owners to monetize their websites. For example, online ads are a very popular and mature way for website owners to make money by displaying advertisement on the websites. How is gray computing compared to online ads in terms of user experience impact and revenue?

Online ads can be visually annoying. Users often do not have a choice but to view what

Table 3.12: User Experience Impact of Web Worker and Ads

	Avg Page Load Time	Frame Rate while Scrolling	
		Average FPS	Minimum FPS
Original Website (with Ads)	8.5s	46	17
With Web Worker	8.5s	45	17
After Ads Block	3.2s	56	31

is displayed. Many ads exist in the main UI thread and the ads are becoming increasingly dynamic and interactive in order to draw attention. As a result, web pages may take longer to load because the need to fetch various ad related resources. Web pages may become less smooth due to the extra system resources consumed by ads. In comparison, gray computing runs silently on a background thread and is less likely to affect user experience. In order to better understand the user experience impact of ads versus gray computing, we conducted experiments to compare the user experience impact of gray computing and ads in terms of two metrics.

First, we compared the impact of ads and gray computing on a website's page load time. We used Adblock to temporarily block all the ads and recorded the page load time before and after blocking. We tested on 100 different URLs on 20 different domains (the URLs are chosen to include a moderate number of ads). As shown in Table 3.12, the average page load time without ads blocking is 8.5s and only 3.2s when the ads are blocked. We can see ads significantly slow down the page load time of web pages. While for gray computing, Web Workers are executed in the *onload()* function, so the computational tasks are started only when the page load finishes. Our experiments confirmed that the page load time is the same with and without Web Worker tasks. Therefore, gray computing has an advantage in page load time compared to ads.

Second, we measured the FPS rate while page scrolling. Browser cache is disabled to avoid affecting rendering speed. Depending on the number and types of ads displayed on the web pages, there are different degrees of slow down to the FPS rate while scrolling.

While for gray computing, the frame rate is consistent while the background CPU utilization level is below 50%.

Gray computing has less user experience impact than ads. But can gray computing help website owners make as much money as ads? We use Google Adwords as an example, the cost per click advertisers pay is around $0.5, the average click through rate is around 0.3% [91]. Since Google as the platform will take some money, the website owners will get less than $0.0015 for one visitor. For gray computing, suppose the average visiting time is 5 minutes and average processing power of visiting clients is equivalent to an EC2 m3.medium instance, which is priced at $0.06/hour. Suppose the utilization rate is 50%. Website owners can get $0.06/2*5/60 = $0.0025 computation value from one visitor. Because website owners need to give part of the revenue to users as incentives, the websites will make less than $0.0025. Depending on the computational task, length of stay of visitors, and power of visitors computers, gray computing can generate comparable revenue value to ads and with less impact on the user experience of activities, such as scrolling and page load speed.

Another perspective is ads only bring revenue to the website owners but gray computing can bring income for both the website owners and users. Further, some tasks that are performed by the users machine can be beneficial to the user (e.g., automatic tagging of friends faces in photos, automatic summarization of reviews to help the user evaluate a product, etc.).

In summary, we are not arguing gray computing is better than ads, because ads are still a useful way for business marketing and for customers to get information. But gray computing can be a viable supplement or alternative for website owners to make money. Website owners may consider reducing the number of ads to improve user experience and deploy some gray computing tasks instead.

3.10 Answering Research Question 7: User Incentives for Gray Computing

There are clearly significant legal and ethical questions around the concept of gray computing if computing tasks are offloaded to website visitors without their consent. To avoid legal and ethical issues, deployers need to obtain explicit consent from users. But why would users be willing to offer their computing resources? There must be incentives provided to motivate them. In the following paragraphs, we provide several potential strategies for incentivizing users.

First, gray computing can be used to support scientific research, such as computational needs in biology, mathematics, and astronomy, as well as computational needs in the life sciences and medicine. Because such interests may be related closely to a visitor's interest, a visitor may be more willing to contribute to computing tasks of these research areas. The distributed computational tasks can be deployed on the websites of non-profit organizations such as environmental, public welfare, academic/educational organizations and government websites.

In addition to supporting scientific and medical research, gray computing can also be applied to general commercial websites that may offer reward incentives to motivate visitors to help with computational tasks. For instance, users visiting online retail sites can be asked if they are willing to participate in the computation process in order to earn shopping points in return. The rewards can also be in other forms besides monetary incentives. For example, for online web games, websites can reward players with virtual goods and money. Such rewards are attractive to players while at almost no cost to the websites. In fact, online games could be provided freely if those playing the games agree to participate in gray computing tasks. This could allow an online game vendor to be a broker for harnessing compute cycles for computational needs of clients who may pay for the service.

There are some other evidences supporting the application of gray computing. For example, Karame et al. [92] proposed the concept of microcomputing in which computing can serve as an exchange medium. Many online content providers, such as online news-

Figure 3.5: Computing can serve as an exchange medium.

papers, videos and games, offer services of small unit value but of large quantity. Surveys have shown that users are reluctant to authorize credit card payments with small transactions due to complex procedures. Content providers are not willing to charge for small transactions as well because the cost of processing a small value transaction can be more than the profits earned [93]. However, content providers cannot always offer free services. They need to find a business model to generate revenue to sustain their online presence. Gray computing can serve as an exchange medium where the website visitors agree to perform computational tasks assigned by website owners while viewing the content for free, and the websites gain profits from the computational tasks accomplished by visitors while not disturbing the users.

Another attractive perspective of gray computing is it can serve as an alternative to online advertisements. Even though most websites seem to be free to view, websites commonly exploit visitors' computational resources to deliver advertisements, collect user statistics, and perform other tasks. Many websites have banners, pop-up windows for advertisements, or other content that utilize bandwidth or computing power of visitors' computers. Online advertisements can be annoying to users and may increase page loading time. If a website must exploit something from its visitors, background computing tasks can be a choice more acceptable than online advertisements because they are less visually intrusive and do not impact page loading time.

There are many possible incentives for users to participate in an opt-in model. One potential motivation is that specific functionality is tied to opting in, which is already the model employed by JavaScript-heavy sites where functionality stops working if JavaScript is turned off. For example, Facebook could automatically tag your friends in photos and notify you when you are tagged in photos, but only if you opt into the gray computing computation that allows this. A feature-based exchange that benefits the user can be a powerful incentive. Another approach would be to allow the user to turn off a percentage or all ads on the page in exchange for opting into the computation. In exchange for agreeing to run background computations, the user benefits with faster page load times, better scrolling / UI experience, and a less cluttered page. Finally, projects such as BOINC and Folding@Home show the public has sufficient interests in scientific research. Gray computing is light-weight and does not need users to install any client-side software, which offers the potential to attract a large group of people.

Another incentive for users is for a website to offer some form of rewards. For instance, users visiting online retail sites can be asked if they are willing to participate in the computation process in order to earn shopping points in return. The rewards can also be in other forms besides monetary incentives. For example, for online web games, websites can reward players with virtual goods and money. Such rewards are attractive to players while at little to no cost to the websites. The concept can be applied to many online content providers such as online newspapers, videos, and games. They offer services of small unit value but of large quantity. Gray computing can be another way for websites to aid in monetizing their services.

3.11 Empirical Results

In this section, we present some empirical results we have produced so far in assessing gray computing applications.

3.11.1 Experimental Environment

Browser client: We used a desktop computer for our experiments with an Intel Core i5-3570 CPU clocked at 3.5GHz and 8GB of memory. The computational tasks were implemented in JavaScript and executed on Mozilla Firefox 31.0.

Cloud instance: We used an Amazon EC2 m3.medium instance (ECU=3) and Ubuntu 14.04 64bit operating system for benchmarking cloud data processing performance.

Price for cloud computing: See Table 3.13 and 3.14. $P_{unit} = \$0.067/hour$. We chose a duplication factor $d = 2$ for most use cases, because it guarantees a relatively high accuracy as we have shown in Section 3.5. We also chose a conservative successful task completion ratio of $\mu = 0.8$, which is representative of the Hadoop scheduler efficiency with an average page view time of roughly 30s.

We compute the cost saving: $U = T_{cloud} * I * P_{unit} - C_{toclient} - C_{toserver} - C_{request}$ for each task, as shown in Table 3.19. A data processing task is cost-effective for distributed processing with browsers if U is a positive number.

Table 3.13: A Subset of Amazon Cloud Service Type and Price
(As of October 2017)

Service	Subtype	Price	ECU
EC2	t2.micro	\$0.013 /Hr	varied
	m3.medium	\$0.067 /Hr	3
	m3.large	\$0.133 /Hr	6.5
	c3.large	\$0.105 /Hr	7

3.11.2 Empirical Case Study of Gray Computing

In order to quantitatively understand the cost effectiveness of gray computing framework, we examine some practical applications such as face detection, sentiment analysis and bioinformatics applications on gray computing. We compute the cost saving for each application to find out which types of example applications are cost-effective in a gray

Table 3.14: A Subset of Amazon Cloud Service Type and Price
(As of October 2017)

Service	Subtype	Price
S3	Transfer IN To S3	Free
	S3 to CloudFront	Free
CloudFront	To Origin	$0.02/GB
	To Internet (First 10TB)	$0.085/GB
	To Internet (Next 40TB)	$0.08/GB
	To Internet (Next 100TB)	$0.06/GB
	To Internet (Next 350TB)	$0.04/GB
	HTTP requests	$0.0075/per 10000

computing model.

3.11.2.1 Face detection

Overview. For the first gray computing case study, we chose face detection, which is a data processing task that Facebook runs at scale on all of its photo uploads to facilitate user tagging in photos. Face detection is a task that most Facebook users would probably consider to be beneficial. Face detection also has an interesting characteristic in that it can be run on the data that is already being sent to clients as part of a web page (e.g., the photos being viewed on Facebook). Notice in our cost model, a large portion of the cost comes from sending input data from the server to clients. This cost becomes prohibitively expensive for data intensive tasks. Data transfer costs appear inevitable, since it is impossible for the clients to compute without input data. However, there are some circumstances when the input data needs to be sent to the client anyway, such as when the data is an integral part of the web page being served to the clients. In these cases, no extra data transfer costs are incurred. Facebook has 350 million photos uploaded every day [94]. Face detection in photos is a relatively computationally intensive task. Our hypothesis was that significant cost savings could be obtained through gray computing.

Experimental Setup. To test whether offloading face detection tasks to gray computing is cost-effective, we conducted experiments for face detection tasks on both JavaScript

in the browser and OpenCV, which is a highly optimized C/C++ computer vision library, in the cloud. For the JavaScript facial detection algorithm, we used an open-source implementation of the Viola-Jones algorithm [95]. For the Amazon cloud computing implementation, we use the same algorithm but a high performance C implementation from OpenCV 2.4.8. There are more advanced face detection algorithms that achieve better accuracy, but need more computing time, which would favor browser-based data processing.

Empirical Results. The results of the experiment are shown in Table 3.15. We can see that the computation time is approximately linear to the total number of pixels or image size. This result is expected because Viola-Jones face detection uses a multi-scale detector to go over the entire image, which makes the search space proportional to the image dimensions.

Analysis of Results. Suppose the average resolution of the 350 million photos being uploaded is 2 million pixels, which is less than the average resolution of most smartphone cameras, such as the 12 million pixel (12 megapixel) iPhone 6S camera. It takes $1.7s * 3.5 * 10^8/3600h = 165,278h$ of computing time for an EC2 m3.medium instance to process these photos. With our utility function $U = T_{cloud} * I * P_{unit} - 1/\mu * d * I * P_{transferout} - O * P_{transferin} - C_{code} - C_{request}$, where $P_{transferout} = 0$ because the photos already exist in clients so there is no additional cost for transferring photos to the clients. The only extra cost is transferring the face detection codes. In our experiment, $I_{code} = 50KB$. The extra bandwidth consumed in a month is $350 * 10^6/5 * 50 * 10^{-6} * 30 = 105TB$. $C_{code} = 0.085 * 10 * 1000 + 0.08 * 40 * 1000 + 0.06 * 55 * 1000 = \$7350$. For each photo, the returned computing result should be an array of coordinations of rectangles, which is rather small, so O can be ignored. The client does not need to fetch the photos but one HTTP request to return the results. We assume the client processes 5 photos at a time (which takes around 7s on a browser with average hardware). We chose a duplication factor $d = 1$ since this application is not very sensitive to security. Thus, $C_{request} = 350 * 10^6/5 * 0.0075/10^4 = 52.5$, and $U = 165278 * 0.067 - 7350 - 52.5 = \$3671$ could be saved each day by distributing

Table 3.15: Computing time comparison for face detection tasks.

Image Dimension	Number of pixels	Size	JS Computing Time(s)	EC2 Computing Time(s)
960*720	0.69 million	82KB	0.67	0.56
1000*1500	1.5 million	156KB	1.05	1.30
1960*1300	2.55 million	277KB	1.70	2.15

this task to browser clients rather than running it in Amazon EC2. That is a yearly cost savings of roughly $1.34 million dollars. The actual algorithm Facebook uses is likely to be more complex than the algorithm we used and a larger T_{cloud} is expected. Therefore, the cost savings could be even larger than we calculated.

3.11.2.2 Image Scaling

Overview. Another example of a useful gray computation that can be applied to data already being delivered to clients is image scaling. Websites often scale image resources, such as product images, to a variety of sizes in order to adapt them to various devices. For instance, desktop browsers may load the image at its original size and quality, while mobile clients with smaller screens may load the compressed image with lower quality. The Amazon Kindle Browser talks directly to EC2 rather than target websites and receives cached mobile-optimized versions of the site and images that are produced from large-scale image processing jobs.

We can offload image compression tasks to gray computing. Similar to face detection, the photos are already being delivered to the clients, so there is no added cost for transferring data from the server to clients. After loading the images, each client will do the scaling and compression work and then send the scaled images for mobile devices back to the server. When the number of images to process is large, the saved computation cost could be substantial.

Experimental Setup. There are many image scaling algorithms and they achieve different compression qualities with different time consumptions. We wanted to measure

Table 3.16: Computing Time Comparison for Image Resize Tasks.

Image Dimension	Size	JavaScript Computing Time(s)			EC2 Computing time(s)		
		Scaling ratio 30%	50%	70%	30%	50%	70%
960*720	82KB	0.14	0.35	0.66	0.18	0.22	0.25
1000*1500	156KB	0.29	0.69	1.34	0.43	0.45	0.58
1960*1300	277KB	0.48	1.2	2.08	0.66	0.73	0.86

JavaScript's computation speed on this task and did not want to focus on the differences in efficiency of the algorithms. We chose bicubic interpolation [96] for image scaling on both the browser and server. Bicubic interpolation is implemented in JavaScript and C++ for both cloud and browser platforms.

Empirical Results. The experiment results in terms of computation speed are shown in Table 3.16.

Analysis of Results. To obtain a quantitative understanding of the cost savings by offloading the image scaling tasks to gray computing, we again use Facebook as an example and make several assumptions: suppose the average resolution of the photos being uploaded is 2 million pixels and the average scaling ratio is 50%. 350 million uploaded photos daily take $0.55s * 3.5 * 10^8/3600h = 53,472h$ for one EC2 m3.medium instance to scale and compress. With our utility function $U = T_{cloud} * I * P_{unit} - k * I * P_{transferout} - O * P_{transferin} - C_{code} - C_{request}$. Again $P_{transferout} = 0$ because the photos already exist in clients so there is no additional cost for transferring images to the clients. The only extra cost is transferring the image scaling algorithm code. In our experiment, $I_{code} = 3KB$. The extra bandwidth consumed in a month is $350 * 10^6/5 * 3 * 10^-6 * 30 = 6.3TB$. $C_{code} = 0.085 * 6.3 * 1000 =$ \$535.5. We assume the client processes 10 photos at a time (which takes around 10s on a browser with average hardware). We chose a duplication factor $d = 1$ since this application is not very sensitive to security. $O = 3.5 * 10^8 * 80KB/10^6 = 2.8 * 10^4 GB$. $C_{request} = 3.5 * 10^8/10 * 0.0075/10^4 = 26.25$. Therefore, $U = 53472 * 0.067 - 2.8 * 10^4 * 0.02 - 535.5 - 26.25 =$ \$2461 is saved each day by distributing this task to browsers, which aggregates to

$898,219 a year.

There are many techniques that can be used to further improve the quality of the resized images, such as sharpening, filtering, etc. These techniques require additional computation time (larger T_{cloud}), so the amount of money saved could be further increased if these techniques are applied to uploaded images.

3.11.2.3 Sentiment Analysis

Overview. Sentiment analysis [97] refers to using techniques from natural language processing and text analysis to identify the attitude of a writer with respect to some source materials. For instance, Amazon allows customers to review the products they purchased. It would be useful to automatically identify the positive/negative comments and rank the products based on the customers attitude. Since the comment context is sent to the websites' visitors anyway, there will be no extra cost incurred due to the data transferred from servers to clients.

Experimental Setup. There are many machine learning algorithms proposed for sentiment analysis in the literature [97]. We implemented a Naive Bayes Classifier, a simple but quite effective approach, for sentiment analysis. The classifier takes as input a user review and predicts whether the review is positive or negative. The classifier is implemented with JavaScript for the browsers and Python for EC2. We trained our classifier with a movie review dataset [98] containing 1000 positive and 1000 negative reviews. We collected movie reviews from the Internet as test data and partitioned them into 10 files each of size 200KB. Then we used the trained classifier to predict the attitude of each review item and recorded the time needed.

Empirical Results. For an input size of 200KB, the prediction time of browsers with JavaScript is 0.3s and for the same task running on an EC2 m3.medium instance, the prediction time is 0.7s.

Analysis of Results. For 1GB of input data, the cost saving is $1000/0.2 * 0.7/3600 *$

$0.067 = \$0.065$. The size of sentiment analysis code I_{code} is 5KB. To obtain a quantitative understanding of how much money can be saved by offloading the sentiment analysis tasks to gray computing, we use Facebook as an example and make several assumptions: suppose each user uploaded photo has an average of 1Kb comments. Since Facebook has 350 million photos uploaded daily, there is $350 * 10^6 * 1/10^6 = 350GB$ comments in total. If Facebook wants to analyze the attitude of every comment of the photos, the cost is $350 * 0.065 * 365 = \$8304$ a year.

3.11.2.4 Word count

Overview. Word counting is the classic use case that is used to demonstrate Big Data processing with MapReduce. Word counting requires determining the total occurrence of each word in a group of web pages. It is a task that requires a relatively large amount of textual input with a very small amount of computational work.

