SEVENTH FRAMEWORK PROGRAMME

Information & Communication Technologies
Trustworthy ICT

NETWORK OF EXCELLENCE

A European Network of Excellence in Managing Threats and Vulnerabilities in the Future Internet: *Europe for the World* [†]

Deliverable D7.4: Final Report on Cyberattacks

Abstract: In this deliverable, we will *"report our research results in the area of Cyberattacks"*. We begin by putting our work in the context of the SysSec roadmap, and specifically how our research results address the various threats identified in different fields. The report is divided into four different chapters, one for each identified field, containing *Attacks on Web Applications and Services*, *Attacks on Smart and Mobile Devices*, *Attacks on Privacy* and *Attacks on Social Networks*.

Contractual Date of Delivery	August 2014
Actual Date of Delivery	December 2014
Deliverable Dissemination Level	Public
Editors	Sotiris Ioannidis, Thanasis Petsas
Contributors	All *SysSec* Partners
Quality Assurance	TUBITAK-BILGEM

[†]The research was funded by the European Union, within the Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 257007.

The *SysSec* consortium consists of:

FORTH	Coordinator	Greece
Politecnico Di Milano	Principal Contractor	Italy
Vrije Universiteit Amsterdam	Principal Contractor	The Netherlands
Institut Eurécom	Principal Contractor	France
IICT-BAS	Principal Contractor	Bulgaria
Technical University of Vienna	Principal Contractor	Austria
Chalmers University	Principal Contractor	Sweden
TUBITAK-BILGEM	Principal Contractor	Turkey

Document Revisions & Quality Assurance

Internal Reviewers

1. Ali Rezaki (TUBITAK-BILGEM)
2. Konstantina Drakou (FORTH)

Revisions

Ver.	Date	By	Overview
0.1.0	3/12/2014	*Editors*	Changes suggested by the final consistency review.
0.0.9	26/11/2014	#1	Final review of deliverable consistency.
0.0.8	24/11/2014	#2	Further updates by internal review.
0.0.7	21/11/2014	#1	Second review by QMC on deliverable concerning format, style, and content.
0.0.6	24/10/2014	#1	Thorough review of QMC for all chapters.
0.0.5	10/10/2014	#2	Detailed review of introduction, conclusions, and roadmap relations.
0.0.4	23/9/2014	*Editors*	Quality check by all chapter contributors.
0.0.3	15/9/2014	*Editors*	Merging of packages, references, macros, etc.
0.0.2	10/8/2014	*Editors*	Input received by partners.
0.0.1	3/7/2014	*Editors*	Outline and preliminary article selection complete.
0.0.0	17/4/2014	*Editors*	First outline of document.

Contents

List of Figures

1 Introduction

1.1 Cyberattacks in the SysSec Research Roadmap

One of the focus areas of the *SysSec* project was to improve our understanding in new and emerging types of cyberattacks, as well as to advance the state-of-the-art in the area of detection and mitigation of such cyber threats. The *SysSec* project had outlined the main candidate topics that should be investigated in the 2011 First Report on Threats on the Future Internet and Research Roadmap [56], which then was enriched with the 2012 *Second Report on Threats on the Future Internet and Research Roadmap* [57] and finally with our updated roadmap, the 2013 *Red Book: Roadmap for Systems Security Research* [58] providing examples of such attacks, information about their impact, as well as the state-of-the-art of the current defensive tools and techniques against them.

The candidate topics that have been identified as a result of brainstorming and discussion from relevant working groups within the *SysSec* project were the following: Social Engineering, Web Services and Applications, Big Data and Privacy, Critical Infrastructures, Smart, Mobile and Ubiquitous Appliances, Insiders and Network Core Attacks. Considering that the area of cyberattacks is very wide, our research will cover four of the most prevalent fields as outlined in our roadmap. In the first roadmap [56], we paid particular attention on the *Web Services and Appliances*. Thus, during the first year, our research was basically related with attacks on Web Applications and Services. In our second roadmap [57], we placed more emphasis on *Smart, Mobile and Ubiquitous Appliances* and on *Privacy*. Thus, during the second year of our research focus on cyberattacks in these two different fields. In our updated roadmap, in the Red Book [58], the most important field was the emerging *Social Networks*, and therefore, our third year of research was mainly focused on attacks on such networks. In Section 1.2, there is an outline of the topics covered in this report. The topics are listed

in the order specified by the roadmap. In Section 1.3, there is a list of other works of the SysSec project related with cybersecurity that are not covered in detail due to the limited space of this report.

In the *Review of the state-of-the-art in Cyberattacks* [108], an overview of the most critical categories of cyber attacks was provided that threaten the modern computer systems and networks based on the most recent publications appeared in the top conferences in the field including Memory Attacks and Exploitation Techniques, Attacks on Devices, Denial of Service Attacks, Social Network and Privacy attacks, Web attacks, etc.

1.2 Research Topics Covered in This Report

This report provides the selected research work that the *SysSec* partners have conducted during the project in order to address issues in the area of cyberattacks identified previously and included in *SysSec* roadmaps.

As the area of cyberattacks is very broad covering very different kinds of threats from Web Services to Critical Infrastructures and Ubiquitous Appliances as already mentioned in the previous section, our research will cover three of the most prevalent fields in the area as drawn from our roadmaps. These fields of cyberattacks are:

- Attacks on Web Applications and Services (Chapter 2)
- Attacks on Smart and Mobile Devices (Chapter 3)
- Attacks on Privacy (Chapter 4)
- Attacks on Social Networks (Chapter 5)

The report is divided into three different chapters, one for each distinct field. First, in Chapter 2, we present our research work related with web-based cybersecurity threats. In Section 2.2, there is a study related with the security issues that occur in web applications and services due to cross-domain policies. Then, we proceed with presenting a framework for JavaScript randomization aiming at detecting and preventing Cross-Site Scripting (XSS) attacks in Section 2.3. Then, in Section 2.4, we present a malicious document scanner that combines static and dynamic analysis to detect threats contained in PDF files that can reach Internet users through a variety of channels such as rogue web sites, file-sharing networks, removable media, or as attachments to email messages. Second, in Chapter 3, we present our research work related with attacks on smart devices. Specifically, in Section 3.2, we introduce various techniques that can be used by Mobile Malware in order to evade the Android dynamic analysis through emulation detection. Moreover, we evaluate these techniques in different Android dynamic analysis tools and online services that analyze Android malware.

In Section 3.3, we present a threat scenario that targets WebView apps and show its practical applicability in a case study of over 287,000 Android apps. Finally, in Chapter 4, we introduce our research work related to attacks on users' privacy. In Section 4.2, we implement a framework for minimum information disclosure across third-party sites with social login interactions in order to solve the privacy issues that arise by this kind of login mechanisms. Then, we focus on the negative impact that social networking sites may have on privacy of their users and ways we can protect against these cyber threats in Section 4.3. Finally, in our last Chapter 5, we present our research work related to cyber threats on social networks. In Section 5.2, we present how spammers can exploit social networks in order to harvest the users' email addresses that can be used on a later stage in order to satisfy their malicious intentions. We then, continue with attacks against authentication mechanisms in Section 5.3, and social network profile cloning attacks in Section 5.4.

1.3 Other Cybersecurity Related Works of SysSec Consortium

In this section, there is a list of other works of the SysSec project related with cybersecurity which are not covered in detail due to the limited space of this report.

- Claudio Criscione, Fabio Bosatelli, Stefano Zanero, and Federico Maggi. **Zarathustra: Extracting WebInject Signatures from Banking Trojans.** In Proceedings of the *12th Annual International Conference on Privacy, Security and Trust (PST)*. July 2014, Toronto, Canada.

