Portland State University
PDXScholar

Dissertations and Theses | Dissertations and Theses

1-1-2010

Addressing Automated Adversaries of Network Applications

Edward Leo Kaiser
Portland State University

Let us know how access to this document benefits you.
Follow this and additional works at: http://pdxscholar.library.pdx.edu/open_access_etds

Recommended Citation
Kaiser, Edward Leo, "Addressing Automated Adversaries of Network Applications" (2010). *Dissertations and Theses.* Paper 4.

10.15760/etd.4

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

Addressing Automated Adversaries of Network Applications

by

Edward Leo Raymond Kaiser

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Wu-chang Feng, Chair
Wu-chi Feng
Byrant York
Jonathan Walpole
Scott Burns

Portland State University
©2010

ABSTRACT

The Internet supports a perpetually evolving patchwork of network services and applications. Popular applications include the World Wide Web, online commerce, online banking, email, instant messaging, multimedia streaming, and online video games. Practically all networked applications have a common objective: to directly or indirectly process requests generated by humans. Some users employ automation to establish an unfair advantage over non-automated users. The perceived and substantive damages that automated, adversarial users inflict on an application degrade its enjoyment and usability by legitimate users, and result in reputation and revenue loss for the application's service provider.

This dissertation examines three challenges critical to addressing the undesirable automation of networked applications. The first challenge explores individual methods that *detect various automated behaviors.* Detection methods range from observing unusual network-level request traffic to sensing anomalous client operation at the application-level. Since many detection methods are not individually conclusive, the second challenge investigates how to combine detection methods to *accurately identify automated adversaries.* The third challenge considers how to leverage the available knowledge to *disincentivize adversary automation* by nullifying their advantage over legitimate users.

The thesis of this dissertation is that: there exist methods to detect automated behaviors with which an application's service provider can identify and then systematically disincentivize automated adversaries. This dissertation evaluates this thesis using research performed on two network applications that have different access to the client software: Web-based services and multiplayer online games.

For my parents and Ashley.

ACKNOWLEDGEMENTS

This dissertation represents the culmination of my formal educational journey. Along the way, countless people shaped my education in ways both large and small. I would like to express my gratitude to all those who influenced me.

I would like to thank my advisor, Professor Wu-chang Feng, for his guidance throughout the course of this work. When I was stuck on a problem, his valuable advice always offered me a new perspective from which to approach the problem. From providing me with a framework for my research to reading and patiently editing my papers, each of his many contributions nudged me incrementally closer to completing my dissertation. I would also like to express my appreciation to Professor Wu-chi Feng, Professor Bryant York, Professor Jonathan Walpole, and Professor Scott Burns for serving on my dissertation committee.

I have benefited greatly from my fellow graduate students at Portland State University and OGI. Their insights, discourse, and foosball challenges provided the grist for much of my research. Among them are Chris Chambers, Francis Chang, Thanh Dang, Akshay Dua, Bill Howe, Alex Ross, Phillip Sitbon, and Jim Snow.

I would also like to thank the excellent staff at Portland State University and OGI for their expert administrative assistance and willingness to share their knowledge of school procedures. Special thanks to Kathi Lee, René Remillard, Beth Phelps, Dana Director, Lorie Gookin, Barb Mosher, and Cindy Pfaltzgraff.

Finally, I would like to thank my family. My wife, Ashley, for her faith in me, seemingly limitless patience, and editorial skills. And my parents and brother for their unflagging encouragement throughout my educational journey.

Thank you to all my colleagues, friends, and family. With such wonderful people in my life, I look forward to the exciting challenges that lie ahead.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

API	Application Programming Interface
ASLR	Address Space Layout Randomization
CAPTCHA	Completely Automated Turing test to tell Computers and Humans Apart
CPU	Central Processing Unit
DLL	Dynamic Link Library
DOS	Denial-of-Service
FPS	First Person Shooter
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IAT	Import Address Table
IA-32	Intel Architecture 32-bit
I/O	Input/Output
IP	Internet Protocol
MMORPG	Massively Multiplayer Online Role Playing Game
POW	Proof-of-Work
RMT	Real Money Trade
RTS	Real Time Strategy
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator

Chapter 1

INTRODUCTION

1.1 THE AUTOMATION PROBLEM

By design, the Internet supports a perpetually evolving patchwork of networked applications and services. Popular applications include the World Wide Web, online commerce and banking, email, instant messaging, blogging, multimedia streaming, and multiplayer online video games. Practically all networked applications have a common objective: to directly or indirectly process requests generated by humans. Specifically, the above applications all directly inform, entertain, or transact business with human users. Other fundamental Internet services like the Domain Name System (DNS) [81] or the Network Time Protocol (NTP) [80] are used by those higher level applications and thus indirectly process human requests; users rarely query them directly.

Malicious Users Automate Their Participation. Often malicious (i.e., adversarial) users of network applications employ automation to gain an advantage over or simply frustrate the non-automated users. That advantage translates to unfair access to the service, sometimes completely denying human users any access at all. Such adversaries have political, reputational, or fiscal motivations for employing automation. Adversaries have little legal disincentive to automate since they can simply operate in jurisdictions with weak cyber-laws and enforcement. Exacerbating the situation, there are few technological barriers to automate as the automation of network applications is inexpensively achieved through software agents commonly referred to as "*Internet robots*," or simply as "*bots*."

Adversary Automation Impacts Application Viability. Adversarial automation affects not only legitimate users but it also threatens the very viability of networked applications. An application's developer or proprietor (hereon referred to as the "*service provider*") runs the application to distribute ideas or goods. Adversarial automation makes the application less accessible, less efficient, and more costly for the average user and consequently results in lost revenue for the service provider.

Some adversaries intentionally attack the availability of a service. Adversaries perform these attacks (referred to as denial-of-service attacks) out of some grudge for the service provider or when hired to do so by a competitor of the service provider. When the service is unavailable, the service provider loses potential customers and the corresponding revenues. Other adversaries unintentionally lower the efficiency of an application while performing their automation. For example, email spammers intend to solicit business rather than disrupt legitimate correspondence. Other adversaries, like automated ticket scalpers, drive up the cost of popular commodities. While the scalpers turn a quick profit, legitimate customers pay a higher price for the tickets than they otherwise would have. The price represents revenue that could have been captured by the original vendor, yet may have been forgone in an effort to make the tickets available to a wider market.

Additionally, the perception of automated adversaries running rampant within an application (e.g., cheaters in online games) hurts the reputation of the service provider. In saturated and very competitive markets, a service provider's tarnished reputation will discourage participation by legitimate users and can cause the application to fail even though it may be accessible, efficient, and inexpensive.

While we oppose adversarial automation, we support service provider automation as it stands to make applications more accessible, more efficient, and less expensive for users. Specifically, we argue that *service providers must have an automated approach to mitigating automated adversaries.* This dissertation presents methods for mitigating adversarial automation in networked applications.

Legitimate Automation Exists. While uncommon, not all client automation is harmful. In fact, some types of client automation are beneficial when properly identified and operated according to directives laid out by the service provider. One notable example is the prevalence of "*web crawlers*" which algorithmically index the World Wide Web on behalf of search engines such as Google. When restrained from overwhelming a website's resources, web crawlers benefit a website as it becomes locatable through web-searches by potentially interested users. The important distinction is that *legitimate automation supports rather than undermines the goals defined by the service provider.*

1.1.1 Examples of Adversarial Automation

This section discusses specific examples of adversarial automation which negatively impacts various network applications. These examples are classified by the main advantage that each type of automation provides: repetition, accurate timing, or action precision. Some examples demonstrate that clever adversaries apply a hybrid human-software approach; automating all possible tasks of the application and merely engaging a human to circumvent existing anti-automation mechanisms. Table 1.1 on page 14 summarizes these examples.

Repetition-Based Automation

Repetition-based automation performs very simple, structurally repetitive tasks tirelessly, more quickly, and less expensively than hired humans could do alone. Repetition is the dominant advantage that automation provides to adversaries as evidenced by the breadth of the examples presented next.

Port Scans. At the network and transport layers of the Internet [130], port scans are often a precursor to other network-level attacks. A port scan is an automatically-generated sequence of packets sent to a target network in order to probe and enumerate its connected hosts. The port scan lists all the networked machines and the network services running on each of them.

Port scans are effective because network services are commonly built upon one of two transport layer protocols: the Transmission Control Protocol (TCP) [91] or the User Datagram Protocol (UDP) [90]. Both protocols rely upon ports (hence the name "*port scan*") to identify and partition traffic to different network applications running on the same machine. Network services wait for client requests at well-defined application-specific ports. By probing and eliciting a response from a specific port, a port scanner can determine whether or not a specific application is running on the target machine.

The adversary perpetrating an automated port scan is often searching for computers running application software with an unpublished or unrepaired exploit which they intend to attack before it is fixed. This type of port scan, called a "*port sweep*," probes a small number of ports across a large number of machines. At a later time, the adversary may manually subvert one or more of the vulnerable machines as a gateway into the private network (e.g., corporate, military, or government networks) or use a large set of vulnerable hosts as the initial hit-list of machines to infect with an automated network worm [107].

By themselves, port scans do not catastrophically disrupt the applications they probe. However, by mitigating port scans an application service provider can reduce the likelihood of being targeted by an imminent attack that might bring down their application or co-opt their machines into attacking other network hosts or applications.

Network Worms. A network worm is a self-replicating malicious computer program. The worm has two functions: to infect additional network hosts and to deliver a malicious payload programmed by the adversary. The first function often involves port scans to search for new network hosts with identical vulnerabilities. While the occasional port scan is not disastrous, a worm's port scan traffic grows exponentially as more network hosts are infected. As demonstrated by the Slammer Worm in January 2003 [82], this can have a crippling effect on the network and impact the accessibility of all network applications.

The second function of the worm is to execute its malicious payload. In a growing number of cases, the payload directly assaults the privacy of the machine owner and is hence referred to as "*spyware*." This payload seeks out and reports back any sensitive personal information found on the machine. Specialized spyware known as a "*key-loggers*" record passwords entered by users via their keyboards.

Most commonly though, worms install a simple process that awaits commands issued by the adversary at a future time. In effect, the adversary uses the worm to create an army of subservient networked machines (referred to as a "*botnet*") ready to perpetrate other repetitious attacks. Botnet machines (individually referred to as "*zombies*") are co-opted without their owner's knowledge and the adversary attempts to operate them in a manner that minimizes the likelihood that owners discover and remove the zombie processes. Botnets range in size from hundreds to tens of thousands of zombie machines. In some remarkable cases, botnets consisting of hundreds of thousands of zombies have been recorded [21, 23].

Denial-of-Service Attacks. Denial-of-Service (DOS) attacks directly assault the accessibility of an application by overwhelming the service provider with bogus requests. The unrelenting barrage of bogus requests exhausts all the service provider's resources so that it cannot respond to legitimate user requests. Distributed Denial-of-Service (DDOS) attacks often leverage botnets to radically scale the resource imbalance in the adversary's favor. This also allows the adversary to assign the attack effort over many zombie machines to reduce the likelihood that individual zombies are discovered and repaired. Vulnerable service provider resources include network bandwidth, computation cycles, and machine memory.

Regardless of the application, network bandwidth is a vulnerable service provider resource. As an expensive ongoing operating cost, service providers rationally provision only enough bandwidth to handle the expected client load for the application with some bandwidth to spare. When an adversary has greater bandwidth at their disposal than the service provider (e.g., the adversary controls a large botnet), it can swamp the application with a flood of network packets.

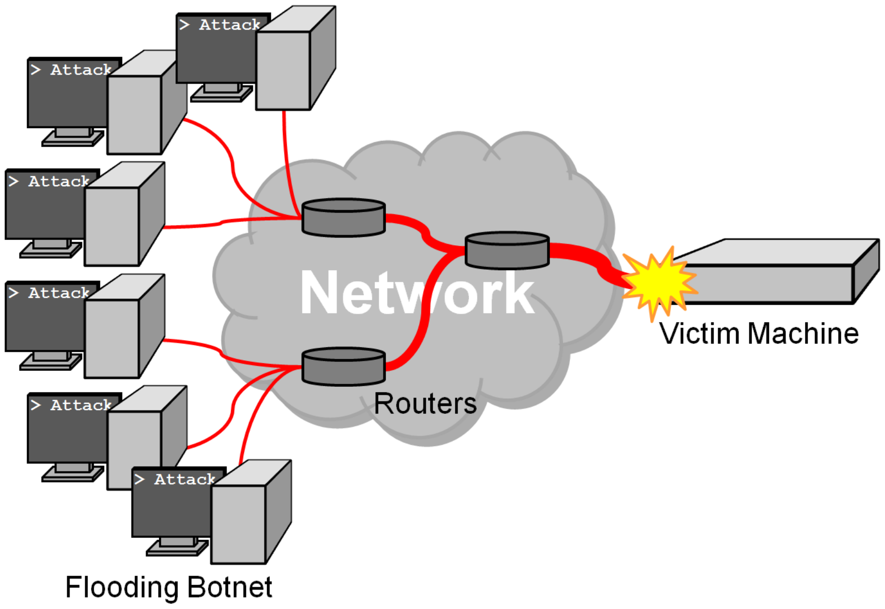


Figure 1.1: Example of flooding-based Distributed Denial-of-Service attack. Each zombie machine generates a small, steady, packet flow. The flows converge as they approach the victim and eventually fully congest a network link to the point where packets, including those from legitimate users, are discarded.

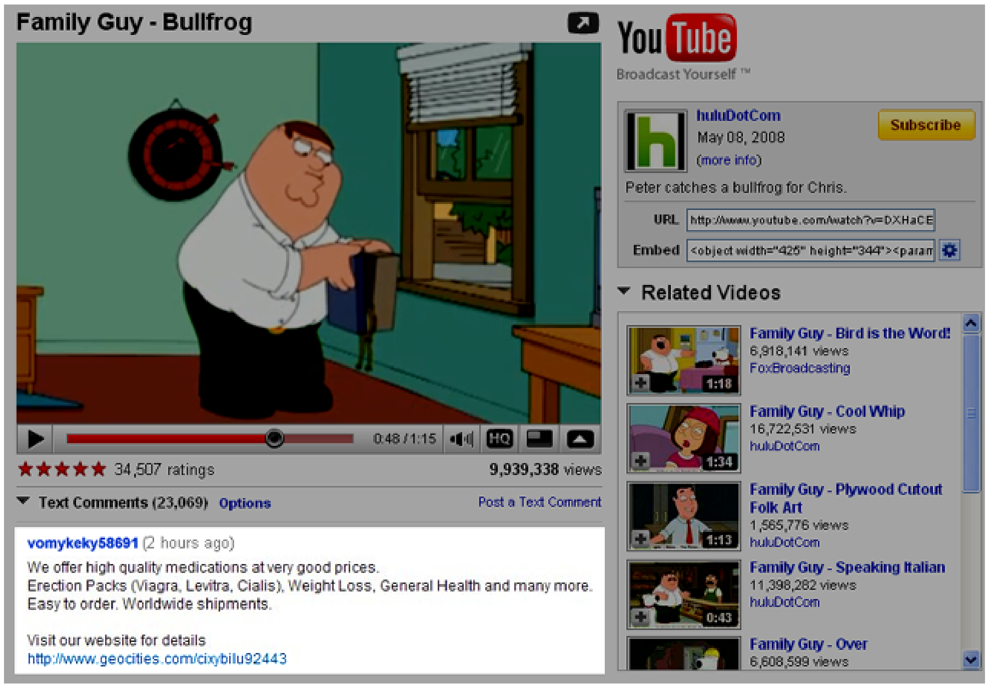
The attack congests the service provider's connection to the Internet as illustrated by Figure 1.1. In this state, network packet buffers are filled to capacity and are forced to discard packets. During a persistent packet flood, many packets belonging to legitimate clients are discarded, thereby disrupting or completely denying communication with the service provider.

To orchestrate a computation-depletion DOS attack, an adversary employs automation to repeatedly send application-specific requests that are difficult for the service provider to process. These requests are often easy for the adversary to generate but are hard for the service provider to dismiss outright since they adhere to the application's protocol. The complex yet bogus requests consume a disproportionately large amount of the service provider's processor cycles, reducing the cycles it has left to process legitimate requests. This critically lowers the application's efficiency and negatively impacts its accessibility.

Memory consumption attacks operate in much the same manner. An intent adversary repeatedly sends application-specific requests to participate in the application's protocol just to the point where the service provider allocates some of its limited memory to transacting with the adversary. Using only a few machines, an adversary can keep all of the service provider's memory resources committed to connections waiting for further correspondence that will not come. Although those connections will eventually timeout and be reset, the adversary can continue to reconnect and reoccupy the resources as frequently as they timeout.

Email and Instant Messaging Spam. Spam is a nuisance that practically every email user has experienced at some time or another. Spam is any unwanted solicitous or duplicitous electronic message. While widely recognized as a problem for email, spam is also a problem for other electronic communication methods, including instant messaging and messaging within online video games. Solicitous spam often advertises goods at unbelievably low prices in an attempt to sell imitation goods to the victim or to outright defraud them. Deceptive spam messages, known as "*phishing*" attacks, trick recipients into believing that the sender is the administrator of their email account, online bank account, or another web-based account. Through this deception, the adversary elicits victims to reveal their passwords or other personal information. An adversary on a spam campaign typically employs a botnet to generate and deliver messages to a large, indiscriminate set of recipients en masse. While the success rate for any single message is low, the success of even a small fraction of messages makes this automation immensely profitable for the adversary.

Comment Spam. Similar to spam, adversaries flood websites that allow users to submit comments with solicitous messages. Often an adversary uses a specialized web crawler to locate websites that accept user comments and then automatically enter the spam message. This nuisance is colloquially referred to as "*crap-flooding*" by bloggers frustrated about the impact of this indiscriminant attack on their blogs.



vomykeky58691 (2 hours ago)
We offer high quality medications at very good prices.
Erection Packs (Viagra, Levitra, Cialis), Weight Loss, General Health and many more.
Easy to order. Worldwide shipments.

Visit our website for details
http://www.geocities.com/cixybilu92443

Figure 1.2: Solicitous spam comment posted on YouTube. The comment (highlighted in the lower left corner of the image) is advertising "quality medications at very good prices" by following a link to a temporary website.

While comment spammers may be trying to solicit business directly from readers, spammers also have another goal which is unique to this application: bolstering the rank of their website in search engine results. Specifically, they intend to improve their rank within Google search results, hence this attack is also known as "*Google bombing*." This motivation exists because the PageRank algorithm [87] that powers the Google search engine sorts search results partially by how well each webpage is connected to the rest of the Internet. In particular, a result is biased towards the top of the results list if other popular websites link to it numerous times, effectively favorably endorsing it. Ironically YouTube, which is owned by Google, is a fashionable website for adversaries to comment spam as shown in Figure 1.2. Those comment spammers hope to transform some of YouTube's massive popularity into improved ranking of their own temporary, commercial website.

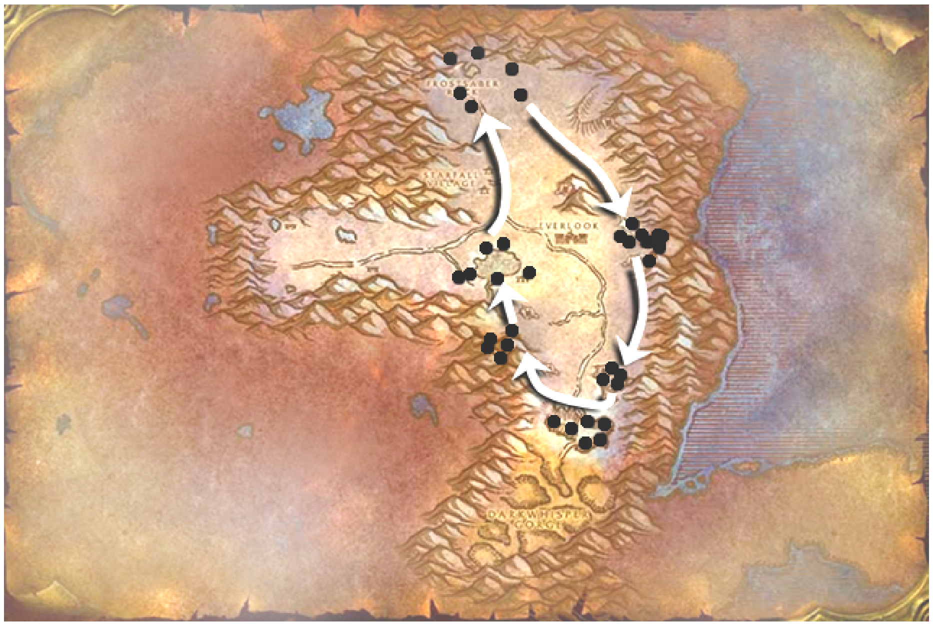


Figure 1.3: Gold farmer botting path. This path was historically used by gold farming bots to mine valuable ore in the World of Warcraft zone Winterspring. Dots indicate the spawn locations of precious Rich Thorium Ore nodes.

Leveling & Grinding Bots. In massively multiplayer online role playing games (MMORPGs) like World of Warcraft [13], a player's social status is closely tied to the exceptionality of their in-game avatar. Specifically, players strive to have high-level avatars with rare equipment. Game developers make this goal non-trivial so that players must continue to play the game (while paying a monthly subscription fee) to accomplish this. Furthermore, to keep the game interesting, the developer periodically patches the game to include new, more powerful equipment and allow avatars to attain new, previously unattainable levels. Individuals without the patience to play the game and achieve these goals resort to automated software to play their avatar on their behalf. The act of mindlessly and continuously hunting virtual monsters is referred to as "*grinding*." Automated software that grinds character levels, in-game currency, or virtual loot are called "*gliders*" and are of great concern to game developers because they shorten the normal life cycle of virtual assets.

Game automation spawned the Real Money Trade market (RMT) [108] where virtual gold and items are exchanged for real currencies. Enterprising individuals in foreign countries exploit cheap labor to manage the automation of multiple avatars simultaneously [58]. The laborers (disparagingly referred to as "*gold farmers*") dislodge avatars that get stuck on terrain obstacles, avoid vigilante players attempting to disrupt them, and dispute game moderators investigating player-generated reports about their automated behavior. Figure 1.3 shows a circuit within a World of Warcraft zone used at one time by gold farming bots to gather precious ore and gems.

Besides undermining the game's enjoyment for legitimate players, the continuous stream of complex in-game actions that gold farming bots undertake has significant resource implications for the service provider. A recent campaign to eliminate gold farmers in the MMORPG entitled EVE Online discovered that 2% of the total player accounts were involved in gold farming. After those accounts were banned from the game, the game's developer observed a dramatic 30% reduction in computation load [17].

Leveling characters and grinding virtual goods using automated programs has very high profit margins. As a consequence, the RMT market has become so saturated with competing adversaries that they have resorted to spamming in-game communication channels to advertise cheap gold or avatar leveling services. Referred to as "*gold spamming*," this activity demonstrates the willingness of adversaries to apply additional automation techniques to the target application in order to selfishly increase their own benefit.

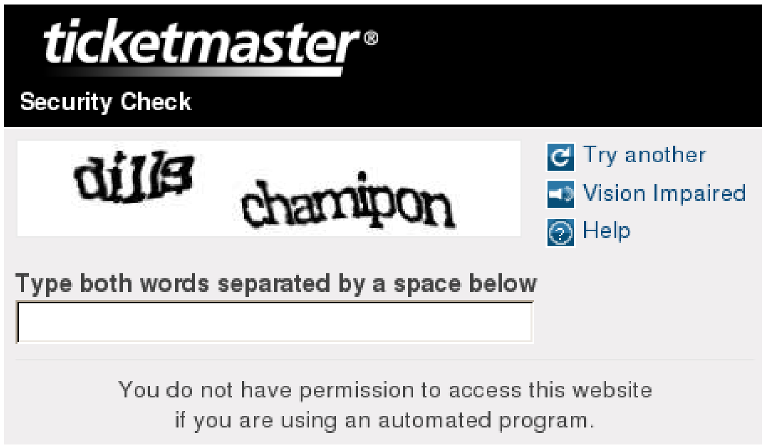
Furthermore, the unexpected and popular RMT market has so far gone unregulated. As a result, numerous illegal and more morally reprehensible behaviors have become rampant: activities such as theft of account information, money, or credit card numbers, and even money laundering. While some nations are considering virtual property ownership laws, addressing the root problem of automation would be a first step towards safeguarding consumers in that market.

Click Fraud. Website advertising often uses the pay-per-click model in which advertisers pay websites for each user that clicks on an ad and is redirected to the advertised website. While the monetary amount paid per user is small (roughly a few cents per click-through), adversaries have learned how automation can exploit this through "*click fraud*." There are two click fraud variations that an adversary may employ. The first approach directly benefits the adversary by using an automated script to simulate clicks on the ad as if it was a legitimate user. Through this approach, the adversary defrauds advertisers with ads placed on their website since they collect payment for user click-throughs that did not actually occur. The second approach is less direct in that an adversary will seek ads from their competitors. Using a specialized web crawler, the adversary targets competitor ads with fraudulent clicks in an effort to deplete the competitor's advertising budget before many legitimate users see the ads.

Timing-Based Automation

Timing-based automation performs online tasks at the exact moment that maximizes benefit for the adversary. This may involve coordinating many network machines or simply executing the protocol more precisely than humanly possible.

Auction Sniping. Adversaries use automation software called an "*auction sniper*" to monitor online timed auctions. The software places a winning bid at the last moment possible (often only seconds before the auction closes) giving other potential bidders no opportunity to outbid the adversary. This allows the adversary to purchase the item without engaging in a bidding war with other interested parties. While not illegal in every jurisdiction or explicitly against rules laid out by many online auction websites, automated auction sniping is frowned upon by many users because it cheats sellers out of higher sale prices and means that goods are not sold for what they are truly worth. This in turn negatively affects the profits of the service provider as a portion of their revenue is realized as percentage-based transaction fees for auction sales.


ticketmaster®

Security Check

dills champion

Try another

Vision Impaired

Help

Type both words separated by a space below

You do not have permission to access this website
if you are using an automated program.

Figure 1.4: Example TicketMaster CAPTCHA. This challenge was presented to a user attempting to purchase event tickets online.

Ticket Bots. When tickets for popular events such as Hannah Montana concerts go on sale online, they sell out almost instantly. Unfortunately, a significant number of the tickets are purchased by scalpers looking to turn a quick profit [110]. The scalpers use world-wide botnets to automate the navigation of the vendor's website and engage in transactions to purchase all tickets within minutes of them becoming available. Many ticket vendors fruitlessly employ CAPTCHAs (Completely Automated Public Turing tests to tell Computers and Humans Apart [117]) like the one shown in Figure 1.4 to deter automated adversaries. Scalpers resort to hybrid human-machine approaches that automates the entire transaction except for outsourcing the task of CAPTCHA solution to inexpensive foreign labor willing to solve them quickly and for less than a penny each [43].

While the scalpers turn a quick profit, legitimately interested concertgoers must pay a higher price for the tickets than they otherwise would have. This increased price clearly represents revenue that could have been captured by the original ticket vendor or performers, yet may have been forgone in an effort to make the event available to a wider audience. Consequently ticket scalpers are not simply "finding the true market value for the tickets" as they might claim, but instead are directly undermining the ticket vendor.

Precision-Based Automation

Precision-based automation performs application tasks with better precision than humans could possibly achieve unaided or reveals secret information that allows humans to interact with the application more precisely than intended by design.

Aim Bots. Some automation scripts called "*aim bots*" improve a player's aim within First Person Shooter (FPS) games. At the extreme end, these bots yield perfect aim where every shot the player takes is fatal for their opponents. This becomes frustrating (and easy to observe) for their honest opponents. Aim bots are considered cheating and their use is strictly against game rules. Rampant cheating, including the use of aim bots, can destroy the reputation of the game and dissuade potential players from purchasing the game.

Map & Wall Hacks. Some programs automatically reveal secret game data to the player which they definitely should not know. For example, "*map hacks*" are used by cheaters in Real Time Strategy (RTS) games to reveal the entire layout of the battlefield and precisely indicate the location and status of opponent units. In FPS games, "*wall hacks*" operate similarly by allowing the cheater to see their opponents through opaque walls. This is not only useful for strategic purposes, but also allows the cheater to precisely target opponents and subsequently kill them using powerful weapons that can shoot through walls. Like the aim bot nuisance, cheaters using map hacks or wall hacks have an unfair competitive advantage over their opponents, negatively impacting the game adoption and sales.

Poker Odds Bots. Automated programs called poker odds bots reveal the odds of winning and recommend a course of action to online poker players. These bots essentially reveal card statistics in a more straightforward fashion than those shown to their opponents. Using this automation, the adversary gains a precision advantage which helps them make fewer uncertain actions, thereby giving them an unfair advantage. This automation is used as the gateway automation to fully automated bots that play the game and collude to defraud human players.

Application & Adversarial Approach	Automation Type		
	Repetition	Timing	Precision
All Network Applications			
Port Scans	✓		
Network Worms	✓		
Denial-of-Service Attacks	✓		
Email & Instant Messaging			
Spam	✓		
World Wide Web			
Comment Spam	✓		
Online Commerce			
Click Fraud	✓		
Ticket Bots	✓	✓	
Auction Snipers		✓	
Multiplayer Video Games			
Leveling & Grinding Bots	✓		
Aim Bots			✓
Map & Wall Hacks			✓
Poker Odds Bots			✓

Table 1.1: Summary of adversarial automation examples. The examples are grouped by the various applications they affect. A few examples attack at the network protocol level, thus any network application may be vulnerable to such an automated attack. The remaining automation examples exploit features specific to the various applications.

Example Summary

Table 1.1 summarizes the automation examples discussed, grouped by the types of network applications which they affect. Most adversarial automation performs highly repetitive tasks and, similar to industrial automation, those tasks become immensely profitable at large scales. Techniques and methods for mitigating the discussed automation examples are presented through the course of this dissertation.

1.1.2 Root Causes of Adversarial Automation

The End-to-End Principle [97] is a central design principle of the Internet. In brief, the principle states that network and application protocol features are only justified at the lower layers of the network system if they optimize network-wide performance, all other complexity should be in the end hosts. This principle yields the lightweight and stateless Network Layer (Layer 3 in the OSI model [130]) which has allowed the Internet to become the dynamic patchwork of innovative applications that it is today. Unfortunately, the resulting properties of the Internet that make novel network applications possible without reengineering the network are the very same properties that facilitate adversarial automation. This section highlights these properties and argues that an application service provider cannot prevent adversarial automation, only detect and disincentivize it.

No Central Administrative Authority. Since the underlying Internet network architecture comprises an amalgamation of communication networks owned by organizations with radically different objectives (e.g., governments, companies, and academic institutions) there does not exist a single administrative authority over the entire Internet. Specifically, no one organization can guarantee that a host's entry point to the Internet (i.e., one of the many communication networks) will monitor and police the data packets that it sends. This is particularly true of packets destined for remote networks due to the phenomenon known as "*hot-potato routing*," a policy where each network forwards packets to the next nearest network, only performing the minimum amount of routing and processing required. Furthermore, no one organization can guarantee or mandate that all network hosts must run anti-virus software or do not run known-malicious applications. Consequently, *the Internet lacks strict host attribution* and it is practically impossible to stop adversarial behavior at its source. This is even more apparent when considering dynamic IP assignment, where an adversary may use a new network address each time they connect to the Internet.

Unpoliced Internet Architecture. The Internet was designed to extensibly support new and unpredicted applications. Since applications operate with different goals, behavior that may be desirable for one application may be intolerable for another application. Thus, no global definition or consensus exists to dictate exactly what constitutes adversarial behavior, automated or otherwise. Insofar as an application's clients do not interrupt or impede other applications, defining undesired behavior is left up to the individual service providers. As a result of the open architecture and the lack of a central administrative authority, *anyone can send any data to any recipient claiming to be any sender* and regarding the inbound data *it is largely up to the recipients to deem what is inappropriate.*

Uncertainty About Host Automation. Even if communication networks had the incentive or desire to monitor and filter all communication from hosts connected to it, since the distinction between wanted and unwanted network packets is defined by an application service provider at a distant network location it would be impractical to query for each packet. This means that intermediate network devices generally cannot know with certainty whether data packets will be accepted or discarded, or even if the packet results from adversarial automation. To exacerbate the problem, adversaries do not confess when caught, but instead attack new victims or adapt their adversarial approach. Furthermore, a network host may not be malicious in all contexts and they might not even know that they are malicious in some contexts (consider the case of a user trying to browse a website while they are unknowingly also running zombie software attacking that same website). Finally, evidence of prior malicious behavior and correct classification as an adversary may only be temporarily valid because the network host could reform (consider the above case after the user discovers and removes the zombie software running on the machine).