Experimental Setup. We compared the cost-effectiveness of running a word count task in JavaScript versus the Amazon Elastic Map Reduce (EMR) service (using a script written in Python). The Amazon EMR experiment was configured with 1 master node (m1.medium) and 2 slave nodes (m1.medium).

Table 3.17: Computing time comparison for wordcount.

Input size	Browser(JavaScript)	Amazon EMR
18MB	13.7s	94s
380MB	3min	6min

Empirical Results. The experiment results in terms of computing speed are shown in Table 3.17. For 1GB of input, $C_{request}$ is very small, the cost saving is $1/0.38 * 6/60 * 0.109 * 2 - 0.14 * 2.5 = \$ - 0.29$.

Analysis of Results. As can be seen, the cost saving is a negative number which means the value of the computed results is less than the cost of transferring the text data to the clients. Word counting is not an ideal application for distributed data processing with gray

computing because it is a data intensive but computationally simple task. However, if the word count was being run on web pages delivered to the clients, the data transfer cost would not be incurred and it would be as cost effective as in face detection and image scaling.

3.11.2.5 Pairwise Sequence Alignment with the Smith-Waterman Algorithm

Overview. In bioinformatics, sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Designing high-performance algorithms for aligning biological sequences is a challenging problem. Biological sequences are made up of residues. In DNA sequences, these residues are nucleic acids (e.g.,ATCG), while in protein sequences, these residues are amino acids.

DNA replication errors cause substitutions, insertions, and deletions of nucleotides, leading to "edited" DNA texts. Since DNA sequences are subject to insertions and deletions, we cannot directly compare the i-th symbol in one DNA sequence to the i-th symbol in the other. Instead, Edit distance is used. Edit distance is defined as the distance between two strings as the minimum number of editing operations needed to transform one string into another, where the edit operations are insertion of a symbol, deletion of a symbol, and substitution of one symbol for another.

The Smith-Waterman algorithm is a well-known dynamic programming algorithm for performing local sequence alignment for determining similar regions between two DNA or protein sequences. The algorithm was first proposed by T. Smith and M. Waterman in 1981 [99]. One main drawback of the SmithWaterman algorithm is its high time complexity: To align two sequences with lengths m and n, $O(mn)$ time is required. Many algorithms (*e.g.*, BLAST [100] which uses a heuristic algorithm) tried to improve the speed, but at the cost of accuracy.

While BLAST is faster than Smith-Waterman, it cannot guarantee the optimal alignments of the query and database sequences as Smith-Waterman, so some matches between

query sequence and database sequence may be missed. Thus, the Smith-Waterman algorithm still performs significantly better than BLAST in terms of accuracy [101, 102].

Experimental Setup. We used SSEARCH36.0, which is the Smith-Waterman algorithm implementation from the FASTA sequence analysis package [103] for the cloud computing test. The JavaScript implementation was obtained by porting the BioJava 3.0 implementation to JavaScript using the Google Web Toolkit 2.6.

Empirical Results. The experiment results can be found in Table 3.18.

Table 3.18: Computation time of pairwise sequence alignment (S1:query sequence, S2:reference sequence).

S1 Length	S2 Length	ssearch36 (cloud)	JavaScript
2096	3583	1.84s	5.79s
993	9904	2.8s	9.24s
2096	9904	5.3s	16.8s

Analysis of Results. It can be seen that the computation time is approximately linear to the length of the query sequence and reference sequence. To get a quantitative understanding of the cost savings, we assume that a research lab wants to align 10^7bp (10MB) of newly produced sequences. They need to align these sequences to a database of 10^{10}bp (10GB) to identify the function of the sequence. Suppose the average length of a produced sequence is 1000 residues (about 1KB), and we partition the database into chunks of roughly 50KB. The cost to run this task using cloud servers is: $14s/3600 * 0.044 * 10^5 * 2 * 10^4 = \$342,000$. The cost to run this task with browsers given the duplication factor $d = 2$ is: $0.12 * 2.5 * 60 * 2 * 10^5 * 10^3/10^6 = \$3,600$. Because the search is done on a large dataset and the best match can happen in any place, even when there are small amounts of malicious clients, the probability they affect the final result is very small. Therefore, the duplication is actually not necessary for this task and the cost can be further reduced to $0.12 * 60 * 2 * 10^5 * 10^3/10^6 = \$1,440$. We can see that the cost to run a Smith-Waterman search on a large dataset is prohibitive for normal labs, while the cost is acceptable with a distributed browser-based approach.

3.11.2.6 Multiple Sequence Alignment

Overview. Multiple Sequence Alignment (MSA) is a sequence alignment of three or more biological sequences. MSA is important in many applications such as phylogenetic tree construction and structure prediction. Finding the global optimum of MSA for n sequences has been proven to be an NP-complete problem [104]. Heuristic approaches try to find the multiple alignment that optimizes the sum of the pairwise alignment score. Progressive methods first estimate a tree, then construct a pairwise alignment of the subtrees found at each internal node.

MUSCLE (MUltiple Sequence Comparison by Log-Expectation) [104] is one of the most widely-used and best-performing multiple alignment programs according to published benchmark tests, with accuracy and speed being consistently better than CLUSTALW [105].

Experimental Setup. We used the PREFAB3.0 database for our experiment. The database contains 1,932 alignment cases averaging 49 sequences of length 240. All the MSA test cases are below 45KB in file size, which is feasible for transferring through a web page. We used latest implementation of MUSCLE (3.8.31) [106] for the cloud computing test. The JavaScript version was obtained by compiling the source code of MUSCLE with Emscripten 1.1.6 64bit.

Empirical Results. We ran the muscle program on 100 alignments randomly chosen from the PREFAB dataset. The processing time on a m1.small instance on average was 0.675s/KB. For the JavaScript version, the processing time on the desktop on average was 2.13s/KB.

Analysis of Results. We apply the cost model to the experiment result we obtained. The cost savings per GB is: $10^6 * 0.675/3600 * 0.044 - 0.12 * 2/0.8 = \$7.95$. This shows the browser-based approach is more cost-effective than a cloud computing solution.

3.11.2.7 Protein Disorder Prediction

Overview. Many proteins contain regions that do not form well-defined 3-dimensional structures in their native states. The accurate recognition of these disordered regions is important in molecular biology for enzyme specificity studies, function recognition, and drug design. The detection of a disordered region is also essential in structural biology since disordered regions may affect solubility or crystallizability. RONN (Regional Order Neural Network) [107] is a software approach, using bio-basic function neural network pattern recognition, to detect natively disordered regions in proteins.

Experimental Setup. We acquired a local version of RONN in C++ from the RONN authors. The JavaScript version is obtained by compiling the source file in C++ with Emscripten 1.1.16. The input of the RONN algorithm is a sequence of proteins in FASTA format. We used 100 protein sequences with an average length of 5,124 residues from the PREFAB datasets.

Empirical Results. The processing time on a m1.small instance is an average of 15.9s/KB. For the JavaScript version, the processing time on a desktop is an average of 17.2s/KB.

Analysis of Results. We apply the cost model to the experiment results we obtained. The cost savings per GB is: $10^6 * 15.9/3600 * 0.044 - 0.12 * 2.5 = \$194$. This shows that a browser-based approach is more cost-effective than cloud computing for RONN.

3.11.2.8 Summary

Table 3.19 shows the comparison of cost saving per GB for different tasks. As we can see, computationally intensive tasks such as rainbow table generation and bioinformatics algorithms have very high cost saving per GB and thus are more suitable for gray computing. One characteristic of these tasks is they often involve repetitive calculation of multiple iterations. Some other promising problems include Monte-Carlo simulations, simulated annealing, evolutionary/genetic algorithms, etc. Face detection, image scaling and

Table 3.19: Comparison of cost saving per GB for different tasks. The higher this value, the more computationally intensive the task, and the more suitable the task for our distributed data processing with browsers.

Task	Cost savings $ per GB
Rainbow table	∞
Protein disorder	194
MSA	7.95
Face detection	0.09
Sentiment analysis	0.065
Image scaling	0.008
Word count	-0.29

sentimental analysis tasks have moderate cost saving per GB, but considering the massive volume of the corresponding image/text data, the potential is still huge. For tasks such as word count, they are more data-intensive instead of computationally intensive. Only very simple processing has been done on the data. Therefore, it is not worth to distribute the data to some remote clients for processing.

Chapter 4

Browser Resource Stealing Attack

4.1 Overview

In our previous work with Gray Computing, we consider the cases where websites obtain explicit consent from users for background computation. However, the use of Web Worker does not require explicit permission from users. Given the huge computing power Gray Computing can tap, it is possible that malicious attackers use gray computing to exploit unwittingly users' computing resources.

In this chapter, we analyze a new potential attack vector – *browser-based resource stealing attacks*. These attacks are performed by embedding JavaScript code in the webpages delivered to users in order to gain access to the processing power on their machines. While users are surfing the Internet, these background resource-stealing JavaScript tasks allow the attackers to use victims' browsers to execute computational tasks.

"Great Canon", a large-scale DDoS attack against Github [9] in 2015 has shown the immense computational potential that resource-stealing attacks can tap. The JavaScript code in the analytics services of Baidu, China's largest search engine, were hijacked to include JavaScript code that instructed visitor's browsers to send HTTP requests to a series of victim Github pages. The attack was able to send 2.6 billion requests per hour at its peak [10]. The result was the largest DDoS attack in Github's history and caused Github to be intermittently unavailable for 5 days. The attack drew massive public attention as it only leveraged the resources of the browsers of unsuspecting website visitors and was not due to compromised machines or unpatched vulnerable application versions.

To launch Web Worker attacks, a large amount of browsers need to visit a compromised webpage. One way of doing this is to hack a high-traffic website and inject malicious JavaScript code. However, popular websites are usually well-managed and protected by

strict security measures, therefore not easy to compromise. Alternatively, attackers can create their own websites and propagate the sites through online advertisement services. Online advertisement is a mature business model on the Internet. Various Ad providers provide platforms for businesses to display banners or image Ads on websites to promote marketing messages.

As part of this research, we show that current ad networks, such as Google AdWords, allow attackers to perform large-scale computational resource theft by displaying ads that launch Web Worker tasks. Our analysis shows that attackers can use a number of strategies to generate ads that generate large numbers of impressions but cost very little due to their low click-through rate. Despite their low cost, each ad impression allows an attacker to steal computational resources from the browser that the ad is displayed in. For many applications, we show that the economics of an ad-based resource stealing attack are attractive to an adversary.

As the core language of the Web, JavaScript has been used by attackers in Cross Site Scripting (XSS), drive-by download [108], cross-site request forgery [109], and other attacks. However, the computing capability of JavaScript and the computing resources that can be stolen from browser clients has been largely undervalued. In this paper, we assess these resource stealing attacks which utilize the computing power of JavaScript to perform distributed computing for various malicious purposes. Complex attacks, such as distributed password cracking, were not feasible before because of their significant impact on foreground JavaScript performance, which would alert users or make them leave the page, but are now possible due to three major JavaScript advances in recent years. 1) JavaScript used to be based on a single-threaded architecture and intensive computation would significantly slow down the page and draw users' attention. With the Web Workers API introduced in HTML5, resource-intensive computation can happen in a background thread. Our experiments [7] have shown that Web Worker computation in the background has no discernible impact on the foreground user experience for a multicore CPU. 2) Popular browsers com-

pete fiercely to optimize their JavaScript engines and there has been a significant increase in JavaScript computational speed. Modern browsers are able to complete complex and computationally intensive tasks such as 3D rendering and image processing [110]. Furthermore, techniques such as asm.js [69] and NativeClient [111] can further improve JavaScript performance to near-native levels. 3) As a major UI evolution, tabs instead of windows in browsers, lead to parallel browsing behavior of users and web pages that are left open for long periods of time. Users tend to keep more tabs open and each tab open for longer, which gives background Web Worker tasks associated with a particular web page a longer period to operate and steal computational resources.

Open Question $\Rightarrow$ Are Browser Resource Stealing Attacks a Significant Threat Vector? In past work [7], we have demonstrated the feasibility of legitimate uses of Web Worker background computing to perform computational tasks beneficial to both the browser owner and the website operator, which is termed *Gray Computing*. However, it remains a question whether these same Web Workers, if used by attackers, can be a significant threat vector. In particular, we would like to know what attacks could be launched through Web Workers? Are Web Worker-based attacks economical and attractive to attackers even when they have other choices such as renting botnets or cloud computing?

People have shown interest in misusing browsers for web attacks. Lam et. al [44] propose Puppetnets which is a distributed attack infrastructure misusing web browsers. However, the attacks assessed were limited to single-threaded web pages performing tasks that were not computationally intensive, such as worm propagation, click fraud, etc. In contrast, Web Worker-based attacks run in the background and can do substantial computational work without impacting foreground JavaScript performance and tipping off users to their presence.

The resource stealing attacks with Web Workers that we assess in this paper can be seen as browser-based botnets where the bot is not an infected computer, but a browser visiting a compromised website. The concept of a browser botnet was first proposed by

Kuppan [45]. They introduced the idea of using Web Workers as a potential attack vector for a DDoS attack. However, they did not consider the cost to launch such attacks and whether the economics and available computing power would be attractive to attackers. Pellegrino et. al [46] present a preliminary cost analysis on browser-based DDoS attacks. Their analysis is limited to DDoS attacks while we analyze a broader set of computational tasks, such as distributed password cracking, and provide concrete comparisons with cloud computing providers.

Contributions.

- We demonstrate that current ad networks allow Web Workers to be spawned by ad impressions, which creates a significant threat for mass resource stealing. We provide a deep analysis of the economics of resource stealing through ad impressions and show that it is an attractive target for many types of computationally intensive applications.

- We analyze the economics behind resource stealing attacks and compare the cost models for Web Worker attacks, botnets, and cloud computing. The cost models take into account the bandwidth/request costs to distribute the computing tasks and allow the quantitative comparison of the cost of different approaches to see whether Web Workers are economically attractive.

- We evaluate the performance and user experience impact of Web Worker resource stealing attacks. We take advantage of several state-of-art projects to optimize JavaScript computing speed and port a number of computationally intensive and potentially malicious applications to JavaScript. We also compare the computing performance with/without Web Workers as well as performance differences of resource stealing attacks across web browsers.

- We assess various ways that resource stealing attacks could misuse a browser's resources, including DDoS attacks, distributed password cracking, rainbow table gen-

eration, and cryptocurrency mining. For each attack vector, we compare the pros and cons of launching attacks through a botnet, cloud computing provider, and Web Workers in a browser.

- We evaluate the attacks in the latest context of cloud computing and mobile computing. We assume that the attackers are able to use cloud services to build their distributed browser attack infrastructure. We also compare the cost of running the computing tasks in a browser versus on the cloud to see whether it is cost effective for attackers to deploy resource-stealing computations in browsers. In addition to traditional browsers running on desktops or laptops, we also cover mobile device performance in our evaluations. Several unique challenges and threats to mobile devices of Web Worker resource stealing attacks are identified and assessed.

4.2 Research Questions

4.2.1 Research Question 1: What Types of Attacks can Browser Resource Stealing Attack Launch?

The attacks we analyze include DDoS attacks, distributed password cracking, rainbow table generation and cryptocurrency mining. Although these attacks are commonly conducted with botnets, dedicated servers, or cloud computing, it still remains an open question as to whether or not it is practical to launch these attacks with Web Workers and JavaScript.

4.2.2 Research Question 2: How Does the Web Worker Variant of the Resource-stealing Attack Compare to the Non-browser-based Attack?

The most popular way for attackers to launch DDoS attacks is usually to build or rent a botnet. With the emergence of cloud computing, rented cloud virtual machines also become an alternative for attackers to conduct malicious computation at scale. It is critical

to know how the browser-based Web Worker attack variants compare to the non-browser based approach in terms of scale, detectability, etc.

4.2.3 Research Question 3: Is Browser Resource Stealing Attack Economically Attractive to Attackers?

The primary motivation for many cyber attacks today is profit [37]. For example, attackers can lease or sell their botnet and earn millions of dollars [112]. Therefore, one important question is that whether launching Web Worker resource-stealing attack is economically viable compared to attacking through other mediums such as rented botnets and cloud computing.

4.2.4 Research Question 4: How can Browser Resource Stealing Attack be Detected and Prevented?

If browser resource stealing attack is feasible and can bring significant damage, it is essential for us to find countermeasures to detect or prevent these attacks to ensure a safer web browsing environment.

4.3 Answering Research Questions

4.3.1 Answering Research Question 1: Feasible Attack Types

4.3.1.1 DDoS Attack

Distributed Denial-of-service (DDoS) attacks have been known for many years. DDoS attacks are characterized by an explicit attempt to make an online service unavailable by overwhelming it with traffic from multiple compromised systems [113]. Attackers usually inject slave or "bot" computers with remote command and control services. The compromised computers are known as bots since they are under the control of the attackers and

are used to send requests simultaneously to the target services. DDoS attacks have been the most prominent threat for website owners in recent years. Studies show the average loss for companies incurred from DDoS attacks is estimated to be $1 million per day [114]. Web Workers could be a serious new threat for DDoS attacks, since they do not rely on compromising a host, but instead getting it to visit a specific web page. The architecture of a potential Web Worker DDoS attack is shown in Figure 4.1.