- Rafael A Rodrguez Gmez, Gabriel Maci Fernndez, Pedro Garca Teodoro, Moritz Steiner, and Davide Balzarotti. **Resource monitoring for the detection of parasite P2P botnets.** Computer Networks, Elsevier B.V., pages 302311, June 2014.

- Jelena Isacenkova and Davide Balzarotti. **Shades of Gray: A Closer Look at Emails in the Gray Area.** In Proceedings of the *9th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2014)*. June 2014. Kyoto, Japan.

- Nick Nikiforakis, Federico Maggi, Gianluca Stringhini, M Zubair Rafique, Wouter Joosen, Christopher Kruegel, Frank Piessens, Giovanni Vigna, Stefano Zanero. **Stranger Danger: Exploring the Ecosystem of Ad-based URL Shortening Services.**. In Proceedings of the *2014 International World Wide Web Conference (WWW)*. April 2014. Seoul, Korea.

- Giancarlo Pellegrino and Davide Balzarotti. **Toward Black-Box Detection of Logic Flaws in Web Applications.** In Proceedings of *the Network and Distributed System Security Symposium (NDSS)*. February 2014, San Diego, USA.

- Zlatogor. Minchev and Luben Boyanov. **Smart Homes Cyberthreats Identification Based on Interactive Training.** In Proceedings of the *3rd International Conference on Application of Information and Communication Technology and Statistics in Economy an d Education (ICAICT-SEE)*. December 2013, Sofia, Bulgaria.

- Dennis Andriesse, Christian Rossow, Brett Stone-Gross, Daniel Plohmann, Herbert Bos. **Highly Resilient Peer-to-Peer Botnets Are Here: An Analysis of Gameover Zeus.** In Proceedings of the *8th IEEE International Conference on Malicious and Unwanted Software (MALWARE'13)*. October 2013, Fajardo, Puerto Rico, USA.

- Matthias Neugschwandtner, Martina Lindorfer, Christian Platzer. **A view to a kill: Webview exploitation.** In Proceedings of the *6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*. August 2013, Washington, DC, USA.

- Andrei Costin, Jelena Isacenkova, Marco Balduzzi, Aurelien Francillon, Davide Balzarotti. **The Role of Phone Numbers in Understanding Cyber-Crime Schemes.** In Proceedings of the *Annual Conference on Privacy, Security and Trust (PST)*. July 2013, Terragona, Spain.

- Davide Canali, Davide Balzarotti, Aurelien Francillon. **The Role of Web Hosting Providers in Detecting Compromised Websites.** In Proceedings of the *22nd International World Wide Web Conference (WWW)*. May 2013, Rio de Janeiro, Brazil.

- Federico Maggi, Alessandro Frossi, Stefano Zanero, Gianluca Stringhini, Brett Stone-Gross, Christopher Kruegel, Giovanni Vigna. **Two Years of Short URLs Internet Measurement: Security Threats and Countermeasures.** In Proceedings of the *22nd International World Wide Web Conference (WWW)*. May 2013, Rio de Janeiro, Brazil.

- Davide Canali and Davide Balzarotti. **Behind the Scenes of Online Attacks: an Analysis of Exploitation Behaviors on the Web.** In Proceedings of the *2013 Network and Distributed System Security Symposium (NDSS)*. February 2013, San Diego, CA, USA

- Luben Boyanov, Zlatogor Minchev and Kiril Boyanov. **Some Cyber Threats in Digital Society.** In *International Journal "Automatics & Informatics*". January 2013.

- Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, Christopher Kruegel. **DISCLOSURE: Detecting Botnet Command and Control Servers Through Large-Scale NetFlow Analysis.** In Proceedings of the *2012 Annual Computer Security Applications Conference (ACSAC)*. December 2012, Orlando, FL, USA.

- Christian Rossow, Christian Dietrich, Herbert Bos. **Large-Scale Analysis of Malware Downloaders.** In proceedings of the *9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. July 2012, Heraklion, Greece.

- Theodore Scholte, William Robertson, Davide Balzarotti, Engin Kirda. **Preventing Input Validation Vulnerabilities in Web Applications through Automated Type Analysis.** In Proceedings of the *36th Annual IEEE Computer Software and Applications Conference (COMPSAC)*. July 2012, Izmir, Turkey.

- Zhang Fu, Marina Papatriantafilou. **Off The Wall: Lightweight Distributed Filtering to Mitigate Distributed Denial of Service Attacks.** In *Proceedings of the 31st IEEE International Symposium on Reliable Distributed Systems (SRDS)*. October 2012, Irvine, CA, USA.

- Zhang Fu, Marina Papatriantafilou, Philippas Tsigas. **Mitigating distributed denial of service attacks in multiparty applications in the presence of clock drifts.** In *IEEE Transactions on Dependable and Secure Computing (TSDC), Volume 9, Issue 3*. May 2012.

- Zhang Fu, Marina Papatriantafilou, Philippas Tsigas. **Mitigating distributed denial of service attacks in multiparty applications in the presence of clock drifts.** In *IEEE Transactions on Dependable and Secure Computing (TSDC), Volume 9, Issue 3*. May 2012.

- Luca Invernizzi, Paolo Milani Comparetti, Stefano Benvenuti, Christopher Kruegel, Marco Cova, Giovanni Vigna. **EVILSEED: A Guided Approach to Finding Malicious Web Pages.** In proceedings of the *2012 IEEE Symposium on Security and Privacy (SP)*. May 2012, San Francisco Bay Area, CA, USA.

- Kaan Onarlioglu, Utku Ozan Yilmaz, Engin Kirda, Davide Balzarotti. **Insights into user Behavior in Dealing with Internet Attacks.** In proceedings of the *19th Annual Network & Distributed System Security Symposium (NDSS)*. February 2012, San Diego, CA, USA.

- Matthias Neugschwandtner, Paolo Milani Comparetti, and Christian Platzer. **Detecting Malwares Failover C&C Strategies with SQUEEZE.** In Proceedings of the *2011 Annual Computer Security Applications Conference (ACSAC)*. December 2011, Orlando, FL, USA.

- Christian Platzer. **Sequence-Based Bot Detection in Massive Multiplayer Online Games.** In Proceedings of the *13th International Conference on Information and Communications Security (ICICS)*. November 2011, Beijing, China.

- Christian J. Dietrich, Christian Rossow, Felix C. Freiling, Herbert Bos, Maarten van Steen, Norbert Pohlmann. **On Botnets that use DNS for Command and Control.** In Proceedings of the *7th European Conference on Computer Network Defense (EC2ND)*. September 2011, Gteborg, Sweden.

- Danesh Irani, Marco Balduzzi, Davide Balzarotti, Engin Kirda, Carlton Pu. **Reverse Social Engineering Attacks in Online Social Networks.** In Proceedings of the *8th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. July 2011, Amsterdam, The Netherlands.

- Francesco Roveta, Luca di Mario, Federico Maggi, Giorgio Caviglia, Stefano Zanero, Paolo Ciuccarelli. **BURN: Baring Unknown Rogue Networks.** In proceedings of VizSec 2011. July 2011, Pittsburg, PA, USA.

- Georgios Kontaxis, Demetris Antoniades, Iasonas Polakis, Evangelos P. Markatos. **An Empirical Study on the Security of Cross-Domain Policies in Rich Internet Applications.** In proceedings of the *4th European Workshop on System Security (EuroSec)*. April 2011, Salzburg, Austria.

- Zhang Fu, Marina Papatriantafilou, Philippas Tsigas. **CluB: A Cluster Based Proactive Method for Mitigating Distributed Denial of Service Attacks.** In proceedings of the *26th ACM Symposium on Applied Computing (SAC)*. March 2011, TaiChung, Taiwan.