For these three main reasons, adversarial automation cannot be prevented at the adversary's network ingress. Instead, each application must learn to detect and disincentivize adversarial behavior as it applies to that specific application.

1.2 RESEARCH CHALLENGES

This dissertation seeks to address adversarial automation. The task of mitigating adversarial automation presents the following three research challenges:

1.2.1 What methods can be used to detect automated behavior?

Quickly and accurately detecting automated behavior is the fundamental challenge to thwarting automated adversaries. Since the adversaries outperform humans, there must exist some characteristic of their operation that allows them to do so. Distinguishing this characteristic and determining how to detect it is the first challenge to towards thwarting adversarial automation of that application.

1.2.2 How can detectors be combined to best identify adversaries?

There may be more than one method to detect the adversary, and each of those detection methods may be individually inconclusive. For any given application, the service provider may have a number of individually inconclusive detectors at their disposal. The next challenge involves how to best combine them to create a single metric with a more conclusive result. Specifically, what is sought is a single value that ranges from "not adversarial" to "adversarial" (i.e., a probability $\in [0.0,\ 1.0]$) to describe each client. This prediction value must be calculable in an accurate and efficient manner.

1.2.3 How can adversarial automation best be disincentivized?

Provided prediction values regarding adversarial automation, the service provider may leverage this knowledge to discourage such behaviors. Specifically, the challenge is how to disincentivize automated behavior so that adversaries behave indistinguishably from and thus have no advantage over non-automated human peers. Of particular interest are mitigation approaches that gracefully handle cases where the prediction value is gaining certainty but is not yet conclusive.

1.3 DISSERTATION OUTLINE

This dissertation explores the outlined challenges using research on two popular networked applications where the service provider has varying access to the client: multiplayer online video games and web-based services. In multiplayer online video games, the game's service provider (hereon referred to as the "*game developer*") implements and releases the only client software that is authorized to interact with the server software. This means the game developer has firm control over what all legitimate client operation looks like.

In contrast, web-based services operate over the HyperText Transfer Protocol (HTTP) [40] which simply dictates request and response formats. Any web browser that adheres to the HTTP protocol can therefore access the service. Behavior outside of the protocol (e.g., how frequently to request data or what to do with it) is browser-dependent and varies amongst browser implementations and browser-addons. As such, a web service provider has little control over exactly what legitimate client operation looks like.

This dissertation is divided into chapters focusing on each of the three research challenges. Chapter 2 investigates the detection of adversarial automation that manifests itself as cheating in online multiplayer video games. In this application, banning adversary accounts is a proven, effective, automation disincentive since the adversary loses their software purchase as well as the subscription fees and time invested up to that point. However, existing detection approaches are error-prone, inefficient, non-automated, and expensive to maintain. Exploiting the game developer's direct access to the game client, we explore a novel approach to video game cheat detection: anomaly-based detection. In this approach, the application automatically learns how the client operates on different machines through partial client emulation. Using continued, random, and remote audits, the server validates client execution and flags unexpected execution as possible cheat behavior. The evaluation of this research focuses on detection accuracy and integrity.

Chapter 3 investigates the combination of individual detectors to create a predictor likely to produce a conclusive result. Specifically, the chapter looks at reducing a set of heterogeneous detectors within multiplayer online video games to establish a metric for identifying malicious, automated players. This research is realized as a novel reputation system for multiplayer online games. Treating a player's peers as detectors (i.e., each peer's observations provide clues regarding the maliciousness of other peers), the disincentive simply follows: players will avoid known malicious peers, peers who will at the same time garner unwanted scrutiny from the game developer. The reputation system is designed to minimize need for direct access to the game client to enforce trustworthy operation. The evaluation of this research explores both its efficiency and collusion resistance, properties necessary for combining largely untrusted detectors. Experiment results indicate that this approach may be applicable to the web-based applications, like those discussed in the next chapter.

Chapter 4 investigates the use of proof-of-work to disincentivize automated adversaries of web-based applications. In this setting, service providers have no access to monitor the operation of client software so they must rely upon detectors completely external to the client software. Given an established accurate predictor regarding the probability a client is automated, we show that service providers can wield proof-of-work challenges to quickly thwart adversaries. The proof-of-work paradigm requires clients to solve computational puzzles which are scaled in difficulty proportionally to the adversarial metric. More adversarial clients are given very difficult puzzles to solve, while less adversarial clients are given trivial puzzles to solve. This approach disincentivizes automated adversarial behaviors by computationally taxing them. The evaluation of this research explores the effectiveness of the approach in isolating automated adversaries.

Chapter 5 reviews the contributions of this dissertation and summarizes some of the key findings of this research.

1.3.1 Thesis Statement

The research discussed throughout this dissertation is guided by the following thesis statement:

> *There exist methods to detect automated behaviors with which an application's service provider can identify and disincentivize automated adversaries.*

Chapter 2

DETECTION METHODS

2.1 INTRODUCTION

The fundamental challenge to thwarting automated adversaries is to quickly and accurately detect their automated behavior. Towards this goal, this chapter explores the problem of cheating in multiplayer online games. Multiplayer online games serve as the ideal application for research focused on developing methods to detect adversarial automation. In this application, a well established and proven disincentive already exists: adversaries are banned, forfeiting their software purchase as well as the subscription fees and time they have invested up to that point. Given such an effective disincentive, the only challenge to defeating automation plaguing multiplayer online games is accurate detection.

Unfortunately, existing cheat detection approaches [31, 76, 115] are error prone, inefficient, and expensive to maintain. These approaches require the game developer to identify, obtain, catalog, and then continuously search for cheat software operating on the client machine. In addition to the above shortcomings, those approaches are completely incapable of detecting cheats which the developer cannot obtain and catalog.

Since the game developer has a thorough understanding of the game client software, it follows that the developer could instead focus detection efforts on abnormal client execution. In this chapter, we show that such an approach is both more efficient and accurate, and able to detect cheats which the developer cannot easily catalogue. Throughout the chapter, we refer to the adversary as a "cheater" and the service provider as a "game developer" or "developer."

2.2 THE CHEATING PROBLEM

Multiplayer online games simulate complex virtual environments. Due to the expense of server computation resources as well as the player's sensitivity to network latency, games are designed to offload as much computation as possible to the game client. That client software is expected to run accurately and keep secret game state hidden from the player. Cheat software violates this trust by altering the simulation locally to give the cheater an unfair advantage (e.g., perfect aim, X-ray vision, or teleportation).

While the motives vary from game to game, cheating has become widespread. Many underground communities write and sell cheats ranging from automated bots that treasure hunt virtual items, level up characters, and attain ranks for cheaters unwilling to play the game, to cheats that provide perfect aim and reveal secret knowledge for cheaters unable to win unaided. Studies have shown that such automated bots behave in a repetitious fashion consuming a disproportionate amount of the costly server computation resources [17]. Furthermore, legitimate players are frustrated by cheaters to the point where they seek other games that are more robust to cheating. Since losing existing and potential paying players directly impacts revenue, developers have a second powerful incentive to thwart automated cheating in their games.

2.2.1 A Distinct Security Problem

Cheaters have one clear advantage over developers – they control the machines on which they cheat. This means that the cheater can grant the cheat software all the necessary privileges and may run it before anti-cheat software is ever loaded. State-of-the-art cheats conceal their presence by modifying the operating system, disabling or spoofing anti-cheat software, and even cloaking their code just before routine anti-cheat software runs. Fortunately, *cheats are a weak threat compared to other security problems* like rootkits, botnets, and worms.

First, most cheats embed themselves within the game process to easily access game data and functionality and this drastically limits their ability to conceal their presence, let alone remain hidden indefinitely. For these reasons, game developers can focus their efforts on a small search space yet detect a majority of cheaters.

Second, at the level of individual cheaters, the problem is not urgent. The problem only becomes catastrophic for game developers if cheating becomes widespread or is believed by most players to be largely unaddressed. As a result, there is no need for rigorous cheat prevention or immediate containment; sensitive information (such as passwords and personal information) is not being stolen and the machine is not being used to attack other network hosts.

Third, cheating damage is easily undone once discovered. By confiscating a cheater's ill-gotten gains and disabling their account, the cheater can no longer affect legitimate gameplay. The full extent of cheating damage is easily determined since game developers keep transactional logs regarding persistent virtual world state (character levels, wealth, and win-loss records) so cleanup is comprehensive. Since disabling the cheater's account annuls all invested time and confiscates both the software purchase and paid subscription fees, cleanup directly punishes the cheater. To an automation-based adversary of networked applications, this is an uncommon, severe consequence for being detected.

Finally, due to the long-term connected nature of online games, the server has many lengthy client interactions during which the cheater need only be detected once to halt their disruption. Reversing an adversary's traditional advantage, *the cheater must anticipate and guard against every detection technique to succeed while the developer need only detect a single unauthorized change to thwart them.* With no need for urgency, being able to eventually detect the cheater is sufficient to address the cheating problem, which is supported by the widespread use of cheat detection. The next section discusses the various methods which cheaters employ to modify and automate the game client.

2.2.2 Cheating Methods

To further understand the cheating problem, it is important to survey the methods that are currently being used by adversaries to cheat[1]. At present, most computer-based video games (and hence most cheats) are implemented for the Windows operating system. The cheat techniques described in this section are illustrated using Windows functions, however, the methods are general and apply to other operating systems. In practice, the methods presented are typically used in conjunction with each other to implement cheat software.

Authorized, Automated Data Read

This method automates the collection of information that is presented and available to the cheater. Such cheats typically use legitimate APIs to learn game data without directly interacting with the game process. Because the APIs serve legitimate purposes (mostly for accessibility), their abuse is difficult to detect.

- Using the Graphics Device Interface (specifically `BitBlt()` and `GetPixel()`) to dump pixel information from the screen and discern game state.

This method is used by bots that automate actions, like FishBuddy for World of Warcraft which automates the casting, hooking, and storing of virtual fish.

Unauthorized Data Read

This method, also known as "*information exposure*," accesses hidden game data that should not be revealed to the player. Specific techniques include:

- Using a packet sniffer to extract game data from unencrypted network traffic.
- Using APIs (such as `ReadProcessMemory()`) to remotely read game memory.
- Using a thread within the game process to access game data.

[1]This research appeared as a survey in NetGames 2008 [38].

Cheats that employ this method include map-hacks that reveal the location of enemy units behind the "*fog of war*" in Warcraft III, wall-hacks that reveal the exact locations of enemy players behind walls in Counter-Strike, and Kick-Ass Map that reveals mob locations beyond the player's view in World of Warcraft.

Unauthorized Data Write

One of the simplest methods to alter game behavior is to directly modify data within the game process. By changing the data that the game uses, cheaters can gain abilities that their opponents cannot. Techniques for performing unauthorized data writes include:

- Using APIs (like `WriteProcessMemory()`) to remotely rewrite game data.
- Using a thread within the game process to modify game data.

Cheats that use *static* data writes include modifying the gravity constant so that cheaters can climb walls in World of Warcraft and modifying memory-mapped wall textures to make them transparent in Counter-Strike. While these examples are extremely easy to detect since they modify data to invalid values for the duration of the cheat, other cheats *dynamically* toggle data between valid values in an illegitimate way. For example, a cheat for Battlefield 2 continuously changes the team the cheater is on in order to trick the game client into revealing enemy locations via the radar.

Cheat Engine [18] is a tool that uses this method to locate and dynamically modify game data. In particular, it modifies the cheater's coordinates and direction to implement teleport cheats, Z-hack cheats (where the cheater is kept a fixed distance from the opponent during battle), and direction cheats (where the cheater is made to face the opponent at all times). Cheat Engine is also used to illegally modify a character's attributes such as experience or health level.

Code Injection

Many cheats change the operation of the game by altering the game code or running their own code within the game process. To achieve this, cheaters can inject their code into the process during the loading of the game or from an external process while the game is running. There are a myriad of ways that code can be injected including:

- Using `WriteProcessMemory()` to *hotpatch* (i.e., overwrite) game code to implement new functionality or make particular game operations always or never happen. For example, one can disable flash grenade blinding effects in Counter-Strike by changing jump instructions that call the flash effect into `NOP`s. Hotpatching is feasible if the changes can be done within the size constraints of the original function.

- Using `WriteProcessMemory()` to write to *code-caves*, pockets of allocated but unused memory between existing game functions. The use of code-caves provides some stealth if the anti-cheat only scans the game's original code locations.

- Allocating new memory using `VirtualAllocEx()` and writing to it using `WriteProcessMemory()`. This is not stealthy, but it facilitates injecting an arbitrary amount of cheat code into the game process.

- Loading a Dynamic Link Library (DLL) containing the cheat payload by either using `LoadLibrary()`, by hooking `LoadLibrary()` as it is called for other game DLLs, or by modifying registry entries such as `AppInit_DLL` to have the payload DLL loaded automatically with the game. Tools that support this technique include Winject and INJLIB.

Cheats that inject code are prevalent and include aiming automation tools like BlackOmega [11] for Battlefield 2, Ecstatic [30] for Counter-Strike, and HL2 Hook [48] for Half-Life 2.

Thread Manipulation

Once cheat code is injected into the game, it must be executed. The most common techniques involve manipulating threads within the game process by:

- Using *detours* (or *trampolines*) to temporarily hijack an existing game thread [53]. The detour redirects game function calls to injected cheat code by hotpatching a handful of bytes at the beginning of game functions. The overwritten bytes include a jump instruction that points to previously injected cheat code. Depending on the intention, some detours will execute part or all of the game function after executing the cheat code.

- Injecting a new thread via `CreateRemoteThread()` to execute cheat code alongside the game's normally operating threads.

Direct Function Calls

Many cheats change the operation of the game by directly calling operating system or game functions as needed for the desired behavior. This is especially true for automation bots which take input from authorized or unauthorized reads, make decisions, and then directly call game code to take action. Techniques include:

- Using I/O APIs (`keybd_event()` or `mouse_event()`) to directly generate game input signals which would otherwise be sent by the keyboard/mouse interrupt handler.

- Using an injected or hijacked thread to directly call functions from within the game process.

The use of direct function calls is prevalent in bots and is a key component of Hoglund's World of Warcraft Implant bot [52].

Function Pointer Hooks

Similar to detours, cheats may modify function pointers in the game or operating system in order to redirect execution to cheat code. Function pointers are prevalent in any running process whether it is within the game's code, the libraries it uses, or the operating system. By overwriting function pointers in order to execute injected cheat code rather than modifying the functions themselves, this method passes integrity checks that only examine the game's code. Techniques include:

- Return address hooks that modify pointers stored in the stack so that functions return to injected code rather than their caller. This technique is the basis for return-to-libc attacks [106].

- Overwriting function pointers in game code that implement run-time binding of operations or jump table implementations of switch statements in C/C++.

- Import Address Table (IAT) hooks that replace the game process' table of function pointers for functions exported by loaded DLLs such as DirectX, DirectInput, WinSock, or kernel DLLs.

- Hooks that replace entries in system tables such as the Interrupt Descriptor Table (IDT), the System Service Dispatch Table (SSDT), or the I/O Request Packet (IRP) Function Table.

- Structured Exception Handler (SEH) hooks that replace exception handler pointers on the stack with addresses for cheat code.

- Using the Windows API to hook message handlers across all running processes via `SetWindowsHookEx()`.

Function pointer hooks are prevalent in bots across all game genres such as HL2Hook/CSHook and speed hacks implemented using Cheat Engine [18].

External Processes

In this method, the cheater employs an external process that modifies or tampers with the game process. Instances of this method include:

- Using previously described operating system APIs to access and tamper with the game process such as `ReadProcessMemory()`, `WriteProcessMemory()`, `CreateRemoteThread()`, and `VirtualAllocEx()`.

- Sending Windows messages such as mouse and keyboard events to the game process via APIs like `SendMessage()`. For example, in first-person shooters, automated events are sent in order to perform recoil suppression (e.g., `WM_MOUSEMOVE`).

- Using `DebugActiveProcess()` to attach to the game process as a debugger and completely control its execution. This technique can be used to either change the game code itself or to hijack game process threads to load cheat code from libraries.

Examples include recoil suppression cheats in first-person shooters that inject mouse events from a remote process and any cheats that use the debugger interface to gain control of the game process.

File Replacement

In this method, the cheater modifies the game binary, the game's data files, the libraries the game uses, or kernel modules on disk before they are loaded. While this method has been used in the past, file integrity checks by anti-cheat software have rendered this less popular. Specific examples include wall hacks in first-person shooters that replace game texture files with transparent alternatives.

Hardware Facilities

Due to the difficulty in correctly measuring hardware state from software, some cheats like Cheat Engine and Hoglund's Governor exploit hardware features. The following methods are specific to Intel processors but are applicable to other processors.

- Cheats may tamper with the Interrupt Descriptor Table Register (IDTR) of the CPU which stores a pointer to where the Interrupt Descriptor Table (IDT) resides. This leaves the original table intact yet points to a completely different table containing pointers to cheat code.

- Processors typically support hardware debug registers that stop execution and cause an exception to occur whenever particular code locations are reached or when particular memory locations are accessed. By using this facility, cheat software can hijack game execution without explicitly injecting debugger interrupt instructions into the original game code.

- To access game memory, cheats can tamper with the memory management subsystem of a processor including its control and segment registers. For example, tampering with IA-32 control registers (CR0-CR3) allows a cheat to modify read-only pages or hide memory pages where cheat code resides.

- Model Specific Registers (MSRs) can also be used to tamper with the game and operating system in a variety of ways. One specific example is the SYSENTER_EIP_MSR register on IA-32 CPUs that holds the address of the "fast" system call function. By modifying this register, a cheat can hook essential system calls underneath a game.

- Another stealthy method is to have the game run virtualized and implement the cheat into the virtual machine or hypervisor. With hardware support for virtualization, such an approach can make detection extremely difficult [131].

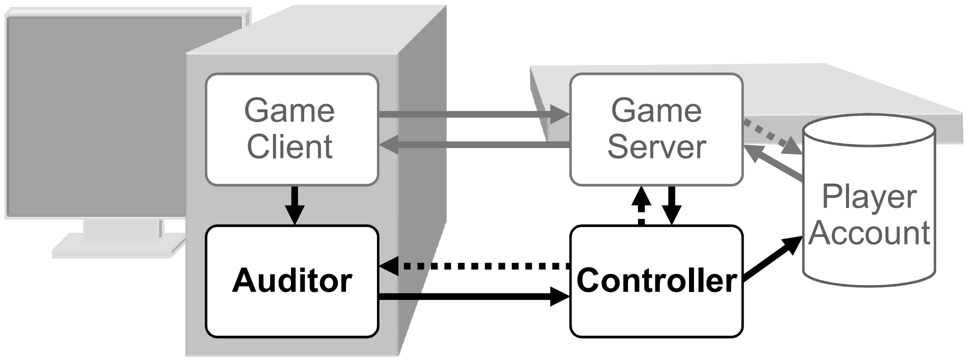


Figure 2.1: Addition of Fides to the client-server game architecture. Interaction between the game and the Fides Auditor and Controller is shown: dashed arrows represent request traffic while solid ones represent data flow. Any detected cheating is recorded to the player account so that the developer can respond appropriately.

2.3 THE FIDES APPROACH

Our approach for cheat detection is the Fides approach[2,3] which, to our knowledge, is *the first anomaly-based software-integrity detection approach* in the literature that is useable on commercial-off-the-shelf online games. The Fides system (shown in Figure 2.1) dovetails with prevalent client-server game architectures to minimize the modifications necessary while efficiently detecting cheats that directly modify game clients. Cheats completely external to the game process (e.g., cheating through collusion or mechanical "*roboting*") is not addressed by this approach.

Fides is a generalized cheat detection approach that is game-independent; the approach works across games, game genres, operating systems, and hardware architectures. Fides adapts anomaly-based application integrity research to the game domain, thereby avoiding the detractions of existing approaches. Specifically, Fides does not require human intensive maintenance and can quickly detect cheats without first knowing their operational minutia.

[2]Fides was the goddess of trust in Roman mythology.
[3]This research appeared in a paper at ACM CCS 2009 [67].

Anomaly-Based Cheat Detection. There are two possible cheat detection approaches: signature-based detection and anomaly-based detection. *Signature-based* detection learns what cheats look like and actively searches for these patterns while the game is running – scanning not only the game but every active process on the system. Anti-spyware advocates observe that this maintenance-heavy methodology is the current choice of game developers [51], likely because it can be implemented after game release.

In contrast, *anomaly-based* detection learns how the legitimate unmodified game client operates and periodically audits it, searching for unexpected deviations that indicate the presence of cheat software. This detection approach only inspects the client process which is feasible since most cheats embed themselves in the client process rather than manifest as an external process. As a result the search space is well bounded and other processes running concurrently affect neither the speed nor the accuracy of this methodology.

Restricting the search space to the game process yields security benefits too. Foremost there is *no risk of privacy breach* [76]; sensitive data that exists in unrelated software processes is not read and cannot be leaked. Additionally, adversaries cannot exploit the scanning of unfamiliar processes by injecting false positives. This avoids attacks similar to the case in which a message containing the binary pattern of a cheat signature was broadcast to an IRC channel belonging to a prominent game clan, was falsely detected by signature-based detection (Punkbuster [31]), and caused 300 legitimate players to be incorrectly banned [85].

Anomaly-based detection is cheat-independent and requires fewer developer resources (i.e., manpower and storage) to maintain since the game is readily accessible for study by its own developers. *Cheats do not need to be captured and studied to create signatures as only knowledge about the game is used.* The perpetually increasing collection of cheats, cheat variations, and polymorphic cheats does not inflate the knowledgebase which only changes when the game software is updated (i.e., patched).

Furthermore, anomaly-based detection is *not reactionary.* Signature-based detection relies on manually finding cheats and cataloging their signatures which causes a lag between when a cheat is first used and when its signature can first be detected. This lag is often artificially increased by the developer to avoid tipping off cheaters as to when and how they were caught. Legitimate players, however, may perceive that personally observed cheating behavior goes unpunished.

For anomaly-based detection to work properly, all host variation that affects the game client (i.e., library versions and their memory locations) must be accommodated to ensure that all legitimate game operation is recognized and not misclassified as cheat behavior. With such accommodations, anomaly-based detection is advantageous in terms of efficiency, maintenance costs, and accuracy.

Continued Random Remote Measurement. A novel Fides feature is that it performs continued random remote measurements of the game client during gameplay. The system comprises a simple client-side Auditor that is directed by a robust server-side Controller. The Auditor supports a selection of parameterized functions to measure client state and return the results to the Controller. The system complexity is located in the Controller which dictates the audit strategy (i.e., what gets measured as well as when) and validates all measurements taken. This continued sampling approach reduces client-side overhead.

Minimizing the Auditor's complexity facilitates strengthening it against attack. The use of techniques like execution entanglement, rapid polymorphism, and lightweight tamper-resistant coprocessors can provide stronger assurances about the Auditor's integrity and measurement accuracy.

Additionally, employing a partially randomized strategy and placing it in the Controller avoids telegraphing audits and allows the developer to change strategy surreptitiously. The "*fear of the unknown*," specifically not knowing when cheat techniques believed to be detection-proof become obsolete, has been an effective deterrent for would-be cheaters [52]. Continued random measurements will eventually detect a persistent cheater due to the always-on nature of the game.

Partial Client Emulation. Another novel feature of the Fides system is that the Controller partially emulates the game client to accommodate any client system variation that would affect measurement validation. The Controller includes routines (explained in detail later) for mapping the client's virtual memory and learning its execution patterns during player logon. This compiled knowledge is relevant for the duration of gameplay and is used to validate each Auditor measurement; deviation indicates the presence of cheat software.

Static game data and code are learned by parsing the game client's binary and library files, and rebasing them to match the layout on the client machine. The Auditor or client (via the server) must relate those pertinent library details to the Controller at client logon. Dynamic data is semantically identified using source code and debugger files so that the Controller may query the server to corroborate those values. For this approach to validate dynamic data values, the server design must be adapted to respond to local queries about player state.

Client execution patterns are learned by disassembling the mapped code sections and creating a graph of legitimate execution describing the range of and relationship between all client functions. This knowledge allows the Controller to know what code should be executing given an instruction pointer, and whether specific code locations represent `CALL` instructions linking two legitimate functions.

To better learn commercial-off-the-shelf games (which may be obfuscated to prevent reverse engineering and may not supply its source code) that run in common machine environments, the emulator includes tools for sampling and profiling legitimate client execution in a secure server-side environment. These tools can provide client understanding where static learning routines cannot.

While the Auditor and Controller in Figure 2.1 are shown separate from the game components, they could be implemented within the game client and game server. Locating the Auditor in the client enables polymorphic patching discussed in detail in Section 2.4.3 on page 60. Locating the Controller in the server streamlines the validation of dynamic data.

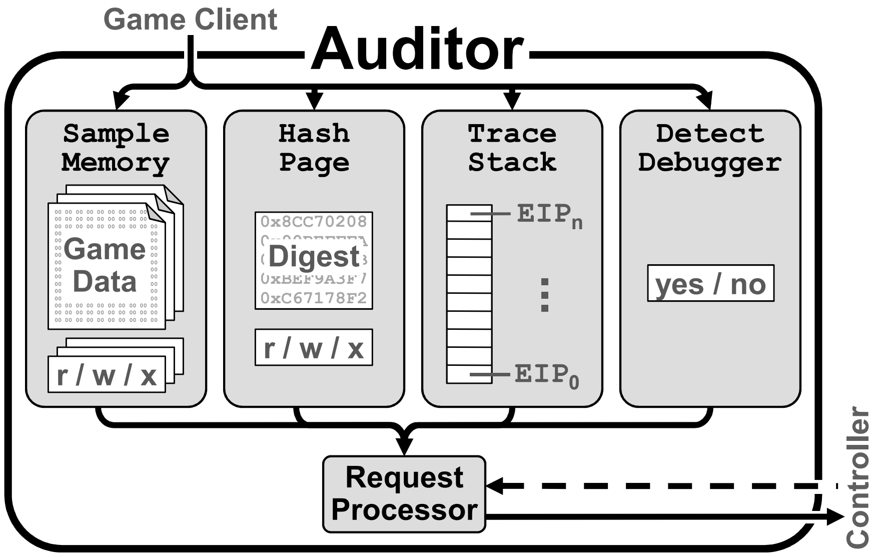


Figure 2.2: Internal structure of the Fides Auditor. The Auditor responds to requests from the Controller and executes one of the four supported measurement routines that collect client state directly from the game client process.

2.3.1 The Auditor

The Fides approach predominantly involves server-side software since it is the game developer who is interested in the detection results. However, to audit the game client requires the addition of a client-side component. That component is the Auditor (shown in Figure 2.2) which accepts instructions from the server-side Controller, performs the requested measurements, and returns the results.

The Auditor is intentionally kept very simple for three reasons. First, game-specific features are kept out of its design so that it may be used by any number of game developers to audit their games, or even by other service providers to audit their network applications. Second, its simplicity allows one to assure its correct operation, reducing the likelihood that incorrect operation will later be discovered and require correction. Third, its simplicity facilitates the use of expensive cryptographic integrity-assuring operations like entanglement or tamper-resistant coprocessors to verify its integrity.

Auditor Measurements

The Auditor currently supports four measurement routines which are described in detail below. Table 2.1 shows the cheat methods best detected by each of the measurement routines.

Sample Memory. This routine returns the contents and read-write-execute permission flags of a specified range of the game client's virtual memory. This routine is best used to detect cheats that modify dynamic data. Since the specified memory range is expected to contain dynamic data, every game client thread must be briefly suspended to get a quiescent reading.

Hash Page. This routine uses a cryptographic hash function (e.g., SHA1 [84]) to hash the specified memory page and return the digest along with the page's permission flags, facilitating efficient detection of cheats targeting static data and code. If unspecified, the page currently executed by a randomly selected thread is hashed. This measurement does not require suspending any game client threads because the target memory page should be static. Whenever possible, use of this routine is preferable over memory sampling because it saves network bandwidth and minimizes the server-side state necessary to validate the results.

Trace Stack. This routine suspends one client thread, chosen at random if unspecified, and obtains the current instruction pointer (`EIP`) and stack pointer (`EBP`) which are the representative registers of the stack frame (i.e., client function executed). Recording all encountered instruction pointers, the routine recursively descends the stack, obtaining each previous stack frame by dereferencing the stack pointer of the current frame, until it reaches the frame corresponding to the thread entry point (i.e., `EBP = NULL`) at which point the measurement is done and the client thread may be resumed. The n recorded instruction pointers list the function calls that the client thread used to get from the entry point to the current point of execution and are returned, facilitating the detection of any cheats that hijack game execution (redirecting it to unrecognized locations or between unrelated functions).

Measurement	Cheat Methods Detected
Sample Memory	Dynamic Data Manipulation
Hash Page	Static Data Manipulation Code Manipulation Code & DLL Injection File Replacement Memory Management Manipulation
Trace Stack	Thread Injection Thread Hijacking Function Pointer Hooking Direct Function Calls Register Manipulation
Detect Debugger	Software Debugging Hardware Debugging

Table 2.1: Cheat methods best detected by Auditor measurements. In general, page hashing and stack tracing are the preferred measurements. Page hashing can detect all the static cheat methods that memory sampling can, yet is much more efficient in terms of network bandwidth. Stack tracing can detect all the cheat methods that alter game execution.

Detect Debugger. This routine detects whether the game client is being manipulated by a cheat attached to it as a debugger, and returns the Boolean result to the Controller. If the game client has a debugger attached, the corresponding flag in its Process Execution Block (PEB) should be set to true (although the first thing such cheats do is set the flag to false) which can be tested via the `CheckRemoteDebuggerPresent()` function. Alternatively, the routine attaches to the game client as a debugger using `DebugActiveProcess()` which always fails if another debugger is already present. Attaching as debugger is relatively expensive (roughly $9ms$) and should only be done if the first test is negative. This measurement routine detects all software debugger cheats and most hardware debugger cheats.

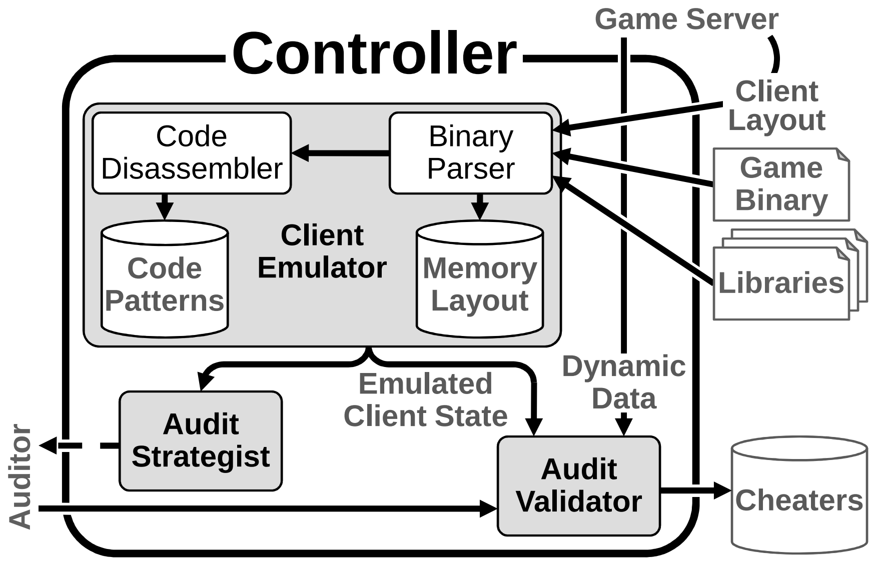


Figure 2.3: Internal structure of the Fides Controller. The Controller hosts all of the system complexity and dictates what should be measured by the Auditor. Arrows show the data flow between external data sources and the Client Emulator, Audit Strategist, and Audit Validator components.

2.3.2 The Controller

The Controller (shown in Figure 2.3) hosts the complexity of the Fides system, the bulk of which is the Client Emulator. The Client Emulator learns the game client properties that remain constant (i.e., static data, code, and function relationships) and identifies dynamic data sections. The Audit Strategist uses the compiled knowledge to create an audit strategy (possibly tailored to the game) and orchestrates the strategy during gameplay. The Audit Validator uses emulated state to directly verify audits of static game client data and code. To verify audits of dynamic game client state, the semantics of the sampled memory are extracted from the emulated state so that the actual values can be validated by what the server dictates they should be at that time. This accommodates most susceptible dynamic variables since they normally change slowly (i.e., on the order of seconds or minutes) and smoothly (e.g., character movement, health, and ammunition).

Client Emulator

The hardware and software of the machine on which the client runs varies from player to player, and even changes between play sessions for individuals who play on different machines. Such variation affects the memory layout and contents of the client process. For example, the libraries loaded by the game differ between operating systems and system versions. Additionally, memory layout differs between executions on systems that randomly rebase libraries via Address Space Layout Randomization (ASLR) [101].