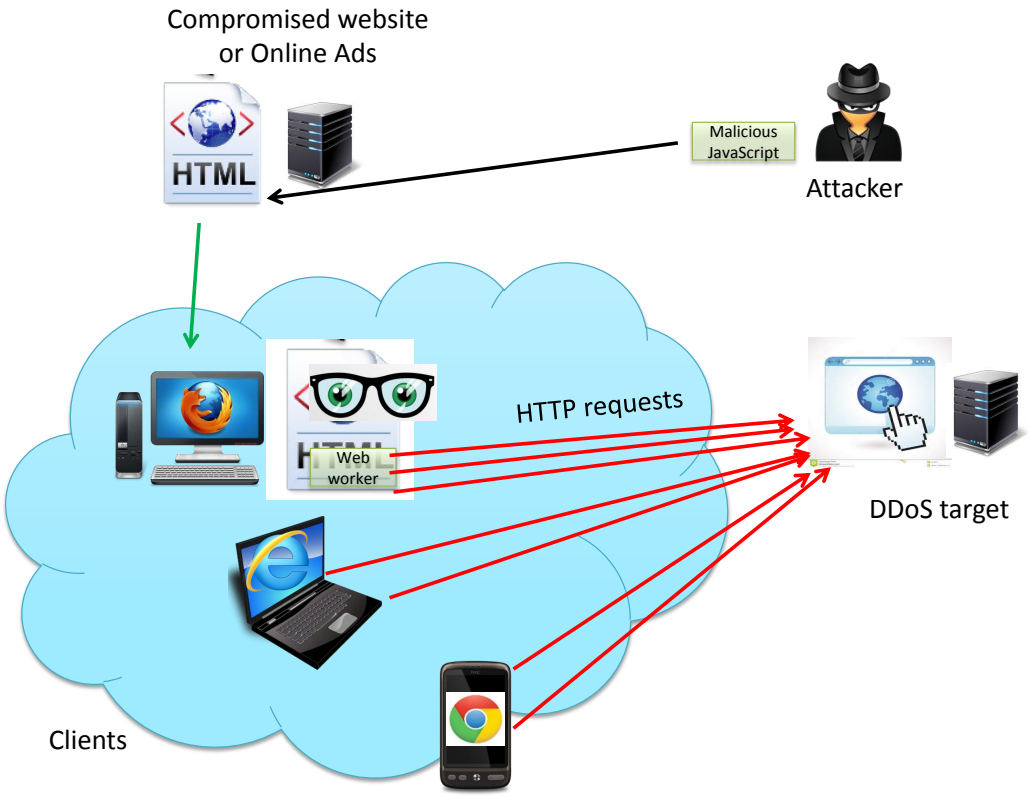

Figure 4.1: Architecture of a Web Worker DDoS Attack.

There are various ways attackers can make client browsers send requests to target services. For example, attackers can embed a src address in an image tag: pointing to the victim service. Or an attacker can send an AJAX GET or POST request from JavaScript. In the past, sending AJAX requests to external domains was impossible due to the restrictions of the "same-origin policy" enforced by most websites. The "same-origin policy" permits scripts running on a webpage to access the DOM

of the same domain only. However, with the Cross Origin Resource Sharing (CORS) capabilities introduced with HTML5, scripts are allowed to make cross-origin AJAX requests to other domains. The attackers can send multiple HTTP GET or POST requests to any website in a background loop of a Web Worker. Although the request needs the website to return a HTTP header with "Access-Control-Allow-Origin: *" for the request to fetch contents, the attackers are not really interested in reading the response. As long as the requests have been sent, the server needs to spend resources and create threads to handle incoming requests. For POST requests, the server will also need to wait for the post data, which can lead to the hung request handling threads in the web server.

One concern for Web Worker DDoS attacks is when sending the first request to the target URL. If the response does not contain the "Access-Control-Allow-Origin" header with a "*" value, the browser will refuse to send subsequent requests to the this URL. However attackers can bypass this restriction by making every request unique (e.g. adding a random query-string parameter).

Another concern is that CORS requests will leave an "Origin" header pointing to the website sending the CORS requests. The server being attacked can use a rule-based filters to block requests with specific "Origin" headers. However, in our experiments, we found only XMLHttpRequest will leave an "Origin" header while embedding a URL in an image source does not add an "Origin" header in the request.

4.3.1.2 Distributed Password Cracking

Password cracking is a computationally intensive task. Cracking involves applying a hash algorithm to a candidate plaintext password and comparing the result to the ciphertext. Cracking a password with moderate strength can take days or months for a single computer. For a relatively strong password, the cracking time can often be insurmountable. Distributed computing is therefore an attractive approach to satisfy the computing demands that attackers need for password cracking.

There are two basic cracking methods: brute-force attacks and dictionary attacks. For brute-force attacks, every possible combination of the keys are searched exhaustively. The search space increases exponentially with key length. Therefore, for a password with sufficient length, brute-force attacks are impractical. To distribute a brute-force cracking attack across a set of compromised hosts is quite simple. A range of the plaintext is sent to each client(e.g. 'aaaaa' to 'azzzz'). The client computes the hash of each plaintext and returns the ciphertext to the server to see if a match has been found. The dictionary attack does not search every combination. Instead, it searches from a given list which contains the most likely candidates. The search space of dictionary attack is greatly reduced compared to a brute force attack. To distribute the dictionary attack, a fraction of the list needs to be sent to the clients. For both attack modes, a coordinating server is needed to make sure clients are working on non-duplicated tasks.

There are some other cracking methods such as pattern matching and word list substitution. These approaches generally lie between the brute-force attack and dictionary attacks. They use some rules to improve efficiency and require less data transfer to the clients than a pure dictionary attack.

Three cracking modes are examined in our experiment: SHA1, SHA512 and PBKDF2-SHA512. PBKDF2 is a key derivation function that repeats certain cryptographic hash function many times on the input to make brute force cracking more difficult. We choose SHA512 and an iteration number of 1000 in our experiment.

For non-browser based cracking, professional tools are used. John the Ripper (JtR) [115] is one of the most famous password recovery programs and is used as a benchmark in various areas. JtR supports different attack modes, such as brute force attacks and dictionary attacks, and covers a variety of hash algorithms including MD5, SHA, etc. OclHashcat [116] is a GPU-accelerated password recovery tool. We use these two tools on Amazon EC2 to compare against the cracking speed of Web Workers in a browser.

For Web Worker password cracking, we ported the MD5, SHA1 and SHA512 algo-

rithms from their C/C++ versions to JavaScript with Emscripten and NativeClient. The logic for checking passwords is still implemented in JavaScript and the cracking logic is implemented in the Web Worker. For distributed password cracking, a coordinating server is needed to dispatch tasks to all the clients. This server accepts requests from clients and returns the brief task information such as the task ID and task input data. Obviously, the more concurrent clients there are, the higher the throughput the server needs to support.

4.3.1.3 Rainbow Table Generation

For non-browser based table generation, we used RainbowCrack [117] to generate the rainbow table. We use the 'rtgen' command provided by the program with 100,000 chains of length equal to 10,000. For the browser client, we experimented on hand written JavaScript code, as well as writing equivalent C++ code to the 'rtgen' function and porting it to JavaScript using Emscripten.

The coordinating command and control server has similar duties as in password cracking. This server accepts requests from clients and returns the brief task information such as the task ID and task input data, so that clients can then fetch the correct data in S3 separately. The introduction of a storage server, as shown in Figure 4.2, reduces the load on the task distribution server by serving data for processing directly out of the cloud-based storage system. The task distribution server is not involved in serving the large volumes of data that need to be processed or storing the results.

4.3.1.4 Cryptocurrency Mining

Cryptocurrencies are digital currencies which use cryptography to secure the transactions and to control the creation of new units. Some of the well-known cryptocurrencies include Bitcoin, Litecoin, etc. Usually cryptocurrencies are mined by people who have installed and run clients such as cpuminer and cgminer on their computers in their spare time. However, the attackers could migrate the mining algorithms to JavaScript and embed them

to regular webpages as Web Worker tasks. When visitors view the compromised websites, they became slaves mining for the attackers unwittingly and making profits for the attackers. In this section, we analyze how much profit Web Worker mining could produce for an attacker.

Crytocurrency mining represents a class of 'attack' where attackers do not really harm anyone but immorally steal the computing resources of users for profit. The task does not necessarily need to be Bitcoin mining but can be data analytics or information processing that were supposed to be conducted on expensive servers. The ideal applications for attackers are those with the property of being 'embarrassingly parallel', where the tasks between clients have zero dependency.

4.3.2 Answering Research Question 2: Comparison of Browser-based Attack and Non-browser-based Attack

4.3.2.1 Comparison of Web Worker and Traditional DDoS Attacks

One characteristic of Web Worker DDoS attacks is that the attacks happen in the application layer. Traditional DDoS attacks come in a number of forms. Some attacks happen in the network layer, such as UDP floods or SYN floods [118]. Some attacks utilize amplifications, such as DNS amplification attacks [119]. However, JavaScript cannot access the network layer, therefore only application layer attacks are possible to launch from via Web Workers.

The size of a Web Worker DDoS attack depends on the traffic of the compromised websites. Popular websites commonly have 10K-100K concurrent visitors with a substantial number of unique IP addresses. In comparison, a botnet typically has a few thousand bots and unique IP addresses.

Given the severe impact of DDoS attacks, many vendors provide DDoS protection and mitigation solutions to website operators. The solution usually consists of a set of techniques including blacklists, IP filtering, etc. The challenge is to differentiate legitimate hu-

Table 4.1: Comparison of Web Worker DDoS attack and botnet-based DDoS attack.

	Web Worker DDoS	Botnet-based DDoS
Attack types	Application layer	Various types: SYN, UDP, DNS amplification
Scale	Depends on the traffic of websites	Rentable, usually consists of thousands of bots
Cost	Almost no cost	Rented for around $25 per 1000 bots per hour
Client quality	High, large IP pool	Low, majority from underdeveloped countries
Detectability	Difficult to detect	C&C server can be traced
Mitigation	Difficult to filter	Can be filtered with rules or IP signatures

man traffic from botnet traffic. The detection and mitigation of Web Worker DDoS attacks will be much more difficult than Botnet-based DDoS attacks because network flood attacks, such as UDP floods and SYN floods, can be resolved at the hardware level but application layer attacks of Web Worker come from real users with real IP addresses. It is even possible to simulate real application transactions through a series of HTTP requests. The attacker looks just like a legitimate visitor. Other features also make Web Worker DDoS attacks potentially harder to detect. 1) Unlike a botnet with a fixed pool of IP addresses that can be blacklisted, the client composition of Web Worker DDoS are continually changing as visitors come and leave from a compromised website. The transience also makes forensics harder because a user becomes a bot only when they are visiting a compromised website. 2) The clients won't be permanently compromised and will only transiently be involved. The detection would require detecting a relationship between browsing behavior and obvious outbound attacks from a client while visiting a particular website.

A high-level comparison of the Web Worker and traditional attack variant is shown in Table 4.1.

4.3.3 Answering Research Question 3: Cost Model for Analyzing the Cost Effectiveness

In this section, we describe the cost model for browser resource stealing attack with Web Workers, rented botnets, and cloud computing. There are two types of costs associ-

ated with each attack: the acquisition cost, which is the cost to set up the attack infrastructure, and the maintenance cost, which is the cost for performing the attack and keeping the resource stealing infrastructure operational. A comparison of the cost breakdown for different attack vectors is shown in Table 4.2.

For attacks leveraging cloud computing infrastructure, there is no acquisition cost as you can purchase as many computing resources as possible from cloud service providers such as AWS or Azure. The maintenance cost for cloud computing would be the price the service providers charge you for actual usage.

For attacks with botnets, there are 2 ways to acquire a botnet. One can either grow their own botnet by exploiting vulnerabilities and propagating malware or they can rent or buy a botnet in an underground market. It is hard to quantify the cost to grow a botnet as it depends on the expertise of the attacker and the vulnerabilities being used. But there is quantifiable pricing for renting a botnet that we can compare with. For maintaining a botnet for an attack, the attacker has to rent a VPN server or other infrastructure to send command and control messages to the bots.

Table 4.2: Cost Breakdown

Attack vector	Acquisition Cost	Maintenance Cost
Web Worker	Create own websites or hack existing websites	Online ads costs (PPC or visiting time), VPN server for C&C, bandwidth costs
Botnet	Grow own botnet by exploits or rent botnet from underground market	VPN server for C&C, bandwidth costs
Cloud computing	No cost	Paid to service providers for usage

A Cost Model for Web Worker Attacks. Since hacking a website is difficult and the cost is hard to estimate, we focus on the cost analysis of using online advertisements to launch Web Worker attacks. The acquisition cost $AC_{browser}$ in Web Worker attacks is the cost of setting up a website (purchasing a VPN server and domain name). The maintenance cost $MC_{browser}$ is the cost of coordinating an attack with Web Workers. The maintenance

cost consists of three primary parts: 1) the cost of the computing time to run a command and control server to coordinate an attack; 2) the cost of the data transfer to/from the command and control infrastructure; 3) the cost to attract visitors through online advertisement services.

Because Web Worker attacks require the coordination of a larger numbers of clients to be effective, they place much higher demands on the command and control infrastructure that must be considered from a cost perspective. Take distributed password cracking as an example, there has to be a coordinating server which is responsible for dividing the data into subtasks and distributing the data to browsers. The coordinating server needs to instruct browser clients to work on different tasks and ensure that clients report results before they are terminated (*e.g.,* the user leaves the website). The cost to maintain a server for attack coordination C_{server} equals the unit price of the server multiplied by the time it is in use, which we express as:

$$C_{server} = P_{server} * t \tag{4.1}$$

Bandwidth cost $C_{bandwidth}$ is the cost to distribute data from the attack coordination server. Take distributed rainbow table generation as an example, the coordinating server needs to assign the browser clients the range of plaintext hashes to compute, and the generated partial rainbow table from each client has to be sent back to server. The hashing algorithm, coded in JavaScript, also needs to be sent to the clients. These costs can be modeled as:

$$C_{bandwidth} = k * I * P_{toclient} + O * P_{toserver} + C_{code} \tag{4.2}$$

The actual data transfer to clients is almost always larger than the size of the inputs to the clients I because some clients will inevitably leave before their assigned computation is complete. Therefore the data transferred to these clients is wasted. We use μ to represent the successful task completion ratio. The actual volume of data transferred out will be k times the data needed for one copy of the task, where k is equal to $1/\mu$:

C_{code} is the bandwidth cost to distribute the processing code, such as hashing algorithms, to the browser clients. n is the total number of user visits to the website. During each visit, the client may request and complete multiple tasks but only one piece of processing code needs to be transferred. I_{code} is the size of the processing code in JavaScript. $P_{toclient}$ is the unit price of bandwidth cost to transfer data from server to client.

$$C_{code} = n * I_{code} * P_{toclient} \tag{4.3}$$

C_{ads} is the cost to spawn Web Worker tasks through online ad impressions. This cost depends on the online ad providers chosen. C_{ads} can be computed as $C_{ads} = n * C_{CPC}$ where C_{CPC} is the average cost per click. Or $C_{ads} = T_{visit} * C_{unit}$ for a website like Hitleap where T_{visit} is the total visiting time and C_{unit} is the unit cost of buying the viewing time.

In summary, the cost to coordinate an attack with Web Workers is given by:

$$C_{browser} = C_{server} + C_{bandwidth} + C_{ads} \tag{4.4}$$

A Cost Model for Cloud-based Attacks. To understand the desirability of Web Worker based attacks, it is important to compare their costs against other attack vectors. Another resource attackers can utilize to launch an attack is cloud computing services, such as Amazon EC2. Attackers can rent massive EC2 virtual machines for password cracking. Cloud computing provides elastic computing resources at competitive prices, as low as few cents per hour. There are tools such as StarCluster [120] which automate and simplify the process of building and managing clusters.

There is no acquisition cost for a cloud-based attack. The cost to run the tasks in the cloud is given by MC_{cloud}:

$$MC_{cloud} = T_{cloud} * I * P_{cloud} \tag{4.5}$$

where P_{cloud} is the cost per hour of a virtual compute unit. T_{cloud} is the computing time to process 1 unit of data with one virtual compute unit in the cloud.

A Cost Model for Botnet-based Attacks. The maintenance cost of botnet-based attack MC_{botnet} is similar to Web Worker attacks. However, the bots will not remain active forever. Attackers will lose control over some bots as time goes by due to the awareness of users or anti-virus scanning. We define s as the average percentage of bots lost each day. To maintain the botnet of same size, attacks will have to pay an additional cost $C_{loss} = s * P_{botnet} * t$.

$$MC_{botnet} = P_{server} * t + C_{bandwidth} + C_{loss} \tag{4.6}$$

The acquisition cost of a botnet-based attack AC_{botnet} is the cost to purchase the botnet. $AC_{botnet} = P_{botnet} * n$. P_{botnet} is the unit price to rent or buy bot. n is the number of bots needed to launch an equivalent attack to Web Worker attacks.

4.3.4 Answering Research Question 4: Attack Limitations and Countermeasures

Although we have demonstrated the feasibility of Web Worker-based resource-stealing attacks, these attacks do have some limitations compared to traditional botnets: 1) *The lack of full control over the host* - Traditional botnets infect the computers with trojans or malware. Once infected, the attackers have nearly full control over the infected systems. In Web Worker-based resource-stealing attacks, JavaScript codes execute in a sandbox. The JavaScript cannot normally access the system kernel, local files on disks, etc. Therefore many system-level attacks are not possible. However, this limitation can also be an advantage because traditional anti-virus software cannot easily detect the existence of browser-based resource-stealing tasks. 2) *Transience* - traditional bots can maintain control over a resource as long as victim users do not find themselves infected. Attackers can launch attacks as long as the bots are powered on. For browser-based resource stealing, however, a user's browser participates in the botnet only when a compromised web page is open. If the

user closes the tab, the botnet control terminates immediately. Since the page viewing time of users is often short and unpredictable, browser-based botnets are also transient and unstable. There is no doubt that traditional botnets are still favored as they can take a persistent foothold on a victim's machines with fewer restrictions. However, browser-based botnets have their own advantages. With continued improvements in JavaScript performance and increases in the number of web-connected devices, Web Worker-based attacks might become a significant attack vector. The threat could be potentially great if ad networks are exploited heavily to launch these attacks.

To prevent attackers from misusing online advertisement and launching Web Worker attacks, ad network providers could enforce stricter review processes on the HTML5 ads. However, since Web Worker attacks do not use any dangerous JavaScript functions and do not harm the end-user directly, it can be difficult to distinguish them from benign code. Attackers can also use code obfuscation techniques to make detection even harder. In addition, manual review or machine review can pose a large burden for advertisement networks given the large amount of ads being submitted. Another strategy to prevent online ads becoming a hotbed for Web Worker attacks is adjusting the pricing model. As our cost analysis in Section 4.3.3 has demonstrated, online advertisements are only economically attractive for attackers when they can achieve very low click-through rates and low costs per click. The ad network providers can penalize ads with low click-through rates to help disincentive this type of attack.

One possible countermeasure to Web Worker attacks is to enforce Content Security Policy (CSP). Content Security Policy is a new standard to prevent XSS attacks by defining a field in the HTTP header so that a server can specify the domains that browsers are allowed to make requests to. If the server restricts the white-list domains to only itself, then a Web Worker will not be able to make DDoS requests or contact with a coordinating server for password cracking. However, very few websites currently implement this feature because websites are rely heavily on external resources such as images, Ads, and other

services. Maintaining a white list of legitimate domains requires tremendous effort.