- Theodoor Scholte, Davide Balzarotti, Engin Kirda. **Quo Vadis? A Study of the Evolution of Input Validation Vulnerabilities in Web Applications.** In proceedings of the *15th International Conference on Financial Cryptography and Data Security (FC)*. February 2011, St. Lucia.

- Leyla Bilge, Engin Kirda, Christopher Kruegel, Marco Balduzzi. **EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis.** In proceedings of the *18th Annual Network & Distributed System Security Symposium (NDSS)*. February 2011, San Diego, CA, USA.

2 Attacks on Web Applications and Services

2.1 Introduction

In this chapter, we present our research work related to web-based cybersecurity threats. First, we present an extensive study on the deployment and security issues of cross-domain policies in the web (Section 2.2). This work provides vulnerability awareness regarding the use of cross-domain Flash and Silverlight web policies. Our results reveal more than 6,500 websites with weak policies and, thus vulnerable to attacks. Moreover, we present proof-of-concept implementations of attack scenarios that target weak corss-domain policies for Flash and Silverlight enabled websites. Then, we proceed with presenting a framework for JavaScript randomization aiming at detecting and preventing Cross-Site Scripting (XSS) attacks (Section 2.3). Cross-Site Scripting (XSS) extends the traditional code-injection attack in native applications to web applications. It is considered as one of the most severe security threats over the last few years. One promising approach for dealing with code-injection attacks in general is Instruction Set Randomization (ISR). The fundamental idea behind ISR is that the trusted code is transformed in randomized instances. Thus, the injected code, when plugged to the trusted base cannot speak the language of the environment. To the best of our knowledge, there has been no systematic effort for applying a randomization scheme directly to a client-side programming language, like JavaScript. In this work we present RaJa, which applies randomization directly to JavaScript. Finally, we present a malicious document scanner, we call MDScan, that combines static and dynamic analysis to detect threats contained in PDF files that can reach Internet users through a variety of channels such as rogue web sites, file-sharing networks, removable media, or as attachments to email messages (Section 2.4). The autonomous design of MDScan allows it to be easily incorporated as a detection component into existing defenses, such as intrusion detection systems and antivirus ap-

plications. Our experimental evaluation with real and generated malicious documents, as well as benign PDF files, shows that MDScan can accurately detect a broad range of malicious documents, even when they have been highly obfuscated, while it has a reasonable runtime processing overhead.

2.2 An Empirical Study on the Security of Cross-domain Policies in Rich Internet Applications

An extended version of the study can be found in [82].

2.2.1 Data Collection

To examine both the policy usage and the security issues of cross-domain policies, we created a number of diverse (both geographically and content-wise) lists of websites. Our first and larger list includes the 100K most popular websites, according to Alexa [1]. We believe this represents a complete list of both popular and less popular websites. Furthermore, we compiled a list with the websites of the Fortune 500 companies to fill our global view set. For our geographically diverse set, formed by popular sites in a more local scale, we used lists with the 500 most popular websites, again according to Alexa, in the U.S.A, Great Britain, Germany, France and Greece. Additionally, we have used a list with 500 of the most popular Greek e-shopping websites to form a set of content-specific sites in our country.

For each domain name in those lists we placed the following requests: `http[s]://domain/crossdomain.xml` and `http[s]://www.domain/crossdomain.xml` to download the policy file for Adobe Flash if one existed. A respective batch of requests was also sent to retrieve the policy file for Microsoft Silverlight. We subsequently examined those XML files and identified the domains that did return a valid policy. Our findings are presented in the following section. We focus on the findings from the `http://www.domain` requests as they represent the most common case and their differences with the results of the other requests are minor.

2.2.2 Policy adoption and security

2.2.2.1 Global Penetration

We first examine the cross domain policy adoption at a global scale. Recall that the existence of a crossdomain policy file does not state whether a website supports RIA plugins or not, but whether a website allows RIA plugins to request and receive data from this website. Figure 2.1 (top) presents our findings for the Top 100 and Top 100K websites, while Table 2.1 summarizes

[1] http://www.alexa.com/

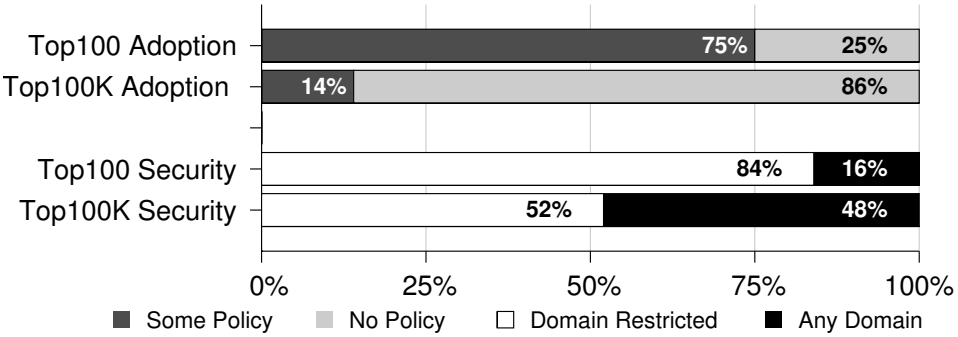


Figure 2.1: Global Alexa Top100, Top100K

	Adobe Flash			Microsoft Silverlight		
	None	Some Policy		None	Some Policy	
Tier		**Restricted**	**Unrestricted**		**Restricted**	**Unrestricted**
Alexa Top 100	25%	84%	*16%*	97%	67%	*33%*
Alexa Top 1K	48%	71%	*29%*	98%	63%	*37%*
Alexa Top 10K	71%	58%	*42%*	99%	38%	*62%*
Alexa Top 100K	86%	52%	*48%*	99%	22%	*78%*
Fortune 500	84%	75%	*25%*	-	-	-

Table 2.1: Adobe Flash Cross-domain Policy and Microsoft Silverlight Client-access Policy adoption for the top 100K Alexa sites.

our findings for all sub-classes examined. We can see that the penetration ratio of domain policies diminishes as the total number of examined sites grows. For example, 75 out of the Top 100 sites serve a policy file, while this percentage drops to only 14% for the Top 100K sites. Furthermore, as we see in the bottom bars of Figure 2.1, security awareness also diminishes. That is, top popular sites seem to restrict their policies more carefully than less popular sites – only 16% are weak cases for the Top 100 vs. almost 50% for the Top 100K sites.

To further understand the evolution of weak implementations as the site's rank moves away from the top, Figure 2.2 plots the cumulative percentage of weak policies in relation to the site's rank. We can observe a clear trend here; the less popular the site is, the higher the probability the website will have implemented a weak policy.

2.2.2.2 Country Penetration

We continue our examination by looking at the cross-domain policy adoption at a more local scale. To do so, we study the policy penetration for the U.S.A. and a number of major European countries. Table 2.2 summarizes our results. Countries like the U.S.A. with a high adoption rate (57%) of

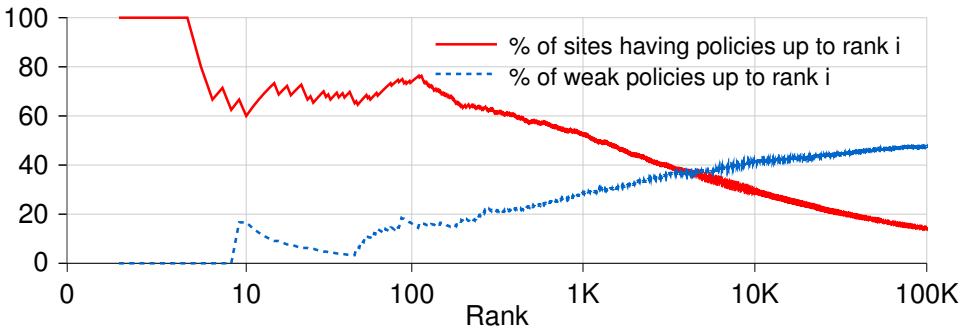


Figure 2.2: Correlation of policy adoption and site popularity rank.

cross-domain policies have fewer weak policies (28% of deployed policies) than countries like Greece where the adoption rate is much lower (36%) but there are more weak policies (39%). This indicates that countries where such policies are more widely used are also more security-aware.