The Client Emulator accommodates for host variation by thoroughly mapping the code and static data and identifying the dynamic data for each client at logon. Pertinent clients layout details (i.e., the name, version, and base location of every library loaded) must be communicated to the Controller either by the Auditor or by the client via the server. The emulator owns a copy of all known and authorized legitimate libraries so no files need to be transferred from the client machine to the Controller. If the client (rather than the Auditor) is tasked to exchange its layout details, it is behooved to accurately report those details otherwise inconsistencies will be instantly detected and classified as cheating.

Game clients may run on operating systems without ASLR (or may be compiled with ASLR disabled – a decision the developer may elect) and share common library content and layout. Specifically, core system libraries (e.g., `KERNEL32.DLL` and `NTDLL.DLL`) often contain thousands of small functions that comprise much of any client application's complexity. The emulator can leverage commonality to reuse significant portions of the emulated state between clients, and learn only truly variable client structure at logon.

The Client Emulator uses two routines for learning client structure: a Binary Parser and a Code Disassembler. The emulator also contains an Execution Sampler and an Execution Profiler to better understand if the client application can be run in a cheat-free virtual machine using a consistent layout.

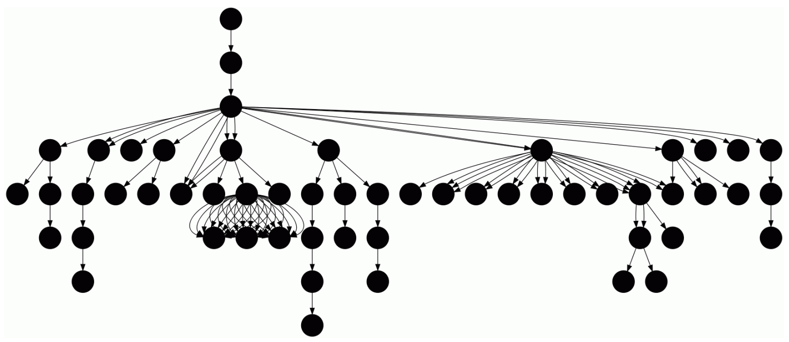


Figure 2.4: Example execution call graph. This partial graph shows the function dependencies originating from the initialization routine of our Homebrew game. Nodes represent functions while directed edges represent specific memory locations (i.e., `CALL` instructions) that relate two functions. Often a second function is called several times, hence the multiple edges connecting any two functions.

Binary Parser. The first learning routine is the Binary Parser which maps the client virtual memory from the executable, all linked libraries, and data files. Using the client-provided layout information, the routine rebases the libraries to know the location and properties (i.e., read-write-execute permissions) of every memory section in the client. Function pointers (e.g., IAT pointers) are corrected so that page hashes may be generated for static data and code sections, reducing emulator state for that data to a small digest per page. If available, the emulator uses source code and debugger database files to learn the semantics of variables in dynamic data sections so the Controller may intelligently query the server for the correct values of a memory sample.

Code Disassembler. The second learning routine is the Code Disassembler which uses the code sections mapped by the Binary Parser to learn the range of and relationship between every function in the game client. The routine starts at the entry point of the executable and traverses the code, observing the memory range of each function and the location of instructions that relate functions, creating an execution graph (like the one shown in Figure 2.4) similar to the "callgraph" model [119]. This knowledge is used when validating stack trace audits.

Execution Sampler. The above routines are sufficient to learn client applications for games designed to work well with Fides, however, to better learn commercial-off-the-shelf games (which may be obfuscated and for which one may not have source code) running in non-ASLR environments the emulator includes an Execution Sampler. The tool executes the client application in a cheat-free environment, like that which the actual client should be running in. The sampler then exhaustively hashes all the non-writeable pages of the game, furnishing the knowledge to the emulator so it may be used to validate page hash audits of the actual client.

Execution Profiler. The emulator also includes an Execution Profiler to learn the instruction range of and the relationship between client functions (specifically observing indirect function calls), reinforcing the execution graph used to validate stack trace audits. This tool executes the client application in a secure, cheat-free environment, attaches to it as a debugger, and uses hardware debugging routines (i.e., register manipulation and single-step interrupts) to step through its execution. The profiler records details about instruction counts, code timing, and function execution frequencies. These execution patterns may also be used to improve audit strategies.

Audit Strategist

The Audit Strategist orchestrates the detection strategy, dictating the ordering, timing, and details of every audit request. Locating all the strategy in the Controller avoids telegraphing measurements, allows the developer to change strategy surreptitiously, and minimizes Auditor complexity. The strategy may be game-specific using developer intuition regarding data or code likely to be attacked, however, a good strategy must employ some randomization to prevent cheaters from predicting audits and developing a successful cloaking routine. As cheaters will be audited countless times while they are online, a strategy with randomness will, with good probability, eventually catch them.

Audit Validator

After measurements have been returned to the Controller, the Audit Validator validates them through different means according to their type as follows.

Hash Page. These audits measure static data, so they should remain unchanged. The validator does a simple binary comparison of the audited digest against the stored digest. Any bitwise difference means that the page has been modified, an indication of tampering.

Sample Memory. Although an easy audit to perform, validating sampled dynamic data is difficult and involves server corroboration. The validator must determine the semantics of the data (i.e., which variable it represents) and query the server regarding what the proper value or range of values should be. Consider the cheat which toggles the team variable to reveal enemy locations via the radar. To validate that variable, the server must share which team the player is on. This is straightforward because there are few valid values and the variable changes infrequently. Next, consider teleport cheats that adjust the character's in-game coordinates. In this case, there are innumerable valid values, the value changes frequently, and due to their use of dead-reckoning techniques, clients will be strides ahead of what the server has recorded. Consequently, the server must provide a range of locations where the player could legitimately be according to game rules.

Stack Trace. Validating a stack audit involves walking the execution graph according to the sequential list of instruction pointers returned by the audit. Client execution has been hijacked if any instruction pointer calls an unknown location in memory (i.e., referencing a nonexistent node) or represents an unrecognized function call (i.e., referencing a nonexistent edge). Indirect function calls are handled by validating every direct call up to it and immediately hash auditing the functions after it ensuring they have not been detoured.

Detect Debugger. The simplest audit to validate; a detect debugger audit should be false otherwise the client is being manipulated by a debugger.

2.4 EVALUATION

To demonstrate the utility and easy deployment of the Fides approach, a software prototype was implemented consisting of two applications (corresponding to the Auditor and Controller) that communicate through a TCP socket. The prototype is game-agnostic and has been tested successfully on a number of games.

Auditor Details. The Auditor employs functions of the Windows debugging API to conveniently read client virtual memory using `ReadProcessMemory()` and access the client's registers using `GetThreadContext()`. When necessary, the Auditor suspends a client thread using `SuspendThread()` since it is three orders of magnitude faster than attaching as a debugger.

Controller Details. The Binary Parser processes files of the Windows Portable Executable (PE) format, maps those files to virtual addresses, fixes the links in each file, and hashes the static sections with SHA1. The implementation challenges included resolving circular linking dependencies, evaluating forwarded exports, handling aliased functions, and locating anonymous import tables. When available, the parser also processes files of the Program Database (PDB) format to uncover un-exported function symbols and identify dynamic data variables.

The Code Disassembler is a basic `x86` disassembler which walks client code starting at the entry point of the binary and follows function calls to determine the range of and relationship between client functions. Implementation challenges included resolving functions that terminate in inconsistent ways, functions that use discontinuous memory, and functions that embed return instructions in the middle (in particular those protected by the Windows implementation of StackGuard [22]).

To better understand the game client structure, the Execution Sampler uses `CreateProcess()` to run the client application local to the Controller since the client and Controller use common non-ASLR operating systems (i.e., Windows XP SP3). The sampler exhaustively hashes every user-space page between `0x00000000` and `0x7FFFF000` that is static (i.e., allocated and non-writeable).

The Execution Profiler also runs the client, but attaches to it as a hardware debugger to learn the function frequency and indirect function calls unhandled by the disassembler. The implementation challenges included avoiding the game's anti-debugging techniques, resolving timing delays caused by the debugger interrupts, and guiding the game into its gameplay loop to be measured.

MEASUREMENT STRATEGY. The Controller uses a straightforward game-independent audit strategy that takes measurements at a specified interval. To avoid predictability, random timing jitter is added, uniformly distributed between -5% and +5% of the interval length. Throughout each experiment, the chosen measurement routine was held constant but the target of each individual measurement was left unspecified. For stack trace audits, this means a thread was selected at random. For page hash audits, this means a thread was selected at random and its executing code page was hashed.

2.4.1 Experimentation

Benchmarking

The Fides Auditor and Controller were benchmarked to evaluate whether the approach is suitable for commercial-off-the-shelf games (in this case the game was Warcraft III [12]) without imposing a significant negative compute burden in exchange for its cheat detection benefits. Using a 2.39GHz Intel Core2 machine, each routine was executed 1,000,000 times recording the average and variance in the number of cycles to complete them. The efficiency of the Auditor and Controller routines are shown respectively in Table 2.2 and Table 2.3.

Warcraft III imports 17 DLLs representing 4,103 total functions, many of which belong to system libraries and are unused. The memory sections represent 1,649 total pages of which 1,619 are flagged as static. This indicates library layout commonality (specifically on non-ASLR systems) could save the Controller from executing the full learning routines for every client.

Auditor Task	Cycles			Time
Null	**82**	±	**18%**	**34.6**ns
Sample Memory	**87,941**	±	**21%**	**36.8**μs
Suspend Threads	54,960	±	22%	22.9μs
Read Memory	11,947	±	35%	5.0μs
Fetch Page Flags	21,854	±	30%	9.1μs
Resume Thread	4,249	±	97%	1.8μs
Hash Page	**112,885**	±	**21%**	**47.2**μs
Suspend Thread	4,843	±	33%	2.0μs
Get EIP	19,760	±	44%	8.3μs
Resume Thread	4,153	±	65%	1.7μs
Read Page	15,597	±	32%	6.5μs
Hash Contents	55,391	±	23%	23.2μs
Fetch Page Flags	11,031	±	42%	4.6μs
Trace Stack	**64,399**	±	**27%**	**26.9**μs
Suspend Thread	4,844	±	25%	2.0μs
Get EIP & EBP	26,042	±	45%	10.9μs
Get Stack Range	15,782	±	42%	6.6μs
Traverse Stack	13,462	±	37%	5.6μs
Resume Thread	4,292	±	31%	1.8μs
Detect Debugger	**23,246,056**	±	**1.5%**	**9.7**ms
Test Debug Flag	2,998	±	43%	1.3μs
Attach Debugger	21,411,428	±	1.2%	8.9ms
Detach Debugger	1,828,041	±	7.0%	0.8ms

Table 2.2: Efficiency of Auditor routines. The average and variance in the number of processor cycles to complete the routines was measured and converted to time.

The results demonstrate that the Auditor's measurements complete very quickly. The three most common routines (sample memory, hash page, and trace stack) operate on the order of tens of microseconds, adding imperceptible overhead (0.05% when auditing roughly once every 100ms) to the game. The most expensive routine (detect debugger) takes 9.7ms and only adds a perceptible hiccup to the game if performed frequently in a tight loop.

Controller Task	Cycles			Time
Null	**85**	±	**19%**	**35.6**ns
Parse Binaries	**236,322,896**	±	**0.1%**	**98.8**ms
(single file)	17,221,136	±	2.5%	7.2ms
Memory Map File	741,448	±	5.0%	0.3ms
Identify Sections	4,774	±	20%	2.0μs
Allocate IAT	388,067	±	19%	0.2ms
Populate IAT	5,193,318	±	2.1%	2.2ms
(single entry)	58,517	±	4.4%	24.5μs
Hash Sections	10,893,211	±	4.5%	4.6ms
(single page)	46,908	±	6.7%	19.6μs
Disassemble Code	**205,290,560**	±	**0.2%**	**85.9**ms
Isolate Function	1,165,818	±	6.3%	0.5ms
Validate Result	*varies as follows ...*			
Sample Memory	*data dependent*			
Hash Page	3,170	±	20%	1.3μs
Trace Stack	10,808,131	±	10%	4.5ms
Detect Debugger	130	±	21%	52.4ns

Table 2.3: Efficiency of Controller routines. The average and variance in the number of processor cycles to complete the routines was measured and then converted to time for convenient analysis.

The Controller's learning routines take under 100ms and do not impose a meaningful burden to the game considering they are only run once at client logon. The number of subtasks to learn a game depends on its complexity – this game is reasonably complex. Similarly, the validation routines are very quick. Validating a hash page audit involves a binary comparison of the 20B digest and validating a debugger audit evaluates the Boolean value, both very efficient routines. Validating a stack trace audit is more involved because each instruction pointer in the list must be looked up as to which function it belongs to, and if it can legally call the next function. The effort to validate such an audit depends on the stack depth and the number of functions in the game.

Homebrew Game

This section evaluates Fides on our Homebrew (i.e., of our own creation) game and four cheats employing different techniques in order to compare the effectiveness of stack trace audits and hash page audits. The game is a simple networked two-dimensional arena combat game in which players jump to-and-from a number of platforms while shooting at each other. The game is single-threaded and uses the Simple Directmedia Library (`SDL.DLL`) which facilitated fast game development by conveniently wrapping graphics functions (in `OPENGL32.DLL`), network functions (in `WSOCK32.DLL` and `SECUR32.DLL`), and system functions (in `KERNEL32.DLL`). The game client executes a short gameplay loop where it checks for input, communicates changes to the server, renders the world, and then sleeps. In total the loop takes $25.8ms$ to complete, equivalent to an average frame rate of $38.8fps$.

To simplify cheating, the game client stores sensitive data (i.e., opponent positions) at fixed locations in memory. The game client also performs shooting actions through the `ShootGun(x,y)` function which fires a shot from the player's current coordinates to the specified `(x,y)`-coordinates. Four game cheats using different cheating techniques were implemented to exploit this function in order to automatically aim the cheater's shots.

The first cheat uses a small hotpatch which statically replaces the input to the `ShootGun()` function with the static coordinates of a target at time of patching – clearly not a very intelligent cheat yet done in place without allocating any additional client memory. The second cheat injected a DLL which detours the `ShootGun()` function to injected code which dynamically aims the shot at an opponent. The third cheat injects a DLL that trampolines the `ShootGun()` function to a `CheatShootGun()` function (contained within the injected DLL) which uses game state to better aim all shots. The fourth cheat injected a DLL and a new game thread to run the injected code. The injected the code loops indefinitely, intelligently executing the `ShootGun()` function every $100ms$.

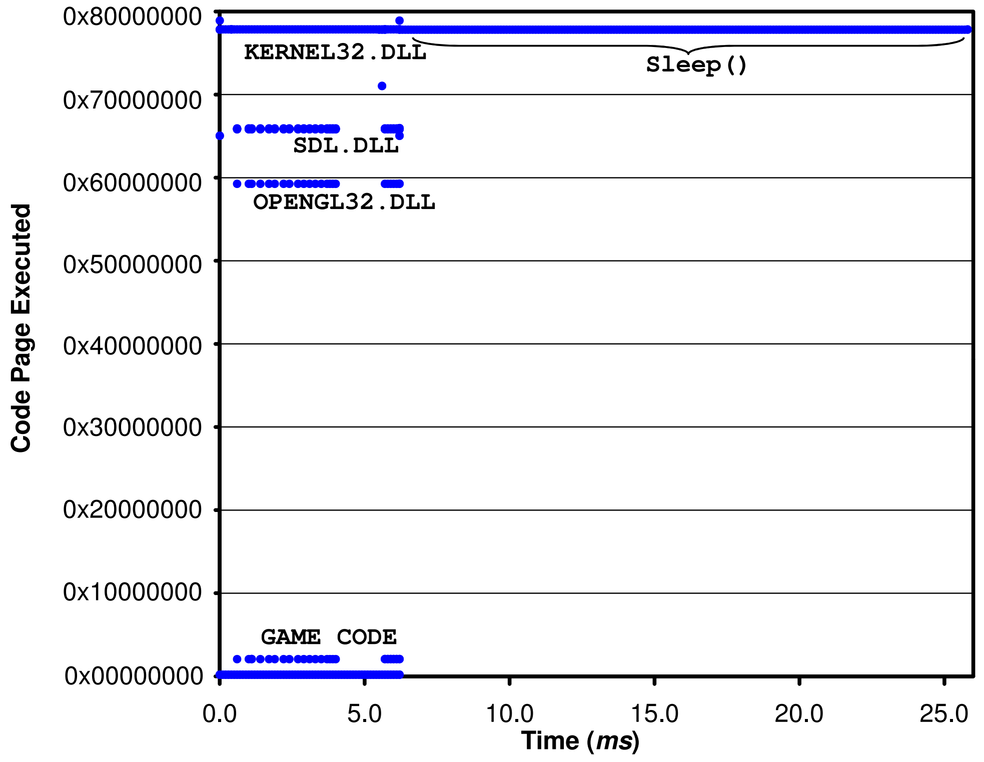


Figure 2.5: Execution profile of the Homebrew gameplay loop. Most of the pages executed represent game code and just a few other DLLs – predominantly `KERNEL32.DLL` because of the `Sleep()` function.

Game Profile. Using the Execution Profiler, the execution pattern of the gameplay loop is illustrated in Figure 2.5. The code pages executed throughout the gameplay loop are shown. Note that there are 65,536 pages between y-axis labels and dots are disproportionately large for legibility. When actively running simulation code, the game frequently calls short library functions that return quickly to the game code. No significant amount of contiguous time is spent on any single code page although many of the executed code pages are clustered in close proximity to each other. Once the loop calls the `Sleep()` function, execution is suspended on that single code page for a significantly large amount of time. Specifically, the gameplay loop spends 76% of its time in `KERNEL32.DLL` almost entirely due to that function.

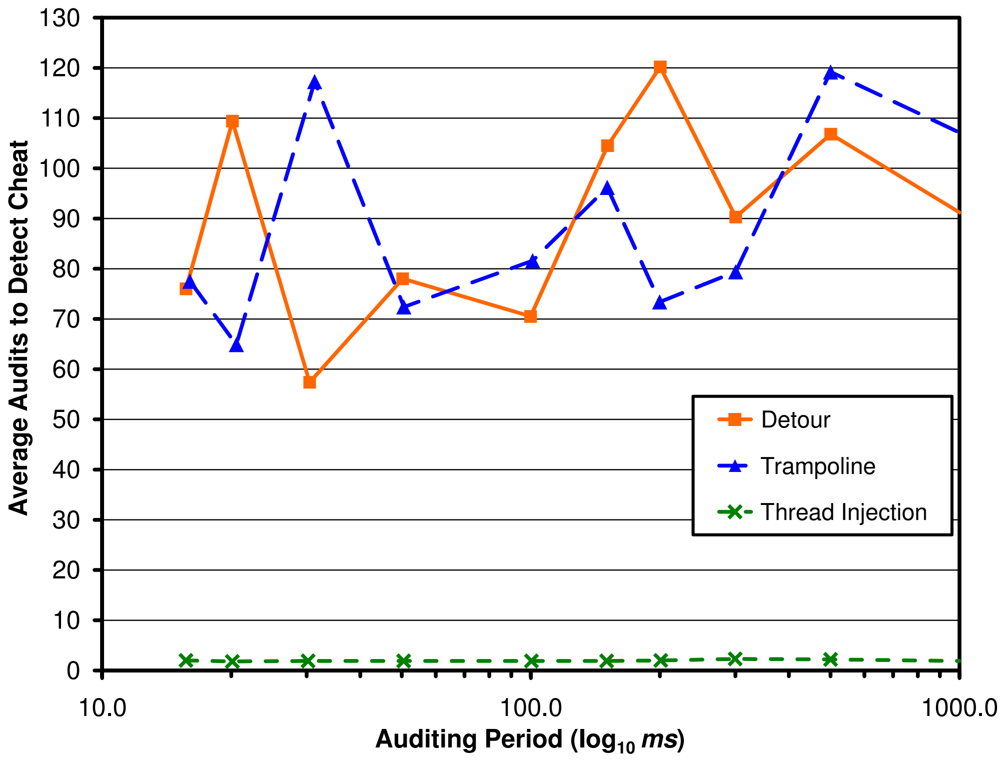


Figure 2.6: Stack trace audits required to detect cheats for the Homebrew game. Three of the four cheats are detectable by stack traces; only in situ hotpatches cannot be detected. In constrast, thread injection is particularly susceptible to detection by stack traces.

Cheat Detection. Figure 2.6 shows the average number of stack trace audits required to detect the cheats. For the cheats that hijack execution, more audits are required for detection because the `ShootGun()` function where the cheats manifest is infrequently executed. Not surprisingly, stack trace audits cannot detect hotpatches which do not change execution flow, but stack trace audits do very well in detecting thread injection. This is because the stack trace has a uniform $\frac{x}{x+y}$ probability of selecting the cheat thread (which is immediately detected as unrecognized, even when idle), where there are x cheat threads and y game threads. In this case the probability of selecting the injected thread is $\frac{1}{2}$ and supports the two audits on average required to detect the thread injection cheat.

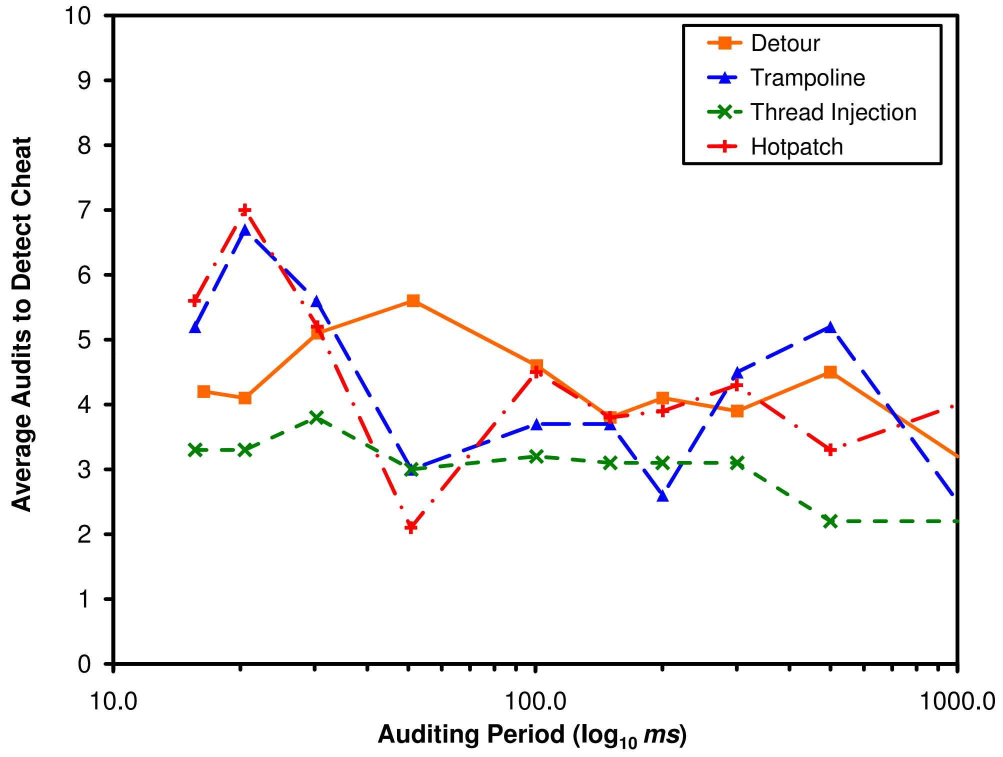


Figure 2.7: Page hash audits required to detect cheats for the Homebrew game. All four cheats are quickly detected by page hashes.

Figure 2.7 shows the average number of page hash audits required to detect the cheats. Each cheat is detected quickly using roughly four audits – expected since the game is single-threaded and spends about a quarter of its time in game code. These results are an order of magnitude faster than the stack trace audits. The small code footprint contributes to the fast cheat detection; while the game does not execute the `ShootGun()` function frequently, other more common functions reside on the same code page so audits indirectly catch the cheat.

This leads to the observation that *careful client design can trade some memory performance for quicker anomaly-based detection.* Hash page audits are able to quickly, sometimes indirectly, catch these cheats not because they modified a lot of code but because the code they did modify was located on the same memory page as frequently executed code.

Commercial-Off-The-Shelf Game: Warcraft III

The Fides approach works best on games designed with it in mind. However, to evaluate that the Fides approach works on commercially-deployed games, this section explores the detection of cheats for the popular Real Time Strategy (RTS) game Warcraft III [12]. The game was released in 2002 and has remained extremely popular since then, especially as the game of choice for professional video game competitions. Weekly, thousands of players continue to play it online in competitive, ladder-ranked matches. The game's developer, Blizzard Entertainment, has patched the game regularly throughout the years to fine-tune game balance and attempt to stay ahead of cheaters. This experimentation was done in October 2008 immediately following a patch release. Within one week, new cheats were released in response to the patch and Fides readily detected them. A month later, the preeminent cheat was released and Fides swiftly detected it.

Warcraft III has several security features that make it complex. The game client is designed so that the executable is merely a launcher: all game functionality is located in a library appropriately named `GAME.DLL`. This library is loaded at runtime via the Windows `LoadLibrary()` function and is obfuscated to hinder disassembly. Similar to the Homebrew game, Warcraft III executes a short gameplay loop where its threads check for input, communicate with the server, render the world, and then sleep. In total, the loop takes only 15.6ms to complete, equivalent to an average frame rate of 64.2fps.

The game client is heavily multi-threaded, employing 22 threads of which only 8 are active during normal gameplay. Threads have assigned tasks that include rendering the world, loading map and texture info, gathering player input, managing network connections, running artificial intelligence routines for path planning and non-player (i.e., computer) opponent strategy, stub server routines for hosting games locally, and routines for performing some anti-cheating and anti-debugging techniques (it attempts to kill any process that attaches as a debugger).

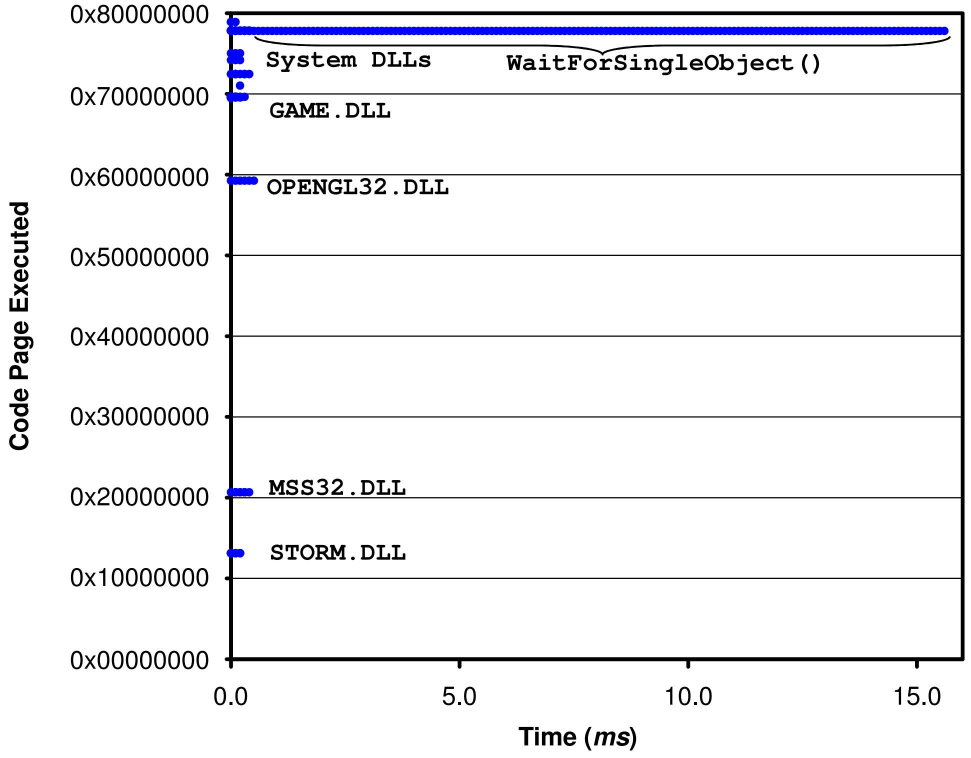


Figure 2.8: Execution profile of the Warcraft III gameplay loop. Like the Homebrew game, the pages executed represent game code and a few other DLLs – predominantly `KERNEL32.DLL` due to the `WaitForSingleObject()` function.

Game Profile. Using the Execution Profiler, the profiles of the threads active during the gameplay loop are shown collectively in Figure 2.8. Game threads execute system functions (in `NTDLL.DLL` and `KERNEL32.DLL`), core game functions (in `GAME.DLL`), render graphics (in `OPENGL32.DLL`), process audio (in `MSS32.DLL`), and access game data (Blizzard games use `STORM.DLL` to load map data from disk). The game uses `WaitForSingleObject()` to sleep while waiting for I/O; if an input event occurs, the function returns early, otherwise it sleeps for the full duration. Without active input, the game threads spend roughly 94% of the loop sleeping. With 22 threads to randomly choose from when auditing the game, the odds of auditing game code modified by a Warcraft III cheat is relatively low when compared to the simple Homebrew game cheats analyzed earlier.

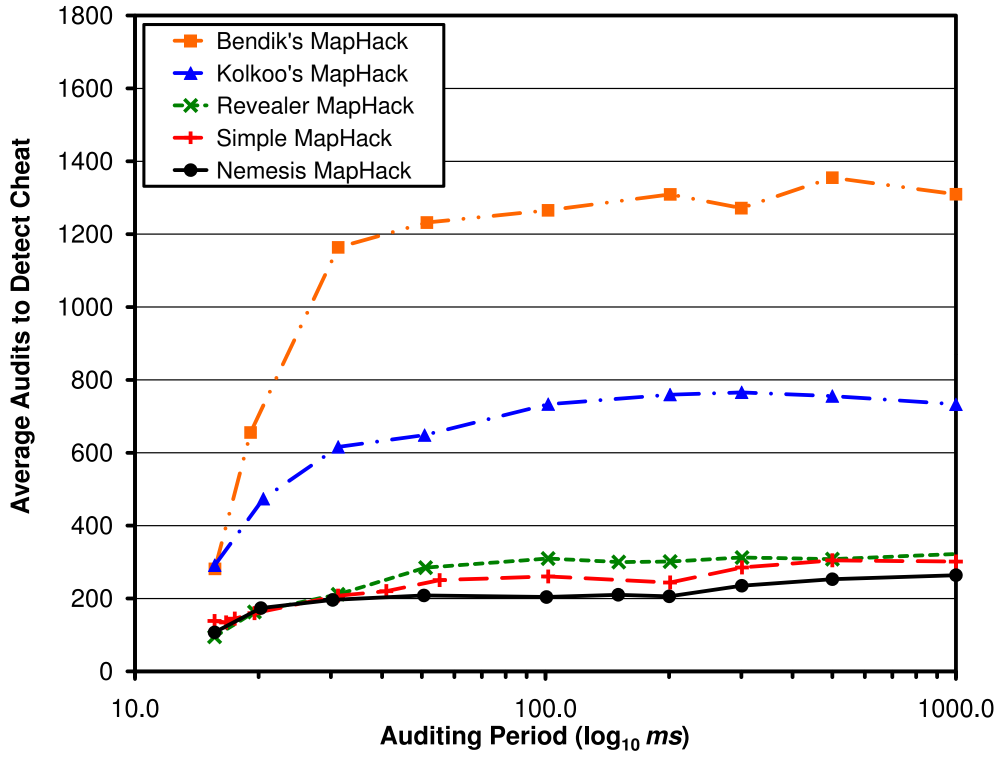


Figure 2.9: Page hash audits required to detect Warcraft III maphack cheats. The five representative cheats shown are easily detectable by page hashes.

Cheat Detection. Several communities are dedicated to cheating in Warcraft III. They produce and sell numerous cheat binaries, very rarely revealing actual source code. Advanced cheats for popular games often have price tags that rival the purchase price of the game itself. Since cheat-software sales are at stake, cheat authors compete to be the first to publish a cheat that works against the newest patch level of the game. Ironically, many authors steal other authors' cheat codes and techniques to spread them to new communities claiming them as their own work – there truly is little honor among cheaters.

When this experimentation was performed Warcraft III was recently patched, meaning a spectrum of first-to-release cheats were readily available. Each cheat was run in isolation by hosting non-ranked Local Area Network (LAN) games to avoid disturbing legitimate players. Fides was able to detect every cheat collected;

the five most prominent cheats (i.e., all original software, not variants rebranding stolen code) being: *Bendik's MapHack* which is very minimalist and reveals no more than hidden units on the main map, *Kolkoo's MapHack* which patches code over a few pages, the *Revealer MapHack* which hotpatches a small number of bytes over fewer pages but also injects code that hooks game input functions so that it may be toggled on or off, the *Simple MapHack* which hotpatches a mere 61 bytes but over a number of pages, and Perma's *Nemesis MapHack* which is the prominent success of this experiment.