For Web Worker DDoS attacks, one way to mitigate them would be to enforce stricter limits on browsers in terms of the number of concurrent requests, number of references to non-local objects, and sustained usage of concurrent requests over time. For example, it may make sense to send a burst of 100-200 requests to several different hosts when a web page first loads, but a sustained request rate of 10,000 requests per minute is not indicative of legitimate behavior. However, these restrictions need to be carefully balanced with the needs of legitimate websites where it is common for webpages to grab resources from different domains for displaying advertisements, accessing third-party APIs, etc.

Countermeasures to application-layer DDoS attack are more difficult than network layer attacks due to the challenge of easily differentiating malicious and non-malicious traffic. One differentiation approach is to try and distinguish between real human traffic and bot traffic by comparing traffic signatures such as request rate, IP, HTTP headers, JavaScript footprint, etc. What makes things more complicated is that there are "benign bots", such as search engine crawlers, monitoring tools, and other automated request generation infrastructure. Traffic classification may misclassify these bots and affect website search engine optimization or monitoring.

For computationally intensive tasks, such as password cracking or rainbow table generation, countermeasures will be tricky to develop. We cannot simply block JavaScript or Web Workers because they are widely used in regular webpages. One possible solution is to monitor the system resource consumption and warn users of abnormal behavior. For example, browsers could have a plug-in which displays the CPU utilization rate or network traffic for each webpage/tab and raise an alarm when viewing a page that has abnormally high resource consumption.

4.4 Empirical Evaluation of Browser Resource Stealing Attack

In order to evaluate the attack economy quantitatively and to provide direct comparison between different attack approaches, we set up experimentation environments consisting of browser clients and cloud instances to facilitate the computation and measurement needed for the cost models described in Section 4.3.3.

4.4.1 Web Worker DDoS Attacks

Distributed Denial-of-service (DDoS) attacks have been known for many years. DDoS attacks are characterized by an explicit attempt to make an online service unavailable by overwhelming it with traffic from multiple compromised systems [113]. Attackers usually inject slave or "bot" computers with remote command and control services. The compromised computers are known as bots since they are under the control of the attackers and are used to send requests simultaneously to the target services. DDoS attacks have been the most prominent threat for website owners in recent years. Studies show the average loss for companies incurred from DDoS attacks is estimated to be $1 million per day [114]. Web Workers could be a serious new threat for DDoS attacks, since they do not rely on compromising a host, but instead getting it to visit a specific web page.

To understand whether DDoS attacks with Web Workers are economically attractive to attackers, we compare the cost of a Web Worker attack with a rented botnet DDoS attacks. A recent trend among cybercriminals is the popularization of "attack as a service". People with little expertise hacking can rent existing botnets from hackers in the black market. The price to rent botnets depends on factors such as the scale and quality of the botnet. For 1 hour of time with 1,000 bots, the price is $25 on average and it is usually difficult and expensive to rent a botnet above 100,000 bots.

Google AdWords supports various Ad forms including text, image and even HTML5 formats. We uploaded a legitimate HTML design file of an Ad. Inside the HTML file, we

included a request to a target server that we set up to assess the feasibility of launching Web Workers from Google AdWords. The difference with password cracking attacks is the users does not need to click the Ad. The requests will be sent any time the ad is shown, regardless of whether or not the user clicks on it. Because AdWords operates on a pay-per-click model, we only need to pay for the clicks but not the views. If the click-through rate (the number of clicks the ad receives divided by the number of times the ad is shown) is low enough, the attacker is able to get a large amount of pageviews while paying very little money. Note that this is exactly the opposite strategy of what a normal businesses would do, as they usually try to increase their click-through rate to attract more customers.

We promoted our website through Google AdWords and Hitleap. In our initial experiment with AdWords, we received 12000 impressions (views) and 29 clicks in 24 hours. The average cost per click (CPC) was \$0.1. The clickthrough rate (CTR) in our case was 0.24%. The average time people saw our Ads was 32s. To lower the cost, we needed a lower CPC and CTR. Google Adwords follows a bid-based CPC model: advertisers compete against other advertisers for the same search keywords. Advertisers with high bid prices will show up first to the visitors. By choosing less competitive keyword terms, we can reduce the CPC. For CTR, a high CTR can usually be achieved with attractive visual effects and high relevance to the keywords in the ads. Therefore, we designed boring ads and selected keywords irrelevant to the ad content.

By carefully choosing the search keywords of the ads, we were able to reduce the CTR to 0.15% with an average CPC \$0.08. Assume a browser is able to send 10000 HTTP requests a minute. If we had more budget and add more keywords for our ads, we could scale up our impressions. Suppose we were able to get 10^6 impressions a day. The average HTTP request rate will be $10^6 * 32/86400 * 10000 = 3.7 * 10^6/min$. The daily cost is $0.08 * 10^6 * 0.0015 = \$120$. For Hitleap, the price for buying traffic is \$0.0375/hour [121]. The daily cost to generate $3.7 * 10^6$ HTTP request per minutes is $0.0375 * 370 * 24 = \$333$. Therefore, AdWords is more favorable than Hitleap for a DDoS attack.

For a botnet-based DDoS attack, the number of HTTP requests a bot can generate depends on a variety of factors such as memory size, network bandwidth, operating system, etc. In our experiment with a HTTP DDoS tool, a bot can send an average of 20,000 requests in a minute. Assume the rented botnet has an online ratio of 50 percent, to generate the same number of requests with botnet, 370 bots are needed. In reality, the purchased botnet will gradually decrease in size due to the awareness of users or anti-virus scanning. An additional cost $C_{loss} = s * P_{botnet} * t$ should be considered. The average percentage of bots lost each day is estimated to be around 6.3% [122]. The cost for botnet-based DDoS C_{botnet} is therefore about $370/1000 * 25 * 24 * (1 + 0.063) = \$236$. The results show Web Worker DDoS attacks have comparable cost to a rented botnet-based DDoS attack. The key factor determining the cost for Web Worker attacks is the click-through rate of the Ads. There is the potential for attackers to adjust their ad strategy to reduce the click-through rate to further decrease the cost of a Web Worker DDoS attack.

To estimate the scale and cost of launching a Web Worker-based attack by compromising existing websites, we use the Forbes.com. If an attacker is able to compromise this website, he or she will have access to 4.4 million unique IP visits per day with a 3 minute average visit time. In our experiments, browsers are able to send around 10,000 HTTP requests a minute. Assume the visits are uniformly distributed, the average DDoS attack traffic would be $4.4 * 10^6 * 3/60/24 * 10^4 = 91$ million requests a minute. In reality, the website traffic is not uniform and thus the peak attack traffic would be even larger.

To generate the same number of requests as the Web Worker attack through Forbes, $91 * 10^6/30000 = 3056$ active bots would needed. Assume the rented botnet has an online ratio of 50 percent. 6102 bots need to be rented and the cost is about $6102/1000 * 25 = \$153$. A cost comparison of Web Worker DDoS and botnet-based DDoS to achieve same attack traffic is shown in Table 4.3.

Table 4.3: Comparison of Web Worker DDoS attack and Botnet-based DDoS attack.

	Web Worker DDoS attack	Rented Botnet DDoS attack
Single client attack traffic	10000 requests/minute	20000 requests/minute
Total attack traffic	91 millions requests/minute	91 millions requests/minute
IP pool size	4.4 million	6102
Total Cost	$0/day	$153/day
Cost per requests per minute	$0	$\$1.68 * 10^{-6}$

Table 4.4: Password cracking speed (hash/sec) for different hash algorithms on different platforms.

Algorithm	JtR (m3.medium)	oclHashcat	Native JS	Emscripten JS	NativeClient
MD5	12M/s	2287M/s	0.35M/s	3.3M/s	6.8M/s
SHA1	6.1M/s	485M/s	0.16M/s	0.83M/s	3.6M/s
SHA512	0.83M/s	64M/s	0.01M/s	0.14M/s	0.51M/s
PBKDF2-SHA512	203/s	4512/s	112/s	123/s	152/s

4.4.2 Distributed Password Cracking

To understand whether password cracking with Web Workers is economically attractive to attackers, we compare the cost of password cracking with Web Workers to cloud computing because cloud computing is considered cost-efficient approach to large-scale cracking tasks [123].

Table 4.6 shows the performance comparison of different browsers. Scripts ported with NativeClient are the fastest but only work under Chrome. For scripts ported with Em-

Table 4.5: Cost comparison for different hash algorithms on different platforms.

Algorithm	Web Worker	Cloud (GPU)
MD5	$67/day	$1.31/day
SHA1	$67/day	$8.42/day
SHA512	$67/day	$21.57/day
PBKDF2-SHA512	$67/day	$38.17/day

scripten, popular browsers such as Chrome, Firefox and Safari show similar performance while IE is more than 50% slower. Although mobile devices such as smartphones or tablets contribute increasing traffic to websites, online advertisement providers provide options to restrict page views to desktops only. So we focus on desktop browsers only.

To make a more convincing estimation, we used browser market share statistics obtained from StatCounter [124] and compute the weighted speed taking into account the performance variation of different browsers. As of October 2017, Chrome(63.6%), Firefox(13.1%), IE(8.3%), Safari(5.9%) are the most used desktop browsers. The weighted MD5 cracking speed is about 4.4M hash/s.

To evaluate attacks with Web Worker, we set up a website using the WordPress software package and embeded a cracking script in a Web Worker loaded in the main page of the website. We also set up a coordinating server in node.js to record performance information. We promoted our website through both Google AdWords and Hitleap.

For Hitleap, 1 million minutes of viewing time can be bought for $625 [121]. We registered our website on Hitleap and used the bought traffic to get other users to view our website. These viewing times will be consumed by other users with a 20s viewing slot. We assume Hitleap can help us attract 100,000 minutes viewing time a day. The daily cost is $62.5.

For AdWords, we created a text ad including a URL of our website. The ads received 12000 impressions (views) and 29 clicks in 24 hours. The cost per click is around $0.1. For those clicks, the average time people stay on our website is 81s. To attract equivalent

Table 4.6: Cracking speed for different browsers.

Browser	MD5	SHA1
Chrome (NativeClient)	6.0M/s	3.6M/s
Chrome (Emscripten)	2.78M/s	0.83M/s
Firefox (Emscripten)	3.1M/s	0.85M/s
IE11.0 (Emscripten)	1.51M/s	0.09Ms

100,000 minutes viewing time as Hitleap, the daily cost will be $7400. Obviously AdWords is not suitable for deploying this attack. To make AdWords economically attractive, we need to create very addictive websites that can hold visitors to stay more than 2 hours for each visit, which is extremely difficult.

To distribute brute force password cracking with Web Workers, the cost is $C_{browser} = C_{server} + C_{bandwidth} + C_{ads}$. For C_{server}, we choose an m3.large instance as coordinating server, $P_{server} = \$0.133/Hour$. For the brute-force attack, the only data that needs to be sent to clients is the start and end of the range of the plaintext, which is negligible. And the client only replies to the server with whether the password has been found. So the data volume transferred from clients to the server $I \approx 0$. For rainbow table generation, the generated table needs to be sent back to the server. However, AWS does not charge for inbound data transfers. So $P_{toserver} = 0$.

The hashing algorithm coded in JavaScript is 50KB. $I_{code} = 50\text{KB}$. The number of unique visits to the website is $n = 10^5 * 60/20 = 3 * 10^5$, which means the hashing algorithm needs to be transferred $3 * 10^5$ times from server to clients. $C_{bandwidth} = 3 * 10^5 * 50 * 10^{-6} GB * 0.09 = \$1.35$. The cost of running a Web Worker attack on this website per day is: $C_{browser} = 0.133 * 24h + 1.35 + 24 = \$28.5$.

To calculate the attack cost using rented cloud computing resources, we use Amazon's Elastic MapReduce Service (EMR) to construct a cluster with GPU instances because GPUs have a better performance/cost ratio than CPU instances as shown in Table 4.4. The unit price for a GPU instance (g2.2xlarge) is $P_{unit} = 0.65 + 0.2 = \$0.85$. As we have discussed in Formula , not all the website visitors' browsing time can be utilized. The effective computing time should consider successful task completion ratio μ. Suppose $\mu = 0.8$, the effective computing time of the compromised website is $10^5/60 * 0.8$ hours. Using the cloud to compute the equivalent number of SHA1 hashes, the rented virtual machine time would be $1333 * 3.6/485 = 9.9h$. The cost of cracking SHA1 with cloud computing is $C_{cloud} = 0.85 * 9.9 = \$8.42$. C_{cloud} is $38.17 for PBKDF2-SHA512 and $975 for rainbow

table generation. Table 4.8 shows the cost comparison of Web Worker cracking and cloud computing cracking for different hash algorithms.

For a rented botnet, the common price of purchasing 1,000 bots from underground market is around 100 dollars [125]. The average computing power of bots varies between different botnets. Most of the bots do not have a high-end GPU. For this comparison, we roughly estimate the average bot's computing capability to be similiar to an m3.medium instance. Assume the bot has an online ratio of 50 percent. For MD5, $88000 * 6.8/12/24/0.5 = 4156$ bots are needed. For SHA1, $88000 * 3.6/6.1/24/0.5 = 4328$ bots are needed. For SHA512, $88000 * 0.513/0.828/24/0.5 = 4543$ bots are needed. The one time cost to purchase the botnet is shown in Table 4.8.

In reality, the purchased botnet will gradually decrease in size due to the awareness of users, anti-virus scanning, etc. So the actual cost of the botnet would be higher.

For SHA1, the cost is $C_{cloud} = 0.85 * 9.9 = \$8.42$. For PBKDF2-SHA512, the cost is $C_{cloud} = 0.85 * 1333 * 152/4512 = \$38.17$. For RainbowTable generation, the cost is $C_{cloud} = 0.85 * 1333 * 1921/2232 = \$975$. As shown in Table 4.8, Web Worker attacks cost more money than cloud computing for SHA1

Some insights we can obtain from the calculations are: 1) the cost savings of the Web Worker variant of the attack are larger for the more complex hashing algorithms. This is understandable because the cost to distribute the computation is fixed. Complex hashing algorithms demand more computational resources and thus create more value to the attacker. 2). The cost savings of the Web Worker approach versus a rented botnet is huge, but GPU instances in the cloud greatly narrow the gap. The inability to use GPU resources limits the attractiveness of the Web Worker attack variant. As GPU APIs for JavaScript become more prevalent, Web Worker attacks will become more attractive. 3). The cost savings are directly affected by the traffic features of the website where the attack was deployed. Longer viewing times of website visitors produce a more stable computing pool and yields large cost savings for the attacker.

Figure shows the relationship between average viewing time of a website and cost savings for a Web Worker password cracking attack. We use the costs of MD5 for demonstration, for a given amount of computation, the cloud computing cost is fixed. The Web Worker computing cost drops as average page view time increases. We can see the Web Worker approach is actually not cost-effective for the attacker when the average dwell time on the website is below 20 seconds. This means websites with short visits are not ideal target for distributed password cracking attacks with Web Workers. These websites still generate values for their computation but the opportunity cost is too large for the attacker. Web Worker attacks are more suitable for launching a DDoS attack which does not need a long-lived connection. However, there are websites where users do linger longer, such as sites with videos that average several minutes in length, online gaming, online education, etc. Web Worker attacks, if they become prevalent, will be biased against these types of websites.

4.4.3 Rainbow Table Generation

Rainbow tables [126] are precomputed tables for reversing cryptographic hash functions. They are widely used for cracking password hashes. The idea is to use a time-space tradeoff and compute the hashes of all the possible plaintext within a length range and store the rainbow table chain. Reduction functions are defined to generate a hash chain so that it is unnecessary to store every plaintext's hash. Once this is done, to crack a given ciphertext, the attacker searches the precomputed table and compares the ciphertext with each entry.

Although searching the rainbow table is much faster than brute-force cracking, the generation of a rainbow table requires huge amounts of computation and may take days or months to finish. Each rainbow table only works for one type of hash method, in other words, you need to generate different tables for MD5, SHA1, SHA256 and their variants. Therefore, distributing the rainbow table generation tasks can be an attractive way to deal with the computational demand. DistrRTgen [127] is a distributed rainbow table generation

project deployed in the volunteer computing platform BOINC. To distribute the tasks, the central server only needs to send a start string and end string of a range of the plaintext to the clients and the clients compute the chain for each plaintext and return the first and last element in the chain.

The key question with distributing rainbow table generation tasks to Web Workers is whether it is cost-effective. This is because although the computing time of a browser client is free, distributing the tasks and receiving the computed results incurs costs for command and control. In our proposed architecture in Figure 4.2, a coordinating server is used to assign tasks to clients and a storage server is used to store the chunks of rainbow table each client computes. For scalability, an economical choice would be using cloud services, such as Amazon EC2 and S3, for command and control. One thing favorable to this approach is the pricing asymmetry of the cloud computing services. Cloud service providers, such as Amazon S3 and Microsoft Azure, only charge for data transfer out of their storage services and not data transfer into their storage services, as shown in Table 3.14.

Answering Research Question 3: Web Worker Rainbow Table Generation Economic Analysis To understand whether rainbow table generation with Web Worker is economically attractive to attackers, we compare the cost to launch the attack with Web Worker and cloud computing because cloud computing is considered cost-efficient and applicable to large-scale computing tasks.