In a similar study, in 2006, Grossman [28] examined the crossdomain policy files for the top 100 sites in the U.S.A., according to Alexa, and the websites of the Fortune 500 list of companies. With a penetration of 36% and 8% respectively, Grossman found 6% and 2% to be wildcarded for any-domain. In January 2011, we re-examined the two lists. Figure 2.3 presents our findings along with the findings of the Grossman study. We see an increase in the penetration of cross-domain policies in 2011. More than 60% of the sites implement a cross-domain policy in the Alexa U.S.A top 100, an increase of almost 173% since 2006. Alarmingly, the number of weak policy deployments has also increased to 21%, exhibiting a growth of 124% since 2006. This indicates that, since 2006, more sites have adopted cross-domain policies, which is not surprising considering the capabilities of Flash technology. What is surprising though, is that the percentage of sites implementing weak policies, has also increased and its increase is almost 70% of the growth rate of Flash technology adoption. In the case of the Fortune 500 sites, the penetration of such policies has doubled while the ratio of weak implementations has remained the same. The two groups of sites have a dice coefficient of 0.14, indicating two very different sets; the first one is more dynamic in terms of Flash technology adoption than the latter and appears to have more secure policies. At the same time, however, it also exhibits an increasing trend towards weak implementations.

2.2.2.3 Category Penetration

Apart from examining local sites in each country, we examine specific site categories as provided by Alexa. More specifically, we inspect the 500 most popular sites in a series of categories. Table 2.3 summarizes our results.

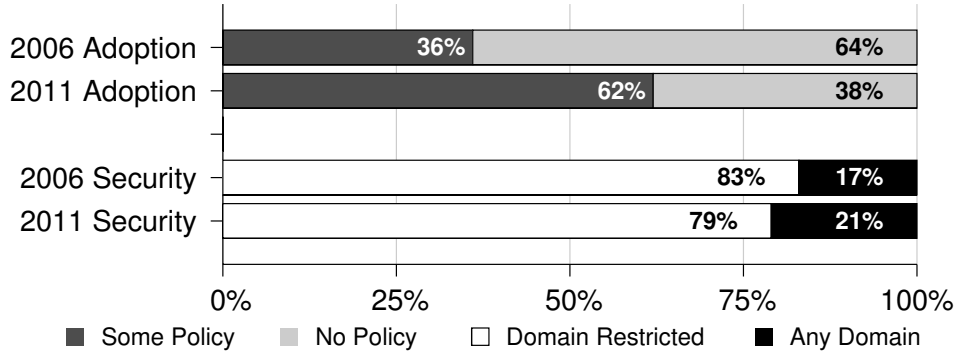


Figure 2.3: Alexa Top100 US

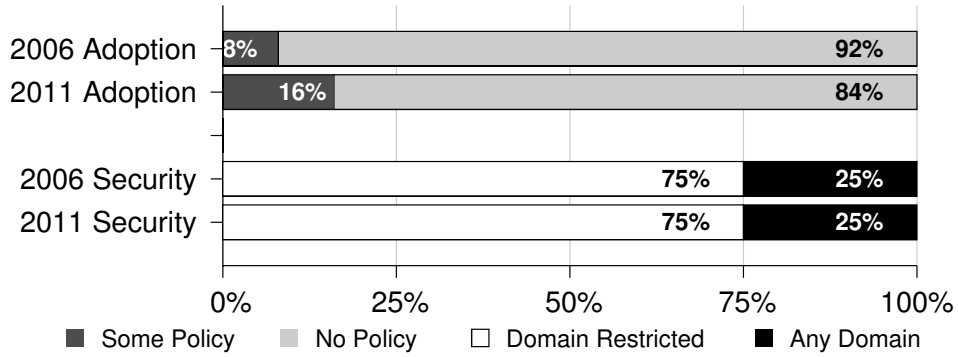


Figure 2.4: Fortune500

Cross-domain policies are adopted in 16% to 46% of the sites, depending on the category. We can see that shopping sites are more aware of the security implications, with only 17% having unrestricted policies. All other categories have unrestricted policies in a larger percentage, ranging from 28% and up to 51%. We also examined the sites for Microsoft Silverlight, but omit the results due to the small adoption rate (less than 2% in all cases) and the lack of space.

Greek E-Shopping Sites: To examine both local and category specific websites we assembled a set with 500 popular Greek e-shopping sites by crawling the catalog of Skroutz [2], a popular Greek e-shopping search engine. Although, the adoption of cross-domain policy files is rather low (2.5%), the percentage of any-domain, unrestricted-access policies reaches an impressive 83.3%. Considering the nature of these sites (i.e., online shopping involving user accounts, personal and financial details), one can imagine the impact of an attack against their users.

[2] http://www.skroutz.gr

	Adobe Flash			Microsoft Silverlight		
	None	Some Policy		None	Some Policy	
Country		**Restricted**	**Unrestricted**		**Restricted**	**Unrestricted**
U.S.A.	43%	72%	*28%*	97%	69%	*31%*
Germany	55%	63%	*37%*	99%	50%	*50%*
Great Britain	58%	66%	*34%*	98%	89%	*11%*
France	63%	62%	*38%*	99%	75%	*25%*
Greece	64%	61%	*39%*	99%	50%	*50%*

Table 2.2: Adobe Flash Cross-domain Policy and Microsoft Silverlight Client-access Policy adoption the top 500 sites across different countries.

	Adobe Flash		
	None	Some Policy	
Category		**Restricted**	**Unrestricted**
Sports	54%	49%	*51%*
Health	85%	54%	*46%*
Society	75%	55%	*45%*
Adult	84%	60%	*40%*
Arts	52%	63%	*37%*
News	62%	64%	*36%*
Science	83%	66%	*34%*
Recreation	67%	68%	*32%*
Home	82%	69%	*31%*
Computers	68%	72%	*28%*
Shopping	68%	83%	*17%*

Table 2.3: Policy adoption and security evaluation for the top-500 sites in a series of content categories.

2.2.2.4 Administrative Overhead

In the previous sections, we have seen that an important number of cross-domain policies are vulnerable to attacks. In this section, we examine whether the creation and management of these policy files require a high administrative overhead. To this extent, we measure the number of directives in each policy file found in the top 100K sites in the global set of Alexa. Without the use of wildcards for sub-domain or any-domain white-listing, each domain to be allowed access must be declared in a directive of its own. Figure 2.5 plots the Cumulative Distribution Function (CDF) of the length (number of directives) of the policy files. One may notice that 10% of those policies has more than 10 directives, 5% has more than 20 and there are sites that span over 100 or even 400 directives. Such long policies can prove hard to effectively maintain and could contain weaknesses that may go unnoticed.