Perma is viewed in many communities as the foremost Warcraft III cheat author. His Nemesis MapHack (which he advertises as "undetectable") is the sequel to his infamous *Zerocraft* cheat which went undetectable for just short of two years before it was discontinued because Blizzard finally obtained a copy and developed a signature that could detect it. In an attempt to prevent Blizzard from obtaining a copy of the Nemesis MapHack, the procedure to obtain a copy requires one to have a profile with good standing in his home cheat community[4] and purchase the cheat for $25. Amazingly, this is more than the current purchase price of the game. The Nemesis MapHack has more features than the other cheats mentioned and is heavily obfuscated to prevent Blizzard (as well as rival cheat authors) from learning its tricks.

Figure 2.9 shows that the Warcraft III cheats are detected using upwards of two hundred Hash Page audits, which is not surprising given that the game has 22 threads and sleeps roughly 94% of the time. At this scale, one can see a trend that the number of audits required to detect a cheat tapers off. This occurs because at larger audit frequencies the inter-arrival randomness grows to the point where the randomness (between ± 5% of the frequency) surpasses the loop duration of 15.6ms so that the probability of detection asymptotically approaches the probability to detect the cheat with a single random audit.

[4]We used an account that had been established for previous anti-cheating research.

The probability of detecting a cheat grows with respect to the quantity of changes that the cheat makes to the game (i.e., a cheat that modifies more code pages becomes easier to detect). This makes the feature-rich "undetectable" Nemesis MapHack among the easiest cheats for Fides to detect. Fides did not need to reverse engineer it. The number of audits required to detect Nemesis taper off somewhere around 300, meaning it can be detected in about 5 minutes when auditing at a leisurely frequency of once per second, or in 20 seconds when auditing at a more aggressive frequency of once every $100ms$. In contrast, Bendik's MapHack which few changes to the game would take 22 minutes to detect when auditing once per second, or 2 minutes to detect when auditing once every $100ms$.

Memory Layout Commonality

While benchmarking the Fides approach, we observed that commonly loaded libraries to common locations (specifically on non-ASLR systems) could spare the Controller from generating and storing new emulation state for each client. This experiment evaluates how consistently game-client memory is laid out in a commercial game (i.e., Warcraft III again) to show that significant potential exists for reuse of emulation state.

A small program was implemented to execute the game client and then measure its virtual memory layout. The game client's memory regions are flagged with read, write, and executable permissions when allocated and may be altered during execution. In general, virtual memory is either code (i.e., flagged as executable), static data (i.e., flagged readable but neither writable nor executable), or dynamic data (i.e., flagged readable and writeable but not executable). Of particular interest to this evaluation are memory regions that should remain constant throughout gameplay: code and static data. Those regions are stored as page hashes during emulation so any commonality between clients means fewer hashes need to be stored. In contrast, dynamic data regions require server corroboration so the associated emulation state is already reduced to only meta-data.

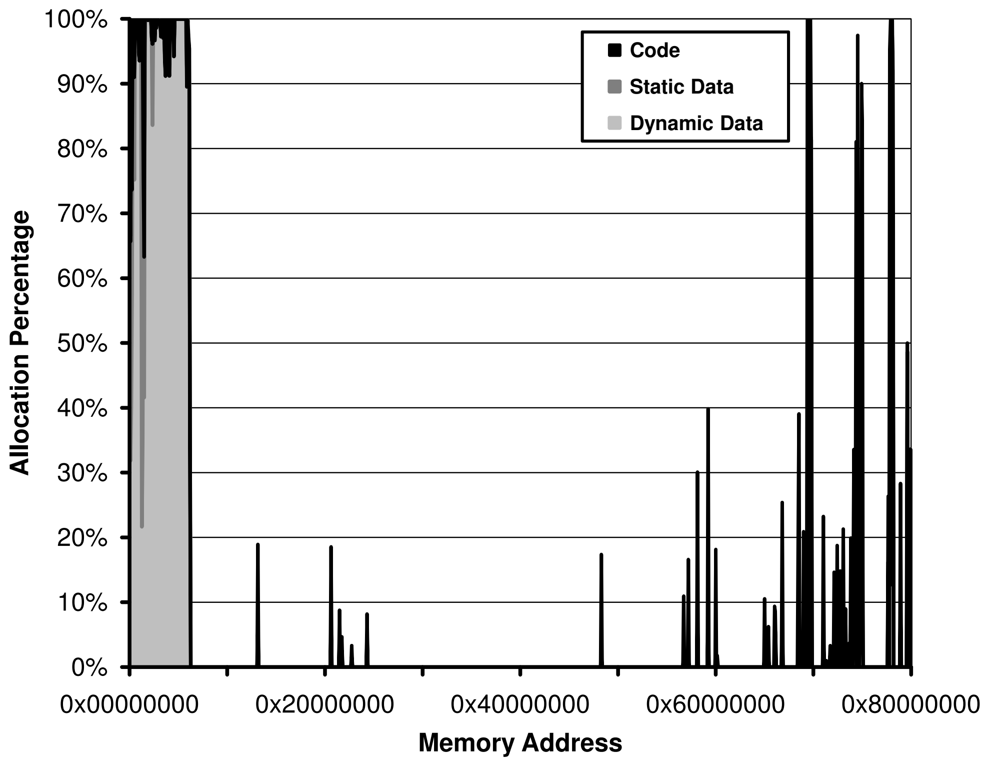


Figure 2.10: Warcraft III virtual memory usage. The percentage (stacked) of each type of page usage within 512-page regions. Here dynamic data includes reserved, but presently unused memory.

The program ran the Warcraft III game client 1,000 times through the same map on two different Windows XP SP3 machines. Figure 2.10 shows the memory layout by type in Warcraft III. The first observation is that the memory space is truly sparse: less than 10% of the available virtual memory is used. Many libraries select default base addresses that are far from other libraries so that they will not likely conflict when loaded, resulting in the spread out code observed.

Typical of most Windows applications, the executable entry point is located at address `0x40000` which accounts for that spike of code amid the static and dynamic data. Below it resides some language localization files and stack space for each game thread. Above the entry point is the heap, which in this game is predominantly used for map data.

Memory Type	Machine #1	Machine #2	Both
Code	**1.4%**	**1.4%**	**1.4%**
constant	*100.0%*	*96.2%*	*90.8%*
Static Data	**1.0%**	**1.0%**	**1.0%**
constant	*98.8%*	*94.6%*	*87.2%*
Dynamic Data	**3.9%**	**3.0%**	**3.5%**
similar	*29.5%*	*55.6%*	*11.3%*
Reserved Memory	**3.5%**	**3.4%**	**3.5%**
similar	*64.6%*	*93.5%*	*49.4%*
Unallocated	**90.3%**	**91.2%**	**90.6%**

Table 2.4: Memory layout commonality. Memory allocation is broken down by type. The table shows how consistently each type is laid out.

Table 2.4 presents the memory breakdown for the 1,000 runs on each of the two machines and how they compare to each other. The results show that code and static data occupy 1.4% and 1.0% of virtual memory, respectively. As expected, both code and static data are allocated consistently on the same machine and are roughly 90% consistent between the two machines. This confirms that indeed significant portions of emulation state (i.e., hash page digests) may be reused between sessions and between clients.

Dynamic data and reserved memory (i.e., virtual memory set aside but not yet allocated on physical storage) each occupy roughly 3.5% of virtual memory. Dynamic data predominantly represents map data and reserved memory is saved for map data that has yet to be loaded. The range and size of these dynamic data sections vary considerably between executions and machines. This may be partially due to the fact that in each game execution, the player's homebase is randomly assigned to one of eight possible locations and means different map data may be loaded initially. Map data is essentially static since it is loaded from static game files on disk and it could be loaded to specific locations and have write-permissions removed so that it can be audited by page hashes.

2.4.2 Limitations

This section recognizes the following limitations to the Fides approach:

Attacks on the Auditor. Cheaters own the systems on which they cheat so they may tamper with any component to evade detection. Likely an adversary would directly target the Auditor by replacing the results it returns to the Controller with clean measurements. This is an easier problem to handle than securing the game client itself because the Auditor is less complex. Furthermore, a number of techniques infeasible for protecting the game client could be used to bolster Fides against such attacks including directly accessing memory, audit entanglement, Auditor and client polymorphism, or leveraging tamper-resistant coprocessors. These techniques are elaborated further in the discussion.

Elementary Measurements. While the measurements presented in this paper detect current and foreseeable cheats, cheaters may evolve methods that the measurements cannot detect efficiently. It is important to note that Fides is not restricted to the four presented measurements and can easily be extended to include new measurement types. In particular, one might consider memory watchpoints and code timing if hardware support became available. Memory watchpoints would observe how frequently code or variables are accessed and could detect cheats that operate external to the game client through unauthorized data reads. Similarly, code timing would observe abnormally long execution of game functions or the entire gameplay loop. This can detect cheats that manipulate the game through a debugger or virtual machine.

Timing Difficulties. Network latency and jitter add timing inaccuracy at a resolution several magnitudes greater than that which code executes at, preventing any remote software integrity system (not only Fides) from predicting a client's exact execution state upon receiving an audit packet. Such systems are thus forced to validate the correctness of whichever state the client software is observed to be in rather than demand that the client be in precisely one state.

External Cheats. Fides focuses on cheats that affect the proper execution of the game client and does not address cheating external to it. For example, online poker cheaters collude by sharing private information (i.e., the cards in their hand) through side-channels in order to defraud legitimate players. Other cheaters employ robotic peripherals to automate repetitive or precision-based gameplay (e.g., the Guitar Hero robot [125]). Cheats that never modify the client cannot be caught by detection of anomalous client execution but instead may be caught by detection of anomalous player behavior (e.g., observing wins correlated to unusual gameplay, abnormal grouping patterns, highly erratic player skill, or even abnormally precise game input).

Poor Game Design. Cheat detection may not address behavior that is against the spirit-of-the-game yet is possible without ever modifying the client or employing external devices. For example, many networked games incorporate "player achievements" that are noteworthy in-game objectives (publicized whenever a player accomplishes one) but when implemented improperly are trivialized by players who design custom levels to do so. Such problems should be solved by better game design, in this case disallowing players from attaining achievements while playing custom levels vetted by the game developer.

Macros and Keybindings. Many developers relax the prohibition on automation by allowing players to customize their game interfaces with macros and keybindings for authorized game commands. Games like World of Warcraft use execution-tainting mechanisms to distinguish permitted customization from excessive automation [126]. For accurate anomaly detection, these interface customizations should be isolated to well-defined regions of client memory where they will not interfere with audits of game code or data. Indirect jumps that access them can be easily validated if the jumps are restricted into those isolated regions. Furthermore, the client must commit to the customizations (i.e., send them to the server) before using them so that the server may verify their legality and Fides may validate that they are not modified at inappropriate times during gameplay.

2.4.3 Discussion

Future Work

There are a number of techniques that could be explored to strengthen the Fides approach (specifically the Auditor) against evasion by the cheater.

Direct Access. Locating the Auditor within the client process provides direct access to its virtual memory without relying upon system functions like `ReadProcessMemory()`. This sidesteps attacks that hook those system calls in order to return bogus measurements. At the same time, eliminating function calls for reading process memory speeds up the Auditor meaning that audits which suspend client threads will have even less impact on performance. However, this is only a partial solution since the Auditor will become easily preemptable and thus vulnerable to cloaked cheats like the Governor [51].

Audit Entanglement. Fides may authenticate audited data by entangling the Auditor's measurement routines with time-sensitive cryptographic computations, similar to Pioneer [100]. Each entangled computation would depend upon a random nonce (sent as part of the audit request) and be constructed such that modification to the measurement routine must alter either the correctness or timeliness of the generated entanglement token. Using entanglement, an Auditor response would only be valid if both the measured data and the correctly computed entanglement token are returned within the time limit. Invalid or late entanglement tokens would indicate the client's use of cheat software.

Auditor Polymorphism. Rather than support defined measurement types, the Auditor could instead accept and execute short auditing routines crafted arbitrarily by the Controller. The measurement types and targets may be changed surreptitiously at any time, dynamically adjusting how the client is audited. Cheaters cannot evade detection by simply using static virtualization; they must understand precisely what each audit routine is measuring and formulate coherent responses (through their own client emulation).

To effectively manipulate data collected by the Auditor, cheaters must completely interpret every audit routine and emulate legitimate client operation sufficiently to generate correct responses on the fly. This is challenging for cheaters to accomplish in a timely fashion, especially when auditor polymorphism is used in conjunction with audit entanglement. Cheaters who fail to completely cloak their changes or virtualize the entire system will be detected by the first measurement that cannot be spoofed, similar to how "undetectable" virtualized rootkits (e.g. BluePill [95]) are detected when unpredicted system functions (like `cpuid()` [92]) behave erratically.

Auditor polymorphism would make the Fides approach more like the Warden system [76] in that frequent updates to client-side detection software keeps cheaters on guard. This allows the system to remain agile enough to adopt new measurements in order to detect new cheat techniques.

Client Polymorphism. Using this technique, the game server periodically instructs the game clients to shuffle their memory layouts by rebasing loaded libraries to arbitrarily specified new locations. This changes the structure of the client without affecting legitimate operation, *dynamically adjusting how the client must be targeted by cheats.* Runtime library rebasing may be done similarly to how in-memory or reflective DLL injection [39, 104] loads libraries from within the process' memory (i.e., not from disk) at runtime.

Each time a new client layout is specified, cheaters must adjust their software accordingly, particularly code hotpatching long jumps or code overwritten during the rebasing. Cheaters attempting to manipulate measurement data must quickly correct their own emulation and system virtualization to remain consistent with legitimate clients. The Controller has an advantage since it can update its emulated state before making the changes known to clients, however, cheaters cannot respond until after they know the layout changes. Additionally, this technique could reduce emulator state by dictating the same new layout for all clients using common libraries.

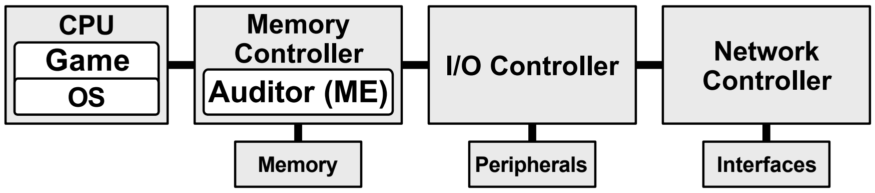


Figure 2.11: One functional location of Intel's Manageability Engine (ME). In the memory controller, it has unimpeded access to all game memory and I/O.

Hardware-Based Stealth Measurements. The Fides system was designed so that the Auditor could leverage a hardware component within the client machine for providing tamper-resistant measurements of the game. The hardware component must be isolated from the host processor so that it cannot be affected by the system owner, yet it must have sufficient access to measure the system to determine if cheat software has compromised the game client.

The Intel Active Management Technology (AMT) [56] platform is one such suitable hardware component. Figure 2.11 shows the current architecture of the AMT, specifically the location of the trusted Manageability Engine (ME). While the ME is not a full-blown CPU (i.e., it lacks the speed and features necessary to run the game client), it has been used to detect rootkits [32] and peripheral automation cheats [98]. The ME would be a good location for the Auditor because it has access to the entire contents of physical memory, it is controllable through a secure (i.e., authenticated and encrypted) network connection, and it only executes signed code thereby assuring players that their privacy is safe.

While challenges remain to employing such a hardware component (e.g., register and memory caching, and virtual memory to real memory mapping), hardware support could facilitate additional measurements for detecting more subtle forms of cheating. Specifically, the addition of memory watchpoints and code timing would allow the Auditor to observe unusual memory accesses in terms of timing, source process, and frequency.

2.5 RELATED WORK

Anomaly-Based Detection. Anomaly-based detection has been explored as a solution in research to many strong security problems like intrusion and rootkit detection [55, 71, 88, 120]. In such applications, the detector must understand the characteristics of a complex multi-faceted system. Extensive knowledge is required to perfectly characterize all legitimate operation, so these approaches instead accept a small misclassification rate in exchange for more manageable state and quick detection results. In contrast, the cheating problem has a limited search space (i.e., the well-defined game client software) which can be efficiently learned using static analysis, and the game application tolerates slower adversary detection over the misclassification of legitimate players.

The branch of research most related to our approach is anomaly-based application integrity checking [16, 34, 50, 99, 119], which validates application behavior from the vantage point of a secure operating system. These approaches work well when an adversary has difficulty infiltrating the host system, however, they are inappropriate for the cheating problem where the adversary owns the machine and can readily alter the operating system to disable detection tools.

Other research applies anomaly detection to player behavior [19, 44, 74, 129]. These approaches detect suspicious player behavior (e.g., colluding) without verifying game state by instead analyzing high-level player behaviors and win-loss statistics. While those approaches can be used in combination with our approach or integrity-based approaches, they are very game-dependent and require a deep understanding of the game rules, game maps, and what normal behaviors look like. Acquiring an understanding of "normal" player behaviors often requires a large volume of trusted gameplay samples and can be expensive in terms of human involvement. This difficulty is exacerbated in persistent games (such as massively multiplayer online games) where the popularity of in-game activities changes over time (sometimes in unpredictable ways) as the virtual world evolves.

Anti-Cheat Software. Many game developers use anti-cheat software; Blizzard games (e.g., Diablo II, StarCraft, Warcraft III, and World of Warcraft) use the Warden system [76] and Valve games (e.g., Counter-Strike, Left4Dead, and Team Fortress) use the Valve Anti-Cheat (VAC) [115]. Numerous other games (e.g., Battlefield 1942, Call of Duty, and Quake) support the use of third party anti-cheat systems like PunkBuster [31]. Unfortunately, existing commercial anti-cheating systems use signature-based detection, promiscuously scanning each and every process on the client machine which leads to accuracy errors and real attacks on the detection mechanisms [85].

Trusted Computing. Trusted computing approaches leverage secure cryptographic software, virtual machines, or hardware components to verify that the application code is operating as intended on the remote machine. Sometimes referred to as a "*root-of-trust*," the trusted system is used as the starting point to assure the integrity of the application. Prominent systems include Terra [41], the TPM architecture [96], and Pioneer [100]. While trusted computing approaches quickly discover changes made to the application, they add overhead to the application's execution that may be prohibitively expensive for the real-time demands of responsive gameplay. Instead, trusted computing approaches could be used to guarantee the proper operation of the Auditor which is smaller, has fewer real-time requirements, and can be used to audit any game.

Proof-of-Correctness. Proof-of-correctness approaches focus on cheat prevention rather than cheat detection [10], ensuring that the game client is running according to game rules and physics. While proof-of-correctness approaches prevent some unauthorized manipulation of client state, they can only slow or frustrate cheating efforts since the adversary controls the hardware and operating system of the machine. Even for console gaming systems, where the developer has a better understanding of and design input regarding the underlying hardware, cheating has proven impossible to eradicate and detection remains a vital tool at the game developer's disposal.

2.6 CONCLUSION

This chapter investigated the problem of detecting automated and adversarial behavior in the context of detecting cheaters in multiplayer online video games. In this application, cheaters own the hardware that the client software runs on and as a result have a clear advantage over game developers in terms of control over system operation. The research contributions of this chapter are:

- We clearly *defined the cheating problem* affecting multiplayer online games and identified properties that distinguish this problem from other traditional security problems. The chief differences are that cheats target specific, well understood software, do not demand urgent response, and inflict damage that is easily repaired once discovered.

- We *enumerated the state-of-the-art in cheating methods.* Cheat techniques range from data manipulation to code injection and execution manipulation. Most cheats do not leverage hardware techniques such as register manipulation or hardware debugging at present, but that may change in the future.

- We *proposed and evaluated a novel anomaly-based cheat detection approach* for multiplayer online games through remote validation of client execution. The Fides system comprises a server-side Controller which specifies how and when a client-side Auditor measures the game client. To accurately validate measurements, the Controller partially emulates the client and collaborates with the game server. In evaluation, we showed that Fides is able to efficiently detect several existing cheats including one advertised as "undetectable."

Likely, the Auditor may become the next target of attack by cheaters. Future work involves research into a number of techniques that may bolster it against attack, including locating it on secure tamper-resistant hardware, entangling it with cryptographic computations, or adding run-time client polymorphism.

While an application service provider may possess an efficient automation detector like Fides, it may not be sufficient to identify every adversary. Improved adversary identification may require combining multiple detectors. It may also involve cooperation between service providers, whether those service providers support similar or different applications. It may also involve input from clients.

The next chapter discusses how to combine multiple detectors to better identify adversaries. This research focuses on an approach which treats clients as detectors but also explains how the approach may be extended to leverage strong detection systems like Fides.

Chapter 3

ADVERSARY IDENTIFICATION

3.1 INTRODUCTION

The previous chapter explored an approach for definitive detection of automation and software tampering. Like many real-world adversaries (e.g., in sports, business, and daily interactions), network adversaries find that behaving outside the defined rules is beneficial so rather than be deterred by accurate detection methods, they instead adapt to avoid the most prominent detection method. As a result, application service providers must resort to multiple means to detect adversarial automation. How to aggregate detectors to make a definitive decision is an important research problem. This chapter investigates a method to do this.

Despite the fact that accurate detection methods exist, rule enforcers have limited resources and cannot observe all infractions. Legitimate participants are obliged to help by reporting malicious behaviors but they are often intimidated into silence. In team sports like cycling, this is known as the *doping dilemma* [102]: athletes who play by the rules are intimidated by teammates and trainers into remaining silent while athletes who break the rules continue to get ahead. Participants expect rule enforcers to catch and punish the rule breakers, yet without their participation that does not always happen. Our approach is to leverage the power of the masses as accurate detectors to allow them to congregate exclusively with like-minded individuals. This approach would allow legitimate participants who want a level playing field to disassociate with rule breakers and compete against legitimate competitors. In such a system, humans are effectively very abundant, yet individually inconclusive adversary detectors.

Given a collection of adversary detectors that may individually be inconclusive or untrustworthy, the next research challenge addressed by this dissertation involves how to best combine detectors to increase the likelihood of a predictive and conclusive result. Specifically, this research seeks a single metric that ranges from "not adversarial" to "adversarial" (i.e., a probability $\in [0.0,\ 1.0]$) to describe each client. This provides an analog control that can operate disincentive mechanisms based on how adversarial or automated each client's behavior appears.

This chapter explores this research challenge by federating a set of homogenous independent detectors. This work is realized through the creation of a reputation system for multiplayer online games. Again, the multiplayer online game application is ideal for research since well established disincentives exist. The application is particularly suited for research into reputation systems because the players themselves are abundant detectors and require little game developer effort to harness. By treating every player's peers as detectors (i.e., their observations provide clues regarding the maliciousness of other peers) the most basic disincentive follows: players will dissociate with and cease to play with players who are known to behave badly.

Legitimate players have in some cases created guilds whose primary goal is to locate automated gold farmers within massively multiplayer online games through word-of-mouth reports. Provided with better tools like a reputation system, those guilds may become more efficient at tracking automated players. By analyzing the resulting social patterns, the game developer may focus man-powered investigation on the most suspicious players, identifying not just antisocial players but also those players employing automation to cheat. This allows the developer to institute harsh disincentives like systematically segregating adversaries from legitimate players, confiscate ill-gotten gains, and impose temporary or permanent bans. Furthermore, if game developers insist on using signature-based automation detection, a reputation system would give developers a more efficient method for finding and cataloging cheats being used.

3.2 THE PROBLEM WITH ONLINE BEHAVIOR

Like the users of many other network applications, players of multiplayer online games interact with each other using aliases which may be graphically represented by customizable avatars. An alias hides the player's real-world identity and allows them to be immersed in the virtual world. Players invest considerable time, effort, and even money improving the noteworthiness of their in-game persona – taking pride in any renown that they achieve (colloquially referred to as "*e-fame*").

Unfortunately, the anonymity provided by aliases also leads to a number of antisocial behaviors [113]. Of particular concern are players who cheat by automating the acquisition of wealth and items, or automating their actions to misrepresent their abilities and receive false recognition from peers. Other examples include players who scam their peers (often exploiting a poorly designed facet of the user-interface) to steal virtual wealth or items. Sometimes powerful players "*grief*" weaker players by playing in a manner that makes the game unenjoyable for them (e.g., "*corpse-camping*" a victim who must resurrect at the same location). Disgruntled players harass their peers through in-game text or voice chat. Some players simply act selfishly with poor sportsmanship (e.g., "*rage quitting*" the game when it greatly inconveniences their peers).

Although these behaviors are against the spirit of the game and many of its rules, limited policing resources mean that only the most grievous infractions are ever investigated. Often the virtual community is so large that a malicious player may negatively affect many peers before sufficient complaints are raised to warrant developer attention. Even then, the game rules regarding in-game actions, especially social behavior, are so vague that infringing players are given several warnings before incurring significant repercussions. Innocent players are thus posed with a dilemma: either they only play with peers they know and trust "*in real-life*" (i.e., from outside the game) or they must risk playing with peers they meet in-game who may behave poorly.

3.2.1 The Case for Reputation Systems

Reputation systems are commonly used in Web applications where the service provider acts as a broker between participants. Often those parties are selling items or services as independent vendors (e.g., eBay auctioneers and Amazon affiliates). Increasingly, reputation systems are being used in applications where clients do not have commercial relationships with each other, but instead judge the veracity of information shared by one another (e.g., Slashdot.org forums).

The appeal of such systems is that they grant clients self-determination about who they will interact with or trust, yet are relatively simple for service providers to implement. At the same time, reputation systems indentify clients who exhibit exceptionally negative behavior so that moderators may efficiently investigate them. This reduces an application's reliance on trouble-ticket systems which preoccupy game moderators with assuring complainants that the perpetrators are being investigated rather than focusing all their efforts on investigation. As discussed in more detail later, reputation systems could incorporate information from other sources, such as an anomaly-based detector.

Definitions

The complete set of game players is denoted by $Players$. A *rating* is the value corresponding to the opinion of one player (the *rater*, $i \in Players$) about another player (the *ratee*, $j \in Players$) and is denoted by $r_{i,j} \in [-1.0,\ 1.0]$, where a negative rating corresponds to dislike or distrust, and a positive rating corresponds to like or trust. A rating of 0.0 represents an unknown or neutral relationship.

The ratings for a given ratee j are combined to formulate their *reputation*, denoted by $R_j \in [-1.0,\ 1.0]$ with a similar interpretation of affinity. A rater's reputation dictates how influential their ratings are in determining the reputation of their peers: disliked raters will have little or no influence while liked raters will have more influence.

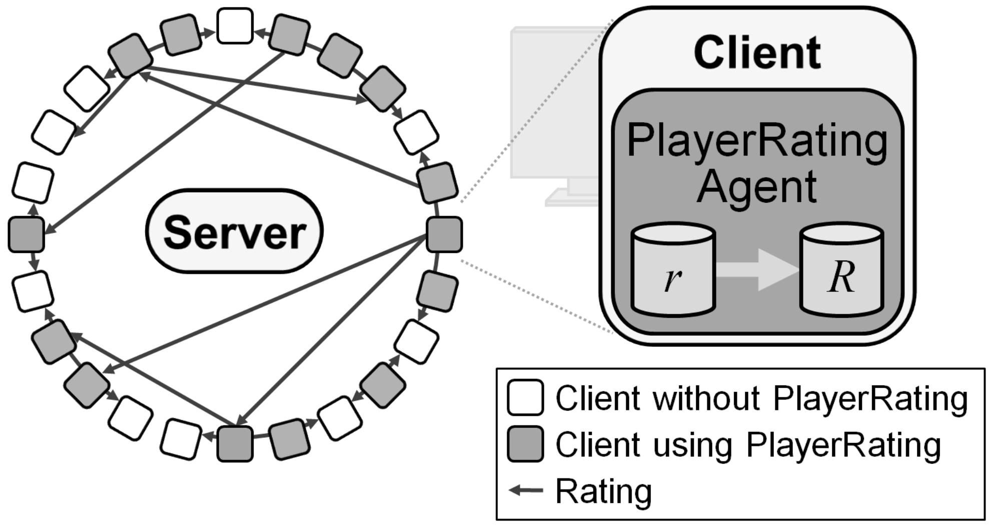


Figure 3.1: Addition of PlayerRating to the game system at large. Each client using a PlayerRating agent combines ratings from liked (i.e., trusted) peers to form a personalized view of reputations regarding all other peers.

3.3 THE PLAYERRATING APPROACH

Our approach is the PlayerRating system[1] which is, to our knowledge, *the first distributed reputation system specifically designed for online games.* The intuition behind the system is that a person trusts the opinions of their friends and friends-of-friends about unmet peers. Furthermore, each opinion is weighed based on their respect for the friend claiming the opinion; close friends will have more influence than mere acquaintances. The system (shown in Figure 3.1) has participating players run PlayerRating agents within their game clients. Each agent enables a player to rate peers and leverage ratings from liked peers to determine the reputability of unmet peers. The player-specific reputations are bound to in-game aliases (preserving real-world anonymity) and provide a best effort prediction of the type of behavior one might expect from one's peers. This allows well-behaved players to more easily congregate and avoid antisocial players.

[1]This research appeared in NetGames 2009 [65].

The novel features of PlayerRating are:

Experience Sharing. While some in-game mechanisms exist for recording social relationships (e.g., friend-list and ignore-list) that data is never propagated. PlayerRating transparently shares player relationship data using strictly authenticated in-game channels to avoid Sybil attacks [26]. Sharing data allows a player to warn peers about malicious peers. In return, peers rate other peer's behavior which allows the player to learn of malicious peers without having to personally encounter them. This reduces the likelihood that a legitimate player will inadvertently interact with a malicious peer.

Personal Perspective. Using shared relationship data, PlayerRating locally learns the game's social network from each player's perspective: players determine where they fit into the network by rating peers that they like or dislike. The system propagates trust through positive ratings to predict the player's perceived reputation of peers that they have yet to meet. As we show in the evaluation (and have observed in practice) the system works well without knowing every rating, but improves as more ratings are learned.

Fine Granularity. Existing in-game social tools are too coarse, restricting player relationships to like, dislike, or ambivalence. In the PlayerRating system, player relationships are analog, allowing a player to express their like or dislike of each peer to different degrees on a Real-valued spectrum. As such, the PlayerRating system could sort and better match players for group play, and focus expensive non-automated policing resources on the most disruptive players.

Distributed Operation. The PlayerRating system operates in a distributed fashion: each participating player runs a PlayerRating agent within their game client that determines the reputation of their peers, accounting for their personal perspective based on who they like and dislike. Participation is optional in the sense that a player may choose not to rate their peers, however, they cannot prevent their peers from rating them. Furthermore, a player must accurately rate their peers before the agent can accurately calculate meaningful peer reputations.

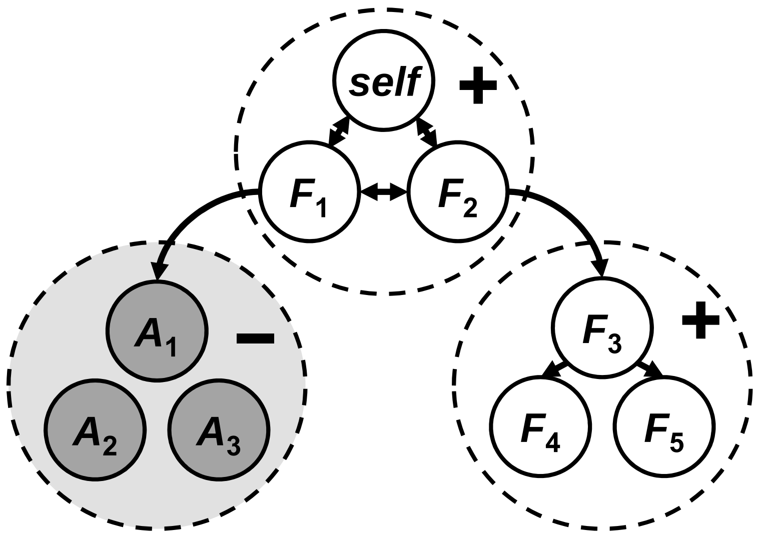


Figure 3.2: Example social network of players. The observer (*self*) belongs to a clique with friends F_1 and F_2. Friend F_1 dislikes A_1 so the observer will probably also dislike A_1 and her clique. Friend F_2 likes F_3 so the observer will probably like F_3 and her friends F_4 and F_5, although with less certainty.