Table 4.7 shows the rainbow table generation speed (chain/sec) on different platforms. To distribute rainbow table generation tasks to browsers, the cost $C_{browser} = C_{server} + C_{bandwidth}$. Using our previous example in Section 4.4.1, suppose the attacker compromised Forbes.com, which has traffic of 4.4 million visits per day. Each visit lasts 3 minutes on average. The attacker would be able to use $4.4 * 10^6 * 3/60$ hour of browser computing time. For C_{server}, we choose an m3.xlarge instance as coordinating server, $P_{server} = \$0.28/Hour$. For the brute-force attack, the only data need to be sent to clients is the start and end of the range of the plaintext, which is negligible. The clients do

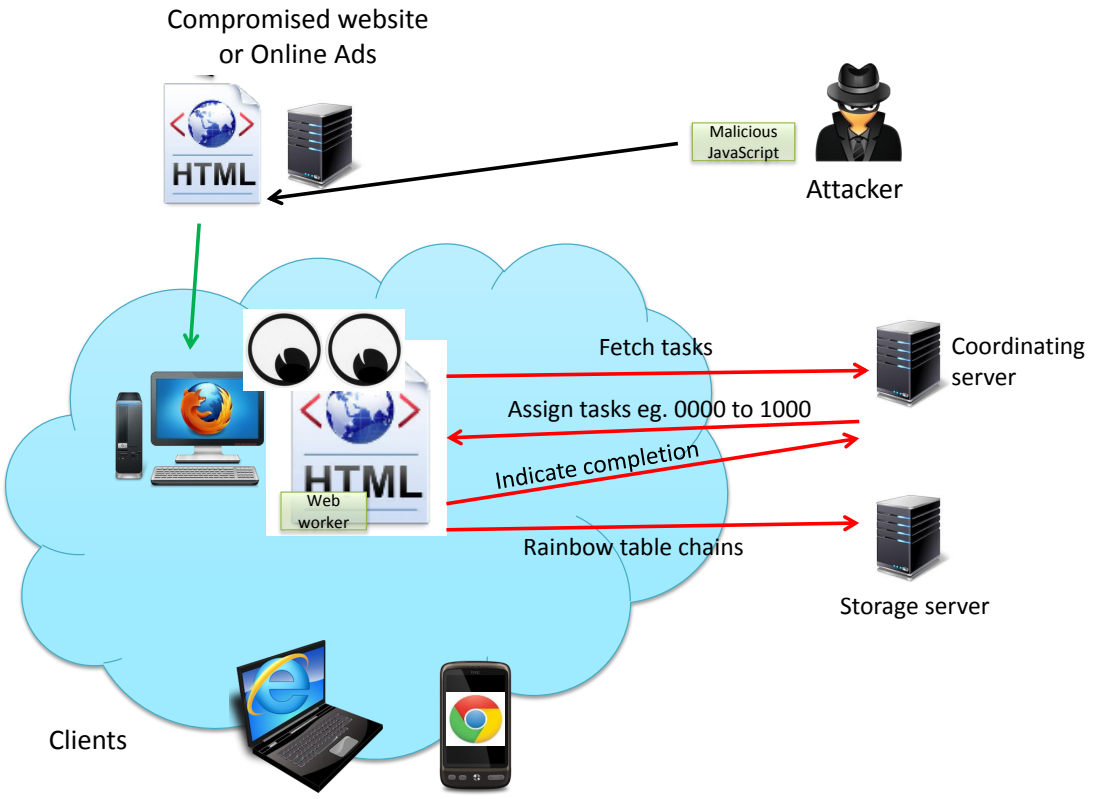

Figure 4.2: Architecture of distributed rainbow table generation.

Table 4.7: Rainbow table generation speed (chain/sec) on different platforms.

Task	m3.medium	Native JS	Ported JS
Rainbow Table	2232c/s	451c/s	1503c/s

need to return rainbow table which takes few MB but data transfer from client to server is free. So we only need to compute C_{code}. The hashing algorithm coded in JavaScript is 60KB. $I_{code} = 60\text{KB}$. Suppose $\mu = 0.8$, $d = 2$. The number of unique visits to the website is $n = 4.4 * 10^6$. The cost of Web Worker attack on this website one day is: $C_{browser} = 0.28 * 24h + 4.4 * 10^6 * 60 * 10^{-6} GB * 0.09 = \$30.48$.

To launch the attack with cloud computing, we use Amazon's Elastic MapReduce Service (EMR) to construct a cluster with CPU instances. The unit price for m3.medium is $P_{unit} = 0.087 + 0.022 = \$0.11$. To run the same load of computation using cloud ser-

Table 4.8: Cost comparison of Web Worker and cloud computing for rainbow table generation.

Task	Web Worker	Cloud
Rainbow Table	\$67/day	\$975/day

vice, the time needed would be $4.4 * 10^6 * 3/60/2 * 0.8 * 1503/2232 = 59358h$. So about $59258/24 = 2469$ CPU instance need to work simultaneously to achieve equivalent computing capability. The cost $C_{cloud} = 0.11 * 59358 = \$6529$. As shown in Table 4.8, $C_{browser} < C_{cloud}$. The attackers will find distributing the task with Web Workers more worthwhile than using cloud computing.

4.4.4 Cryptocurrency Mining

We experimented with two cryptocurrencies: Bitcoin and Litecoin. They have the top 2 market capitalizations and represent two families of cryptography algorithms. The mining of Bitcoin relies on repeated SHA256 hashing while Litecoin uses the scrypt algorithm. For Web Worker mining with Bitcoin, we used BitcoinPlus, which is a web service that provides APIs for people to embed a JavaScript miner on their own website. The JavaScript miner can connect to any mining pool via the HTTP stratum proxy using the account information attackers registered in the pool. In this way, the payout will be credited directly to the attackers. For Litecoin, the same framework was used. We ported the core algorithm 'scrypt' from C code to JavaScript with Emscripten and constructed our own JavaScript Litecoin miner.

As of November 15, 2016, the difficulty factor of mining for Bitcoin is 254,620,187,304 and the price in the market is \$712 per coin. In our experiment, our JavaScript miner reaches a maximum speed of 0.5MHash/s, the time needed to generate a block (25 bitcoins) is $difficulty * 2^{32}/hashrate = 2.187 * 10^{15}$ seconds [128]. Suppose an attacker compromised a website of a million visits per day with an average visiting time of 20 minutes.

Table 4.9: Bitcoin mining speed of different hardwares. CPU, GPU and ASIC data is from [2].

Hardware	Bitcoin mining speed
AntMiner S4 (ASIC)	2Thash/s
ATI Radeon 6970 (GPU)	365Mhash/s
Intel i7 2600k (CPU)	18.6Mhash/s
JavaScript	0.5Mhash/s

$10^6 * 20 * 60$ seconds of computing power can be used each day. The attackers can earn only $10^6 * 1200/(2.187 * 10^{15}) * 25 * 712 = \$0.01$ a day on average.

For Litecoin, our JavaScript Litecoin miner on Chrome can run with a speed of 0.25KHash/s. Using the payout data from f2pool, one of the largest mining pools in the world, 0.00832777 LTC coin can be made for 1MHash/s per day. The price of Litecoin as of November 15,2016 is \$3.88. The attackers can earn $0.00832777 * 0.25 * 10^3 * 10^6 * 20 * 60/87600/10^6 * 3.88 = \$0.11$ a day. For both Bitcoin and Litecoin, the reward is extremely low even for a high traffic website.

In summary, it is not worthwhile to do Bitcoin or Litecoin mining attack with Web Workers. Partially because the increasing difficulty of mining and partially because the huge performance gap between the specialized mining hardware used by some mining operations and the commodity computers as shown in Table 4.9.

Chapter 5

Featureless Web Attack Detection with Deep Network

5.1 Overview

In this chapter, we focus on an application of Gray Computing in cyber security. Deep learning has achieved great success recent year in computer vision [12], speech recognition [13], natural language processing [14], etc. Many deep learning libraries and framework in JavaScript emerge such as Keras.js [16], WebDNN [17], deeplearn.js [18]. These libraries implemented low-level linear algebra and matrix calculation which are essential for training and applying deep learning models. These libraries enable us to build deep learning application through web browser.

Web servers and web applications have been attractive targets for attackers. SQL injection [129], cross site scripting (XSS) [130] and remote code execution are common web attacks against web applications. These attacks can break down web services, steal sensitive user information and cause significant loss to both service providers and users.

Protecting web applications from web attacks proves to be a challenging task. Even though developers and researchers have taken various measures such as firewalls, intrusion detection systems (IDSs) [131] and defensive programming best practice [132] trying to make web applications safe, web attacks still remain to be a major threat today. More than half of the web applications during a 2015-2016 scan contained high security vulnerabilities such as XSS or SQL Injection [133]. And the hacking attacks cost the average American firm $15.4 million per year [134]. The Equifax Data Breach in 2017 [135] was a serious cyber attack which results in the expose of 143 million American consumers' sensitive personal information. The attacker utilized a vulnerability in Apache Struct. Although the vulnerability was disclosed and patched in March 2017, Equifax did not take any action until 4 months later. The estimated insured loss from the breach is up to 125 million dollars.

There are several reasons why existing intrusion detection systems do not work well as expected: 1. In-depth domain-knowledge in web security and skills are needed for web developers and network operators to deploy such systems [136]. Usually an experienced security expert is needed to determine what feature is relevant to extract from network packages, binaries or other input for intrusion detection systems. However, due to the large demand and relatively low entrance of web development, many developers still lack the necessary knowledge of secure coding practices. 2. Many IDSs rely on rule-based strategy or using supervised machine learning algorithms to learn what are normal requests and what are attack requests. A large amount of labeled training data is needed to train the supervised learning algorithms. However, labeled training data is hard and expensive to get. In addition, the labeled training data obtained is likely to be heavily imbalanced: attack requests are more difficult to get than normal requests, which poses challenges for classification problem [137]. Besides, new kinds of attacks and vulnerabilities emerge everyday. Rule-based or supervised learning approaches can distinguish existing known attacks but may easily miss the new attacks. 3. There are several existing research addressing web attack detection using unsupervised learning. However, they use manually created problem-specific features. The algorithms such as PCA and SVM achieve acceptable performance but the false positive are still too high for practical deployment. Even 10% of false positive can transform to thousands of legitimate users being affected by intrusion detection system.

Open Question $\Rightarrow$ Can we build scalable and resilient cyber infrastructure that can autonomically detect in-process cyber-attacks and adapt efficiently, scalably, and securely to thwart them? Given the challenges existing intrusion detection systems face, an infrastructure that requires less expertises and labeled training data is needed.

In this chapter, we evaluate the feasibility of an unsupervised/semi-supervised learning approach for web attack detection based on deep learning. Our work is motivated by the great success deep learning has achieved recent years in computer vision [12], speech

recognition [13], natural language processing [14], etc. Especially, deep learning has shown not only capable of classification but also automatically extracting features from high dimensional raw input. End-to-end deep learning [15] refers to the approaches where deep learning is responsible for the entire process from feature engineering to prediction. Raw input is fed into the network and high-level output is generated directly. For example, in speech recognition, traditional approaches try to solve the problem by layer-to-layer transformation from audio to phonemes, to words and finally to transcripts. While end-to-end deep learning systems erase the intermediate steps and use audio input to directly learn the output. In this research, we are trying to explore the potential of end-to-end deep learning in intrusion detection systems.

Our proposed approach uses Robust Software Modeling Tool (RSMT) [138] to extract web applications' runtime call traces. RSMT is a late-stage (post-compilation) instrumentation-based toolchain targeting languages that run on the Java Virtual Machine (JVM). RSMT permits arbitrarily fine-grained traces of program execution to be extracted from running software. During an unsupervised training epoch, traces generated by test suites are used to learn a model of correct program execution with stacked denoising autoencoder. Then a small amount of labeled data is used to calculate reconstruction error and set up a threshold to distinguish normal and abnormal behaviors. The two-phase approach utilizes the vast amount of unlabeled data as well as labeled data to boost the performance. During a subsequent validation epoch, traces extracted from a live application are classified using previously learned models to determine whether each trace is indicative of normal or abnormal behavior.

5.2 Overview of Research Questions

In this section, we describe key research questions for detecting web attacks with machine learning.

Research Question 1: How to monitor and characterize application runtime be-

havior? Static analysis approaches, which analyze applications' source code and search for potential flaws, suffer from the drawbacks thats Therefore, a runtime analysis approach is preferred. What is the best way to instrument the applications in order to monitor and character application runtime behavior? Also, instrumenting application will inevitably incurs a performance overhead. If the performance overhead is significant, it will affect the throughput of the web applications. How can we reduce the performance to an acceptable degree?

Research Question 2: How can we detect various kinds of attacks that have significantly different characteristics? There are different types of web attacks such as SQL injection, cross site scripting, remote code execution and file inclusion vulnerabilities. They use various forms of attack vector and exploit different vulnerabilities inside the web applications. Therefore, they may exhibit completely different characteristics. Many existing approaches for intrusion detection are designed to detect only one kind of attack. For example, A grammar based analysis that works on SQL injection detection will not work on XSS. Can we characterize the normal behaviors instead and detect different kinds of attacks all together?

Research Question 3: Can we develop intrusion detection systems without web security domain knowledge? Traditional ways of creating intrusion detection systems involve rule-based approach where domain-specific knowledge in web security is indispensable. Experienced security experts are needed to determine what feature is relevant to extract from network packages, binaries or other input for intrusion detection systems. The feature selection process is usually tedious and time-consuming. Even experienced engineers will often rely on repetitive trial-and-error processes. In addition, with the quick technology update cycle and new tools and packages available everyday, even the web security experts sometimes cannot keep up with the latest vulnerabilities. Therefore we are wonder is it possible to abandon the feature engineering and build intrusion detection systems with featureless approach.

Research Question 4: How to solve the labeled training data problem in supervised learning? Machine learning-based intrusion detection systems rely on labeled training data to learn what should be considered normal and abnormal. However, labeled training data can be difficult and expensive to collect in large scale for many real-world web applications. For example, normal request training data can be generated with load testing tools, web crawlers or unit tests. But if the application has vulnerabilities, then the generated data may also contain some abnormal requests and these slipped in abnormal data could undermine the performance of supervised learning approach.

Abnormal training data is even more difficult to obtain. It is hard to know what kind of vulnerabilities the system has and what attack the system will face. Even we manually create some attack requests against target application, it might not cover all the possible scenarios. Also different types of attacks have different characteristics, it is difficult for supervised learning methods to capture what attack requests should look like. Supervised learning approaches may distinguish existing known attacks well but can easily miss the new attacks. Besides, new forms of attacks and vulnerabilities emerge everyday. It is difficult to keep up with the daily disclose of web vulnerabilities, especially when current web applications frequently need to depend on many third-party packages.

Research Question 5: How can we store and communicate the detection results effectively? When Web attacks happen, the web service providers may not discover the attack immediately. Even if they do, they may prefer the ignore or delay the disclosure of the attack to minimize the impact. However, this will leave users' sensitive information at risk and prevent users from taking immediate action to reduce losses. In the Equifax data breach, Equifax revealed the attack until months after the attack happened. Is it possible that we can make the attack disclosure automatic and public so that people can be aware of what is happening with sites that hold their data?

5.3 Solution Approaches

5.3.1 Robust Software Modeling Tool

In this section, we present a high-level architecture of our proposed in-progress web attack detection system.

In order to monitor and characterize web application's runtime behavior, we developed an infrastructure named Robust Software Modeling Tools (RSMT). RSMT is a late-stage (post-compilation) instrumentation-based toolchain targeting languages that run on the Java Virtual Machine (JVM). RSMT permits arbitrarily fine-grained traces of program execution to be extracted from running software. RSMT constructs its models of behavior by first injecting lightweight shim instructions directly into the application binary. These shim instructions enable an RSMT runtime component to extract features representative of control and data flow from a program as it executes but do not otherwise affect application functionality.

As we have described in Section 5.2, different attacks have different characteristic and traditional feature engineering approach cannot have a unified solution for all the attacks. RSMT bypasses the various attack vectors, instead it capture the low-level call graph under the assumption that no matter what the attack types is, it will try to invoke some methods in the server side that it should not have access, or the access pattern is statistically different than the legitimate traffic.

Figure 5.4 shows a high-level workflow of our web attack detection system based on RSMT. The system is driven by one or more environmental stimuli (a). These stimuli are actions transcending the process boundary and can be broadly categorized as manual (human interaction-driven) or automated (test suites, fuzzers) inputs. The manifestation of one or more stimuli results in the execution of various application behaviors. RSMT attaches an agent and embeds lightweight shims into an application (b). These shims do not affect the functionality of the software, rather, they serve as probes that allow the inner work-

ings of the software to be examined. The events tracked by RSMT are typically control flow-oriented in nature, but dataflow-based analysis is also possible. As the stimuli drive the system, the RSMT agent intercepts event notifications issued by the shim instructions. These notifications are used to construct traces of behavior that are subsequently transmitted to a separate trace management process (c). There, traces are aggregated over a sliding window of time (d) and converted into bags of features (e). RSMT can enact online strategies using these feature bags (f). Online strategies involve two epochs: during a training epoch, these traces (generated by test suites) are used by RSMT to learn a model of correct program execution. During a subsequent validation epoch, traces extracted from a live application are classified using previously learned models to determine whether each trace is indicative of normal or abnormal behavior.

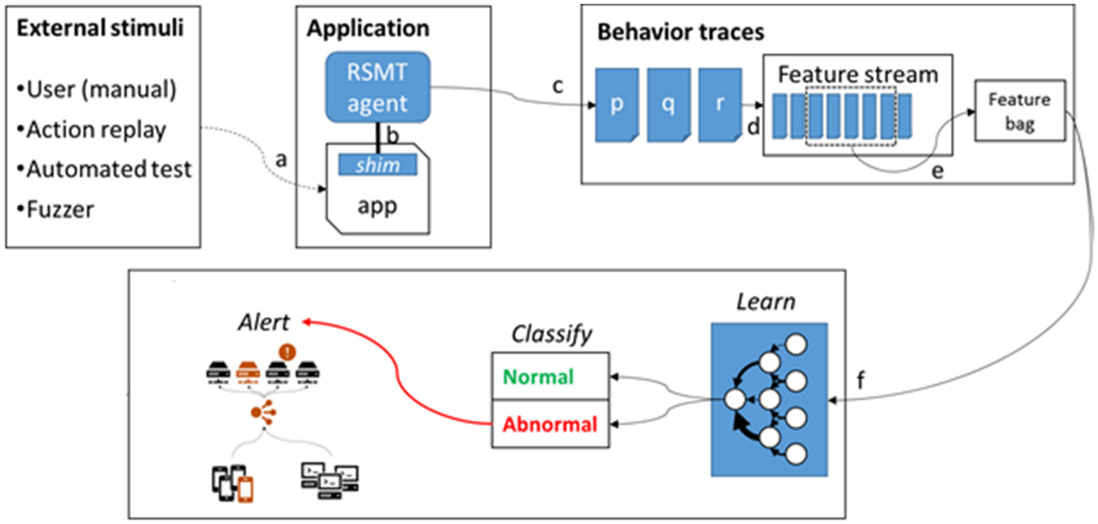

Figure 5.1: The architecture of the RSMT online monitoring and detection model

There are three core components of the RSMT architecture: 1) An application, to which the RSMT agent is attached. 2) An agent server, which is responsible for managing data gathered from various agents. 3) A machine learning backend, which is used for training various machine learning models and validating traces.

5.3.2 Unsupervised Web Attack Detection with Deep Learning

In this section, we present our unsupervised/semi-supervised web attack detection system based on RSMT and deep learning.

Traces Collection with unit tests: A trace is a sequence of directed f-calls-g edges observed beginning after the entry of some method.