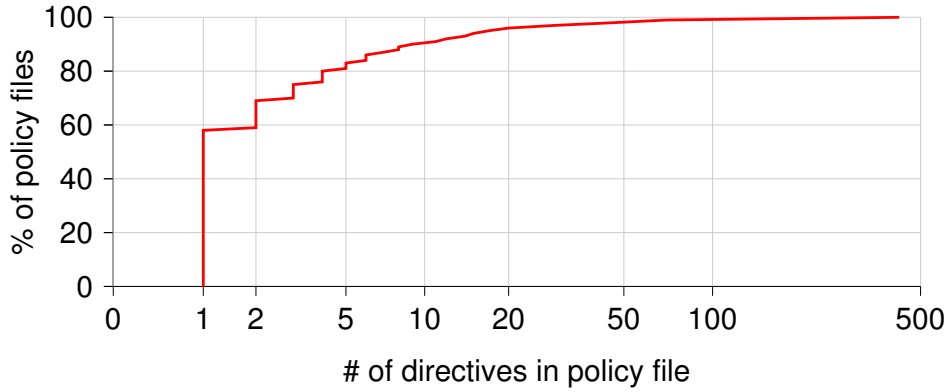


Figure 2.5: CDF of the number of directives (length) per cross-domain policy for top 100K global sites.

2.2.3 Attacks

In the previous section we showed that the percentages of cross-domain adoption and weaknesses are high among the web sites. In this section we present two attack scenarios that leverage a weak cross-domain policy and describe how they can be used for malicious purposes. Furthermore, we present a proof-of-concept implementation for exploiting such weaknesses and provide the technical details. Each scenario has been tested with at least one real case of a weak policy found during our experiments in section 2.2.2.

2.2.4 Setting up the Attack

Prior to discussing the actual attack, we first describe the steps needed by the attacker in order to use a Flash object as an attack proxy. The whole scenario is depicted in Figure 2.6. Initially, the attacker tricks her victim into loading `http://attacker.com/malicious.swf` which is a carefully crafted Adobe Flash object. This object can be masked as a movie, animation or game and thereby conceal its true purpose, or can be completely invisible (e.g., zero dimensions on an HTML page) and accessed as part of a perfectly legitimate site via an iframe in the form of an advertisement. As soon as `malicious.swf` loads, it can execute arbitrary code and open a standard network socket with a malicious server running on the attacker's home computer. Socket connections to network destinations are governed by the same rules as cross-domain site access but this is a server the attacker controls. For that matter, the Flash player inside the victim's browser issues a cross-domain access request towards the malicious server, which in return provides the necessary cross-domain policy (Listing 2.1).

As soon as the victim's Flash player examines the policy, it establishes a network connection with the attacker's server. Through that connection,

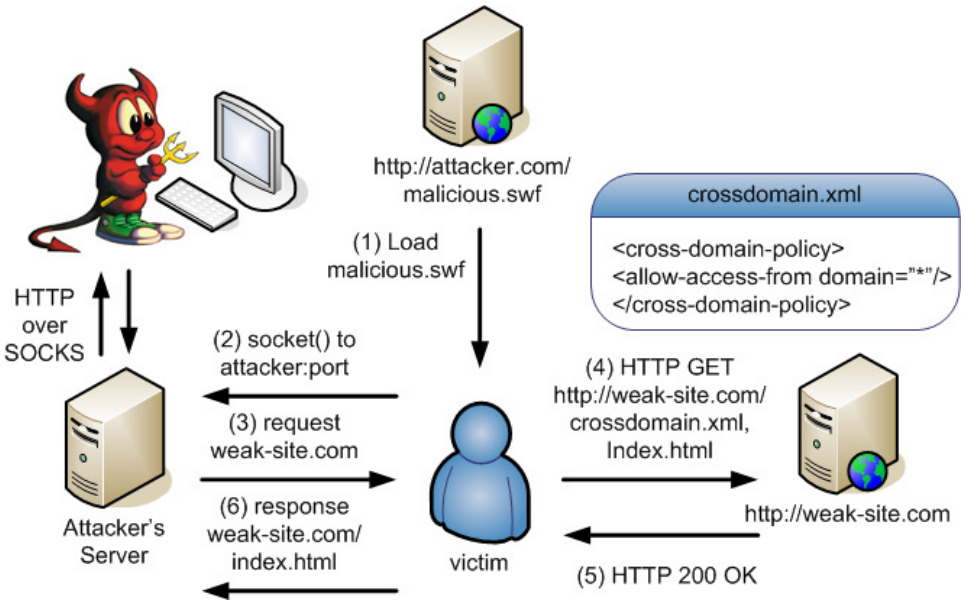


Figure 2.6: Attack Proxy Design

Listing 2.1: Cross-domain Policy enabling socket connections from the victim

```
1       <allow-access-from domain="attacker.com"
2         to-ports="8080"/>
```

`malicious.swf` receives control messages describing the URLs for which it will issue HTTP GET requests (steps 2-3 in Figure 2.6). Since, malicious.swf is executed inside the victim's web browser, the victim is the one who places those HTTP requests on the network (step 4). As soon as the request is served back to `malicious.swf` (step 5), it forwards the received content through the network socket back to the attacker's control server (step 6). At this point the attacker receives the response to an HTTP request that the victim has placed on his behalf.

The set of URLs the attacker is able to instruct the victim to fetch on his behalf, is dictated by the set of sites on the web with weak cross-domain policies. While this may seem as a serious restriction on the attacker's request-issuing capabilities, in Section 2.2.2, we found more than 6.5K sites containing weak policies. Furthermore, in special categories such as Greek e-shopping sites, more than 80% of those that have a deployed policy are vulnerable to this attack.

Steps 4 and 5 can differ, depending on the actual attack carried out. In the next sections we describe two possible attack cases. Both cases were

implemented and tested in real world scenarios, using a proof-of-concept deployment of the Figure 2.6 components.

2.2.4.1 Abusing cookies and credentials

In the first attack scenario, the attacker abuses the victim's web credentials when accessing sites with weak policies. By instructing `malicious.swf` to issue an HTTP request towards `http://site.requiring.login.com`, the victim's browser will in fact send along any cookies it has, if the victim is already authenticated for that site (e.g. victim's web mail, e-shopping account). So, for instance, the attacker could instruct the victim to fetch `https://mail.webmail.com` and receive a response with the victim's web mailbox contents or `https://shopping-site.com/cart?action=add&item=foo` to add an item to the basket. Some sites employ Cross-site Request Forgery (CSRF) tokens which are, in essence, random nonces returned to the (legitimate) user upon logging in to a site. Thus, any attacker cannot issue a direct request without knowing the nonce. However, as with login cookies, CSRF tokens are also appended by the victim's browser in the upstream path from `malicious.swf` to the destination site or can be read from the HTTP headers included in the site's responses. It should be noted that such requests are not recorded in the browsing history and do not leave other evidence, such as cache files, on the victim's computer.

2.2.4.2 Laundering web attacks

An attacker can launder various types of attacks, such as XSS and SQL injections, by issuing them through the victim's browser. The attacker's target site must have a weak policy so that the victim can issue the malicious requests even if the attacker does not require the receipt of HTTP responses. Furthermore, sites with weak security policies are also likely to be vulnerable to other attacks.

On a different note, an attacker could frame the victim by associating him with frowned-upon activity. For instance, he could abuse the victim's credentials for posting offending messages on a web discussion group, with a weak policy, that the victim is a member of or place network requests towards controversial content through the victim's browser.

2.2.5 Lessons Learned

In this work we conducted an extensive study regarding the adoption and implementation of the cross-domain policy for RIA object access. When the policy is not specifically crafted to match a website's design and structure, an adversary can leverage the weak implementation to deploy various attacks targeting the site's users. We found that 14% of the top 100K sites

have adopted cross-domain policy files, and almost 50% of them have a weak implementation. Furthermore, the top 500 sites we examined from a country-level point of view present an adoption rate of 50%, with weak implementation percentages ranging from 28% to 39%. When examining top e-shopping websites in various countries, we found up to 83% of them implementing a weak policy. We also presented real world examples of weak policy implementations and attacks that can be carried out against them. Finally, our proof-of-concept attack implementation highlighted the security implications of misconfigured cross-domain access policies.