Assumptions

An assumption underlying all reputation systems is that past performance is a reliable indicator of future behavior. However, another assumption at the core of reputation systems is that players may reform and improve their conduct at a later time although this will not excuse past transgressions. The player population forms a weighted directed graph (like the example shown in Figure 3.2) where each vertex represents an individual player (*uniquely identified* by a player ID) and edges represent ratings. The following assumptions are made:

- *Ratings are subjective.* Each player is allowed only one rating for each peer yet the rating should encompass the rater's perception of every interaction with the ratee. How the rater judges interactions and assigns ratings is subjective and entirely up to them. To accommodate new interactions, a rating may be updated or withdrawn by the rater at any time.

- *Ratings are asymmetrical.* A related assumption is that any two players may have different opinions of each other and their peers. Even the very best of friends often have different opinions regarding their peers.

- *Raters are authenticated.* Interplayer communication passes through the game server in practically all online games. At the server, each message is authenticated with respect to the sender's (i.e., rater's) player ID. For this reason, the approach only accepts firsthand ratings and does not accept "hearsay ratings" (i.e., forwarded ratings) since they cannot be authenticated in the same way.

- *Players are self-esteemed.* It is assumed that all players always view themselves with the highest regard: as absolutely likeable and trustworthy so $R_{self} = 1.0$. Similarly, ratings from peers about themselves (i.e., $r_{i,j}$ where $i = j$) are ignored since they will always be 1.0.

- *Positive ratings are transitive.* Insofar as positive ratings represent trust, positive ratings and reputations are transitive. Specifically, a peer trusted by a trusted peer (i.e., a friend of a friend) becomes trusted, albeit with less certainty. The opposite may not be true, so the ratings from distrusted peers are not necessarily useful.

- *Ratings follow a power-law distribution.* Players will self-organize following a power-law distribution [2] like other social systems [9, 15, 72, 93]. Much of the graph will be sparse as most players will create few ratings – even in systems with monetary incentive (e.g., eBay) only 60% of users ever generate ratings [25]. However, the power-law-based "*small-world*" model of Watts and Strogatz [124] indicates that a few well-connected players will result in a short average path length between any two players (i.e., only a few degrees of separation). These few well-connected players are fundamental for the system to yield predictions for many of the peers a player may encounter.

Design Goals

Players in multiplayer online games compete over recognition and limited in-game resources. Often they will adopt any tools that make the game more enjoyable or help them outperform their peers. To be suitable for this application, a reputation system should facilitate more positive player interactions yet also prevent abuse. Specifically the system should:

- *Support Incremental Deployment.* As a new tool, the system must accommodate gradual adoption, otherwise no one will use it in the first place. Specifically, the system must calculate peer reputations as accurately as possible given only partial graph information. Graph updates may be sent frequently to quickly increase the utility of the system for new adopters.

- *Encourage Participation.* Players cannot be expected to rate every player they interact with, yet reputation systems become more accurate with more ratings. Related to the previous requirement, the system must encourage participation by being easy to use and immediately beneficial.

- *Resist Abuse.* While encouraging participation is important, preventing abuse is equally important. Malicious players should not be able to increase their own reputation, even through collusion. Otherwise ill-gotten reputation could be used to slander legitimate players, or worse, lure trusting players into scams.

- *Incur Minimal Overhead.* As the goal of the system is to improve the overall gameplay experience, it must not distract players at inopportune times or degrade game performance. This means the system should mostly operate in the background and be efficient in terms of computation, storage, and communication.

3.3.1 Agent Algorithms

The PlayerRating system operates in a distributed fashion: each participating player runs a PlayerRating agent within their game client that determines the reputation of their peers, accounting for their personal perspective based on who they like and dislike. This section illustrates system features using a World of Warcraft [13] user-interface implementation [60], however, the system can easily be adapted to other games and game genres.

The system is intentionally designed so that players may optionally participate; they may choose not to rate their peers but they cannot prevent their peers from rating them. Furthermore, a player must accurately rate their peers before their agent can accurately calculate meaningful peer reputations.

Each player rates their peers when it is convenient to do so, presumably while socially interacting with the ratee or shortly thereafter. Ratings are shared with peers and expired periodically. Using all currently known ratings, the PlayerRating agent calculates each ratee's reputation as the average of every rating about them, weighted by the influence (a function of reputation) of the corresponding raters. Ratings are not absolute, but instead express a ratee's reputation relative to that of the rater. Thus, a ratee cannot be more reputable than the most reputable rater who rated them, preventing collusion via positive feedback loops. These calculations are done in an iterative fashion (similar to how Google PageRank [87] operates) through repeated calls to the `UpdateReputations()` function.

Initialization

The `Initialization()` routine of a PlayerRating agent is executed only once, when the player first installs the PlayerRating agent. The routine simply involves zeroing all data (ratings, reputations, and related variables) and setting the player's own reputation R_{self} to 1.0. This routine need not be executed each time the player executes the game client because the data will persist between play sessions.

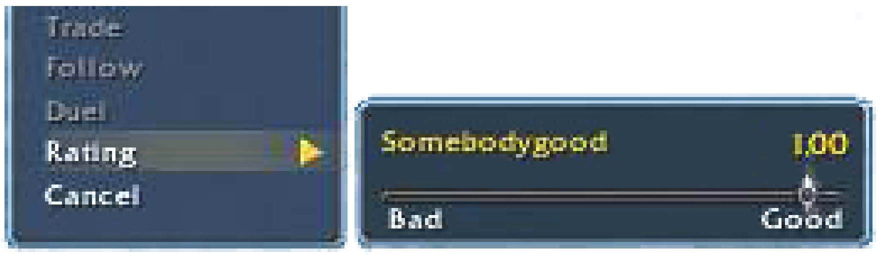


Figure 3.3: User interface addition for rating peers. The PlayerRating rating slide-bar was added to the peer interaction menu in World of Warcraft. The menu is only displayed when the player chooses to interact socially with a peer.

Recording Ratings

The PlayerRating agent unobtrusively extends the game's user interface (like the World of Warcraft interface addition shown in Figure 3.3) to enable the player to easily rate their peers. As defined earlier, ratings run along a single dimension and represent the rater's overall impression of the ratee. It is possible to extend ratings to other dimensions corresponding to criteria specific to player skills (e.g., how good the player is at a particular role like being the team healer $r_{i,j,healer}$) or specific behaviors (e.g., negatative actions like intentionally killing teammates $r_{i,j,team\_killer}$ or excessively swearing $r_{i,j,swearing}$). Adding extra dimensions would linearly increase system state, computation, and communication overheads. Some criteria may not be useful enough to justify such overhead and warrant further investigation.

The recording routine, `RecordRating()`, is used to record both personally created ratings and those ratings disseminated by one's peers. The freshness of a peer's rating $r_{i,j}$ is indicated by setting a corresponding time-to-live variable $ttl_{i,j}$ to the maximum value TTL_MAX and it may be expired if it later becomes irrelevant. Subsequent receipt of a previously recorded peer rating reaffirms the peer's commitment to that rating, so the rating is updated and the corresponding time-to-live value is reset to the maximum value. By doing so, relevant ratings survive the expiration process.

Sharing Ratings

Ratings are disseminated transparently via data channels which exist to support game modification ("*modding*") and operate similar to in-game chat channels. Only personally-created ratings are broadcast on the PlayerRating channels. All peers listening to the channel may record ratings. A peer may record ratings from an unknown rater (i.e., $R_i = 0.0$) with the idea that in the future either they or someone they trust might determine that the rater is also trustworthy.

Communication is strongly-authenticated since each message must go through the server which validates the sender's alias. To mitigate flooding, the server restricts message length to a couple hundred bytes and limits senders to a few messages every 10 seconds. PlayerRating agents may further limit the ratings they accept to control the growth of their knowledgebase.

Dissemination Policy. The `ShareRatings()` function executes a game-tailored policy to share ratings as quickly as possible and introduce some redundancy over time. Redundant broadcasts ensure that peers who may have been offline during the original broadcasts are notified and reaffirm the continued conviction in previously shared ratings (otherwise they will be expired and discarded by one's peers). Ratings must be shared in a fashion that minimizes the required communication, so retransmission should not be too aggressive.

Strategies used by the World of Warcraft PlayerRating implementation include: broadcasting a rating as it is created or updated, broadcasting the rating after interacting with the ratee, and periodically broadcasting ratings sequentially. In this game, ratings are broadcast to hundreds of players at once so the policy is careful to not overwhelm recipients. Other strategies may include broadcasting ratings randomly or through bulk broadcast when joining a zone, small server, or playgroup. These strategies are particularly useful for First Person Shooter (FPS) games where players join small servers of 8 to 32 players for short durations. In this case a more aggressive policy has less risk of overwhelming recipients.

Expiring Stale Peer Ratings

The persistent nature of online games means that a lot can change while a player is offline. In particular, a player's friends and peers may continue to play the game and interact with other peers whom the player should know about. This affects the PlayerRating system in that one's peers may change or revoke previously shared ratings, or quit playing the game altogether (obviating ratings created by or about them). To handle the possibility that the PlayerRating agent may obliviously possess stale peer ratings, the ratings collected from peers are slowly expired to ensure that each player's knowledgebase only contains relevant data.

Expiry Policy. The `ExpireRatings()` function is run periodically with period T_{dec}. The function ages ratings by decrementing their associated time-to-live values. If a time-to-live value reaches zero, the agent has not received any reaffirmation of the peer's conviction behind the corresponding rating, so it is expired and removed from the knowledgebase. Since players have accurate knowledge of their own ratings, those ratings do not need to be expired. The maximum lifetime of a peer's rating which is not reaffirmed is limited to:

$$\textit{maximum rating lifetime} = T_{dec} \times \text{TTL\_MAX} \tag{3.1}$$

As a part of the game client, the PlayerRating agent only runs while the player is online so rating lifetime is measured in terms of *played time* (referred to as "*pTime*" with units "*pHours*", "*pDays*", etc.). While both T_{dec} and TTL_MAX are currently default system settings set to a common value for all agents, players can individually adjust their local values to change the maximum lifetime of ratings, thereby controlling the amount of state kept by their PlayerRating agent. To be clear, locally modifying either T_{dec} or TTL_MAX does not affect in any way how the player's peers store, use, or share ratings.

By associating a time-to-live value with each rating, the PlayerRating agent is able to determine how old each rating is at any particular moment. The next routine exploits this to infer how recent and thus how relevant each rating is.

Algorithm 1 Pseudocode for the `UpdateReputations()` function.

```
1: $R', w \Leftarrow \varnothing$
2: for all $i \in Players$ do
3:     $w_\Delta \Leftarrow \texttt{Influence}(R_i)$
4:     for all $j \in Players$, $r_{i,j} \neq 0.0$ and $\neg\texttt{IsSelf}(j)$ do
5:         $R'_\Delta \Leftarrow (r_{i,j} \times R_i \times \texttt{Decay}(ttl_{i,j})) - R'_j$
6:         $w_j \Leftarrow w_j + w_\Delta$
7:         $R'_j \Leftarrow R'_j + (R'_\Delta \times \frac{w_\Delta}{w_j})$
8:     end for
9: end for
10: $R \Leftarrow R'$
```

Calculating Reputations

The core routine of the PlayerRating system, `UpdateReputations()` shown in Algorithm 1, uses all known ratings to determine the reputations of one's peers. The algorithm calculates the reputations R' and updates R only once the entire graph has been processed, meaning only line 10 must be atomic. As the bulk of the algorithm is non-atomic, the algorithm may be periodically and incrementally executed with low priority to avoid sudden computation spikes that may result in degraded gameplay at inopportune times.

One iteration of the algorithm loops over all the players (lines 2-9) and their ratings (lines 4-8). Each rater's influence is calculated (line 3) by a monotonically non-decreasing function of their reputation: the `Influence()` function. In the World of Warcraft implementation, this function calculates influence as the square of positive reputation and as zero otherwise:

$$\texttt{Influence}(R_i) = \begin{cases} (R_i)^2 & \text{if } R_i > 0.0 \\ 0.0 & \text{otherwise} \end{cases} \tag{3.2}$$

Using this influence function, disliked and unknown peers have no influence (all their ratings may be skipped) while liked peers have quadratically more influence.

For each non-zero rating where the ratee j is not the actual player running the agent *self* (ratings about the player are ignored), *the relative rating is decayed, weighted, and averaged* (lines 5-7) with all other ratings about the ratee used to formulate the ratee's new reputation R'_j. Averaging relative ratings ($r_{i,j} \times R_i$) means that a ratee cannot be more reputable than their most reputable rater and that reputations will always fall on the same scale as ratings (i.e., $[-1.0,\ 1.0]$). The ratings are weighted by the rater's influence.

More recent interactions (and thus ratings) are more important than older ones. Since the rating's age is already maintained via its time-to-live value, the time relevance of the rating is logically calculated using the `Decay()` function which should be monotonically non-increasing with respect to older (i.e., smaller) time-to-live values. The World of Warcraft implementation uses a linear decay to avoid large changes when ratings are expired and removed from the knowledgebase:

$$\texttt{Decay}(ttl_{i,j}) = \frac{ttl_{i,j}}{\text{TTL\_MAX}} \tag{3.3}$$

The `UpdateReputations()` routine is designed so that peers who are further away (i.e., have more degrees of separation) from the player will be more likely to have reputations that approach zero (i.e., unknown). The reasons for this are twofold. First, with more degrees of separation between any two people comes less familiarity and trust, so those peers really are relatively unknown. Second, this prevents distant peers with a few highly positive ratings from becoming disproportionately influential with regards to the ratings they create, especially when compared to peers closer to the player (i.e., one's close friends).

Furthermore, the routine is designed to equilibrate to stable values for all reputations as the routine is iterated. The routine requires iteration because the social network is in constant flux: ratings may be added, changed, or removed at any point in time by one's peers.

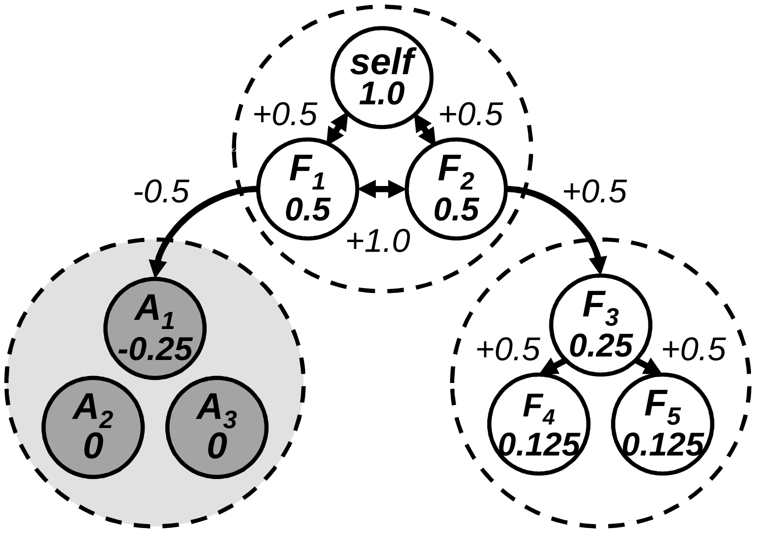


Figure 3.4: Iterating `UpdateReputations()` over the example social network. Labeled edges are ratings and labeled vertices are the reputations (from the point of view of *self*) achieved upon reaching equilibrium in three iterations.

Figure 3.4 numerically illustrates the example network from Figure 3.2 after reputations calculated by *self* have reached equilibrium. Before the first iteration, $R_{self} = 1.0$ and all other reputations are 0. After one iteration, the two peers closest to *self* have $R_{F_1} = R_{F_2} = 0.5$ and all other peer reputations remain at 0. In the second iteration, the PlayerRating agent calculates the reputation for A_1 by:

$$R_{A_1} = \frac{r_{F_1,A_1} \times R_{F_1} \times \texttt{Influence}(F_1)}{\texttt{Influence}(F_1)} = \frac{-0.5 \times 0.5 \times 0.25}{0.25} = -0.25$$

The positive reputation for F_3 is calculated similarly through F_2. The reputations for F_1 and F_2 only remain unchanged because of the example numbers used:

$$\begin{aligned} R_{F_1} &= \frac{r_{self,F_1} \times R_{self} \times \texttt{Influence}(self) \ + \ r_{F_2,F_1} \times R_{F_1} \times \texttt{Influence}(F_1)}{\texttt{Influence}(self) \ + \ \texttt{Influence}(F_1)} \\ &= \frac{0.5 \times 1.0 \times 1.0 \ + \ 1.0 \times 0.5 \times 0.25}{1.0 \ + \ 0.25} = 0.5 \end{aligned}$$

In the third iteration, F_4 and F_5 are discovered and none of the previously calculated reputations change. At this point, the network is discovered as fully as possible given the two ratings *self* created.

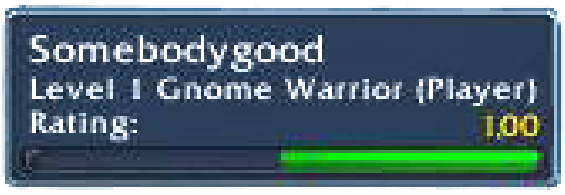


Figure 3.5: Peer reputation displayed in a tooltip. This World of Warcraft tooltip is displayed when the player places the cursor over the peer's character.

Reputation Lookup

The `LookupReputation()` function simply returns R_j (or 0.0 if the peer is completely unknown). It is helpful to remind the player of their rating for that peer at the same time by returning $r_{self,j}$. Both lookups are inexpensive and the information can be presented in many graphical and numerical ways, such as the addition to a player tooltip shown in Figure 3.5.

3.4 EVALUATION

The PlayerRating system design facilitates incremental deployment since a player does not need to rate peers and choosing not to participate does not prevent peers from rating them. The system also encourages participation since a player can only benefit from accurate peer predictions if they first accurately rate their peers. This section evaluates the system in terms of meeting the two measurable design requirements: resistance to abuse and low system overhead.

For evaluation, the system was implemented as an offline C++ application that was optimized for speed. This allowed experiments to be run without impacting real players and demonstrates how efficient the system might be if implemented directly within the game client. When implemented as a user-interface modification, it would be written in the game's scripting language. In World of Warcraft, that language is Lua which roughly uses the same memory footprint but executes at $\frac{1}{30}$ of the speed of C++ for equivalent programs [20].

Characteristic	Value
Ratees	30,000
Raters	3,919
1 ≤ outlinks ≤ 10	2,402
10 < outlinks ≤ 100	1,230
100 < outlinks	287
Ratings	101,842
positive	76,101
negative	25,741
Mean Ratings	3.4
positive	2.5
negative	0.9
Sparseness	0.000113

Table 3.1: Characteristics of the Slashdot Zoo subset. Raters follow a power-law distribution with respect to their outlinks.

3.4.1 Experimentation

To demonstrate the efficiency and collusion resistance of the PlayerRating system, experiments were performed on an emulated player population constructed using 30,000 identities from the Slashdot Zoo [72]. This is a reasonable number of player identities since World of Warcraft census indicate that realms support up to this many players [122]. Summarized in Table 3.1, this subset preserves the power-law characteristics of the original set.

Convergence to Equilibrium

This experiment shows how quickly PlayerRating agents converge to equilibrium in the worst case: when all ratings are learned instantly. In normal operation, fewer ratings update simultaneously due to limits of the communication channel. Within one agent, reputation instability is measured as the root-mean-square-difference ($RMSD$) between iterations of the `UpdateReputations()` function calculated by:

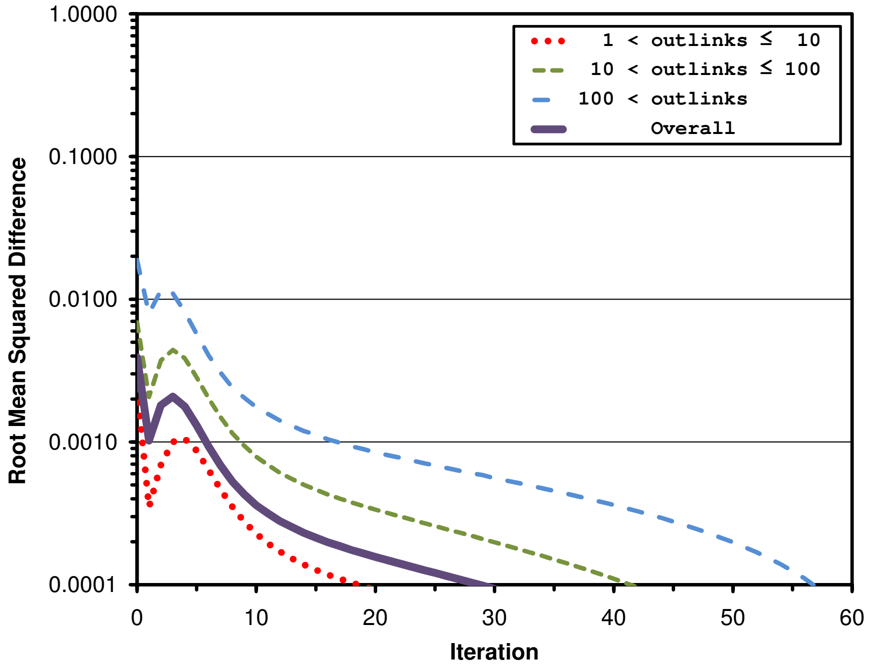


Figure 3.6: PlayerRating reputation convergence. The average *RMSD* for players with at least one rating, after instantaneously discovering the ratings graph, converges towards equilibrium as the `UpdateReputations()` function is iterated.

$$RMSD = \sqrt{\frac{\sum_{i \in Players} (R'_i - R_i)^2}{|Players|}} \tag{3.4}$$

where a value of zero means the system has completely reached equilibrium and any other value indicates that some reputations are still in flux.

Figure 3.6 shows the average *RMSD* for various players with at least one rating. The results indicate that overall the average participant system converges quickly. Players who rate fewer peers converge quickly since their connected graph is generally smaller, while players who rate many peers converge more slowly since their connected graph is larger. These results mean that malicious peers cannot generate a number of ratings that would cause instability. The small bump occurs at iteration 3 because that is the first iteration that can uncover a cycle in a completely new graph, possibly propagating trust back to a known peer.

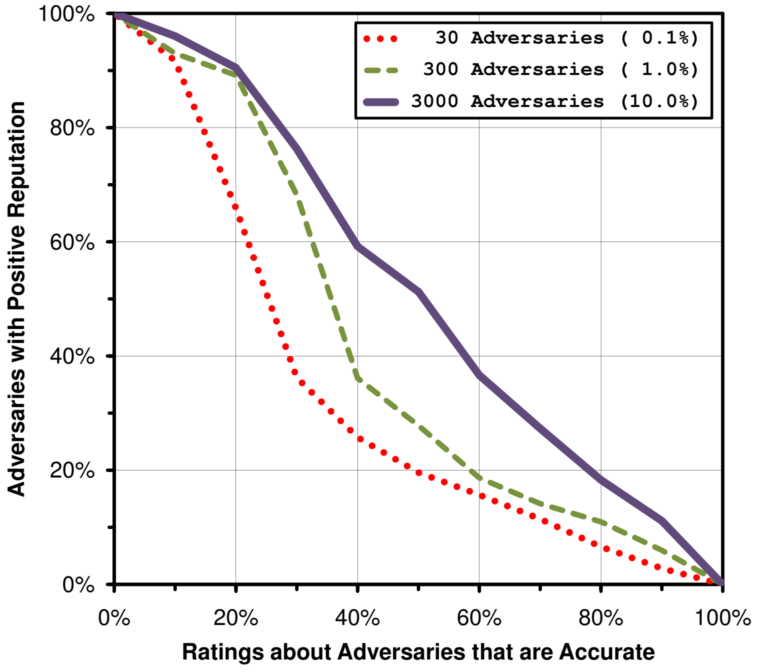


Figure 3.7: PlayerRating collusion resistance. The percentage of colluding adversaries that incorrectly obtain positive reputation as viewed by the average player is plotted against the percentage of accurate (i.e., negative) ratings about individuals in the adversary population.

Collusion Resistance

This experiment shows the ability of PlayerRating agents to accurately identify and isolate a population of colluding adversaries (i.e., correctly assign them negative reputation). Initially, the colluding adversaries join the game and immediately establish a fully connected positive rating sub-graph among themselves. Then, the adversaries play in an entirely legitimate fashion in order to obtain positive ratings from the original player population (assigned randomly according to the power-law distribution and sparseness of the original ratings). The leftmost data plotted in Figure 3.7 shows that at this early stage all the adversaries have obtained positive reputation as expected because they have not yet acted maliciously, betraying their earned trust.

As adversaries begin to betray their previously-earned positive reputation by committing malicious acts, they acquire negative ratings (which reverse some previously-obtained positive ratings). As the percentage of positive ratings decreases, the percentage of negative (i.e., accurate) ratings increases and fewer adversaries retain positive reputations. Eventually, the last adversaries abuse their earned trust and are re-identified with negative ratings, leaving no adversaries with positive reputations.

The figure shows that larger colluding populations retain positive reputation longer than smaller colluding populations. This occurs because the colluding population is fully connected, meaning that their rating links grow quadratically and quickly outnumber the original rating links. In the largest adversary population of 3,000 colluding adversaries (which represents 10% of the original player population) the rating links vastly outnumber the original links: *adversary outlinks* $= 3,000 \times 2,999 = 8,997,000 = 88.3$ times the original *outlinks*.

Computation Overhead

The PlayerRating agent runs within the game client process which requires fast execution to minimize any impact on the game's performance. Specifically, the agent absolutely must not affect the game's playability at critical gameplay moments. To measure the PlayerRating agent's speed, benchmarks of its various functions were performed on an Intel Core 2 Quad system (Q6600/2.4GHz). The results shown in Table 3.2 each represent an average of 10,000 executions. Most of the agent's functions operate on the order of μs. The slowest operations are merely on the order of ms. They are the `ExpireRatings()` function which must iterate over all stored ratings to decrement their time-to-live values and the portion of the `UpdateReputations()` function that commits R' to R. Fortunately, these both of these operations are infrequently performed (at periods greater than a $pHour$) and may be scheduled to coincide with the next logon, loading screen, or idle time when they will impact the player's gameplay the least.

Function	Theoretical	Experimental	
	Bounds	Cycles	Time
`Initialization()`	$O(1)$	473	$0.2\mu s$
`RecordRating()`	$O(1)$	33,539	$14.0\mu s$
`ShareRatings()`	$O(outlinks)$	*policy dependent*	
`ExpireRatings()`	$O(\lvert r \rvert)$	30,267,923	$12.7ms$
`UpdateReputations()`	$O(\lvert r \rvert)$	18,300,514	$7.7ms$
process rating	$O(1)$	300	$0.1\mu s$
process rater	$O(\lvert Players \rvert)$	26,089	$10.9\mu s$
commit R′	$O(\lvert Players \rvert)$	10,047,723	$4.2ms$
`LookupReputation()`	$O(1)$	752	$0.3\mu s$

Table 3.2: Efficiency of PlayerRating routines. The theoretical bounds and experimentally measured computation cycles (also expressed as time for analysis) to execute the various routines of a PlayerRating agent are given.

Memory Overhead

With many gaming machines having gigabytes of physical memory and game client's leaving so much virtual memory unused, having a small memory footprint is less important than being fast. While memory overhead is a lesser concern, PlayerRating agents require extremely little state. Specifically, neutral/unknown ratings (i.e., with value 0.0) are not useful and need not be stored. Furthermore, ratings from peers with negative reputations will not be used so they can be discarded. Similarly, ratings about the player will have no bearing on any of the algorithms (recall $R_{self} = 1.0$ always) so they need not be stored either.

In summary, a PlayerRating agent's state is reduced simply to the number of relevant ratings kept and the peer reputations. This is far less than the square of the number of players due to the sparseness of the social graph:

$$total\ state = |r| + |R| \ll |Players|^2 \tag{3.5}$$

For example, the ratings and reputations throughout the evaluation merely require 2.4MB of application memory.

3.4.2 Limitations

The current limitations of PlayerRating involve changes to player account information that is not public and therefore is not accessible by a user-interface modification. Specifically, a player may quit the game, transfer their character to another server, or rename their character (although only to another unique name). While large-scale Sybil attacks are not possible due to restrictions on how frequently these actions may be performed, some system inaccuracy (i.e., rating duplication may exist until those ratings expire). Such temporary inaccuracies could be avoided if the system was aware of relevant changes to peer accounts and peer ratings made while the player was offline. This would obviate the need for rating expiration in the first place.

External to the game, peers sometimes sell their accounts for profit, although this practice is often against the game's Terms of Service Agreement. Characters changing ownership in this fashion may have an abrupt change in behavior, making existing ratings about them obsolete. Players with positive ratings about these peers may be briefly misled until those ratings are corrected.

Finally, adversaries may attempt small-scale Sybil attacks [26] by creating additional game accounts. In general, this is prohibitively expensive because creating a game account involves purchasing a copy of the game and paying a subscription fee. However, some games like World of Warcraft offer temporary trial accounts (10-day trial accounts are already abused for gold spamming) which may facilitate short-term Sybil attacks. This may be addressed by incorporating the character's level in the `Influence()` function (i.e., ratings from high-level characters are more relevant than ratings from low-level characters). This would immediately mitigate trial accounts since those accounts expire before an adversary could reach maximum character level (it takes roughly 300 $pHours$ to reach maximum level). If PlayerRating were built into the game, ratings from trial accounts would be easy to ignore and need not be propagated at all.

3.4.3 Discussion

There are several possible PlayerRating applications. Currently the system is publicly available as a World of Warcraft mod [60] and may be adopted by any player willing to do so. We used the system (along with in-game and real-life friends) for a number of years before quitting the game. It is our experience that the system successfully warns players when first interacting with peers who have behaved badly in the past. If implemented within game algorithms that match players for group play, reputation scores could be compared to weigh the likelihood that the group will get along well. In this sense, PlayerRating could completely replace existing friends-list and ignore-list tools by simply treating peers with positive ratings as friends and ignoring peers with negative ratings.

Platforms that support multiple game titles (e.g., Sony's PlayStation Network or Valve's Steam) may trivially build a recommender system on top of PlayerRating to focus online marketing at players with friends (i.e., peers they rate highly) who enjoy other titles or specific DownLoadable Content (DLC). This would effectively state: "your friend from game X also plays game Y, you might enjoy it too." Furthermore, such a system could recommend User Generated Content (UGC) – especially if a rating dimension was applied to modders ($r_{i,j,modder}$) regarding the quality of their creations. Ultimately, this may become an effective approach to reduce a game developer's time and expense in creating content for new games.

Reputation systems like PlayerRating may be used to federate adversary detectors in other applications where the clients do not directly interact with and rate each other. For example, independent Web services may use such a system to learn which of its clients are bots participating in a denial-of-service attack. In this setting, the system may incorporate ratings from peers and input from other detectors like Fides. Since these sources of information are not being rated by the same criteria with which they rate clients, the system could augment the `Influence()` function to account for the historical accuracy of each rating source.

3.5 RELATED WORK

Existing In-game Tools. Many games offer simple tools for tracking peers that a player likes (e.g., friend-list) or dislikes (e.g., ignore-list). Extensions like PlayerNotes [86] add annotation capabilities to those tools so that the player may be reminded why they befriended or ignored peers. While these tools are easy to implement and are present in many online games, they do not share such preferences and thus cannot predict whether or not a player will like a peer they meet in-game for the first time.

Reputation and Recommender Systems. A common approach to improving online social interaction is the use of reputation systems, such as eBay's Feedback system [29] or Slashdot's Karma system [105]. Related to reputation systems are recommender systems (where value is assessed for objects rather than people), the most notable of which is the PageRank system that powers the Google search engine [87]. The PageRank system determines the popularity or rank of a webpage as the sum of all supporting evidence (i.e., hyperlinks pointing to it) weighted by the rank of the referring webpages and a decay factor. The algorithm iteratively updates each webpage's rank through random walks of the Web, eventually reaching Web-wide equilibrium. So long as a webpage has many references from popular pages, it will also have a high PageRank.

Deployed reputation and recommender systems, like PageRank, evaluate the global perception of persons or objects yielding rough predictions that do not account for personal preference. For personalization, some approaches (e.g., TrustRank [46] and Personal PageRank [57]) extend PageRank while others (e.g., EigenTrust [68] and email filtering [45]) implement systems that operate in a distributed fashion. Personalized approaches allow people with similar interests (e.g., players who dislike profanity) to congregate and form cliques. Such approaches record peers that one likes or dislikes, and extrapolates personal and shared ratings to predict like and dislike for peers not personally met.

The PlayerRating approach is architecturally similar to the PageRank system, but is computed in a distributed fashion. PlayerRating calculations are tailored to better suit online games and direct exposure to players. PlayerRating leverages both positive and negative ratings, as well as bounding reputations, so that they remain on a fixed scale and can be more easily interpreted by players.