From a starting entry method A, We record call traces up to depth d. We record the number of time each trace being triggered for a request. For example, A calls B one time and A calls B and B calls C one time will be represented as: A-B: 2; B-C: 1; A-B-C: 1. Each trace can be represented as a 1*N vector [2,1,1] where N is the number of different method calls. Our goal is given the trace signature $T_i = \{c_1, c_2, ..., c_n\}$ produced from request P_i, determine if it is an attack request.

There are generally two categories of approaches towards web attack detection:

Supervised learning: such as Nave bayes, SVM, etc. They train a classifier with training dataset consists of traffic labeled as normal traffic and attack traffic. The classifier than classify the incoming traffic as either normal or attack. The problem with supervised approach:

- Hard to get a large amount of labeled training dataset.
- Cannot handle new types of attacks which does not included in the training dataset.

Unsupervised learning: which does not require labeled training dataset. The assumption is: Data can be embedded into a lower dimensional subspace in which normal instances and anomalies appear significantly different. The idea is to use dimension reduction techniques such as PCA or autoencoder for anomaly detection. PCA or autoencoder try to learn a function $h(x) = x$. PCA learns a low dimension representation of the input vector.

The original input X will be projected to $Z = XV$. V contains the eigenvectors and we can choose k eigenvectors with the largest eigenvalues. To reconstruct the original input,

$x = XVV^T$. If all the eigenvectors are used, then VV^T is an identity matrix, no dimensionality reduction is performed, the reconstruction is perfect. If a subset of eigenvectors are used, the reconstruction is not perfect, the reconstruction error $E = ||x - X||^2$. If the training data share similar structure or characteristic, the reconstruction error should be small. An outlier is data that has very different underlying structure or characteristic. Therefore, it is difficult to represent the outlier with the feature we extract, the reconstruction error is larger. We can use the reconstruction error as a standard to detect abnormal traffic.

Autoencoder However, the transformation PCA performed is linear. Therefore it cannot capture the true underlying structure if the relationship to be modeled is non-linear. Deep neural network has achieved great success recent years in computer vision, speech recognition, natural language processing etc [139]. With non-linear activation function and multiple hidden layers, DNN can model complex non-linear functions, which make it an ideal candidate for anomaly detection in web attack. More specifically, we use a special case of neural network called autoencoder.

An autoencoder is a neural network with a symmetric structure. It consists of two parts: an encoder which maps the original input to hidden layer h with an encoder function $h = f(x) = s(Wx + b)$ where s is the activation function; an decoder that produce a reconstruction $r = g(h)$. The goal of normal neural network is to learn a function $h(x) = y$ where the target variable y can be used for classification or regression. While an autoencoder is trained to have target value equal to input value, in other words, to minimize the difference between target value and input value: $L(x, g(f(x))$ where L is the loss function. It penalizing g(f(x)) for being dissimilar from x.

If no constraint is enforced, an autoencoder will likely to learn an identity function by just copying the input to the output, which is not useful. Therefore, the hidden layers in autoencoders are usually constrained to have smaller dimensions than input x. This forces autoencoders to capture the underlying structure of the training data.

Figure 5.3 shows a visualization of normal and abnormal requests represented using the

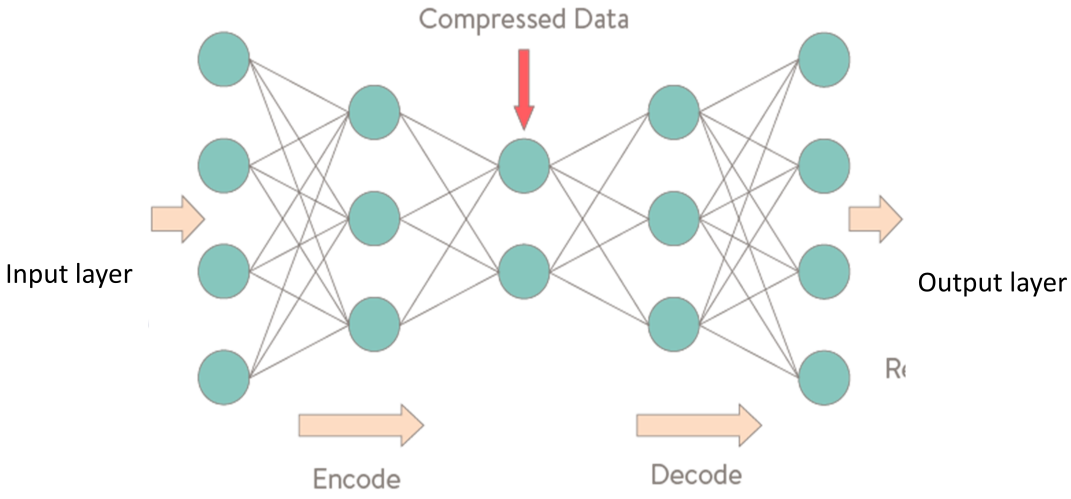

Figure 5.2: Structure of stacked autoencoder.

compressed representation learned from autoencoder using t-Distributed Stochastic Neighbor Embedding (t-SNE) [140]. We can see the abnormality can be easily distinguished in the low-dimensional subspace learned with autoencoder.

When the reconstruction g(f(x)) is different from x, the reconstruction error $e = ||g(f(x) - x)||^2$ can be used as an indicator for abnormality. If the training data share similar structure or characteristic, the reconstruction error should be small. An outlier is data that has very different underlying structure or characteristic. Therefore, it is difficult to represent the outlier with the feature we extract, the reconstruction error is larger. We can use the reconstruction error as a standard to detect abnormal traffic.

Compared to PCA, autoencoder is more powerful because the encoder and decoder function can be chosen to be non-linear to capture non-linear manifold. While PCA only does linear transformation so it can only capture linear structure.

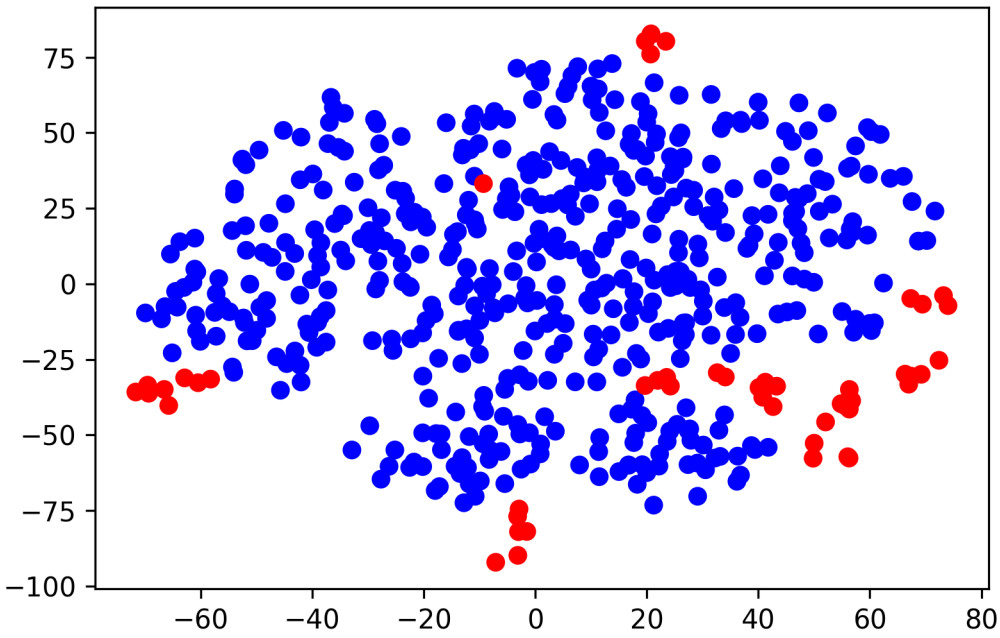

Figure 5.3: t-SNE visualization of normal and abnormal requests. Blue dots represent normal request and red dots represent abnormal requests.

There are several extensions to the standard autoencoder. 1. stacked autoencoder [141]. Autoencoder can contain more than one hidden layer. The output of each preceding layer is feed as the input to the successive layer. For encoder: $h_1 = f(x)$, $h_i = f(h_{i-1})$. For decoder: $g_1 = g(h_i)$, $g_i = g(g_{i-1})$. Deep neural network has shown promising applications in a variety of fields such as computer vision, natural language processing due to its representation power. The advantages also apply to deep autoencoder. To train a stacked autoencoder, a pretraining step involving greedy layer-wise training is used. We first train the first layer of encoder on raw input. After a set of parameters are obtained, we use this layer to transform the raw input a vector represented as the hidden units in the first layer. We then train the second layer on this vector to obtain the parameters of second layers. We repeat this process by training the parameters of each layer individually while keep the parameters of other layers unchanged. 2. Denoising. Denoising is another technique to

prevent autoencoder from learning an identity function. It works by corrupting the original input with some form of noise. The autoencoder now needs to reconstruct the input from a corrupted version of it. This forces the hidden layer to capture the statistical dependencies between the inputs. More detailed explanation of why denoising autoencoder works can be found in [142]. In our experiment, we implement the corruption process by randomly set 20% of the entries of each input as 0.

In our experiment, an denoising autoencoder with 3 hidden layers were chosen. The structure of the autoencoder is shown in figure. The hidden layer contains n/2, n/4, n/2 dimensions respectively. Adding more hidden layers does not improve the performance and are easily overfitted. Relu [143] was chosen as the activation function in the hidden layer.

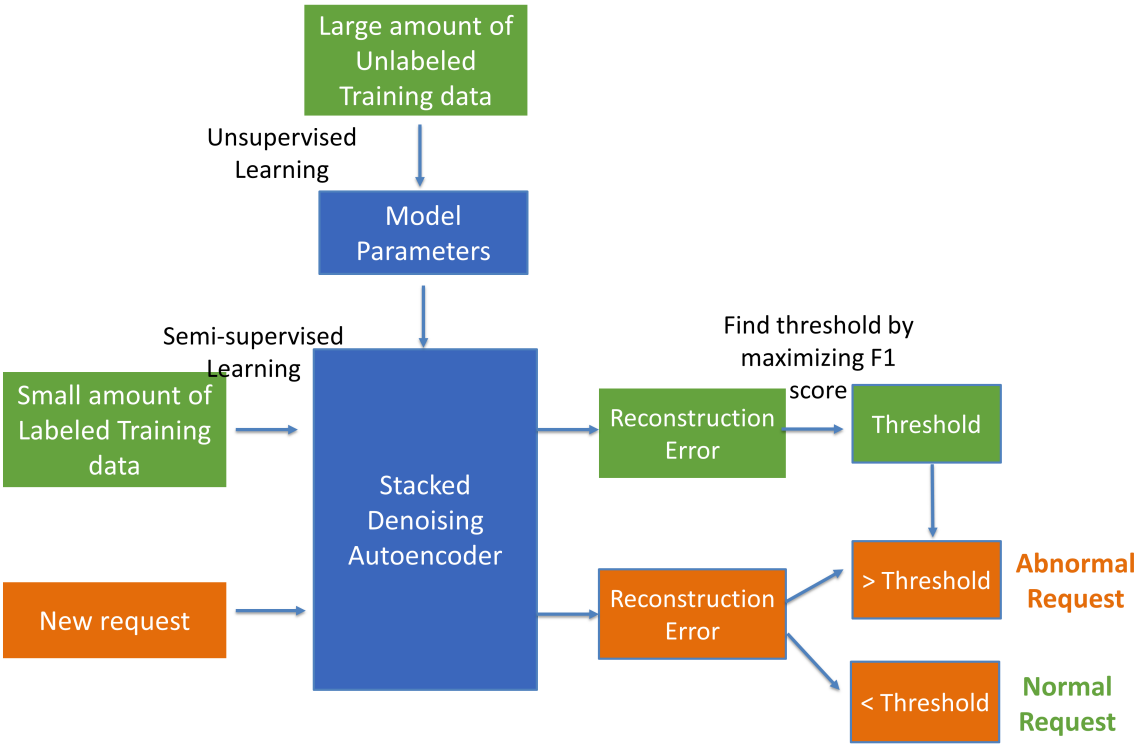

Figure 5.4: The architecture of the proposed unsupervised/semi-supervised web attack detection system.

The architecture of our unsupervised/semi-supervised web attack detection system is shown in Figure 5.4.

1. A large number of Unlabeled training trace is collected with RSMT and simulated user requests. The unlabeled training trace should contain mostly normal requests, but it is acceptable if few abnormal requests slip in.

2. A stacked denoising autoencoder is used to train on the unlabeled training trace. By minimizing the reconstruction error, the autoencoder learn a embedded low dimension subspace which can represent the normal requests with low reconstruction error.

3. A semi-supervised learning step is optional where a small amount of labeled normal and abnormal request data is collected. Normal request data can be collected with running repetitive unit test. Abnormal request data can be collected by manually creating attack requests such as SQL injection, Cross Site Scripting. The transformation learned in unsupervised learning is applied to both normal and abnormal requests and their average reconstruction error is calculated respectively. A threshold for reconstruction error is chosen to maximize metric such as F1 score.

4. If no semi-supervised learning is conducted, the highest reconstruction error for unlabeled training data is recorded and the threshold is set to a value that is higher than this maximum by a adjustable percentage.

5. When a new test request arrived, the trained autoencoder will encode and decode the request vector and calculate reconstruction error E. If E is larger than the learned threshold θ, it will be classified as attack request. If E is smaller than θ, it will be considered as normal requests.

5.3.3 Leveraging Blockchain for Automatic and Public Attack Disclosure

When Web attacks happen, the web service providers may not discover the attack immediately. Even if they do, they may prefer the ignore or delay the disclosure of the attack to minimize the impact. However, this will leave users' sensitive information at risk and prevent users from taking immediate action to reduce losses. In the Equifax data breach, Equifax revealed the attack until months after the attack happened. To prevent such things happen again, we propose an automatic and public attack disclosure architecture by leveraging blockchain technology.

Blockchain is an emerging technology that provides a decentralized way of information sharing over a network of untrusted participants [144]. The concept of Blockchain has achieved great success as the underlying infrastructure of Bitcoin, in which blockchain serves as a public ledger to store all transactions of Bitcoin network. Blockchain is rapidly growing in popularity and has been applied to a wide range of areas such as digital currencies, medical records, intellectual property protection [145].

Blockchain consists of a chain of block. Each block can contain any form of information that can be digitalized such as transaction data, contracts, identities, etc. A block consists of a batch of transactions and a block head which includes the hash of previous block head. An update will add a new block to the end of the existing chain. Blockchain implements protocols on how to valid and distribute new blocks. Each node in the network stores an identical copy of the blockchain and contributes to the process of certifying digital transactions of the network. When a signed transaction is properly formed, it is sent to a few other nodes on the blockchain network for validation. And the few nodes will then send it to their peers. If a majority of the nodes in the network agree the transaction is valid, the new transaction will be added to the blockchain and the changes will be reflected in all copies of the blockchain in the network. The blockchain network relies on miners to aggregate transactions into blocks and append them to blockchain.

Two important characteristics are guaranteed with blockchain which make it attractive

for a wide range of applications: 1) Immutable: You can only insert new records to the transaction history but cannot update or delete existing records. 2) Decentralization: There is no single entity controlling the blockchain. Entities can achieve trust without the need to rely on any central governing authority or third-party control.

Because machine learning algorithms can be computationally expensive. It is natural to have multiple distributed systems to compute the detection. Blockchain can provides a medium for machines to communicate their findings (particularly in the case of unsafe behaviors) effectively. Also, since the attack was written to blockchain, it ensure the public disclosure of the attack. The Blockchain can be used to require K of N agreement since deep learning may not be perfectly accurate.

5.4 Empirical Results

This section presents the experimental evaluation of the proposed system. We first describe the test environment, evaluation metrics. Then we compare the performance of deep learning approach with other benchmarked methods.

5.4.1 Experiment Testbed

Three web applications have been used as testbed for web attack detection.

1. Video upload application. The application is built upon Apache Spring framework with embedded HSQL database. The function of the application is to handle HTTP request for uploading/downloading/viewing video files.

2. PetClinic. Petclinic is a Spring-based application for storing and managing the relationship of owners and their pets.

3. Compress application. The application is built with Apache Commons Compress library. It takes a file as input and output a compressed file in various format.

Our test application is a Spring-Boot webapp engineered in a manner that intentionally leaves it vulnerable to several widely-exploited vulnerabilities. The test emulates the be-

havior of both normal (good) and abnormal (malicious) clients by issuing service requests directly to the test webapps REST API. For example, the test harness might register a user with the name “Alice” to emulate a good clients behavior or “Alice OR true” to emulate a malicious client attempting a SQL injection attack. SQL injection attack is constructed as follows:

Construct queries with permutations/combinations of keywords INSERT, UPDATE, DELETE, UNION, WHERE, AND, OR, etc.

1. Type1: Tautology based: Add statement suc has OR ’1’ = ’1’ and OR ’1’ $<$ ’2’ at the end of the query. For example: SELECT * FROM user WHERE username = ’user1’ OR ’1’ = ’1’.

2. Type2: Comment based: $--$. For example: SELECT * FROM user WHERE username = ’user1’ – AND password = ’123’.

3. Type3: Use semicolon to add additional statement. For example: SELECT * FROM user WHERE username = ’user1’; DROP TABLE users; – AND password = ’123’.

For XSS attack, we add a new method with an @RequestMapping in a controller that is never called in the “normal” set. Then, try calling it in the abnormal set. Also, we Modify an existing controller method with @RequestMapping so that a special value of one of the request paths calls a completely different code path to be executed. Only trigger this alternate code path in the abnormal set.

Object deserialization vulnerabilities can be exploited by crafting serialized objects that, during the deserialization process, will invoke reflective methods that result in unsafe behaviors. For example, we could store ReflectionTransformer items in an ArrayList that result in Runtime.exec being reflectively invoked with arguments of our choice (effectively enabling us to execute arbitrary commands at the privilege level of the JVM process). To generate such serialized objects targeting the Commons-Collections library, we used the ysoserial tool [146].

For the Compression application, 1000 traces were collected. All runs compress 64MB of randomly generated data, but the method of generating the random data is different for each run. For each of $x \in$1,2,4,8,16,32,64,128,256,512,1024,2048,4096, a single chunk of size 64MB/x is generated and duplicated x times. i.e., with x = 4096 the data is very repetitive whereas with x = 1, the data is not repetitive at all.

This illustrates that control flow of compression algorithms is very input-dependent, and its not likely feasible to create inputs/test cases that would exercise all possible control flow paths. To collect normal request data: 1. We use the JMeter [147] test recorder to record test scripts. 2. We use a site spider to generate simulated usage. 3. We also use some manual testing to generate simulated usage (e.g., adding pets, etc.)