2.3 An Architecture for Enforcing JavaScript Randomization in Web2.0 Applications

An extended version of this study can be found in [49].

2.3.1 Architecture

In this section, we present in detail the architecture of our sytsem (called RaJa). We highlight all key components that compose the architecture and code modifications we have performed in existing software. RaJa builds on well-known and highly used open source projects like Mozilla and Apache. Finally, we give a short overview in a collection of tools we have specifically build for system administrators who want to utilize the RaJa framework.

2.3.1.1 Overview

RaJa is based on the idea of Instruction Set Randomization (ISR) to counter code injections in the web environment. XSS is the most popular code-injection attack in web applications and is usually carried out in JavaScript. Thus, RaJa aims on applying ISR to JavaScript. However, the basic corpus of the architecture can be used in a similar fashion for other client-side technologies. In a nutshell, RaJa takes as input a web page and produces a new one with all JavaScript randomized. A simple example is shown in Figure 2.7. Notice that in the randomized web page all JavaScript variables (emphasized in the Figure) are concatenated with the random token `0x78`. All other HTML elements and JavaScript reserved tokens (like `var` and `if`) as well as JavaScript literals (like `"Hello World"`, `"welcome"` and `true`) have been kept intact. The randomized web page can be rendered in a web browser that can de-randomize the JavaScript source using the random token. RaJa needs modifications both in the web server and the web client, as is the case of many XSS mitigation frameworks [78, 87, 74, 50]. In order to perform the randomization, RaJa needs to run as a pre-processor before any other server-side language (like PHP) takes place. RaJa assumes that

```
1  <!-- Original Document.  -->
2  <html>
3  <script>
4  var s = "Hello World!";
5  if (true)
6    document.
7      getElementByName("welcome").
8      text = s;
9  </script>
10 <div id="welcome"></div>
11 </html>
12
13 <!-- Randomized Document.  -->
14 <html>
15 <script>
16 var s0x78 = "Hello World!";
17 if (true)
18   document0x78.
19     getElementByName0x78("welcome").
20     text0x78 = s0x78;
21 </script>
22 <div id="welcome"></div>
23 </html>
```

Figure 2.7: A typical RaJa example.

only the JavaScript stored in files in the server is trusted. Randomizing all trusted JavaScript ensures that any code injections will not be able to execute in a web browser that supports the framework.

A sample work-flow of a RaJa request-response communication is as follows. The RaJa-compliant web browser announces that it supports the framework using an `HTTP Accept` header. The web server in turn opens all files needed for serving the requests and randomizes each one with a unique per-request key. Typically, a request involves several files that potentially host JavaScript, which are included in the final document through a server-side language. For example, PHP uses `require` and similar functions to paste the source of a document in the final web response. RaJa makes sure that all JavaScript involved is randomized. Finally, the web server attaches an `HTTP X-RAJA-KEY` header field which contains the randomization key. The RaJa-compliant web browser can then de-randomize and execute all trusted JavaScript. Any JavaScript source code that is not randomized can be detected and prevented for executing. The potential code injection is logged in a file. We now look into details on how we enable RaJa in the server and client side, respectively.

2.3.1.2 Server Side

In order to enable RaJa in a web server we use two basic components: (a) an Apache module (`mod_raja.so`), which operates as content generator and

(b) a library interceptor which handles all `open()` calls issued by Apache. Although RaJa can be used with any server-side technology, for the purposes of this work we use PHP. Thus, we have configured the RaJa-enabled web server to use PHP as an output filter. For all experiments in this work we use PHP acting as an output filter, but if someone prefers to use PHP as a module and not as a filter, two Apache web servers can be used with the RaJa-enabled Apache acting as a proxy to the PHP-enabled one.

The RaJa Apache module handles initially all incoming requests for files having an extension of `.html`, `.js` and `.php`. This can be configured to support many other file types. For each request, it generates a random key and places it to a shared memory placeholder. It then opens the file in order to fulfill the request. The call to `open()` is intercepted using the `LD_PRELOAD` [30] functionality, available in most modern operating systems, by the RaJa randomizer. The latter acts as follows. It opens the file and tries to identify all possible JavaScript occurrences. That is, all code inside a `<script>` tag, as well as all code in HTML events such as `onclick`, `onload`, etc. For every JavaScript occurrence a parser, based on the Mozilla SpiderMonkey [37] JavaScript engine is invoked to produce the randomized source. All code is randomized using the token which is retrieved from the shared memory placeholder. We analyze in more detail the internals of the SpiderMonkey-based parser below.

The randomized code is placed in a temporary file and the actual `libc_open()` is called with the pathname of the randomized source. Execution is transferred back to the Apache RaJa module. The module takes care for two things. First, it attaches the correct `Content-Length` header field, since the size of the initial file has possibly changed (due to the extra tokens attached to JavaScript source). Second, it attaches the `X-RAJA-KEY` header field to the HTTP response, which contains the token for the de-randomization process. The key is refreshed per request. All randomized code is contained in an internal memory buffer. This buffer is pushed to the next operating element in the Apache module chain. If the original request is for a PHP file, then the buffer will be pushed to the PHP output filter. It is possible that PHP will subsequently open several files while processing `require()` or similar functions. Each `open()` issued by the PHP filter is also intercepted by the RaJa randomizer and the procedure is repeated again until all PHP work has been completed. The size of the final response has possibly changed again, due to the PHP processing. PHP takes care for updating the `Content-Length` header field.

We present the control flow of the RaJa architecture in Figure 2.8 with all eight steps enumerated. We now proceed and present a step-by-step explanation of a RaJa-enabled request-response communication. In Step (1), the RaJa-enabled web client requests `index.php` from a RaJa-enabled web server. In Step (2), the request is forwarded to the RaJa module which in turn in Step (3), generates a key, stores the key in a shared memory frag-

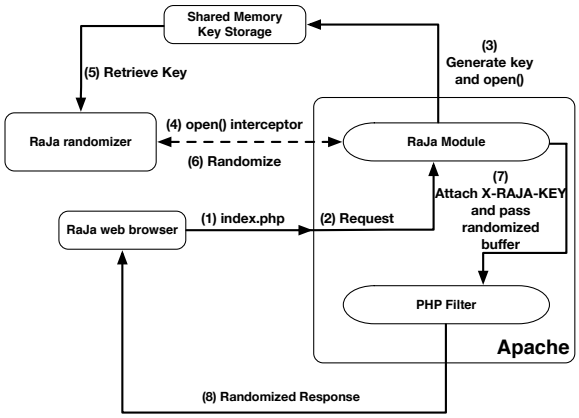


Figure 2.8: Schematic diagram of the RaJa architecture.

ment and opens the file `index.php`. In Step (4), the RaJa randomizer intercepts `open()` and in Step (5), it retrieves the key from the shared memory fragment. In Step (6), `index.php` is opened, randomized, saved to the disk in a temporary file and the actual `libc_open()` is called with the pathname of the just created file. In Step (7), control is transferred to the RaJa module which adds the correct `Content-Length` and `X-RAJA-KEY` header fields. If the file is to be processed by PHP the buffer containing the randomized source is passed to the PHP filter. All `open()` calls issued from PHP will be further intercepted by the randomizer but we have omitted this in Figure 2.8 to make the graph more clear to the reader. Finally, in Step (8), the final document is served to the RaJa-enabled web browser.

2.3.1.3 Randomization

All JavaScript randomization is handled through a custom parser based on the SpiderMonkey [37] JavaScript engine. The RaJa parser takes as input JavaScript source code and it produces an output with all code randomized. For an example refer to Figure 2.7. The original SpiderMonkey interpreter parses and evaluates JavaScript code. In RaJa execution is disabled. Instead, all source is printed randomized with all JavaScript identifiers concatenated with a random token. Special care must be taken for various cases. We enumerate a few of them.