3.6 CONCLUSION

In many networked applications, clients interact using aliases which unfortunately enable a number of antisocial behaviors, including automation. This chapter investigated the problem of identifying adversaries using a federation of abundant, yet sometimes inaccurate and individually inconclusive detectors in the context of identifying misbehaving players in multiplayer online video games. This application is ideal for such research since established disincentives exist and treating players as detectors harnesses abundant information sources. The research contributions of this chapter are:

- We briefly *outlined the problem with online behavior.* Basically anonymity in an (even indirectly) competitive application or environment invites participants to behave badly towards one another. Fueled by financial incentive, automated and adversarial behaviors become ever more common.

- We *proposed and evaluated a novel reputation system for multiplayer online games.* The PlayerRating system runs an agent within each participating player's game client and extends the user interface to noninvasively allow the player to rate their peers. Using the player's ratings along with the ratings shared by their peers, a PlayerRating agent calculates reputations for all peers from that player's perspective. By design, the system facilitates incremental deployment and encourages participation. In evaluation, we showed that the PlayerRating system resists abuse and imposes minimal overhead.

There are a number of ways which a reputation system could be used in multiplayer games. Besides the original purpose of identifying malicious players to avoid, such a system could provide the basis for focused in-game advertising and evaluating both downloadable and user-generated content. Furthermore, reputation systems like PlayerRating could be used to federate adversary detectors in other applications where the clients do not directly interact with and rate each other.

The next chapter discusses how an application service provider may leverage an analog metric (like player reputation) to best disincentivize automated adversaries of Web-based applications. That research focuses on maximizing the application's service for legitimate clients while minimizing service granted to mostly adversarial clients.

Chapter 4

DISINCENTIVIZING ADVERSARIAL AUTOMATION

4.1 INTRODUCTION

The previous chapters explored approaches to detect automated behavior and combine detectors in order to identify application adversaries. Applications that authenticate clients through some form of real-world credibility (e.g., multiplayer online video games which bind player accounts to a credit card or software purchase) can immediately leverage these approaches to penalize and thus disincentivize malicious behavior. In contrast, applications which are susceptible to adversarial automation before they can conveniently authenticate the client (e.g., Web-based commerce) cannot leverage these approaches as easily and must establish a disincentive mechanism which may work even in the absence of strong client authentication. This chapter investigates how to do this.

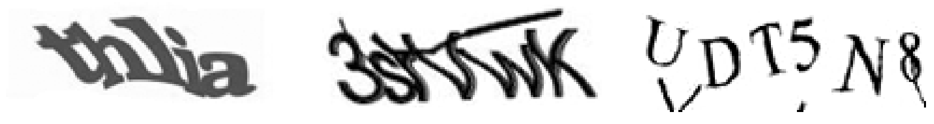
We explore a Proof-of-Work (POW) approach that leverages the various available information sources (including client geographic location) to differentiate clients and thus disincentivize adversarial automation behavior in the context of resource consumption attacks on Web-based applications. Most Internet commerce is done through Web-based applications so they experience a wide variety of resource consumption campaigns against them such as denial-of-service, ticket scalping, comment spam, and click fraud. Web applications are ideal for adversary disincentivization research since they have access to many sources of information about client behavior including load metrics and network address blacklists yet lack effective disincentive mechanisms. We show that our proof-of-work approach is efficient, transparent to human users, and straight-forward to implement.

4.2 THE RESOURCE CONSUMPTION PROBLEM

Service providers use Web-based applications to distribute ideas or sell goods. Rationally, a service provider only provisions enough computation and network resources to handle the expected client load for the application, with some extra to spare. Unfortunately, in most Web-based applications there is a resource imbalance between the server and clients, both in terms of the resources each have available and that each must commit in order to complete a transaction. Specifically, a client may simply send a single request (i.e., network packet) to initiate the protocol, however, the server may need to retrieve data from disk or a database, perform a non-trivial computation, or generate a webpage from dynamic content.

In times of unexpectedly high legitimate interest (referred to as "*flash crowds*" or as the "*Slashdot effect*"), it is possible for the server to be overwhelmed and cease to adequately service any requests. This underscores the resource consumption problem: with or without it being their motivation, automated adversaries (i.e., large botnets) can easily consume all the Web server's resources, often in early stages of the application protocol. Automated attacks on Web-based applications remain a significant problem on the Internet, despite vast efforts to combat them. Examples include comment spam on Web-based forums [89], ticket-purchasing robots [110], click-fraud robots, and denial-of-service attacks.

There are two disadvantages that a Web server has in dealing with this problem. First, network clients are weakly authenticated; it is often not until the very end of a transaction that the client's real-world identity must be revealed, so it is difficult to stop adversaries from consuming server resources in the early stages of the protocol and aborting just before they must commit their own resources or identity. Second, Web applications seek a general audience which is not known a priori, so the server expects that most clients engaging in the protocol intend to complete the transaction.



a) Google b) Yahoo! c) Microsoft Live

Figure 4.1: Unintelligible CATPCHAs. Found on three of the largest Internet websites, these examples represent the sole challenge to creating an online account.

4.2.1 The Shortcomings of CAPTCHAs

A common approach to address the automated resource consumption problem is the use of image-based Completely Automated Public Turing Tests to Tell Computers and Humans Apart (CAPTCHAs) [117]. Each CAPTCHA is a small server-generated image consisting of skewed representations of letters and numbers (like the examples shown in Figure 4.1) that a user must correctly interpret before they are granted access to the protected service.

There are several disadvantages of using CAPTCHAs. One drawback is the user-interface problem they impose [70]; users who are visually impaired are unable to solve the challenge and are thus denied access to the service. Even visually unimpaired users find it increasingly frustrating to solve challenges correctly because they are becoming less readable in order to thwart automated solving software. A second drawback is that automated solvers are improving at an alarming rate and can efficiently defeat most human-readable CAPTCHA constructions [49]. The final and probably most important drawback is that the fixed-difficulty challenge that CAPTCHAs represent fatally limits their utility. CAPTCHAs are designed so the typical user can solve one in roughly 10 to 20 seconds. Enterprising adversaries have outsourced the solving of CAPTCHAs to low-cost foreign data entry specialists who will solve 1,000 challenges per hour for under $3 [43]. From an economic standpoint, the inability to change the "price" of access makes the approach unable to protect valuable online resources [75].

4.2.2 The Case for Proof-of-Work

Proof-of-Work (POW) protocols are an alternative to CAPTCHAs. A few proof-of-work protocols have been proposed in the literature [7, 8, 24, 28, 35, 121, 123]. A proof-of-work (also known as "*cryptographic puzzles*" or "*client puzzles*") protocol extends a client-server request-driven application protocol so that the server issues prospective clients computational challenges of client-specific difficulty. Each client must solve their own computational challenge and return a correct answer to the server before they will be granted service. Every POW protocol is designed such that issuing and verifying challenges is trivial for the server to do, yet solving the challenges is arbitrarily difficult (as set by the server) for the clients to do.

The challenge-response nature of POW protocols makes them similar to CAPTCHA protocols except that POW challenges are solved by the client machine rather than the human user. In removing the human from the protocol, POW protocols may more readily differentiate service on a per-client basis according to the likelihood that each client is adversarial (i.e., through demonstrated behaviors or other detection methods). Clients that are more likely to be adversaries are given extremely taxing challenges to solve, while clients unlikely to be adversaries are given easy challenges to solve. In contrast to the uniform price applied to all clients in CAPTCHA protocols, the server can accurately price individual access to the service using a POW protocol.

While POW approaches are highly configurable in terms of workload, few have made progress towards being deployed. The biggest hurdle to adopting the previously proposed approaches is that they require changes to standard protocols and wide-scale adoption of special client software in order to operate properly, denying all clients who have not installed it. This chapter describes kaPOW, a novel POW approach that is transparent to clients, and POW difficulty policies for disincentivizing automated adversary behaviors using two different information sources: per-client request history and client geolocation.

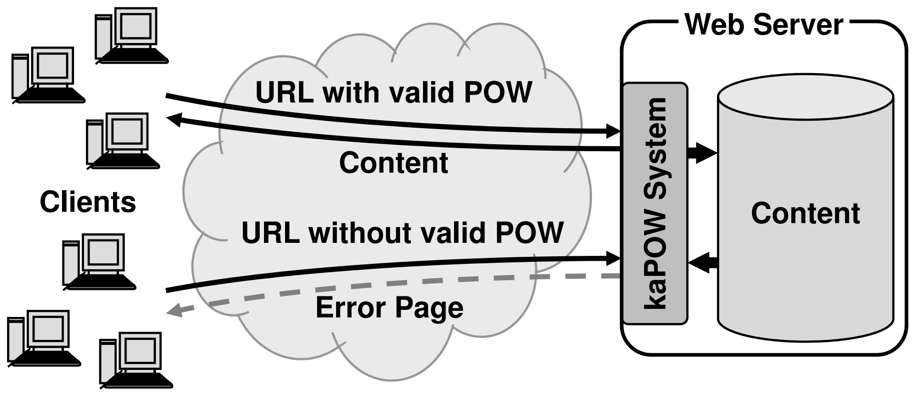


Figure 4.2: Addition of kaPOW to Web server. Clients who attach a valid proof-of-work solution to their requests are granted the content they seek, while clients without a valid solution are ignored and issued a new proof-of-work challenge, server resources permitting.

4.3 THE KAPOW APPROACH

Proof-of-work approaches proposed in the literature [7, 8, 24, 28, 35, 121, 123] have demonstrated the feasibility of implementing proof-of-work systems at the network and transport layers. While the approaches have experimentally proved their effectiveness against portscans and network-level packet-flooding denial-of-service attacks, they have all met resistance to adoption due to one fundamental detraction: they require both clients and servers to adopt specialized software in order to adhere to the protocol.

This led us to develop the kaPOW approach which is the *first transparent proof-of-work approach in literature or in practice* (it currently protects the project's webpage [62]). The kaPOW approach (shown in Figure 4.2) protects Web-based applications by incorporating a simple proof-of-work module at the server. To make the approach transparent to clients, the approach leverages JavaScript which is nearly ubiquitous as recent measurements have shown that JavaScript is enabled in upwards of 95% of all Web browsers [118].

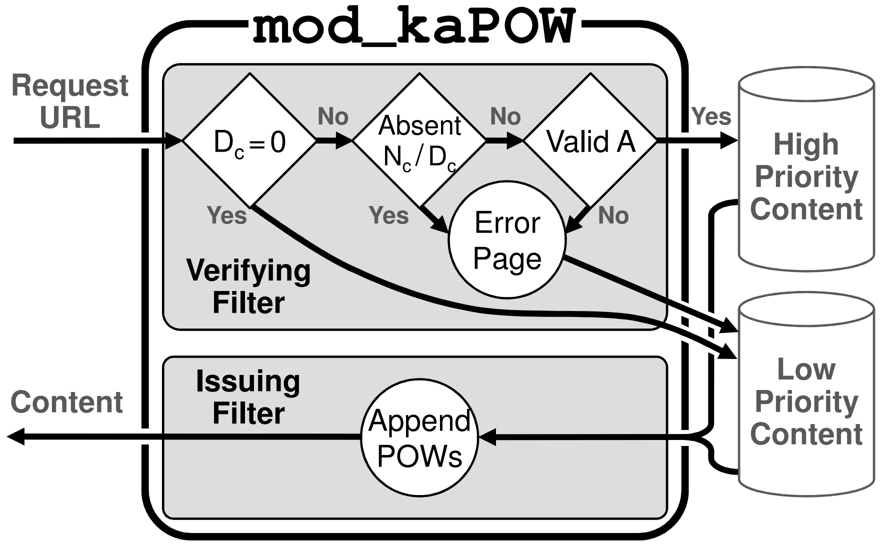


Figure 4.3: Internal structure of `mod_kaPOW`. The system is an Apache module which processes POW-protected URLs and the corresponding content.

The kaPOW approach was implemented as `mod_kaPOW`[1], the Apache [5] Web server module shown in Figure 4.2. The system protects Web content by embedding POW challenges ("*work functions*") in Uniform Resource Locators (URLs) as new query parameters. As webpages are served, the URLs found in any HyperText Markup Language (HTML) tags are updated to include a fresh work function.

When a client's Web browser encounters a POW-protected link, it runs a server-provided JavaScript routine to solve the work function and append the solution to the URL before attempting to follow the link. As elaborated upon later, clients without JavaScript (or with it disabled) are not necessarily prevented from accessing the content.

Upon receiving a request, the server verifies that the URL contains a valid work function and solution before servicing it. If either the work function is stale or the solution is incorrect, the server denies the request and returns an error page containing a link with a new challenge.

[1]This research appeared in a paper at Global Internet 2008 [63].

Work Function	Server Effort Issue()	Server Effort Verify()	Client Effort Solve()	
Exponential				
Time-Lock	$2,500\mu s$	$1.1\mu s$	*deterministic*	D_c
Hash Collision				
MicroMint	–	$D_c \times 1.1\mu s$	*probabilistic*	$D_c \times \lvert range \rvert$
Hash Reversal				
Basic	–	$1.1\mu s$	*probabilistic*	2^{D_c}
k Sub-Puzzles	$1.1\mu s$	$k \times 1.1\mu s$	*probabilistic*	$k \times 2^{D_c}$
TLS	$1.1\mu s$	–	*probabilistic*	2^{D_c}
Hashcash	–	$1.1\mu s$	*probabilistic*	2^{D_c}
Hint-Based	$2.1\mu s$	$1.1\mu s$	*probabilistic*	D_c
Targeted	–	$1.1\mu s$	*probabilistic*	D_c

Table 4.1: Comparison of work function constructions. All of them, save for the Time-Lock construction, are hash based and use probabilistic solution algorithms. The Hint-Based and Targeted Hash Reversal constructions are our contributions.

The Work Function

Before describing the internal operation of the system, we present the construction of the work function employed by this approach. It is important to note that while this work function is currently the best candidate for the kaPOW approach, the approach is not dependent on this work function and could readily adopt another work function construction if necessary for a particular Web application.

There are a number of work function constructions throughout the literature, most of which are hash based. Table 4.1 compares the work functions in terms of the effort to `Issue()` challenges and `Verify()` solutions, as well as the number of "*units of work*" (as a function of the client-specific difficulty D_c) required to `Solve()` them. For hash-based constructions, a unit of work is one execution of the hash function which takes $1.09\mu s$. For the Time-Lock construction, a unit of work is squaring one integer modulo a large pseudo-prime which takes $0.75\mu s$.

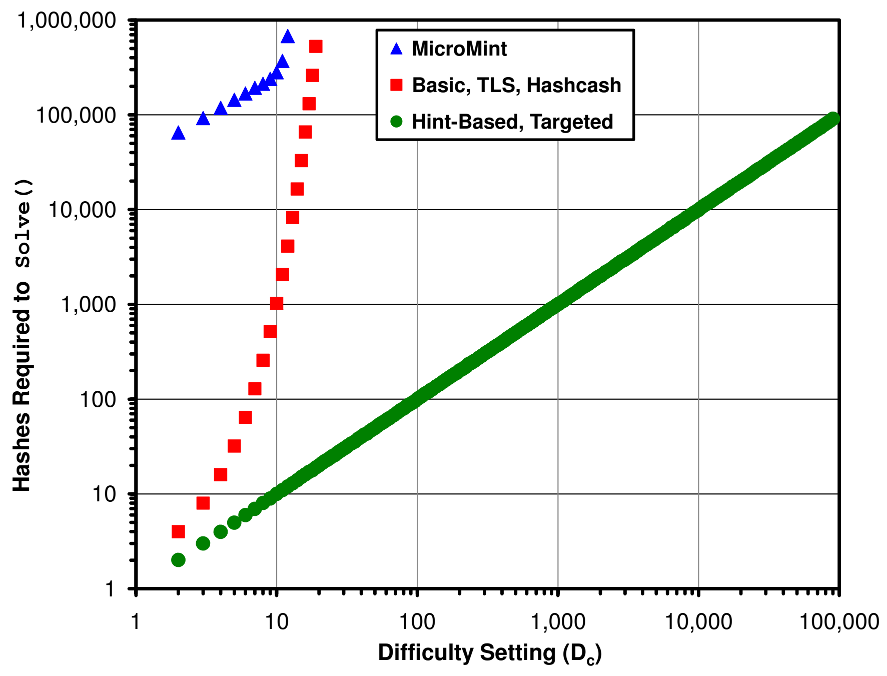


Figure 4.4: Comparison of hash-based work functions. Shown are the hashes required in practice to solve the various work functions across a range of difficulties.

The computationally expensive issuance of the Time-Lock work function makes the construction impractical for combating resource consumption attacks, in particular attacks that flood request packets. In contrast, hash-based work functions are easy to issue and verify. Figure 4.4 shows the effort required to solve the various hash-based work functions. Each data point represents the average number of hashes to solve $10,000$ work functions of that particular difficulty.

The MicroMint, Basic, TLS, and HashCash constructions require effort exponentially proportional to the difficulty D_c. At high difficulty values the coarse work-resolution prevents a policy from accurately setting a client's effort. Consider that after difficulty $D_c = 20$ (requiring 1 million hashes or roughly 1 second to solve) the constructions cannot issue work functions with intra-second granularity and after difficulty $D_c = 29$ (requiring 73 seconds to solve) the constructions cannot issue work functions with intra-minute granularity.

In contrast, the Hint-Based Hash Reversal and Targeted Hash Reversal constructions are able to issue work functions with fine-grained solution effort through the full range of difficulties. The Hint-Based construction achieves this by selecting a solution and giving a hint within D_c steps from that solution. This requires more effort to issue work functions than the Targeted construction, making it comparatively weaker when combating resource consumption attacks such as those based on request flooding.

For efficient issuance and verification, and finely targeted solution effort, the `mod_kaPOW` prototype employs our Targeted Hash Reversal construction[2]. The work function is of the form:

$$H(N_c \parallel D_c \parallel A) \equiv 0 \texttt{ mod } D_c \tag{4.1}$$

where H is a pre-image resistant hash function with output uniformly-distributed (e.g., SHA1 [84]), N_c is a client-specific nonce generated by the module, D_c is the client-specific difficulty set by the module, and A is the solution that the client's JavaScript solver must find. Since both N_c and D_c are fixed by the issuer and H is pre-image resistant, this work function requires the solver to perform a brute-force search to find a value for A that satisfies the equation. The probability that any given value for A satisfies the equation is $\frac{1}{D_c}$, and the expected number of attempts to find a valid value are geometrically distributed with a mean of D_c. This is experimentally validated in Figure 4.4.

This work function is a good candidate because it is efficiently implementable. Specifically, it requires no hashes to issue and an answer can be verified in a single hash. Using a hash function like SHA1 with a sufficiently large digest, this takes only $1.09\mu s$ in software [37]. Additionally, the work function can be expressed compactly: the issuer simply has to send D_c and N_c, and a verifier only needs those parameters and the solution A to verify that the equation is satisfied.

[2]This construction was presented in a paper at Global Internet 2007 [36].

The Server Module

The intelligence of the `mod_kaPOW` system is within an Apache Web server module. Apache provides a rich interface for writing modules that range from those that control how a client accesses a server (such as `mod_rewrite`) to those that dynamically generate content (such as `mod_include`). As a result, Apache lends itself well to supporting a proof-of-work module[3]. The module has two filters; an *issuing filter* that embeds work functions into outbound HTML content and a *verifying filter* that prioritizes inbound requests based on whether or not they contain a correct answer to a valid work function.

To prioritize requests, the server is configured with two virtual hosts. The default low-priority virtual host does not support persistent HyperText Transfer Protocol (HTTP) connections, and tears down any connection after servicing a single request. Only a limited number of low-priority requests are handled concurrently; all excess low-priority requests are rejected with HTTP error code `503: Service Temporarily Unavailable`. Any request demonstrating a correct answer to a valid work function is redirected to the high-priority virtual host which supports more concurrent requests and allows persistent connections. If a client sends a subsequent request with an incorrect answer or an invalid work function, the connection is transferred to the low-priority virtual host to be terminated.

Client-Specific Variables

During the course of their operation, the issuing and verifying filters refer to the client-specific difficulty D_c and nonce N_c. To establish D_c, the module uses a counting Bloom filter [14, 33] to track the load imposed (i.e., requests sent) by each client. The counting Bloom filter is an efficient data structure that offers a tradeoff between its size and the probability of incorrectly assigning a high difficulty to a client. It has no false negatives (i.e., a malicious client will never be issued a trivial

[3]The Apache module naming-convention inspired the name "`mod_kaPOW`".

work function) and the probability of a false positive can be driven arbitrarily low with additional memory.

Given that the Bloom filter uses k different hash functions to index into an array of n counters, the probability of misclassifying a single client from an estimated population of m clients is approximately $(1 - e^{-\frac{km}{n}})^k$. Using a value of k that minimizes that equation, the error is approximated by $0.6185^{\frac{n}{m}}$. Thus to achieve a misclassification rate of less than 0.1% of $20,000$ clients, the Bloom filter requires $288,000$ counters or a total of 1.2MB when using 32-bit counters. The Bloom filter is updated every 10 seconds so that the difficulty is held constant long enough to give clients a chance to respond, but short enough so that the difficulty can respond to sudden changes in load. When the structure is updated, each counter c in the filter is updated according to the following logic:

$$c_{t+1} = \begin{cases} c_t + r_t - \text{DECAY} & r_t \leq \text{DECAY} \\ c_t + 1.01^{r_t - \text{DECAY}} & \text{otherwise} \end{cases} \tag{4.2}$$

which states that the difficulty decays linearly from one time window to the next unless the requests r_t in the latest time period t are greater than the rate of DECAY, in which case those extra requests exponentially increase the difficulty.

The client-specific nonce N_c is created by concatenating the client's identity IP_c, the original unmodified URL, and a server nonce N_s:

$$N_c = IP_c \parallel URL \parallel N_s \tag{4.3}$$

binding the client nonce and thus the entire work function to the client and specific content for a fixed window of time. When the server nonce changes the client nonces also expire, meaning solutions cannot be reused indefinitely. The unpredictable server nonce prevents attacks where adversaries solve work functions offline and stockpile valid answers. The server can update its nonce independently from the Bloom filter and as frequently as needed to keep client solutions fresh, however, the prototype presently updates the nonce and Bloom filter simultaneously.

```
<HTML>
  <HEAD>
    <SCRIPT TYPE='text/javascript' SRC='kaPoW.js'></SCRIPT>
    <TITLE>Sample Content Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Sample Content Page</H1>
    This webpage demonstrates an image and link protected by
    proof-of-work.<BR><BR>
    <IMG SRC='test.jpg?Dc=0' Nc=a53b6145 Dc=10> are solved when
    the page is loaded to avoid delay.<BR>  In contrast,
    <A HREF='/?Dc=0' Nc=52a6c561 Dc=10>
    POW-protected links</A> are solved only when the link is clicked.
  </BODY>
</HTML>
```

Figure 4.5: HTML markup of modified example document. The link to the solution JavaScript file and the added work function variables are highlighted.

The Issuing Filter

The issuing filter scans and parses HTML documents as they are served. It adds work functions to all tags containing URLs as well as a link to the JavaScript solver necessary for the client's browser to solve the work functions. The modification to an example document is shown in Figure 4.5.

The issuing filter includes the solution instructions for work functions through the addition of a link to a JavaScript file (`kaPOW.js`) at the head of the document so that it is retrieved first (if not already cached) and the script may work as the remaining tags are incorporated into the client's in-memory Document Object Model (DOM). Despite containing a URL, this tag does not have a work function because clients need this resource before they can possibly solve any work function.

The issuing filter incorporates work functions into tags by adding the variables N_c and D_c as tag attributes. To avoid accidentally triggering HTML escape sequences, the values are transmitted in hexadecimal. It is important to observe that N_c differs between tags because it is calculated from the original unmodified URL in each tag. The filter also appends a default difficulty of zero ("`Dc=0`") to the actual URL so that clients without JavaScript enabled can follow the link and indicate to the server that they cannot solve work functions. Before sending the content, the issuing filter updates the Bloom filter to note the client's request.

The Verifying Filter

The verifying filter parses request URLs and extracts any appended proof-of-work variables. If the request URL contains the variables N_c and D_c, they must be verified as current and correct before the module does computationally expensive operations such as hashing. If N_c and D_c are valid, the verifier proceeds to check that A satisfies the work function. If everything checks out, the request is accepted by the high-priority virtual host and the content is sent to the client.

There are three reasons why a client's request might be rejected by the verifying filter: the URL has no work function or solution attached, the work function parameters are not current, or the attached solution is not valid. The first two failures may have occurred for a variety of legitimate reasons and are not necessarily indicative of a malicious client.

If the request URL contains no POW parameters, then the client may have been linked to this resource from an external server that has not yet adopted the kaPOW system and hence did not issue a work function to the client. It is also possible that the client arrived at this website by manually entering the URL into the address bar of their Web browser – users are not expected to know nor manually append work function parameters and a solution into URLs since it would be incredibly impractical for them to do so.

If the request URL contains POW parameters but they are invalid, the client may have been directed to this site from an external server that appended its own, different values for N_c and D_c. Alternatively, the user may have spent long enough reading the last webpage that the server has since updated its nonce N_s and has thus expired the stale client nonces N_c embedded in that webpage. The `mod_kaPOW` module ensures that old webpages which may be cached expire at the same frequency that clients do. This is done by updating the HTTP `ETag` value (a metadata string that represents the version of the webpage content) for webpages whenever the server nonce changes.

```
<HTML>
  <HEAD>
    <SCRIPT TYPE='text/javascript' SRC='kaPOW.js'></SCRIPT>
    <TITLE>Error: Invalid POW</TITLE>
  </HEAD>
  <BODY ONLOAD='Solve(document.links[0]);
                window.location.replace(document.links[0].href)'>
    <H1>Invalid POW</H1>
    The requested URL did not have a valid proof-of-work attached.<BR>
    If you are reading this page, it is likely that you do not have
    JavaScript enabled.<BR><BR> If you would still like to try to
    access the content, please click the following link:
    <A HREF='http://maes.cs.pdx.edu/?Dc=0' Nc=52a6c561 Dc=10>
    http://maes.cs.pdx.edu/</A><BR><BR><HR>
  </BODY>
</HTML>
```

Figure 4.6: HTML markup of kaPOW error page. This page is sent in response to a URL that had an invalid work function or an incorrect solution. The script redirecting the Web browser to use current POW parameters is highlighted.

Regardless of the reason, when a request is denied the filter returns an error page to the client such as the one in Figure 4.6. The error page contains some text explaining the error and a single link to the requested content. After it has been processed by the issuing filter, it has a work function embedded into it. The key feature is highlighted; the error page includes an `OnLoad()` script which immediately solves the work function and redirects the browser to use the proper URL. The client's Web browser is instructed to omit this error page from the browsing history so the user may never actually see this page displayed.

A notable exception is this error page will not automatically redirect clients who do not have JavaScript enabled. Recall that the issuing filter embeds "`Dc=0`" into all URLs found within HTML tags. If a client's browser does not have JavaScript enabled it will not solve the work function and will instead use the URL verbatim. When the verifying filter observes a URL with the variable D_c set to zero, it will conclude that the client cannot solve the work function and will accept the request on the low-priority virtual host. This allows non-JavaScript clients to access the content they seek (although there is a chance that the low-priority server is inundated with requests and may not be able to service theirs) without giving adversaries a means to disturb clients who correctly solve the work functions.

The Client Solver

While the client end of the system can be computationally demanding, particularly for malicious clients, it is functionally simple. The browser executes a few scripts found in the JavaScript file (`kaPOW.js`) linked at the head of the HTML document. The `Solve()` script is used to solve individual work functions that the browser encounters. The script takes a tag with a URL as input and extracts the variables N_c and D_c and systematically hashes them with various values for A until Equation 4.1 is satisfied. The script removes existing POW variables embedded in the URL (specifically the "`Dc=0`") and appends the new N_c, D_c, and A to the URL stored in memory for when the browser needs to fetch the referenced resource.

Another script runs as soon as the HTML file is read and hooks into the event triggered when tag elements are added to the DOM. As content tags (such as `<IMG>`) are added, this script calls `Solve()` so that the URL in the tag is valid before the browser fetches that content. As hyperlink tags (`<A>`) are added, they have their `ONCLICK` attribute modified to call `Solve()` so those work functions are only solved if the user chooses to follow the link.

4.4 EVALUATION

To evaluate the effectiveness of the kaPOW approach, `mod_kaPOW` was implemented in C as an Apache v2.0 module. The module supports a simple load-based difficulty policy using a Bloom filter. The Bloom filter's counters start at zero and are updated every 10 seconds to count the request activity in the previous 10 seconds. The Bloom filter's DECAY constant is set to 10 so that as long as a client sends fewer than 10 requests every 10 seconds they will not be issued a work function (i.e., D_c will remain zero). If a client sends more than 10 requests every 10 seconds, the surplus requests exponentially increase the counters so the client will subsequently be required to solve very difficult work functions in order to receive service. The prototype is Web accessible and was demonstrated at SIGCOMM 2008 [62, 64].

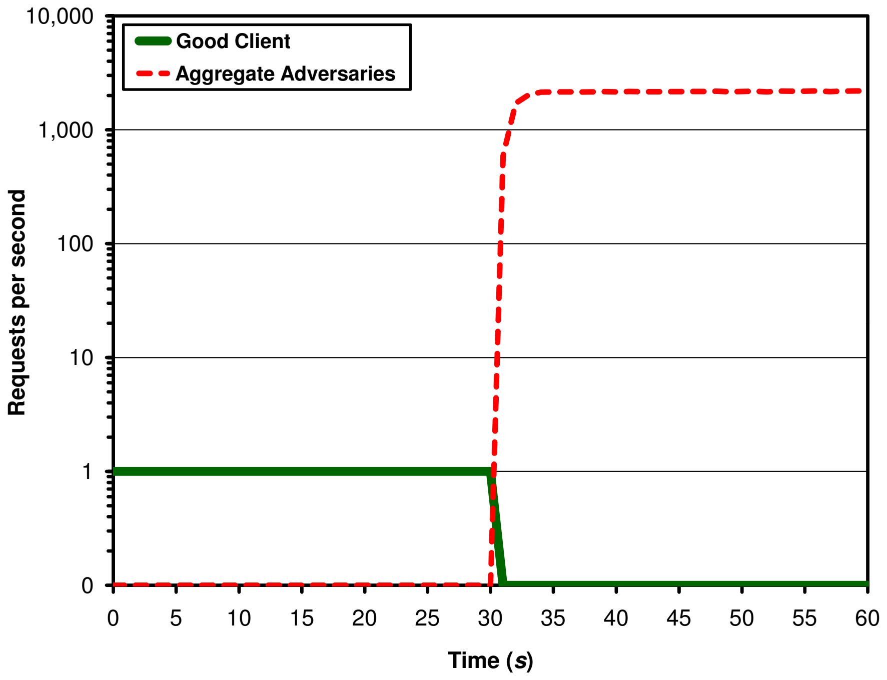


Figure 4.7: Flooders vs. default Web server. Starting at time t = 30, the unprotected server is saturated by the flooders who connect and continuously send requests as fast as possible; the good client cannot get any more requests serviced.

4.4.1 Experimentation

`mod_kaPOW` Thwarting Flooders

In this experiment, we set up a network of six 1.8GHz dual processor Intel Xeon machines connected by Gigabit Ethernet interfaces: a server running Apache v2.0, a good client which requests a webpage once every second, and four flooding adversaries that attempt to saturate the server with requests. While this setup is far from the magnitude of a real botnet, the server is configured to give adversaries an advantage over the good client by only accepting four persistent connections. This means if the four adversaries connect and remain connected, they will deny service to the good client as demonstrated in Figure 4.7. Without `mod_kaPOW`, the adversaries occupy the server indefinitely, sustaining 2,150 requests per second, yet the good client cannot establish a connection to get a single request serviced.