5.4.2 Evaluation Metrics

An ideal system should detect the legitimate traffic as normal and detect attack traffic as abnormal. Therefore, there are two kinds of error exist. The first is false positive (FP) or false alarm, which refers to the detection of benign traffic as attack. The second is false negative (FN), which refers to detecting attack traffic as benign traffic. The goal should be to minimize both FP rate and FN rate. However, a trade-off exists as a more strict algorithm will tend to reduce the FN rate but at the cost of detecting benign traffic as attack. Since anomaly detection is a very imbalanced classification problem: attack test case is much more rare than normal test case. Accuracy is not a good metric because simply predicting every request as normal with give a very high accuracy. So we choose Precision, Recall and F1 score as the metrics for evaluation. Precision = TP/(TP+FP) which penalize false positive, Recall = TP/(TP+FN) which penalize false negative. F1 score is a metric that evenly weights precision and recall.

$$F_1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{5.1}$$

5.4.3 Experiment Benchmarks

There are several techniques we can use to differentiate benign traffic and attack traffic. The first is the **naive approach**. The naive approach will learn a set of method calls from a training set (obtained by unit test or simulated legitimate requests). If encounter a new trace, the naive approach will check if the trace contain any method call that is never seen from the training set. If there is such method, the trace will be treated as attack trace. Otherwise, it is considered safe. The naive approach can detect attack traces easily since attack traces usually contains some dangerous method calls that will not be used in legitimate operation. However, naive approach also suffer from high false positive rate as it is sometimes impossible to iterate all the legitimate request scenario. So a legitimate request may contain some method call that does not exist in training set, which results in blocking a benign traffic.

A more advanced technique is **one-class SVM** [148]. Traditional SVM solves the two or multi-class situation. While the goal of a one-class SVM is to test new data and found out whether it is alike or not like the training data. By just providing the normal training data, One-class classification creates a representational model of this data. If newly encountered data is too different, according to some measurement, it is labeled as out-of-class.

5.4.4 Experiment Result

Table 5.1 and Table 5.2 show the performance comparison of different algorithms on two testbed web applications. For the video upload application, the attack threat is SQL injection and XSS. We can see Autoencoder outperform other algorithms. For compression application, we evaluate the detection performance in terms of deserialization attack.

Figure 5.5 plots the precision/recall/fscore curve along with threshold value. We can observe a tradeoff between precision and recall. If we choose a threshold that is too low, many normal request will be classified as abnormal, resulting in higher false negative and

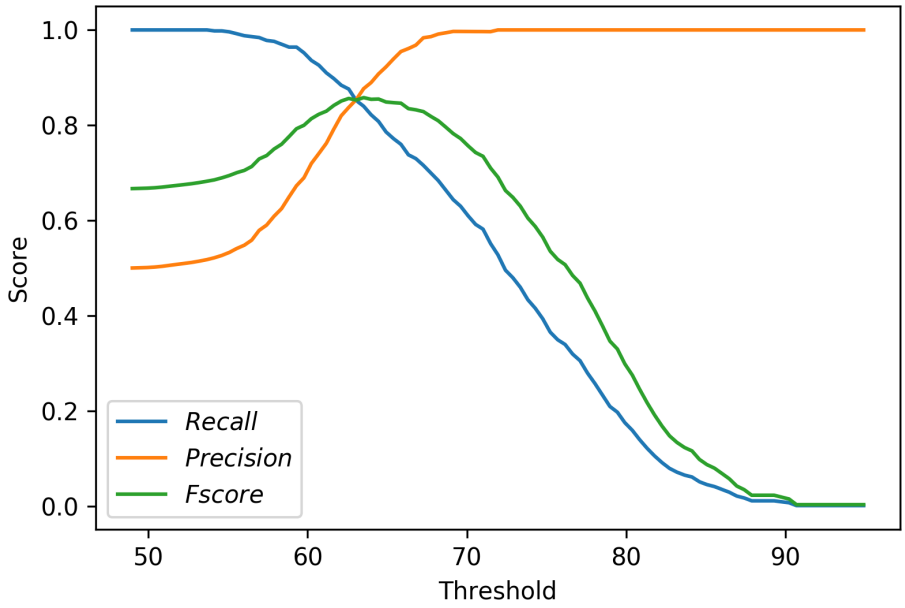

Figure 5.5: Threshold is chosen with max F-score.

low recall score. If we choose a threshold that is too high, many abnormal requests will be classified as normal, leading to higher false positive and low precision score. In our

Table 5.1: Video upload application (about 60 traces): Comparison of performance of different machine learning algorithms for anomaly detection of traces

	Precision	Recall	F-score
Naive	0.722	0.985	0.831
PCA	0.827	0.926	0.874
One-class SVM	0.809	0.909	0.858
Autoencoder	0.898	0.942	0.914

Table 5.2: Comparison of performance of different machine learning algorithms for anomaly detection of traces on Compression Application

	Precision	Recall	F-score
Naive	0.421	1.000	0.596
PCA	0.737	0.856	0.796
One-class SVM	0.669	0.740	0.702
Autoencoder	0.906	0.928	0.918

experiment, we choose the threshold that maximize the fscore in the labeled training data to balance the precision and recall.

Synthetic dataset: to understand how various parameters such as training data size, input feature dimension, test coverage ratio will affect the performance of machine learning algorithms, we manually create synthetic dataset to simulate web application requests. A trace contain N unique method calls. The number of method i being called is A_i.

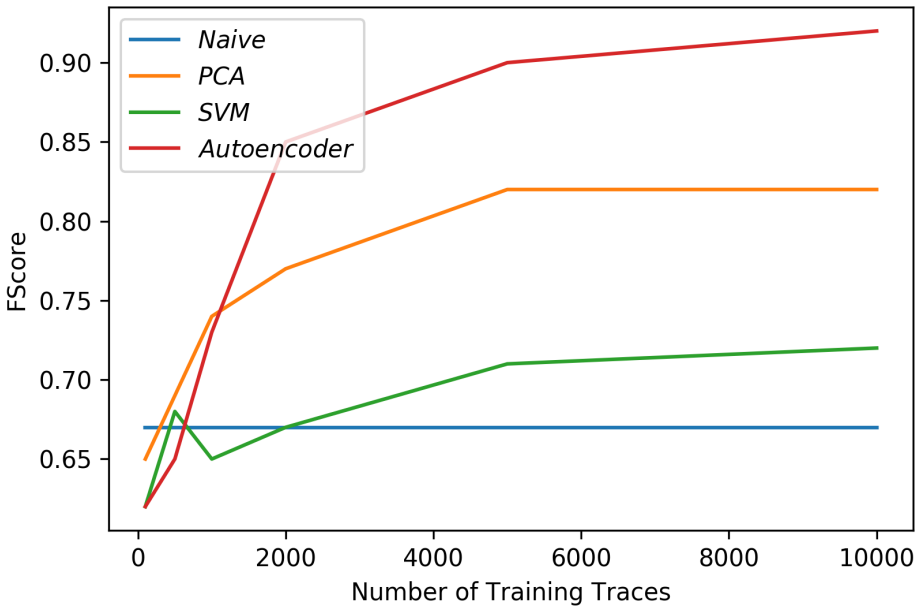

Figure 5.6: Performance of different machine learning algorithm under different unlabeled training data size.

Figure 5.6 shows the performance of machine learning algorithms with different unlabeled training data size. Since the test case contains method call that were not presented in the training data, Naive approach will simply treat every request as abnormal, resulting 100% recall but 0% precision. Both PCA and autoencoder's performance improve as we have more training data. However, when there is limited training data (below 1000), PCA have better performance. Autoencoder needs more training data to converge but once given enough training data, autoencoder outperforms other machine learning algorithms. Our experiments show the autoencoder will generally need at 5000 unlabeled training data to

achieve good performance.

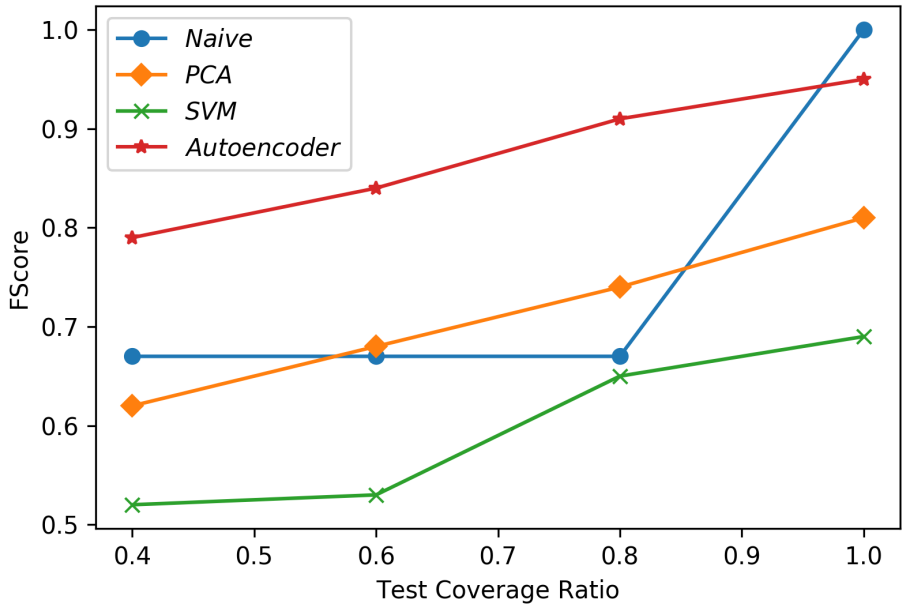

Figure 5.7: Performance of different machine learning algorithm under different test coverage ratio.

Figure 5.7 shows the performance of machine learning algorithms to different test coverage ratio. The test coverage ratio is the percentage of method calls covered in the training dataset. For large-scale web applications, it is sometimes impossible to traverse every execution path and method calls due to path explosion problem [149]. If only a subset of method calls are present in the training dataset, naive approach or other supervised learning approach might classify the legitimate test request with uncovered method calls as abnormal. However, PCA and autoencoder can learn a hidden manifold by finding the similarity in structure instead of exact method calls. Therefore, they can still give a good performance even given only a subset of coverage of all the methods call.

Figure 5.8 shows the performance of machine learning algorithms with different feature dimension. We keep the unique feature ratio as a constant. We can see the difference between autoencoder and other approaches are not significant when the number of feature is small. However, the gap becomes larger as the number of feature keep increasing. The

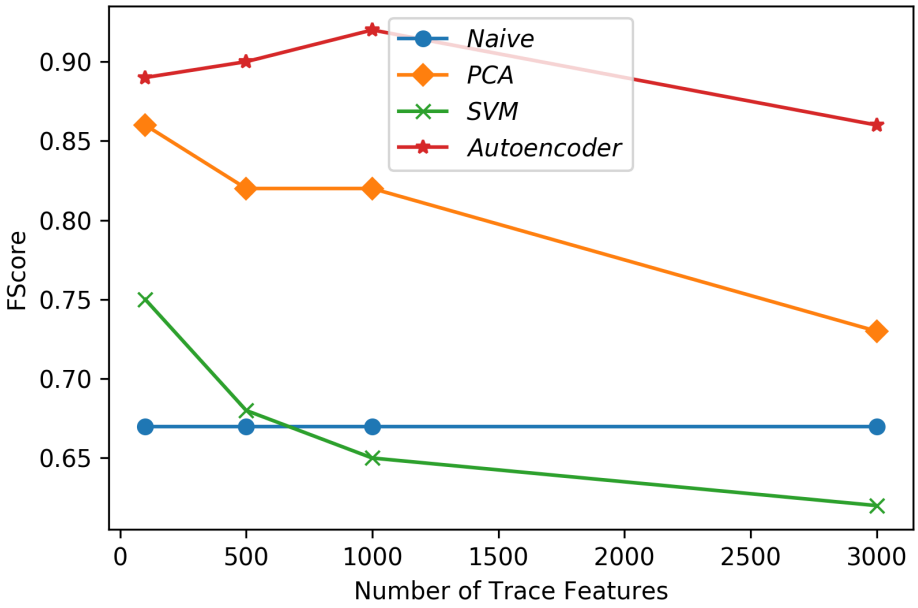

Figure 5.8: Performance of different machine learning algorithm under different input feature size.

autoencoder show robust performance even with complicated high dimension input data.

Figure 5.9 shows the performance of machine learning algorithms to different unique feature ratio. The performance of machine learning algorithms improve as the unique feature ratio increase. This is not surprising because the statistical difference between normal and abnormal requests are larger and easier to capture. For autoencoder, at least 2% of unique features are needed in the abnormal requests for acceptable performance.

The experiment was conduct on a desktop with Intel i5 3570 and GTX 960 GPU on a

Table 5.3: Comparison of training/classification time for different algorithms. The training was performed with the same set of 5000 traces with default parameters specified above. The classification time is the average time to classify one trace over 1000 test traces.

	Training Time	Classification Time
Naive	51s	0.05s
PCA	2min 12s	0.2s
One-class SVM	2min 6s	0.2s
Autoencoder	8min 24s	0.4s

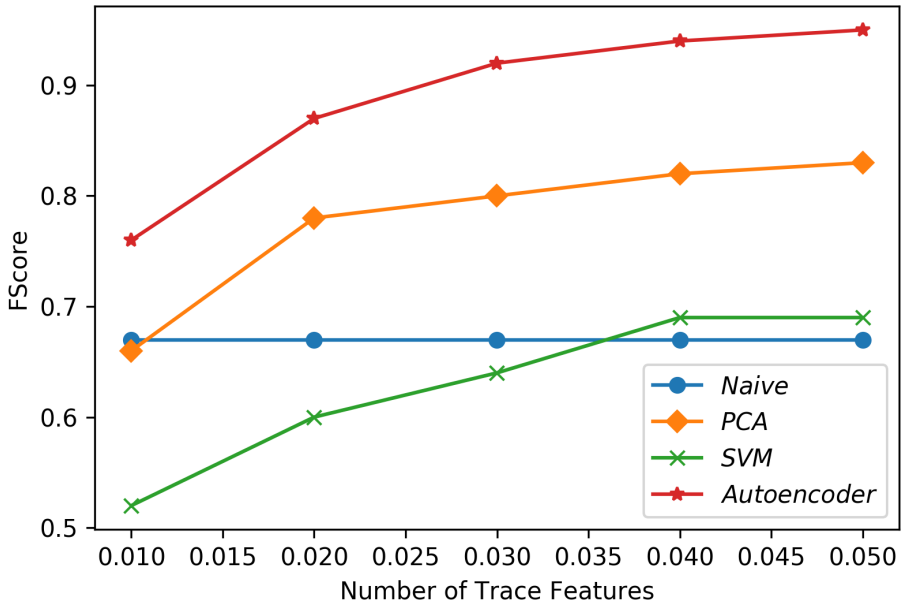

Figure 5.9: Performance of different machine learning algorithm under different unique feature ratio.

Windows 10. Autoencoder was implemented with keras 2.0 with tensorflow backend.

As we can see from Table 5.3, the training time of deep autoencoder is significant longer than other approaches. However, the training does not need to be performed frequently and can be done in offline. Also, existing deep learning framework such as tensorflow provides support for GPU, which can also significantly speed up the training time. For the classification time, with the trained model, all the algorithm can perform classification in an acceptable short period of time.

Although deep learning approach achieve better detection accuracy, the computational cost it needs is still significant more than traditional approaches. However, hardware advances such as Tensor Processing Unit [150] are bringing high performance and low cost computing resources in the future. And we expect the computation cost will not be a bottleneck for the deployment of deep learning for cyber attack detection.

Chapter 6

Conclusions

This dissertation presents Gray Computing, a novel framework for distributed computing for web browsers. Several key research questions have been proposed regarding the challenges Gray Computing will face. Then empirical experiments and theoretical reasoning have been conducted to demonstrate the feasibility of Gray Computing framework. Some real-world potential applications of Gray Computing have been examined.

In the first part, we primarily focus on the legitimate usage of Gray Computing. However, given its great potential and relatively loose constraints, Gray Computing could also be misused by attackers. We continue to evaluate browser resource stealing attack, which is the unauthorized usage of Gray Computing. DDoS attack, distributed password cracking and cryptocurrency mining have assessed in their capacities in browser resource stealing attack. Browser resource stealing attack has been compared with traditional attack vectors such as botnet and cloud computing. The empirical results show browser resource stealing attack is a serious threat to web security and measures need to be taken to prevent malicious exploiting website visitors' computing resources.

In the third part, we focus on an application of gray computing in cyber security. Deep learning has achieved great success recent year in computer vision, speech recognition and natural language processing. Many deep learning libraries and framework in JavaScript emerge such as Keras.js, WebDNN, deeplearn.js. This enables us to build deep learning application through web browsers and offload expensive computation to the client browsers.

In our research, we evaluate the feasibility of an unsupervised/semi-supervised approach for web attack detection. The proposed approach utilizes Robust Software Modeling Tool (RSMT) to derive the call graph from web application runtime when a request has been processed. We then train a stacked denoising autoencoder to encode and reconstruct

the call graph. The encoding process can learn a low-dimensional representation of the raw features with unlabeled request data. The reconstruction error of the request data is used to recognize anomalies. We empirically test our approach on both synthetic datasets and real-world applications with intentionally perturbed vulnerabilities. The results show that the proposed approach can efficiently and accurately detect attack requests ranging from SQL injection, cross-site scripting, deserialization attack with minimal domain knowledge and labeled training data. We also explore the possibility of leveraging Blockchain technology to make attack disclosure automatic and public.

6.1 Summary of Contributions

Gray Computing: Distributed Computing with Web Browsers (completed and presented in Chapter 3). We explore the feasibility, cost-effectiveness, user experience impact, and architectural optimizations for leveraging the browsers of website visitors for more intensive distributed data processing, which we term gray computing. Although previous research has demonstrated it is possible to build distributed data processing systems with browsers when the web visitors explicitly opt into the computational tasks that they perform, no detailed analysis has been done regarding the computational power, user impact, and cost-effectiveness of these systems when they rely on casual website visitors. The empirical results from performing a variety of gray computing tasks, ranging from face detection to sentiment analysis, show that there is significant computational power in gray computing and large financial incentives to exploit its use.

We also derived a cost model that can be used to assess the suitability of different tasks for distributed data processing with gray computing. This cost model can aid future discussions of legitimately incentivizing users to opt-into gray computing. Further, we pinpoint the key factors that determine whether a task is suitable for gray computing and provide a process for assessing the suitability of new data processing task types, which can help in guiding the design of gray computing systems and user incentive programs. We

also present a number of architectural solutions that can be employed to exploit cost and performance asymmetries in cloud environments to improve the cost-effectiveness of gray computing for legitimate uses.