1. Literals. All literals, like strings and numbers, are parsed and directly pasted in the output in their original form.

2. Keywords. All keywords, like `if`, `while`, etc., are parsed and directly pasted in the output in their original form.

```
1  <!-- Original Source.  -->
2  <?php if (user_exists($user)) { ?>
3   var message = <?php echo "Welcome" ?>;
4  <?php } else { ?>
5   var message = "Sign-Up Needed.";
6  <?php } ?>
7
8  <!-- Randomized Source.  -->
9  <?php if (user_exists($user)) { }?>
10  var message0x78 =
11   <?php echo "Welcome" ?>;
12 <?php } else { ?>
13  var message0x78 = "Sign-Up Needed.";
14 <?php } ?>
```

Figure 2.9: Code mixing of JavaScript with alien languages.

3. HTML comments. The original SpiderMonkey removes all comments before evaluation. The RaJa parser pastes all HTML comments in their original form.

4. Language mixing. Typically a web page has a mixture of languages such as HTML, PHP, XML and JavaScript. The RaJa parser can be configured to handle extra delimiters as it does with HTML comments and thus identify other languages, such as PHP, which heavily intermix with JavaScript. We further refer to these languages as *alien languages*.

We augment the RaJa parser to treat occurrences of alien languages inside JavaScript according to the following rules.

- *Rule 1.* An alien language occurrence is treated as a JavaScript identifier if it occurs inside a JavaScript expression.
- *Rule 2.* An alien language occurrence is treated as a JavaScript comment and is left intact if *Rule 1* is not applied.

We conclude to these basic two rules after investigating four popular and large, in terms of lines of code (LOC), web applications. By manually checking how PHP is mixing with JavaScript, we observed that in the majority of the cases PHP serves as an identifier or literal inside a JavaScript expression (see line 3 in Figure 2.9). For a short example of how these two rules are applied refer to Figure 2.9.

2.3.1.4 De-randomization

The de-randomization process takes place inside a RaJa-compliant browser. In our case this is Firefox with an altered SpiderMonkey engine. The modified JavaScript interpreter is initialized with the random token, taken from

the `X-RAJA-KEY` header field. During the parse phase it checks every identifier it scans for the random token. If the token is found, the internal structure of the interpreter that holds the particular identifier is changed so as to hold the identifier de-randomized (i.e. the random token is removed). If the token is not found, the execution of the script is suspended and its source is logged as suspicious.

We take special care in order to assist in coding practices that involve dynamic code generation and explicit execution using the JavaScript's built-in function `eval()`. Each time a correctly randomized `eval()` is invoked in a script, the argument of `eval()` is not de-randomized. Notice that this is consistent with the security guarantees of the RaJa framework, since the `eval()` function is randomized in the first place and cannot be called explicitly by a malicious script unless the random token is somehow revealed. However, this approach is vulnerable to injections through malicious data that can be injected in careless use of `eval()`. For the latter case, the RaJa framework can be augmented with tainting [101, 112, 89, 91].

Self-Correctness. In order to prove that the RaJa parser does not produce invalid JavaScript source, we use the built-in test-suite of the SpiderMonkey engine. We first run the test-suite with the original SpiderMonkey interpreter and record all failures. These failures are produced by JavaScript features which are now considered obsolete. We subsequently randomize all tests, remove all E4X [66] tests because we do not support this dialect, re-run the test-suite with the `raja-eval` (a tool capable in executing randomized source) and record all failures. The failures are exactly the same. Thus, the modified SpiderMonkey behaves exactly as the original one in terms of JavaScript semantics.

2.3.2 Case Studies

In this section we test RaJa with popular web applications. We present our experiences from deploying the framework with existing source code, which is composed by multiple flavors of server-side and client-side code. We, also, highlight various coding idioms and practices we found, while trying to enable RaJa in real-world web applications.

2.3.3 WordPress

WordPress is a popular blog engine based on PHP and MySQL. The web application is composed by approximately 150,000 lines of code as of version 2.9. The source corpus includes PHP, HTML, XML, SQL and JavaScript code, mixed up in various ways. WordPress had many security vulnerabilities in the past and thus it is considered ideal for testing RaJa with it.

We install WordPress in an Apache that supports RaJa. We use our tools introduced in Section 2.3.1 to scan all the source code of WordPress and spot

```
1  /* Case 1.  */
2  echo "<script type='text/javascript'>\n";
3  echo "/* <![CDATA[ */\n";
4  echo "document.write(...);"
5  echo "/* ]]> */\n";
6  echo "</script>\n";
7
8  /* Case 2.  */
9  $actions['quickedit'] =
10  'onclick="Reply.open(\'' .$post->ID. '\',\'edit\');"'
11
12 /* Case 3.  */
13 tinyMCEPreInit = {
14  ...
15  mceInit : {<?php echo $mce_options; ?>},
16  ...
17 };
18
19 /* Case 4.  */
20 var RecaptchaOptions = {
21  lang : '{L_RECAPTCHA_LANG}',
22  index : <!-- IF $CAPTCHA -->
23             {$CAPTCHA}<!-- ELSE -->10<!-- ENDIF --> };
24
25 /* Case 5.  */
26 onsubmit=
27   "return
28   (emptyFormElements(this, 'table')
29   &amp;&amp; checkFormElementInRange(
30   this, 'num_fields', ...);"
```

Figure 2.10: Categorization of all cases that result in faulty randomization due to interference between JavaScript and PHP.

JavaScript snippets. We attempt to randomize all 187 identified scripts. The RaJa randomizer manages to successfully randomize 169 scripts and fail to 18. The failed tests indicate that the randomizer cannot process the scripts up to the end due to code interference between JavaScript and PHP. The successfully passed scripts indicate that the randomizer succeeded on them. It is questionable if the rest of the PHP code has been modified wrongly due to the randomizer. We further proceed and check each file using PHP's lint mode to check the randomized files syntactically. All files succeed on passing the PHP syntax check.

We manually analyze all failed tests and categorize the family of problems that drive the RaJa randomizer to failure. We conclude that there are five general cases where code-mixing between PHP and JavaScript results in broken randomized source:

- *Case 1.* Partial injection of JavaScript source using the PHP built-in function `echo()`. Lines 2-6 in Figure 2.10.
- *Case 2.* String concatenation. Lines 9-10 in Figure 2.10.

```
1  /* Case 1.  */
2  <script type='text/javascript'>
3  /* <![CDATA[ */
4  document.write(...);
5  /* ]]> */
6  </script>
7
8  /* Case 2.  */
9  $content = '\'' . $post . '\',\'edit\'';
10 $actions['quickedit'] =
11  'onclick="commentReply.open('.$content.');"';
12
13 /* Case 3.  */
14 <?php $mce_options_s = "{" . $mce_options . "}"; ?>
15 tinyMCEPreInit = {
16  ...
17  mceInit : <?php echo $mce_options_s; ?>,
18  ...
19 };
20
21 /* Case 4.  */
22 <!-- IF $CAPTCHA -->
23 var RecaptchaOptions = { ... };
24 <!-- ELSE -->
25 var RecaptchaOptions = { ... };
26 <!-- ENDIF -->
```

Figure 2.11: Suggested workarounds for all cases that result in faulty randomization due to interference between JavaScript and PHP.

- *Case 3.* Partial JavaScript code generation by PHP scripting blocks. Lines 13-17 in Figure 2.10.
- *Case 4.* JavaScript code generation by using frameworks' meta languages. Lines 20-23 in Figure 2.10.
- *Case 5.* Markup injections. Lines 26-30 in Figure 2.10.