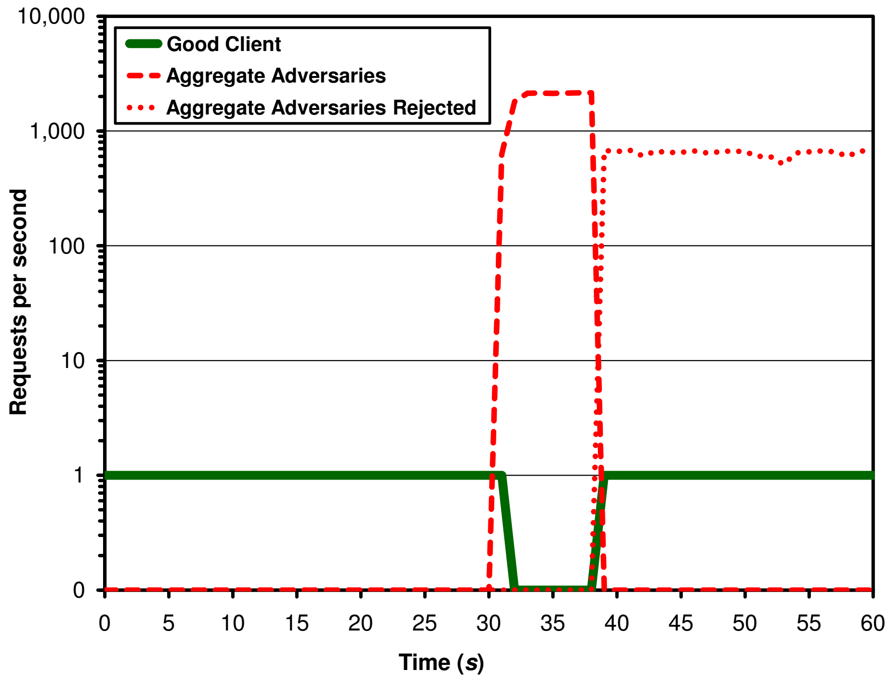


Figure 4.8: Flooders vs. `mod_kaPOW`. Starting at time t = 30, the flooders saturate the server and deny service to the good client. At time t = 39, `mod_kaPOW` updates its Bloom filter; the flooders are then required to solve difficult work functions so all their subsequent requests (without answers attached) are rejected.

In this next scenario, we add `mod_kaPOW` to the server and do not modify the adversary behavior at all. Specifically, flooders ignore POW work functions and simply send as many requests as possible. Figure 4.8 shows that the flooders are only able to deny service to the good client for nine seconds. Until `mod_kaPOW` next updates its Bloom filter counters, the flooders remain connected and sustain 2,150 requests per second. Once the Bloom filter is updated, each adversary is required to solve a difficult challenge. Since they do not send correct answers along with their requests, the requests are rejected and their connections to the server are terminated, restoring service to the good client. In this state, the flooders are only able to sustain 650 rejected requests per second due to the "three-way handshake" required establish a new TCP connection per request.

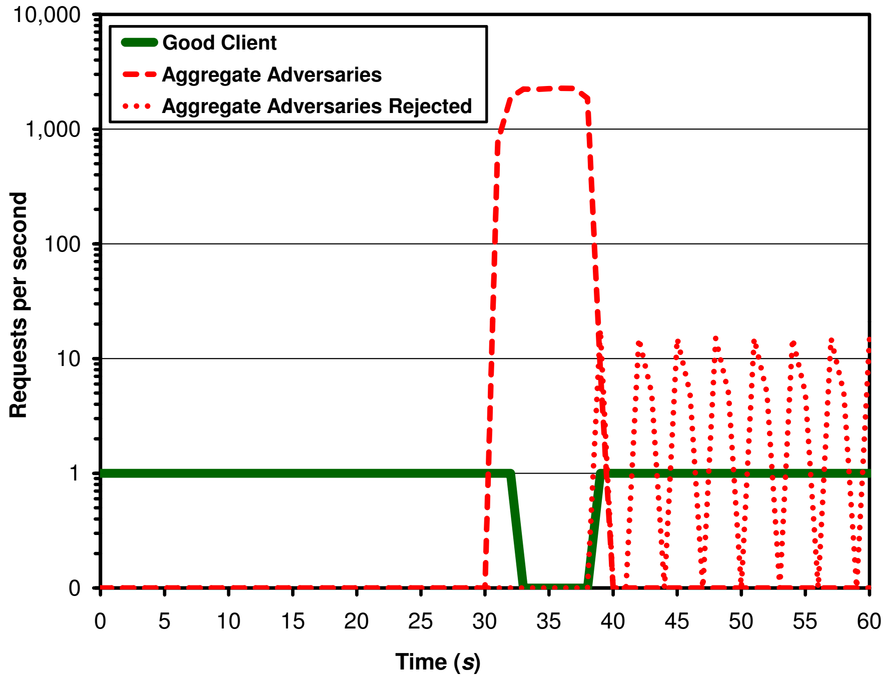


Figure 4.9: Flooders vs. `mod_kaPOW` with `iptables` filter. Starting at time t = 30, the flooders saturate the server and deny service to the good client. At time t = 39, `mod_kaPOW` updates its Bloom filter; the rejected flooders are disconnected and then limited to five connection attempts every second by an `iptables` filter.

The previous scenario showed that the `mod_kaPOW` prototype proficiently repels a flood of requests without valid answers attached. Unfortunately, these requests still consume substantial Web server resources since the server must accept numerous adversary connections just to read and then reject the requests. Figure 4.9 shows the next scenario which employs a simple `iptables` filter to rate-limit incoming TCP connections. By using standard `iptables` matching rules, a network-level ingress filter can restrict each adversary to a reasonable five TCP connections every second. This reduces the volume of rejected flooder requests by two orders of magnitude. Since we did not modify the flooders' TCP protocol, dropping their requests induces normal TCP backoff behavior which contributes to the jigsaw pattern of rejected requests.

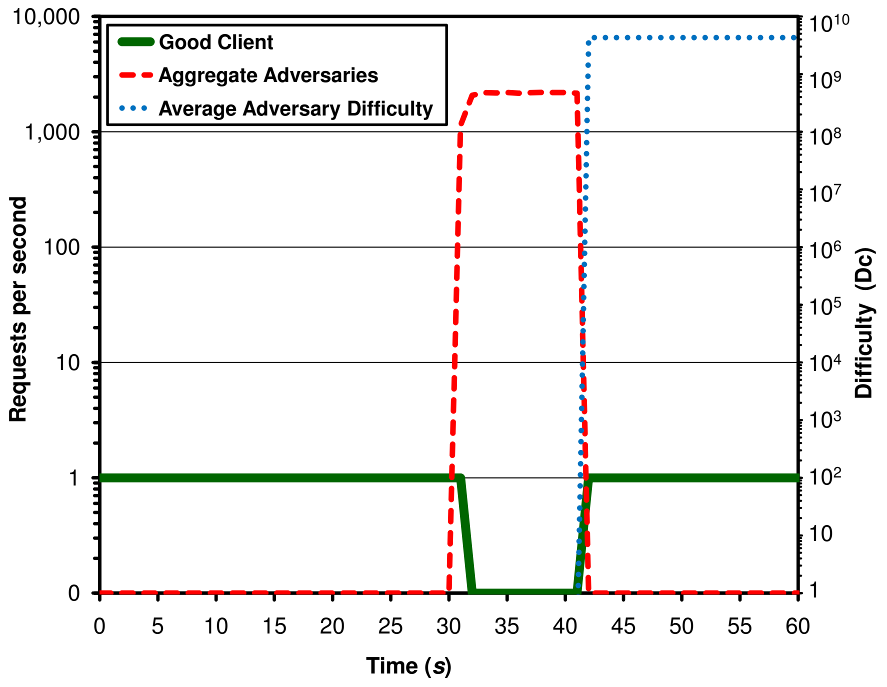


Figure 4.10: Solving flooders vs. `mod_kaPOW`. Starting at time $t = 30$, the solving flooders saturate the server and deny service to the good client. At time $t = 40$, `mod_kaPOW` updates its Bloom filter; the flooders are then issued difficult very work functions which preoccupy them indefinitely so they send no new requests.

This fourth scenario demonstrates that `mod_kaPOW` properly rejects flooding adversaries who attempt to solve any work functions they are issued. Figure 4.10 shows that once again the flooders are only able to deny service to the good client for under 10 seconds: that is, until `mod_kaPOW` next updates its Bloom filter. At that point, the Bloom Filter counters are updated and the adversaries are issued difficult work functions, restoring service to the good client. The flooders work functions require 2^{32} hashes (roughly 1.3 hours) to solve, which preoccupies them with solving the work functions rather than sending requests that will be rejected.

These scenarios illustrate that even when using a relatively simple policy to adjust the work function difficulty D_c, the kaPOW approach can disincentivize automated adversaries to achieve separation between them and legitimate clients.

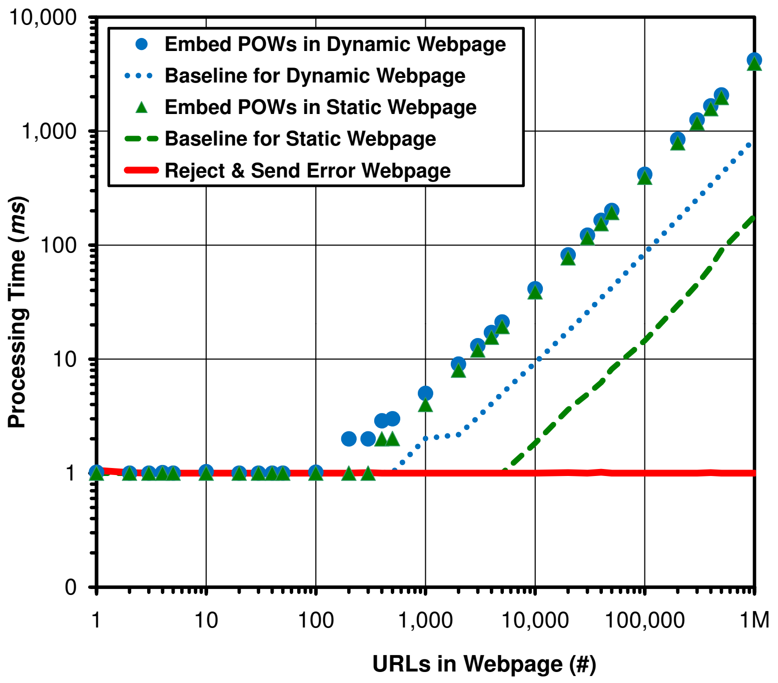


Figure 4.11: Request processing time vs. webpage URLs. The time required to modify dynamic and static webpages, and reject invalid requests is compared.

`mod_kaPOW` Computation Overhead

In this experiment, we used the Apache Benchmark tool (`ab`) [6] to measure the time required to process various Web requests (each data point represents the average of 10,000 identical requests), rounding up to the nearest millisecond.

Figure 4.11 shows the overhead when processing files which contain a variable number of URLs. There is observable overhead when processing large static webpages stored on the disk. The overhead is considerably less when processing dynamically generated webpages because they are appropriately formatted for the filter when they are created. Processing time is independent of file size (i.e., constant) for requests that are rejected for having an invalid solution because the small error webpage is returned instead. The overhead is negligible for all webpages containing fewer than a couple hundred URLs, yet the benefit of rejecting invalid requests for large webpages is substantial.

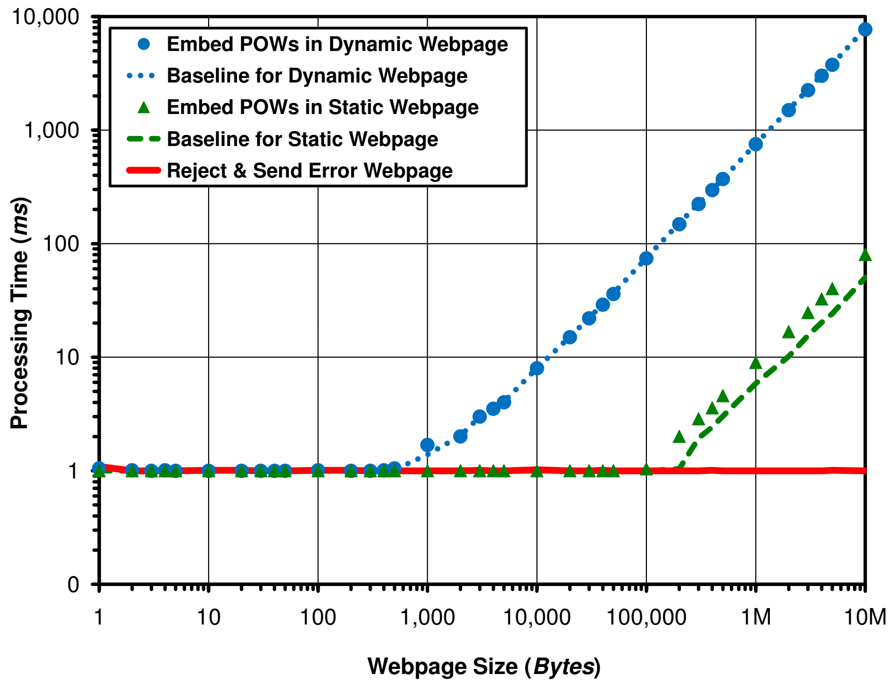


Figure 4.12: Request processing time vs. webpage size. The time required to modify dynamic and static webpages, and reject invalid requests is compared.

Figure 4.12 shows the overhead for processing webpages of variable length containing no URLs. The graph shows that the computation overhead is negligible because no work functions need to be created. The benefit of rejecting invalid requests for large webpages remains substantial.

`mod_kaPOW` Bandwidth Overhead

The kaPOW approach adds proof-of-work challenges to outbound HTML content which requires some additional network bandwidth. This overhead is a linear function of the number of URLs in the original webpage: each tag containing a URL has a work function added (between 15 and 29*Bytes*) in addition to the header's reference to the kaPOW script file (56*Bytes*). The total bandwidth overhead (in *Bytes*) is therefore bounded by:

$$bandwidth\ overhead \ \leq\ (29 \times \#_{\texttt{URLs}}) + 56 \tag{4.4}$$

Geographic kaPOW

The previous experiments have shown that the kaPOW approach can use a load-based difficulty policy to thwart generic packet-flooding adversaries. Many automation attacks like email and comment spam, click fraud, and ticket purchasing bots do not flood packets and thus cannot be deterred using a load-based difficulty policy. The following experiments use a difficulty policy based on client geographic distance to protect the online ticketing application. Event tickets are a \$30 billion market with most revenue coming from online purchases [103]. Automated scalpers instantly snap up all available tickets so that they can resell them at substantially higher prices [110, 111, 112]. To change the economics for scalpers, we introduce a PHP-based kaPOW prototype [61] that sets the work function difficulty proportional to geographic distance from the ticketed event[4].

The key observation is that *most legitimate purchases come from clients located in close geographic proximity to the event.* The policy leverages modern geolocation databases which are 90% accurate in resolving the geographic location of any network client to within 25 miles [42, 78] and adaptively issues distant clients more difficult work functions. In doing so, operators of ticket purchasing networks are forced to acquire significant network resources (i.e., physical machines) in close proximity to each event in order to monopolize all event tickets. This PHP-based approach does not require any changes to the software running on either the client or server and can be readily deployed by current online ticketing applications.

While this work focuses on the online ticketing problem, geographic distance may be used as a heuristic of client legitimacy in other applications as well. For example, online comment spam affecting articles published by regional news outlets could similarly be mitigated using geographically driven proof-of-work. Additionally, Web services with localized content could prioritize local clients during denial-of-service attacks by throttling distant clients.

[4]This research appeared in a paper at Global Internet 2010 [66].

Adversary Model: Online Ticket Robots

Adversary Goal. We assume that legitimate demand for event tickets is sufficient to normally sell them all. As a result, the goal of an adversary operating a network of ticket purchasing robots is to *acquire as many tickets as possible when they become available for sale*. To simplify the adversary model, we further assume that all the tickets to the event are desirable for resale so the adversary will purchase any and all tickets given the opportunity. Since the adversary will always purchase the maximum number of tickets allowed per transaction (usually between four and eight tickets), from hereon we will use the term "ticket" to mean the largest quantity of real tickets that can be acquired per transaction.

General Strategy. Long before tickets go on sale, the adversary establishes control of a botnet. This typically involves stealthily compromising a large number of computers attached to the Internet, or possibly leasing an existing botnet from herders [54]. Individual botnet machines are roughly equivalent to the computers used by legitimate clients in terms of the network and computation resources available to them. Indeed, some legitimate client computers may be unknowingly compromised and running botnet software targeting the very same event that the computer's owner is interested in. As we discuss in more detail later, this scenario becomes more probable when employing geographically-based proof-of-work. In fact, this is a desirable outcome since the owner of the computer will be alerted to the machine's compromise and take steps to remove the botnet software.

Timed to coincide with the start of the ticket sale (i.e., time $t = 0$), the adversary directs the botnet to execute as many ticket purchasing transactions as possible. Since the adversary intends to use the botnet to buyout multiple events or launch other network attacks, the adversary is careful to operate the botnet in a fashion that neither alerts the online ticket vendor of the illegitimate purchase requests nor alerts the true owners of the physical machines as to their misuse (lest it encourage the machine's real owner to remove the botnet software).

For any popular event, there are some legitimate clients (i.e., dedicated fans) who attempt to purchase tickets at the very moment they go on sale. These fans are the only clients who stand a chance versus ticket purchasing robots; clients who decide to buy tickets hours or days after they have gone on sale are clearly too late. Here onwards, these very passionate legitimate clients represent the legitimate client population C. To simplify the evaluation of our approach, we assume that these highly enthusiastic clients also equal the number of tickets available for sale (i.e., $\textsc{Tickets} = |C|$) so that the event would sell out quickly even without the ticket purchasing robots. This allows us to reason that any ticket purchased by an adversary would have otherwise been sold to a legitimate client. In practice, this assumption does not seriously weaken the adversary model since ticket scalpers predominantly target extremely popular events to minimize the risk of purchasing tickets that they cannot resell at a profit before the event occurs.

EXISTING DEFENSES. Online ticket vendors currently track the network addresses of successful ticket purchasers and restrict each address to one purchase per event. As a result, hosts that are behind network address translating proxies are denied by ticket vendors. This means that any adversary who generates a large number of ticket purchase transactions must have an equivalent number of unique network addresses to successfully complete them. Consequently, this restricts any traffic forwarding and tunneling that an adversary may perform, as they must control an equal number of forwarders with unique network addresses.

Geographic kaPOW Proof-of-Work Mechanism

The proof-of-work mechanism in the geographic kaPOW approach is similar to that of `mod_kaPOW` but is instead implemented in PHP, a ubiquitous Web scripting language. This requires no changes to the Web server so it may be adopted by websites that cannot load Apache modules. The approach continues to use Targeted Hash Reversal work functions and a periodically updated secret server nonce to generate temporary client nonces.

Geographic kaPOW Difficulty Policy

The goal of any proof-of-work approach is to maximize the amount of work that adversaries must perform while simultaneously minimizing the work imposed on legitimate clients. The key observation behind geographic pricing is that most legitimate purchasers connect online from a geographic location close to where the event takes place. Commercial geolocation databases have become very accurate at mapping network addresses to their geographic locations. Our hypothesis is that a proof-of-work system where work function difficulty is driven by geographic distance can limit scalping by forcing remote adversaries to perform significantly more work than local clients. Adversaries must then physically own significant resources near event centers in order to monopolize ticket purchases, which may be prohibitively expensive to acquire.

While geographic proof-of-work increases the monetary cost to adversaries by forcing them to have a presence near each event, there are two problems with using network address geolocation databases. The first problem is that non-local and erroneously geolocated legitimate clients will be unfairly penalized. The second problem is that for small events in large event centers, the cost of obtaining sufficient unique local machines to monopolize the event tickets may not be high enough to completely deter automated ticket purchasing. To address these potential problems, the proof-of-work approach could also consider the credit card's geographic billing address when determining the difficulty of a work function, and require that it be distinct. Clients must already provide authentic credit card information including the billing address in order to purchase tickets. Using that information, the system would have another method for determining where clients are geographically purchasing event tickets from, one which is possibly harder to spoof. This would further raise adversary costs by forcing them to obtain and maintain a large number of unique credit cards (and associated mailing addresses) local to each event center.

Function	Requests Serviced per Minute
Serve blank PHP page	36,583
Client geolocation	12,462
Client geolocation and issue work function	12,444
Client geolocation and verify solution	12,412

Table 4.2: Throughput of Geographic kaPOW routines. Prototype ticket server throughput across a range of tasks.

Geographic kaPOW Throughput

Moving on to the Geographic kaPOW prototype, Table 4.2 shows the baseline performance of the prototype on an Intel Core 2 Quad system (Q6600/2.4GHz) running Apache v2.2.9. As the table shows, the server processes over 36,000 blank pages a minute. When IP geolocation is added, the throughput of the system drops by two-thirds due to the overhead of looking up the IP address in the geolocation database. The cost of issuing and validating proof-of-work challenges is negligible compared to that of geolocation resolution. In each case, the performance is more than adequate to support the ticketing application as the capacity of most venues is below the amount of requests the server can process in a minute.

Geographic kaPOW Simulator

The PHP-based kaPOW shows how easily proof-of-work can be added to online ticketing applications. To show that it can mitigate realistic networks of ticket-purchasing robots, however, large-scale experimentation using thousands of robots must be performed. Since such experimentation is impractical, we have instead developed a simulator that closely models the behavior of the prototype server and its clients. To validate that the simulator accurately represents the PHP implementation, we compare the results of the following small-scale experiment on the prototype with the identical experiment in the simulator.

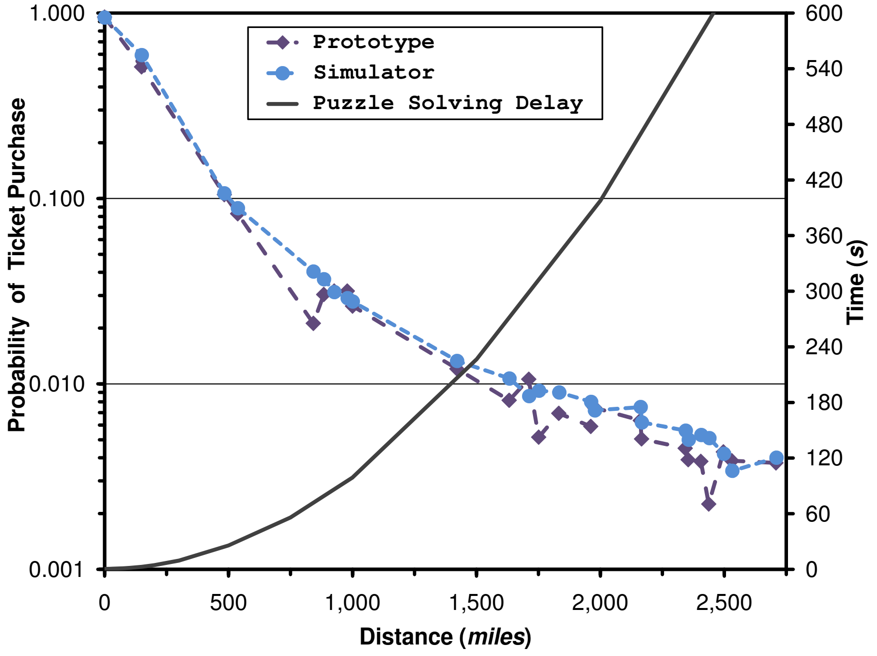


Figure 4.13: Simulator validation. The probability that prototype and simulator clients may purchase a ticket vs. their distance from the event.

The experiment consists of an event in a city on the West coast of the USA for which 100 good clients and 100 adversaries attempt to purchase 100 available tickets. While the legitimate clients are all located near the city, adversaries are randomly distributed across the 25 largest metropolitan areas in the United States proportionally to the region size. As explored later, this distribution improves the adversaries' ability to acquire tickets across all events held across the country. The work function difficulty is set as $D_c = 100{d_c}^2 + 10^6$, alternatives are explored later.

The experiment was performed 10,000 times on both the prototype and simulator. Figure 4.13 shows the probability that clients and adversaries successfully purchase tickets to an event as a function of their distance to the event. As the figure shows, the results from the simulator closely match those from the prototype with local clients having an exponentially higher probability of purchasing a ticket than their distant peers.

Rank	Metropolis	Population	Events
1	New York City, NY	17,799,861	1,756
2	Los Angeles, CA	11,789,487	1,163
3	Chicago, IL	8,307,904	819
4	Philadelphia, PA	5,149,079	508
5	Miami, FL	4,919,036	487
6	Dallas, TX	4,145,659	412
7	Boston, MA	4,032,484	397
8	Washington, DC	3,933,920	388
9	Detroit, MI	3,903,377	385
10	Houston, TX	3,822,509	377
11	Atlanta, GA	3,499,840	345
12	San Francisco, CA	2,995,769	295
13	Phoenix, AZ	2,907,049	286
14	Seattle, WA	2,712,205	267
15	San Diego, CA	2,674,436	263
16	Minneapolis, MN	2,388,593	235
17	St. Louis, IL	2,077,662	204
18	Baltimore, MD	2,076,354	201
19	Tampa, FL	2,062,339	203
20	Denver, CO	1,984,887	197
21	Cleveland, OH	1,786,647	173
22	Pittsburgh, PA	1,753,136	173
23	Portland, OR	1,583,138	156
24	San Jose, CA	1,538,312	157
25	Riverside, CA	1,506,816	154
	Total	**101,350,499**	**10,000**

Table 4.3: Largest U.S. metropolises. The population as well as how many simulated events occur in each of the 25 most populous U.S. metropolises is listed.

Similar to real ticket vendors, the simulated server sells tickets to events throughout the 25 largest metropolitan areas in the United States [114]. Events occur in proportion to each area's population. Further experiments evaluate the adversary's ability to purchase tickets to the 10,000 events shown in Table 4.3.

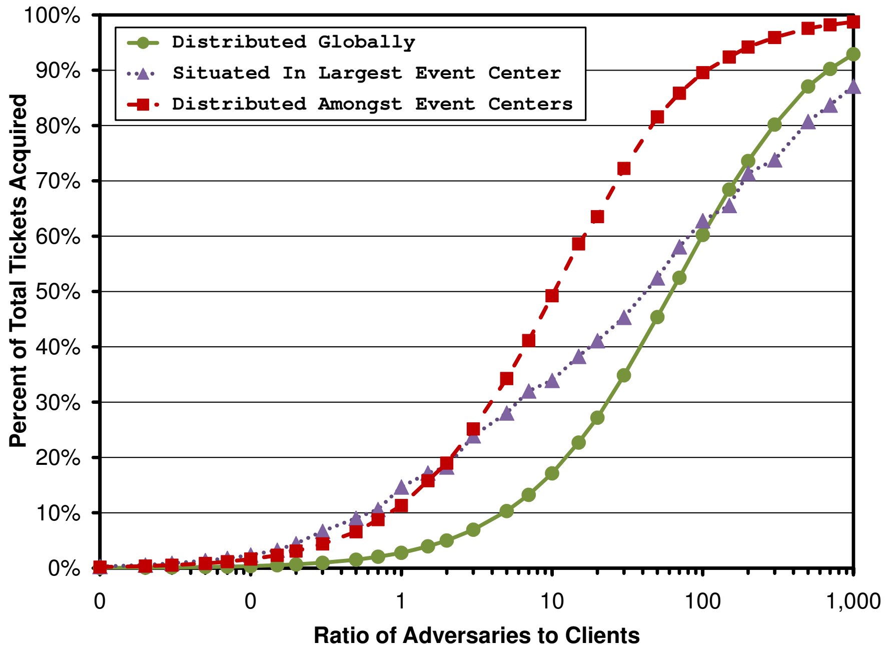


Figure 4.14: Adversary ticket acquisition. The percentage of total tickets acquired by adversaries vs. their ratio to clients, for adversaries employing various geographic distributions.

Ticket-Purchasing Botnet Geographic Distribution

The next experiment explores geographic distribution strategies that the adversary network might take to maximize its success. In each trial, an event location is selected and 2,500 local clients attempt to purchase the 2,500 tickets. The adversary population is exponentially increased to see what percent of the total tickets they can purchase. Once again, the difficulty algorithm is $D_c = 100{d_c}^2 + 10^6$.

Figure 4.14 shows the success of three strategies for distributing adversaries. The first approach assembles adversaries all around the globe like a naïve botnet might. Adversary IP addresses were obtained from the 10,000 worst daily offenders reported by DShield [27]. Not surprisingly, this approach requires orders of magnitude more adversaries than other approaches because many of the bots are far away (i.e., not in North America) from where events are held.

In the second approach, all adversaries are situated in the largest event center: New York City. Acquiring tickets to events in that area is easy, however, acquiring tickets to events held in other areas remains challenging – they must get “lucky” when solving their work functions to have a chance to purchase tickets before local legitimate clients do.

The third approach distributes adversaries throughout the 25 largest areas in the United States in proportion to their population. This simulates the repeated or long-term leasing (from a botnet controller) of only those zombie machines that are geographically desirable to at least one event location. In this approach, each adversary is local to at least some events and on average 5.96% of the adversaries are local to a randomly selected event. Of the three adversary approaches, this one performs the best, particularly in purchasing the last (i.e., highest) percentile of tickets, and is selected for subsequent experiments.

Large Ticket-Purchasing Botnets

The previous experiment qualitatively demonstrated the ability for geographic proof-of-work to slow down an adversary. To quantify the extent at which this is the case, we simulate the performance of the system as the number of adversaries is dramatically increased. Similar to the last experiment, the adversaries in these scenarios are distributed across the 25 largest metropolitan areas and the difficulty algorithm is again calculated by $D_c = 100{d_c}^2 + 10^6$. Figure 4.15 shows the ability of all individuals (clients or adversaries) to purchase tickets with respect to their distance from the event as the population size of adversaries varies. As expected, the probability that an individual may purchase a ticket decreases the further away they are from the event location, so local clients stand a much better chance of acquiring tickets. In addition, as the number of total agents is increased, the probability that an individual may purchase a ticket decreases for agents at all distances simply because there are more individuals competing for the same finite number of tickets.

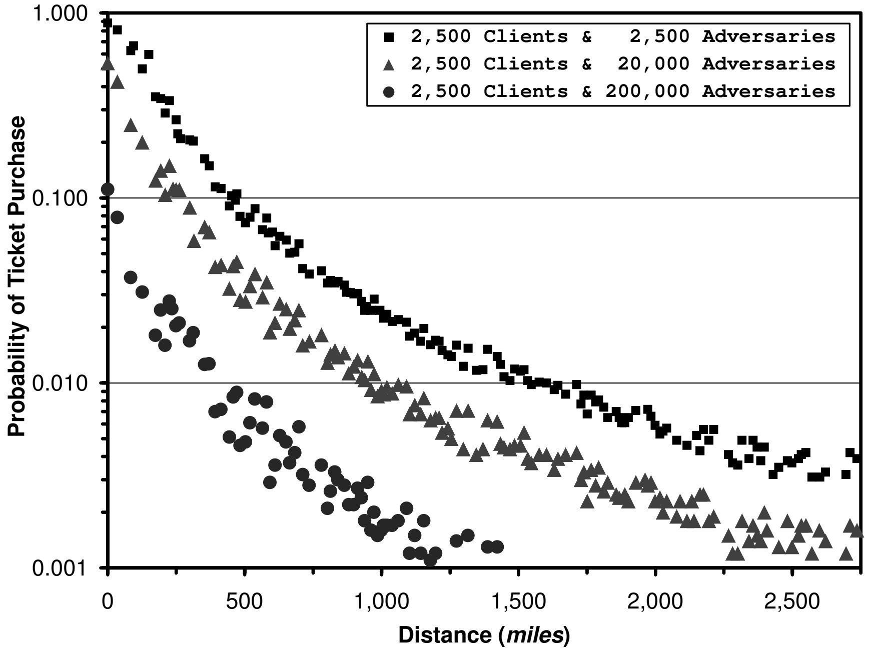


Figure 4.15: Ticket purchase probability. The probability any agent may purchase a ticket vs. their distance from the event for large adversary populations.

As the adversary population is increased significantly versus the legitimate client population, larger numbers of local adversaries A_{local} begin to compete with the legitimate clients. This decreases the percentage of tickets that go to legitimate clients as an increasing percentage of tickets are acquired by adversaries, as shown in Table 4.4. While the adversary network as a whole acquires more tickets across all events, for any specific event, non-local adversaries A_{far} are largely unsuccessful. With increased distance, adversary effectiveness quickly drops off. This is particularly evident in Figure 4.15 when the 200,000 adversaries outnumber the 2,500 clients (and thus tickets) by a ratio of 80 to 1; adversaries beyond 1,500 miles have less than a 1% chance to acquire tickets. As the adversary population increases, individual local adversaries also have a diminished ability to purchase tickets because they are competing amongst themselves (not just legitimate clients) for the limited tickets.

Adversary	Tickets Acquired by		
Population	C	A_{local}	A_{far}
2,500	88.7%	4.9%	6.4%
20,000	56.2%	23.0%	20.8%
200,000	12.9%	51.0%	36.1%

Table 4.4: Ticket acquisition breakdown. The percentage of total tickets acquired by the populations evaluated in Figure 4.15. There are 2,500 legitimate clients and 2,500 tickets available for purchase.