We believe that software engineers will find additional compelling reasons for adopting gray computing as a platform for development of practical applications. We envision a business model that can be based on gray computing to utilize idle computing resources, which can bring benefit to both website owners and visitors.

Browser Resource Stealing Attack (completed and presented in Chapter 4). We evaluated the potential of a using background Web Worker Javascript tasks to steal computational resources from website visitors. We showed that not only can ad impressions be used to launch these resource-stealing attacks, but that there are many tasks ranging from rainbow table generation to DDoS attacks were they are economically attractive to attackers. Web Worker resource-stealing attacks are carried out by embedding malicious JavaScript code in a webpage delivered to normal website visitors. Attackers are able to launch application-layer DDoS attacks or offload computing tasks such as password cracking or rainbow table generation to the background tasks running in Web Workers on browser clients.

In the past, JavaScript was single-threaded, which made it hard to perform this type of attack without impacting user experience and alerting the user to the malicious behavior. Web Workers, however, facilitate this type of attack by allowing JavaScript to run in background threads without impacting user experience, as we showed in our experiments. The attack does not harm the compromised user directly but allows attackers to misuse their computing resources. Based on our quantitative analysis of the costs of launching these attacks through ad networks in comparison with cloud computing, we found that these attacks are economically feasible and have a number of properties that could be attractive to attackers building new botnet types.

Featureless Web Attack Detection with Deep Network (completed and presented in Chapter 5). We presented a featureless unsupervised learning approach for automati-

cally detecting attacks for web applications. We used the Robust Software Modeling Tool (RSMT) to instrument the web application and record the control flow extracted from critical path of execution. Deep denoising autoencoder is used to learn a low-dimension representation of the call graph vector that represent the normal request well. We show that normal instances and anomalies appear significantly different in terms of reconstruction error with the compressed representation. To validate our system, we created several test applications and synthetic trace dataset. We then evaluated the performance of unsupervised learning against these dataset. Our results show unsupervised learning with deep learning can detect web attacks with high precision and recall, while with minimum domain knowledge and labeled training data.

BIBLIOGRAPHY

[1] Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org/u32/benchmark.php?

[2] Bitcoin Mining Speed. https://en.bitcoin.it/wiki/Mining_hardware_comparison.

[3] Web Workers. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

[4] YouTube Statistics. https://www.youtube.com/yt/press/statistics.html.

[5] BOINC. http://boinc.berkeley.edu/.

[6] Top500 Supercomputer. http://www.top500.org/.

[7] Yao Pan, Jules White, Yu Sun, and Jeff Gray. Gray computing: An analysis of computing with background javascript tasks. In *International Conference on Software Engineering*, 2015.

[8] Yao Pan, Jules White, and Yu Sun. Assessing the threat of web worker distributed attacks. In *IEEE Conference on Communications and Network Security (CNS)*, pages 306–314. IEEE, 2016.

[9] DDoS on Github. http://arstechnica.com/security/2015/04/02/ddos-attacks-that-crippled-github-linked-to-great-firewall-of-china/.

[10] https://blog.thousandeyes.com/chinas-new-weapon-great-cannon/.

[11] https://www.abusix.com/blog/5-biggest-ddos-attacks-of-the-past-decade.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[13] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[16] Keras.js. https://github.com/transcranial/keras-js.

[17] WebDNN. https://github.com/mil-tokyo/webdnn.

[18] deeplearn.js. https://github.com/PAIR-code/deeplearnjs.

[19] Luis FG Sarmenta. *Volunteer computing*. PhD thesis, Massachusetts Institute of Technology, 2001.

[20] Francis S Collins, Michael Morgan, and Aristides Patrinos. The human genome project: lessons from large-scale biology. *Science*, 300(5617):286–290, 2003.

[21] Great Internet Mersenne Prime Search. http://www.mersenne.org/.

[22] Distributed.net. http://www.distributed.net/.

[23] Luis FG Sarmenta and Satoshi Hirano. Bayanihan: Building and studying web-based volunteer computing systems using java. *Future Generation Computer Systems*, 15(5):675–686, 1999.

[24] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[25] Folding@Home. http://folding.stanford.edu/.

[26] https://boinc.berkeley.edu/wiki/Publications_by_BOINC_projects.

[27] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency-Practice and Experience*, 17(2-4):323–356, 2005.

[28] Open Science Grid. http://www.opensciencegrid.org/.

[29] David P Anderson. Volunteer computing: the ultimate cloud. *ACM Crossroads*, 16(3):7–10, 2010.

[30] Internet population. http://www.internetlivestats.com/internet-users/.

[31] Tomasz Krupa, Przemyslaw Majewski, Bartosz Kowalczyk, and Wojciech Turek. On-demand web search using browser-based volunteer computing. In *Sixth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 184–190. IEEE, 2012.

[32] Fumikazu Konishi, S Ohki, Akihiko Konagaya, Ryo Umestu, and M Ishii. Rabc: A conceptual design of pervasive infrastructure for browser computing based on ajax technologies. In *Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 661–672. IEEE, 2007.

[33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[34] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of the*

19th ACM International Symposium on High Performance Distributed Computing, pages 95–106. ACM, 2010.

[35] Sandy Ryza and Tom Wall. Mrjs: A javascript mapreduce framework for web browsers. *http://www. cs. brown. edu/courses/csci2950-u/f11/papers/mrjs. pdf*, 2010.

[36] Sean R Wilkinson and Jonas S Almeida. Qmachine: commodity supercomputing in web browsers. *BMC bioinformatics*, 15(1):176, 2014.

[37] Yuri Namestnikov. The economics of botnets. *Analysis on Viruslist. com, Kapersky Lab*, 2009.

[38] Esraa Alomari, Selvakumar Manickam, B. B. Gupta, Shankar Karuppayah, and Rafeef Alfaris. Botnet-based distributed denial of service (ddos) attacks on web servers: Classification and art. *International Journal of Computer Applications*, 49(7):24–32, July 2012. Full text available.

[39] Yinglian Xie, Fang Yu, Kannan Achan, Rina Panigrahy, Geoff Hulten, and Ivan Osipkov. Spamming botnets: signatures and characteristics. *ACM SIGCOMM Computer Communication Review*, 38(4):171–182, 2008.

[40] Michael Bailey, Evan Cooke, Farnam Jahanian, Yunjing Xu, and Manish Karir. A survey of botnet technology and defenses. In *Conference For Homeland Security, 2009. CATCH'09. Cybersecurity Applications & Technology*, pages 299–304. IEEE, 2009.

[41] Guofei Gu, Junjie Zhang, and Wenke Lee. Botsniffer: Detecting botnet command and control channels in network traffic. 2008.

[42] Guofei Gu, Roberto Perdisci, Junjie Zhang, Wenke Lee, et al. Botminer: Clustering

analysis of network traffic for protocol-and structure-independent botnet detection. In *USENIX Security Symposium*, pages 139–154, 2008.

[43] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 129–138. ACM, 2012.

[44] VT Lam, Spyros Antonatos, Periklis Akritidis, and Kostas G Anagnostakis. Puppetnets: misusing web browsers as a distributed attack infrastructure. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 221–234. ACM, 2006.

[45] Lavakumar Kuppan. Attacking with html5. *Presentation at Black Hat*, 2010, 2010.

[46] Giancarlo Pellegrino, Christian Rossow, Fabrice J Ryba, Thomas C Schmidt, and Matthias Wählisch. Cashing out the great cannon? on browser-based ddos attacks and economics. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[47] Alexa. http://www.alexa.com/.

[48] SETI@home. http://setiathome.ssl.berkeley.edu/.

[49] EC2 Pricing. http://aws.amazon.com/ec2/pricing/.

[50] Fredrik Smedberg. Performance analysis of javascript, 2010.

[51] David P Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE, 2004.

[52] Chao Liu, Ryen W White, and Susan Dumais. Understanding web browsing behaviors through weibull analysis of dwell time. In *Proceedings of the 33rd international*

ACM SIGIR conference on Research and development in information retrieval, pages 379–386. ACM, 2010.

[53] Waloddi Weibull et al. A statistical distribution function of wide applicability. *Journal of applied mechanics*, 18(3):293–297, 1951.

[54] Hadoop. http://hadoop.apache.org/.

[55] Derrick Kondo, Filipe Araujo, Paul Malecot, Patricio Domingues, Luis Moura Silva, Gilles Fedak, and Franck Cappello. Characterizing result errors in internet desktop grids. In *Euro-Par 2007 Parallel Processing*, pages 361–371. Springer, 2007.

[56] Luis FG Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18(4):561–572, 2002.

[57] Wei Wei, Juan Du, Ting Yu, and Xiaohui Gu. Securemr: A service integrity assurance framework for mapreduce. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 73–82. IEEE, 2009.

[58] Shanyu Zhao, Virginia Lo, and C Gauthier Dickey. Result verification and trust-based scheduling in peer-to-peer grids. In *Fifth IEEE International Conference on Peer-to-Peer Computing*, pages 31–38. IEEE, 2005.

[59] Wenliang Du, Mummoorthy Murugesan, and Jing Jia. Uncheatable grid computing. In *Algorithms and theory of computation handbook*, pages 30–30. Chapman & Hall/CRC, 2010.

[60] Kan Watanabe, Masaru Fukushi, and Susumu Horiguchi. Optimal spot-checking for computation time minimization in volunteer computing. *Journal of Grid Computing*, 7(4):575–600, 2009.

[61] Karl Aberer and Zoran Despotovic. Managing trust in a peer-2-peer information

system. In *Proceedings of the tenth international conference on Information and knowledge management*, pages 310–317. ACM, 2001.

[62] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[63] Facebook hits. http://www.webpronews.com/facebook-gets-100-billion-hits-per-day-2010-07.

[64] S3 Pricing. http://aws.amazon.com/s3/pricing/.

[65] Unreal Engine in JavaScript. http://www.davevoyles.com/asm-js-and-webgl-for-unity-and-unreal-engine/.

[66] Octane. http://octane-benchmark.googlecode.com/svn/latest/index.html.

[67] WebCL. https://www.khronos.org/webcl/.

[68] Jerzy Duda and Wojciech Dlubacz. Gpu acceleration for the web browser based evolutionary computing system. In *17th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 751–756. IEEE, 2013.

[69] Emscripten. http://kripken.github.io/emscripten-site/index.html.

[70] Asm.js. http://asmjs.org/.

[71] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

[72] http://kripken.github.io/emscripten-site/docs/getting_started/FAQ.html.

[73] Google Web Toolkit. http://www.gwtproject.org/.

[74] Quake. https://code.google.com/p/quake2-gwt-port/.

[75] Eclipse Graphical Editing Framework. http://gefgwt.org/.

[76] Arno Puder, Victor Woeltjen, and Alon Zakai. Cross-compiling java to javascript via tool-chaining. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 25–34. ACM, 2013.

[77] pyjs. http://pyjs.org/.

[78] Opal. http://opalrb.org/.

[79] Closure compiler. https://developers.google.com/closure/compiler/.

[80] http://blog.venturepact.com/8-high-performance-apps-you-never-knew-were-hybrid/.

[81] Battery Status API. https://developer.mozilla.org/en-US/docs/Web/API/Battery_Status_API.

[82] Network Information API. https://developer.mozilla.org/en-US/docs/Web/API/Network_Information_API.

[83] HTML5. https://en.wikipedia.org/wiki/HTML5.

[84] https://tampermonkey.net/.

[85] https://greensock.com/js/speed.html/.

[86] https://webperf.ninja/2015/jank-meter/.

[87] https://technet.microsoft.com/en-us/library/cc749115(v=ws.11).aspx.

[88] Greasemonkey. http://www.greasespot.net/.

[89] Electroly. https://wiki.mozilla.org/Electrolysis/.

[90] James Cipar, Mark D Corner, and Emery D Berger. Transparent contribution of memory. In *Proceedings of the annual conference on USENIX'06 Annual Technical Conference*, pages 11–11. USENIX Association, 2006.

[91] http://www.wordstream.com/blog/ws/2016/02/29/google-adwords-industry-benchmarks.

[92] Ghassan O Karame, Aurélien Francillon, Victor Budilivschi, Srdjan Čapkun, and Vedran Čapkun. Microcomputations as micropayments in web-based services. *ACM Transactions on Internet Technology (TOIT)*, 13(3):8, 2014.

[93] David Hinds. Micropayments: A technology with a promising but uncertain future. *Communications of the ACM*, 47(5):44, 2004.

[94] Facebook statistics. http://expandedramblings.com/index.php/by-the-numbers-17-amazing-facebook-stats/.

[95] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, pages I–511. IEEE, 2001.

[96] Bicubic interpolation. https://en.wikipedia.org/wiki/Bicubic-interpolation.

[97] Bo Pang and Lillian Lee. Opinion mining and sentiment analysis. *Foundations and trends in information retrieval*, 2(1-2):1–135, 2008.

[98] Bo Pang and Lillian Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *Proceedings of the 42nd annual meeting on Association for Computational Linguistics*, page 271. Association for Computational Linguistics, 2004.

[99] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[100] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[101] William R Pearson. Comparison of methods for searching protein sequence databases. *Protein Science*, 4(6):1145–1160, 1995.

[102] Eugene G Shpaer, Max Robinson, David Yee, James D Candlin, Robert Mines, and Tim Hunkapiller. Sensitivity and selectivity in protein similarity searches: a comparison of smith–waterman in hardware to blast and fasta. *Genomics*, 38(2):179–191, 1996.

[103] FASTA Program. http://fasta.bioch.virginia.edu/fasta_www2/fasta_down.shtml.

[104] Robert C Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic acids research*, 32(5):1792–1797, 2004.

[105] MA Larkin, Gordon Blackshields, NP Brown, R Chenna, Paul A McGettigan, Hamish McWilliam, Franck Valentin, Iain M Wallace, Andreas Wilm, Rodrigo Lopez, et al. Clustal w and clustal x version 2.0. *Bioinformatics*, 23(21):2947–2948, 2007.

[106] MUSCLE. http://www.drive5.com/muscle/.

[107] Zheng Rong Yang, Rebecca Thomson, Philip McNeil, and Robert M Esnouf. Ronn: the bio-basis function neural network technique applied to the detection of natively disordered regions in proteins. *Bioinformatics*, 21(16):3369–3376, 2005.

[108] Martin Johns. On javascript malware and related threats. *Journal in Computer Virology*, 4(3):161–178, 2008.

[109] Adam Barth, Collin Jackson, and John C Mitchell. Robust defenses for cross-site

request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88. ACM, 2008.

[110] Unreal Engine 4 in Firefox. https://blog.mozilla.org/blog/2014/03/12/mozilla-and-epic-preview-unreal-engine-4-running-in-firefox/.

[111] NativeClient. https://developer.chrome.com/native-client.

[112] http://www.theregister.co.uk/2012/09/24/zeroaccess-botnet/.

[113] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004.

[114] DDoS Report. http://www.neustar.biz/resources/whitepapers/ddos-protection/2014-annual-ddos-attacks-and-impact-report.pdf.

[115] John the Ripper. http://www.openwall.com/john/.

[116] oclhashcat. http://hashcat.net/oclhashcat/.

[117] RainbowCrack. http://project-rainbowcrack.com/.

[118] Haining Wang, Danlu Zhang, and Kang G Shin. Detecting syn flooding attacks. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1530–1539. IEEE, 2002.

[119] Georgios Kambourakis, Tassos Moschos, Dimitris Geneiatakis, and Stefanos Gritzalis. Detecting dns amplification attacks. In *Critical Information Infrastructures Security*, pages 185–196. Springer, 2008.

[120] StarCluster. https://star.mit.edu/cluster/.

[121] Hitleap. https://hitleap.com/.

[122] Christian Rossow, Dennis Andriesse, Tillmann Werner, Brett Stone-Gross, Daniel Plohmann, Christian J Dietrich, and Herbert Bos. Sok: P2pwned-modeling and evaluating the resilience of peer-to-peer botnets. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 97–111. IEEE, 2013.

[123] http://blog.cryptohaze.com/2012/08/cryptohaze-cloud-cracking-slides-writeup.html.

[124] http://gs.statcounter.com/.

[125] http://www.enigmasoftware.com/malware-service-market-booms-prices-malware-botnet-tools-decline/.

[126] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Advances in Cryptology-CRYPTO 2003*, pages 617–630. Springer, 2003.

[127] DistrRTgen. https://www.freerainbowtables.com/.

[128] Bitcoin Difficulty. https://en.bitcoin.it/wiki/Difficulty.

[129] William G Halfond, Jeremy Viegas, and Alessandro Orso. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[130] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering*, pages 171–180. ACM, 2008.

[131] Roberto Di Pietro and Luigi V Mancini. *Intrusion detection systems*, volume 38. Springer Science & Business Media, 2008.

[132] Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. *ACM SIGOPS Operating Systems Review*, 36(SI):45–60, 2002.

[133] https://www.acunetix.com/acunetix-web-application-vulnerability-report-2016/.

[134] http://money.cnn.com/2015/10/08/technology/cybercrime-cost-business/index.html.

[135] https://www.consumer.ftc.gov/blog/2017/09/equifax-data-breach-what-do.

[136] Noam Ben-Asher and Cleotilde Gonzalez. Effects of cyber security knowledge on attack detection. *Computers in Human Behavior*, 48:51–61, 2015.

[137] Nathalie Japkowicz and Shaju Stephen. The class imbalance problem: A systematic study. *Intelligent data analysis*, 6(5):429–449, 2002.

[138] Fangzhou Sun, Peng Zhang, Jules White, Douglas Schmidt, Jacob Staples, and Lee Krause. A feasibility study of autonomically detecting in-process cyber-attacks. In *Cybernetics (CYBCON), 2017 3rd IEEE International Conference on*, pages 1–8. IEEE, 2017.

[139] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[140] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[141] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.

[142] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[143] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[144] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[145] Melanie Swan. *Blockchain: Blueprint for a New Economy*. O'Reilly Media, Inc., 2015.

[146] ysoserial. https://github.com/frohoff/ysoserial.

[147] http://jmeter.apache.org/.

[148] Yanxin Wang, Johnny Wong, and Andrew Miner. Anomaly intrusion detection using one class svm. In *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, pages 358–364. IEEE, 2004.

[149] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, 2008.

[150] David Schneider. Deeper and cheaper machine learning [top tech 2017]. *IEEE Spectrum*, 54(1):42–43, 2017.