Throughout the 10,000 events, an average 11,872 of the 200,000 adversaries were local to any given event. The local adversaries roughly represent 6.0% of the total adversary population yet account for 58.6% of tickets acquired by the entire adversary population (51.0% of all tickets sold). On average 94.0% (118,128) of adversaries are non-local and manage to purchase only 36.1% of total tickets. The adversary network's success comes at a great cost as 98.9% of the individual adversaries have nothing to show for their arduous proof-of-work computation.

Geographic Difficulty Algorithms

The prior experiments have used a single difficulty algorithm for determining the amount of work a client must perform as a function of its geographic distance from the server. To examine how sensitive our approach is to this algorithm, we examine a number of alternatives. In comparing algorithms, it is helpful to derive the worst-case and best-case scenarios regarding the number of tickets that an adversary population may acquire. The worst case scenario occurs when the server operates without proof-of-work challenges. Given that clients and adversaries arrive at approximately the same time, the percentage of total tickets which the adversaries are expected to acquire is then governed by:

$$\textit{without proof-of-work} \approx \frac{|A|}{|A| + |C|} \tag{4.5}$$

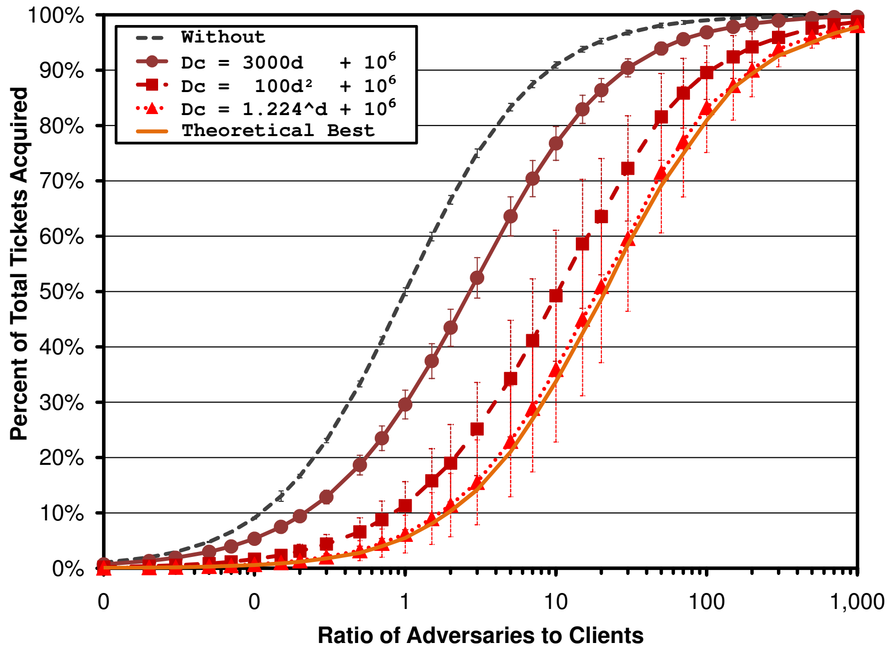


Figure 4.16: Effectiveness of various difficulty algorithms. The percentage of total tickets acquired by adversaries vs. their ratio to clients.

Conversely, the theoretical best that any geographically-driven system using proof-of-work can do is deny all non-local adversaries so that only local adversaries A_{local} compete with legitimate clients for the tickets. The percentage of tickets they acquire is similarly governed by:

$$\textit{theoretical best} \approx \frac{|A_{local}|}{|A_{local}| + |C|} \tag{4.6}$$

Figure 4.16 demonstrates the effectiveness of three different difficulty setting algorithms on impeding adversaries with respect to the theoretical bounds described above. The three algorithms shown in the figure are:

- linear ($D_c = 3000d_c + 10^6$)
- degree-2 polynomial ($D_c = 100{d_c}^2 + 10^6$)
- exponential ($D_c = 1.224^{d_c} + 10^6$)

The average client delay (in seconds) for these functions closely follows the difficulty divided by the number of hashes computable in one second (i.e., $\frac{D_c}{1{,}000{,}000}$). Thus, for these functions the delay is roughly 1 second for legitimate clients (due to the 10^6 constant) and quickly grows to minutes for distant adversaries. As the figure shows, minimal geographic differentiation is needed to give legitimate clients a noticeable advantage over the ticket purchasing robots. With slightly more aggressive differentiation, the system quickly nears the theoretical best curve.

Using the linear difficulty algorithm, remote adversaries are delayed on the order of tens of seconds. In contrast, the polynomial algorithm ramps up the difficulty so that distant adversaries across the country (3,000 miles away) are delayed several minutes. The exponential algorithm is much more severe and delays adversaries further than 100 miles away several minutes. The three algorithms impede adversaries such that the adversaries must multiply their population size by a factor of 2.72, 10.4, and 19.2 (for the respective linear, polynomial, and exponential algorithms) to acquire the same percentage of tickets as if the server were operating without geographic proof-of-work protection. As indicated in the last experiment, this occurs because the adversary population is mostly non-local (i.e., roughly 94%) and they are largely ineffective at securing tickets. As we elaborate upon in the discussion, regardless of whether the adversary employs only local bots or all bots, the kaPOW approach increases the likelihood that the real owner of a compromised machine will discover and remove the botnet software.

The probabilistic nature of solving Targeted Hash Reversal work functions means that in some cases adversaries get "unlucky" and take much longer than expected to find an answer. This means they could do worse than the theoretical best equation dictates (as evidenced by the error-bars reaching below the theoretical best curve). Conversely, in some unfortunate cases adversaries get "lucky" and solve their work functions in fewer hashes than expected and thus obtain more tickets than expected.

4.4.2 Limitations

Probabilistic Solution. As the last experiment in the previous section showed, probabilistic work functions like the Targeted Hash Reversal construction involve a solution strategy that is random. While the theoretically expected solution effort is D_c units of work (i.e., SHA1 hash executions) and is experimentally confirmed, any client attempting to solve a function can get either lucky or unlucky in finding a correct answer. Both possibilities are undesirable. An unlucky legitimate client will likely have the same outcome as a lucky adversary: the legitimate client may be denied service while the adversary receives the service. Unfortunately, the only known deterministic (i.e., non-probabilistic) work function is the Time-Lock puzzle construction [94] which is too computationally expensive to issue and verify. Faster Time-Lock implementations or alternative deterministic constructions may exist and warrant further investigation.

Meaningless Work Functions. Work functions represent busywork used to differentiate access to service between clients and adversaries. After a work function solution is verified, it is discarded. Besides throttling the solver, that solution provides no meaningful result for the client's expended computation effort. Future work may consider work functions where the solution has some meaningful purpose beyond throttling the solver. The research challenge here is finding a work function that can be easily issued and verified, and requires some arbitrary (yet predictable) amount of computation to solve. Additionally, the ideal work function would have compact data representations for both the challenge and solution so they will impose minimal communication overhead like current work functions. One viable approach might require the client to solve two similarly structured challenges like reCAPTCHA does for CAPTCHAs [116]: one with a solution known by the server that is used for verification and a second challenge where the answer is unknown but sought by the server.

4.4.3 Discussion

Proof-of-work forces clients to commit their computational resources before they are granted service. Since their efforts represent busywork, one might consider an approach which simply requires the client to wait an amount of time proportional to D_c. That approach overlooks two benefits of the proof-of-work approach.

First, *proof-of-work deters an adversary from using a single machine to send multiple requests.* If clients were simply required to wait a prescribed amount of time, an adversary would simply flood the server with sequential time-delayed requests. With proof-of-work, the adversary gains little benefit from flooding requests since the challenge must still be solved before service is granted. Additionally, proof-of-work prevents an adversary from using a single machine to concurrently flood other servers protected by proof-of-work since solving simultaneous proof-of-work challenges simply slows down the solution of each rather than provide an advantage.

Second, *proof-of-work increases the likelihood that botnet machines will be discovered and repaired.* Aggressive adversaries solving difficult work functions will incur steep computational penalties which may make individual machines unresponsive to their real owners. This increases the chance that the owner of the machine will investigate the system degradation and fix it (i.e., remove the zombie software). The risk of detection and removal will thus deter adversaries from targeting servers protected by proof-of-work. In the ticketing application, adversaries using local zombie machines also increase the risk of being discovered when conflicting with legitimate owners also attempting to purchase event tickets. Since the ticket vendor allows only one transaction per network address, two outcomes are possible. If the legitimate owner completes their transaction first, the adversary cannot complete a transaction with that machine. If the zombie completes their transaction first, the legitimate owner will get an error message that may lead to discovering and removing the zombie software.

4.5 RELATED WORK

Proof-of-Work Systems. Originally proposed in 1978 by Merkle, computational puzzles where initially proposed for cryptographic key exchange [79]. They were first used to combat resource consumption attacks in 1992 by Dwork [28]. Since then, numerous proof-of-work approaches have been proposed in the literature [1, 7, 8, 24, 35, 59, 83, 121, 123]. These approaches remain unused in practice because they require clients and servers to install new software to understand the protocol. The kaPOW approach presented in this chapter avoids that limitation and is transparent to both clients and servers.

CAPTCHA Systems. The use of CAPTCHA systems [117] is widespread – practically all websites which offer user accounts or allow users to post information employ CAPTCHAs. As discussed at the beginning of this chapter CAPTCHAs suffer three main problems. First, they make the user-interface less accessible, particularly for visually impaired users. Second, that automated solvers have defeated most easy-to-implement CAPTCHA constructions. Third, they represent a small, constant “price” which is less costly than the value of the goods or service which they are used to protect.

Indirection Approaches. An adversary must know where a network service resides in order to attack it. Indirection approaches [69, 73, 109] provide the ability to hide or dynamically relocate a public service in order to prevent malicious clients from reaching the service indefinitely. These approaches make it more arduous for legitimate clients to access the service and the defense rapidly breaks down if an adversary successfully learns the location of the service.

Filtering & Capability Approaches. Filtering approaches [3, 47, 77, 128] have network devices discard unwanted requests close to their sources. Capability approaches [4, 127] distribute access tokens so that clients may attach them to requests to indicate they are wanted. To work efficiently, these approaches require Internet-wide deployment of specialized network devices.

4.6 CONCLUSION

This chapter investigated the problem of disincentizing automated behavior which plagues Web-based applications. In this application, the server has little bearing over the proper execution of client software but instead has numerous information sources available to infer that a client is malicious. The research contributions of this chapter are:

- We defined the *resource consumption problem* affecting Web-based applications. In particular, there exists a resource imbalance between a server and its clients, both in terms of the resources each have available and that each must commit in order to complete a transaction.

- We *proposed and evaluated a novel proof-of-work approach* for Web applications. The kaPOW approach leverages the ubiquity of JavaScript to issue transparent computational challenges to clients – the human user need not be involved in solving the challenge. The approach was implemented twice to demonstrate its efficiency: as an Apache module `mod_kaPOW` and as a PHP-script. In evaluation, we showed that kaPOW is able to repel aggressive request-flooding adversaries.

- We *proposed and evaluated geographic distance as a novel means for indentifying likely adversarial behavior*. This information source was evaluated in the context of fully automated ticket-purchasing scalper networks. Leveraging accurate IP geolocation databases, the system assigns client-specific challenges that are more difficult the further away a client is from the event. Evaluation showed that an adversary network must use up to 19.2 times as many machines to acquire the same percentage of tickets that they would otherwise acquire if the server was unprotected by geographically-driven proof-of-work.

Chapter 5

CONCLUSION

In this dissertation, we have focused on the problem of addressing adversarial automation that affects networked applications. This dissertation validated the following thesis statement:

> *There exist methods to detect automated behaviors with which an application's service provider can identify and disincentivize automated adversaries.*

using research on two popular networked applications, multiplayer online video games and Web-based services. In these two applications, the service provider has varying access to the client. In multiplayer online video games, the game developer implements and releases the only client software that is authorized to interact with the server software. This means the game developer has firm control over what all legitimate client operation looks like.

In contrast, Web-based applications operate over the standardized HyperText Transfer Protocol which only dictates request and response formats. Any Web browser that adheres to the protocol can therefore access the service. Behavior outside of the protocol varies between client implementations so a Web service provider has little control over exactly what client operation looks like.

In validating the thesis statement, this dissertation investigated three research challenges: how to detect automated behavior, how to combine individually inconclusive automation detectors, and how to disincentive adversarial automation.

Methods to Detect Automated Behavior

Chapter 2 investigated the detection of adversarial automation that manifests as cheating in multiplayer online games. In this application, banning cheaters is a proven disincentive since they lose their game purchase, however, existing detection approaches are error-prone, not completely automated, and expensive to maintain. Leveraging the game developer's direct access to the game client, we explored a novel approach to cheat detection: *anomaly-based detection*. Our approach, Fides [67], automatically learns how the game client operates on different machines through partial client emulation. A server-side Controller specifies how and when a client-side Auditor measures the game. Through continued random remote audits, the Controller validates client execution and flags unexpected execution as cheating. The evaluation of this approach demonstrated that it is able to efficiently detect new un-cataloged cheats including one advertised as "undetectable."

Combining Detectors to Identify Adversaries

Chapter 3 investigated the combination of individually inconclusive detectors to produce a more conclusive result. Specifically, this chapter looked at reducing a set of heterogeneous detectors within multiplayer online video games to create a single metric for player maliciousness, whether it is rooted in automation or otherwise. Our approach, PlayerRating [65], is a *novel reputation system for multiplayer online games*. Treating a player's peers as detectors (i.e., each peer's observations provide clues regarding the maliciousness of other peers), the disincentive simply follows: players will avoid known malicious peers, and these peers will garner unwanted scrutiny from the game developer regarding the possibility that they are automated. The evaluation of this approach demonstrated its efficiency and collusion resistance, properties necessary for combining largely untrusted detectors. The results also indicate that this approach may be applicable to Web-based applications that have many information sources at their disposal.

Disincentivizing Adversarial Automation

Chapter 4 investigated the use of proof-of-work challenges to disincentivize automated adversaries of Web-based applications. In these applications, service providers have no access to monitor the operation of client software so they must rely on information sources completely external to the client software. Our approach, kaPOW [63], is a *novel transparent proof-of-work system* which does not require clients to install specialized software. Proof-of-work systems require that clients solve computational challenges before they are granted service, where each challenge is individually scaled in difficulty proportional to a metric of the client's maliciousness. More adversarial clients are given very difficult puzzles to solve, while less adversarial clients are given trivial puzzles to solve. Using request-load as an indication of the likelihood that a client is automated, we showed that service providers can wield proof-of-work challenges to efficiently thwart aggressive request-flooding adversaries. Additionally, using *geographic distance* as an indication that online ticket purchases are being initiated by distant, automated adversaries, we showed that ticket vendors can force adversaries to use up to 19.2 times as many botnet machines to acquire the same number of tickets that they would otherwise acquire if the server was unprotected.

REFERENCES

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately Hard, Memory-bound Functions. In *Proceedings of the 10th Annual ISOC Symposium on Network and Distributed System Security (NDSS '03)*, pages 25 – 39, February 2003.

[2] A. Adamic. Zipf, Power-laws, Pareto: A Ranking Tutorial. Technical report, HP Labs, October 2000.

[3] D. Andersen. Mayday: Distributed Filtering for Internet Services. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, pages 31 – 42, March 2003.

[4] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet Denial-of-Service with Capabilities. In *Proceedings of the 2nd Workshop on Hot Topics in Networks (HotNets '03)*, pages 39 – 44, November 2003.

[5] Apache Software Foundation. `http://www.apache.org`.

[6] Apache Software Foundation. ab: Apache HTTP Server Benchmarking Tool. `http://www.apache.org/docs/2.0/programs/ab.html`.

[7] T. Aura, P. Nikander, and J. Leiwo. DOS-Resistant Authentication with Client Puzzles. In *Proceedings of the 8th International Workshop on Security Protocols*, pages 170 – 177, April 2000.

[8] A. Back. Hashcash: A Denial of Service Counter-Measure. Technical report, August 2002. `http://www.hashcash.org/papers/hashcash.pdf`.

[9] P. Bak. *How Nature Works: the Science of Self-Organized Criticality.* Copernicus, 1996. ISBN 0-387-94791-4.

[10] N. E. Baughman and B. N. Levine. Cheat-Proof Playout for Centralized and Distributed Online Games. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '01)*, pages 104 – 113, April 2001.

[11] Black Omega. `http://www.mpcforum.com/showthread.php?s=d19abd77230cfca490701e3e1ec19ee3&t=183542`.

[12] Blizzard Entertainment. Warcraft III. `http://blizzard.com/en-us/war3/`.

[13] Blizzard Entertainment. World of Warcraft. `http://worldofwarcraft.com`.

[14] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An Improved Construction for Counting Bloom Filters. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA '06)*, pages 684 – 695, September 2006.

[15] M. Buchanan. *Ubiquity: The Science of History or Why the World is Simpler than We Think*. Crown Publishers, 2001. ISBN 0-609-60810-X.

[16] L. Catuogno, I. Visconti, and V. S. Allende. A Format-Independent Architecture for Run-Time Integrity Checking of Executable Code. In *Proceedings of the 3rd International Conference on Security in Communications Networks (SCN '02)*, pages 219 – 233, September 2002.

[17] CCP Games. Operation Unholy Rage: Eliminating Farmers in EVE Online, August 2009. `http://www.eveonline.com/devblog.asp?a=blog&bid=687`.

[18] Cheat Engine. `http://cheatengine.org`.

[19] K. Chen, H. K. Pao, and H. Chang. Game Bot Identification Based on Manifold Learning. In *Proceedings of the 7th Workshop on Network and System Support for Games (NetGames '08)*, pages 21 – 26, October 2008.

[20] Computer Language Benchmarks. C++ vs. Lua. `http://shootout.alioth.debian.org/u32q/benchmark.php?test=all&lang=gpp&lang2=lua`.

[21] E. Cooke, F. Jahanian, and D. McPherson. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *Proceedings of the 1st Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI '05)*, pages 39 – 44, July 2005.

[22] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63 – 78, January 1998.

[23] D. Dagon, C. Zou, and W. Lee. Modeling Botnet Propagation Using Time Zones. In *Proceedings of the 13th Annual ISOC Symposium on Network and Distributed System Security (NDSS '06)*, February 2006.

[24] D. Dean and A. Stubblefield. Using Client Puzzles to Protect TLS. In *Proceedings of the 10th USENIX Security Symposium*, pages 1 – 8, August 2001.

[25] C. Dellarocas, M. Fan, and C. A. Wood. Self-Interest, Reciprocity, and Participation in Online Reputation Systems. Technical report, MIT Sloan, August 2004.

[26] J. R. Douceur. The Sybil Attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, pages 251 – 260, March 2002.

[27] DShield. Distributed Intrusion Detection. `http://www.dshield.org`.

[28] C. Dwork and M. Naor. Pricing via Processing or Combatting Junk Mail. In *Proceedings of the 12th Annual International Cryptology Conference (CRYPTO '92)*, pages 139 – 147, August 1992.

[29] eBay. User Feedback. `http://ebay.com/services/forum/feedback.html`.

[30] Ecstatic Counter-Strike Source Hack. `http://www.mirc-scripts.de/pn/html/modules.php?op=modload&name=News&file=article&sid=225`.

[31] Even Balance. PunkBuster: Countermeasures for Cheaters in Multiplayer Online Games. `http://evenbalance.com`.

[32] J. Evers. Taking on Rootkits with Hardware, December 2005. `http://news.cnet.com/Taking-on-rootkits-with-hardware/2008-1029_3-5992309.html`.

[33] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281 – 293, June 2000.

[34] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy*, pages 62 – 75, May 2003.

[35] W. Feng. The Case for TCP/IP Puzzles. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA '03)*, pages 322 – 327, August 2003.

[36] W. Feng and E. Kaiser. The Case for Public Work. In *Proceedings of the 10th IEEE Global Internet Symposium*, pages 43 – 48, May 2007.

[37] W. Feng, E. Kaiser, W. Feng, and A. Luu. The Design and Implementation of Network Puzzles. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '05)*, pages 2372 – 2382, March 2005.

[38] W. Feng, E. Kaiser, and T. Schluessler. Stealth Measurements for Cheat Detection in On-line Games. In *Proceedings of the 7th Workshop on Network and System Support for Games (NetGames '08)*, pages 15 – 20, October 2008.

[39] S. Fewer. Reflective DLL Injection, October 2008. `http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf`.

[40] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[41] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the 19th ACM Symposium on Operating Systems (SOSP '03)*, pages 193 – 206, October 2003.

[42] Geobytes Inc. GeoNetMap. `http://www.geobytes.com`.

[43] GetAFreelancer. Captcha Entry Projects. `http://www.getafreelancer.com/projects/by-tag/captcha-entry.html`.

[44] S. Gianvecchio, Z. Wu, M. Xie, and H. Wang. Battle of Botcraft: Fighting Bots in Online Games with Human Observational Proofs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, pages 256–268, November 2009.

[45] J. Golbeck and J. Hendler. Reputation Network Analysis for Email Filtering. In *Proceedings of the 1st Conference on Email and Anti-Spam (CEAS '04)*, July 2004.

[46] Z. Gyongyi, H. Garcia-Molina, and J. Pedersen. Combating Web Spam with TrustRank. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)*, pages 576 – 587, August 2004.

[47] M. Handley and A. Greenhalgh. Steps Toward a DoS-resistant Internet Architecture. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA '04)*, pages 49 – 56, August 2004.

[48] HL2 Hook. `http://www.zerogamers.com/downloads/c49-Counter-Strike:-Source-Hacks/f809-HL2-Hook-v13.html`.

[49] S. Hocevar. PWNtcha: A CAPTCHA Solving Library. `http://caca.zoy.org/wiki/PWNtcha`.

[50] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection using Sequences of System Calls. *Journal of Computer Security*, 6(3):151 – 180, August 1998.

[51] G. Hoglund. Keeping Blizzard Honest - Announcing the Release of 'The Governor', 2005. `http://www.rootkit.com`.

[52] G. Hoglund. Hacking World of Warcraft: An Exercise in Advanced Rootkit Design. In *The 10th Annual Black Hat USA Technical Security Conference*, June 2006.

[53] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135 – 143, July 1999.

[54] N. Ianelli and A. Hackworth. Botnets as a Vehicle for Online Crime, December 2005. CERT RFC 1700.

[55] H. Inoue and S. Forrest. Anomaly Intrusion Detection in Dynamic Execution Environments. In *Proceedings of the 10th New Security Paradigms Workshop (NSPW '02)*, pages 52 – 60, September 2002.

[56] Intel Corporation. Active Management Technology. `http://www.intel.com/technology/platform-technology/intel-amt/`.

[57] G. Jeh and J. Widom. Scaling Personalized Web Search. In *Proceedings of the 12th International Conference on World Wide Web (WWW '03)*, pages 271 – 279, May 2003.

[58] G. Jin. Chinese Gold Farmers in the Game World. *Consumers, Commodities & Consumption Online Newsletter of the Consumer Studies Research Network*, 7(2), May 2006.

[59] A. Juels and J. Brainard. Client Puzzles: A Cryptographic Defense Against Connection Depletion. In *Proceedings of the 6th Annual ISOC Network and Distributed System Security Symposium (NDSS '99)*, pages 151 – 165, February 1999.

[60] E. Kaiser. PlayerRating. `http://wow.curseforge.com/addons/playerrating/`.

[61] E. Kaiser and W. Feng. kaPoW Online Ticketing Application. `http://kapow.cs.pdx.edu/geotickets`.

[62] E. Kaiser and W. Feng. mod_kaPoW Demo. `http://kapow.cs.pdx.edu`.

[63] E. Kaiser and W. Feng. mod_kaPoW: Protecting the Web with Transparent Proof-of-Work. In *Proceedings of the 11th IEEE Global Internet Symposium*, April 2008.

[64] E. Kaiser and W. Feng. mod_kaPoW: Putting the 'PoW' in Proof-of-Work. In *The ACM Interest Group Conference on Data Communication (SIGCOMM '08) Demo Session*, August 2008.

[65] E. Kaiser and W. Feng. PlayerRating: A Reputation System for Multiplayer Online Games. In *Proceedings of the 8th Workshop on Network and System Support for Games (NetGames '09)*, November 2009.

[66] E. Kaiser and W. Feng. Helping Hannah Montana: Changing the Economics of Ticket Robots with Geographic Proof-of-Work. In *Proceedings of the 13th IEEE Global Internet Symposium*, March 2010.

[67] E. Kaiser, W. Feng, and T. Schluessler. Fides: Remote Anomaly-Based Cheat Detection Using Client Emulation. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, pages 269 – 279, November 2009.

[68] S. D. Kamvar, M. T. Schlosser, and H. Garcia-molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. In *Proceedings of the 12th International Conference on World Wide Web (WWW '03)*, pages 640 – 651, May 2003.

[69] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proceedings of the ACM Special Interest Group Conference on Data Communication (SIGCOMM '02)*, pages 61 – 72, August 2002.

[70] D. Kesmodel. Codes on Sites 'Captcha' Anger of Web Users. *Wall Street Journal*, May 2006.

[71] C. Kruegel and G. Vigna. Anomaly Detection of Web-Based Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*, pages 251 – 261, October 2003.

[72] J. Kunegis, A. Lommatzsch, and C. Bauckhage. The Slashdot Zoo: Mining a Social Network with Negative Edges. In *Proceedings of the 18th International Conference on World Wide Web (WWW '09)*, pages 741 – 750, April 2009.

[73] K. Lakshminarayanan, D. Adkins, A. Perrig, and I. Stoica. Taming IP Packet Flooding Attacks. In *Proceedings of the 2nd Workshop on Hot Topics in Networks (HotNets '03)*, pages 45 – 50, November 2003.

[74] P. Laurens, R. F. Paige, P. J. Brooke, and H. Chivers. A Novel Approach to the Detection of Cheating in Multiplayer Online Games. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS '07)*, pages 97 – 106, July 2007.

[75] B. Laurie and R. Clayton. 'Proof-of-Work' Proves Not to Work. In *Proceedings of the 3rd Annual Workshop on the Economics of Information Security (WEIS '04)*, May 2004.

[76] Lavish Software. On Warden. `http://onwarden.blogspot.com`.

[77] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling High Bandwidth Aggregates in the Network. *ACM Computer Communication Review*, 32(3):62 – 73, July 2002.

[78] MaxMind Inc. Geolocation and Online Fraud Prevention from MaxMind. `http://www.maxmind.com`.

[79] R. Merkle. Secure Communications Over Insecure Channels. *Communications of the ACM*, 21(4):294 – 299, April 1978.

[80] D. L. Mills. Network Time Protocol Version 4 - Reference and Implementation Guide. Technical report, University of Delaware, June 2006.

[81] P. Mockapetris. Domain Names - Concepts and Facilities. RFC 1034 (Standard), November 1987.

[82] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33 – 39, July 2003.

[83] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host Identity Protocol Architecture. RFC 5201 (Experimental Protocol), April 2008.

[84] National Institute of Science and Technology (NIST). Secure Hash Standard, April 1993. Federal Information Processing Standard (FIPS) 180-1.

[85] NetCoders. The Unerring Punkbuster. `http://forum.netcoders.cc/announcements/14061-unerring-punkbuster.html`.

[86] Netherby. PlayerNotes. `http://wow.curseforge.com/addons/playernotes/`.

[87] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, January 1998.

[88] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - A Coprocessor-Based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179 – 194, August 2004.

[89] phpBB. phpBB: Creating Communites Worldwide. `http://www.phpbb.org`.

[90] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[91] J. Postel. Transmission Control Procol. RFC 793 (Standard), September 1981.

[92] T. Ptacek and N. Lawson. Don't Tell Joanna, The Virtualized Rootkit is Dead. In *The 11th Annual Black Hat USA Technical Security Conference*, August 2007.

[93] J. Rauch. Seeing Around Corners: The New Science of Artificial Societies. *The Atlantic Monthly*, 289(4):35 – 48, April 2002.

[94] R. Rivest, A. Shamir, and D. Wagner. Time-Lock Puzzles and Timed-Release Crypto. Technical report, MIT, March 1996. MIT/LCS/TR-684.

[95] J. Rutkowska. Subverting Vista Kernel for Fun and Profit: Part 2 - Blue Pill. In *The 10th Annual Black Hat USA Technical Security Conference*, August 2006.

[96] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-Based Integrity Measurement Architecture. In *Proceedings of the 13th USENIX Security Symposium*, pages 223 – 238, August 2004.

[97] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. In *Proceedings of the 2nd International Conference on Distributed Computing Systems (ICDCS '81)*, pages 509 – 512, April 1981.

[98] T. Schluessler, E. Johnson, and S. Goglin. Is a Bot at the Controls - Detecting Input Data Attacks. In *Proceedings of the 6th Workshop on Network and System Support for Games (NetGames '07)*, pages 1 – 6, October 2007.

[99] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of the 21st IEEE Symposium on Security and Privacy*, pages 144 – 155, May 2001.

[100] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Integrity and Guaranteeing Execution of Code on Legacy Platforms. In *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP '05)*, pages 1 – 16, October 2005.

[101] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*, pages 298 – 307, October 2004.

[102] M. Shermer. The Doping Dilemma. *Scientific American*, 298(4):60 – 67, April 2008.

[103] B. Siwicki. Big Ticket Items, January 2007. `http://www.internetretailer.com/article.asp?id=20961`.

[104] Skape and JT. Remote Library Injecion, April 2004. `http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf`.

[105] Slashdot.org. Slashdot FAQ: Karma. `http://slashdot.org/faq/com-mod.shtml#cm700`.

[106] Solar Designer. Getting Around Non-executable Stack (and Fix), August 1997. Bugtraq Mailing List.

[107] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149 – 167, August 2002.

[108] A. Starodoumov. Real Money Trade Model in Virtual Economies. Master's thesis, Stockholm School of Economics, June 2005.

[109] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proceedings of the ACM Special Interest Group*

Conference on Data Communication (SIGCOMM '02), pages 73 – 86, August 2002.

[110] R. Stross. Hannah Montana Tickets on Sale! Oops, They're Gone. *New York Times*, December 2007.

[111] StubHub Inc. Tickets at StubHub! `http://www.stubhub.com`.

[112] TNOW Entertainment Group. Tickets at TicketsNow. `http://www.ticketsnow.com`.

[113] M. Tresca. The Impact of Anonymity on Disinhibitive Behavior through Computer-Mediated Communication. Master's thesis, Michigan State University, 1998.

[114] US Census Bureau. List of Populations of Urbanized Areas, 2000. `http://www.census.gov/geo/www/ua/ua2k.txt`.

[115] Valve Software. Valve Anti-Cheat. `https://support.steampowered.com/kb_article.php?p_faqid=370`.

[116] L. von Ahn. The reCAPTCHA Project. `http://recaptcha.net`.

[117] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. In *Proceedings of the 22nd Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '03)*, pages 294 – 311, May 2003.

[118] W3 Schools. Browser Statistics. `http://www.w3schools.com/browsers/browsers_stats.asp`.

[119] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 21st IEEE Symposium on Security and Privacy*, pages 156 – 168, May 2001.

[120] K. Wang and S. J. Stolfo. Anomalous Payload-Based Network Intrusion Detection. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID '04)*, pages 203 – 222, October 2004.

[121] X. Wang and M. Reiter. Mitigating Bandwidth-Exhaustion Attacks Using Congestion Puzzles. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*, pages 257 – 267, October 2004.

[122] Warcraft Realms. World of Warcraft Census. `http://www.warcraftrealms.com/realmstats.php`.

[123] B. Waters, A. Juels, J. Halderman, and E. Felten. New Client Puzzle Outsourcing Techniques for DoS Resistance. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*, pages 246 – 256, October 2004.

[124] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440 – 442, June 1998.

[125] Wired News. Guitar Hero Robot Plays Videogame With Electronic Precision. `http://blog.wired.com/gadgets/2008/11/guitar-hero-rob.html`.

[126] WoWWiki. Secure Execution and Tainting. `http://www.wowwiki.com/Secure_Execution_and_Tainting`.

[127] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting Network Architecture. *ACM SIGCOMM Computer Communications Review*, 35(4):241 – 252, October 2005.

[128] D. Yau, J. Lui, and F. Liang. Defending Against Distributed Denial-of-service Attacks with Max-min Fair Server-centric Router Throttles. *IEEE/ACM Transactions on Networking (TON)*, 13(1):29 – 42, February 2005.

[129] S. Yeung, J. Lui, and J. Yan. Detecting Cheaters for Multiplayer Games: Theory, Design and Implementation. In *Proceedings of the 3rd IEEE Consumer Communications and Networking Conference (CCNC '06)*, pages 1178 – 1182, January 2006.

[130] H. Zimmermann. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425 – 432, April 1980.

[131] D. Zovi. Hardware Virtualization-Based Rootkits. In *The 10th Annual Black Hat USA Technical Security Conference*, June 2006.