We depict examples of all five cases that force RaJa to produce faulty randomized source code in Figure 2.10. All examples are real from the source of WordPress, except for Case 3, where we use an example from phpBB and for Case 5, where we use an example from phpMyAdmin. Some examples are slightly altered for better presentation. We now proceed and discuss each case in detail and provide a workaround, where possible.

Case 1. RaJa cannot handle partial fragments of JavaScript, since the source must be first fully parsed and then randomized. It is hard for the parser to isolate portions of JavaScript injected using the PHP built-in `echo()` function and then re-assemble them in a snippet that can be successfully parsed. The example of this case in Figure 2.10 (lines 2-6) can be solved with a different coding practice suggested in Figure 2.11 (lines 2-6).

Case 2. RaJa cannot handle complex string concatenation of mixed JavaScript and PHP variables, since both languages support all quotation flavors in string literals. Thus, string quoting in PHP interferes with JavaScript quoting and it is hard to isolate one from the other. The example of this case in Figure 2.10 (lines 9-10) can be solved with a different coding practice suggested in Figure 2.11 (lines 9-11).

Case 3. RaJa treats all alien language occurrences as an identifier according to Rule 1 (see Section 2.3.1). There are some cases where PHP code is injected in a location that cannot be handled correctly as a JavaScript identifier (recall Rule 1 from Section 2.3.1). Consider the case in line 15 of Figure 2.10. The curly brackets surrounding the PHP code denote an object inside. Thus, the JavaScript interpreter does not expect to parse a common identifier. The example of this case in Figure 2.10 (lines 13-17) can be solved with a different coding practice suggested in Figure 2.11 (lines 14-19).

Case 4. RaJa cannot randomize JavaScript when is mixed with meta language elements. The JavaScript source cannot be parsed if the meta language code is removed, since the removal leaves two subsequent JavaScript identifiers. The latter produces a syntax error. The example of this case in Figure 2.10 (lines 20-23) can be solved either by extending the parser to identify some meta language elements and ignore them or by giving a different coding practice suggested in Figure 2.11 (lines 22-26).

Case 5. RaJa cannot randomize JavaScript when is mixed with HTML special characters. These HTML characters are translated from a PHP filter before the script reaches the web browser. Before the translation the JavaScript expression is invalid. For example, the most frequently occurring case is when the sequence of HTML entities `&&` is translated to `&&`, which is the logical `AND` in a JavaScript expression. We are not aware of the goal of the web programmers that use this coding tactic. The example of this case in Figure 2.10 (lines 26-30) can be solved by extending the parser to ignore all HTML special characters.

All workarounds suggested are meant for assisting RaJa to handle complex code intermixing between JavaScript and PHP. Web programmers may dislike the coding idioms we suggest and it is not our intention to enforce coding practices. However, we believe that programming aesthetics may be overlooked in favor of security. We further proceed and alter all faulty scripts according to the workarounds we suggest above. The result is a full functional and RaJa-enabled WordPress. We manage to create and administrate a sample blog using a RaJa-enabled web browser. We present an example from a randomized script from the WordPress distribution in Figure 2.12.

```
1  <script type="text/javascript">
2  <?php if ( $user_login || $interim_login ) { ?>
3  setTimeout_0x42( function(){ try{
4  d_0x42 = document_0x42.getElementById_0x42('pass');
5  d_0x42.value_0x42 = '';
6  d_0x42.focus_0x42();
7  } catch(e_0x42){}
8  }, 200);
9  <?php  } else { ?>
10 try{
11 document_0x42.getElementById_0x42('login').focus_0x42();
12 } catch(e_0x42){}
13 <?php  } ?>
14 </script>
```

Figure 2.12: Example of randomized source code from WordPress (`wp-login.php`).

Application	LoCs	Scripts	Passed	Failed
WordPress	143,791	187	169	18
phpBB	213,681	539	512	27
phpMyAdmin	178,447	263	215	48
Drupal	44,780	8	6	2
Total	580,699	997	902	95

Table 2.4: Summary of scripts that RaJa can successfully randomize in four real-world web applications.

2.3.3.1 phpBB, phpMyAdmin and Drupal

We perform similar studies for other three large and popular web applications, namely phpBB, phpMyAdmin and Drupal. In Table 2.4, we present a summary of our complete study. Specifically, we list all JavaScript snippets identified, the number of them that we can successfully randomize, as well as all scripts that fail in randomization.

All failed scripts are covered by the five cases we have already presented. It is worth mentioning that phpBB uses internally a meta-language for better code structuring (see example of Case 3 in Figure 2.10). RaJa can effectively identify and randomize most of phpBB scripts even if the language mixing is triple (PHP, JavaScript and phpBB meta-language). In Table 2.5, we present all scripts in all four web applications that failed due to one of the five cases mentioned above.

Web Application	Scripts	C1	C2	C3	C4	C5
WordPress	18	3	12	3	0	0
phpBB	27	1	0	0	26	0
phpMyAdmin	48	0	43	2	0	3
Drupal	2	0	2	0	0	0
Total	95	4	57	5	26	3

Table 2.5: Categorization of all mixed scripts in all four web applications.

2.3.4 Evaluation

In this section, we evaluate RaJa. We measure the overhead RaJa imposes on the server and client side and we test the framework with real-world attacks from a well-known XSS repository.

Server-Side Overhead. The most crucial part of RaJa performance is the server-side part. All web pages are examined for JavaScript source code and if an occurrence is found the RaJa randomizer performs a full-JavaScript parsing session. This process is repeated for every request. This is vital for the security guarantees of the framework, since the randomization key has to be refreshed in every request. Otherwise, an XSS exploit can perform a request to reveal the key and then launch the actual attack.

In order to measure the overhead imposed on the server-side part, we request a set of web pages that embed a significant amount of JavaScript using the Apache Benchmark (`ab`) [15]. The sample of web pages is collected from the SunSpider [39] suite, which constitutes a collection of JavaScript benchmarks for measuring the performance of JavaScript engines. The suite is composed by nine different groups of programs that perform various complex operations. We manually select three JavaScript tests from the SunSpider suite. The *heavy* test involves string operations with many lines of JavaScript. This is probably the most processing-intensive test in the whole suite, composed by many lines of code. The *normal* test includes a typical amount of source code like most other tests that are part of the suite. Finally, the *light* test includes only a few lines of JavaScript involving bit operations.

We use `ab` over a Fast Ethernet (FE) network. We configure `ab` to issue 1000 requests for the heavy, normal and light web pages to stress a RaJa-enabled server. Figure 2.13 summarizes the results for the case of the `ab` tool connecting to the web server through a FE connection. The RaJa server imposes an overhead that ranges from a few tens of milliseconds to about one hundred of milliseconds in the worst case (the heavy web page). While the results are quite promising for the majority of the tests, the processing time for the heavy page could be considered significant.

Client-Side Overhead. The additional overhead in the client-side originates from the fact that a RaJa compliant web browser embeds a modi-

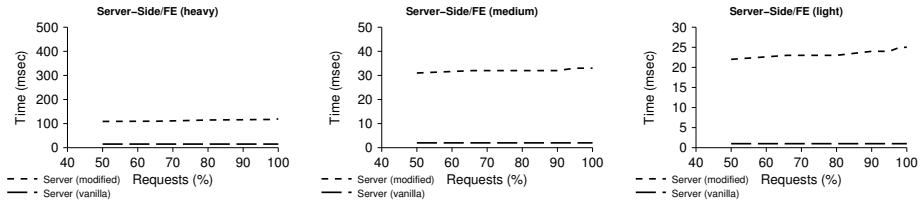


Figure 2.13: Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a Fast Ethernet link.

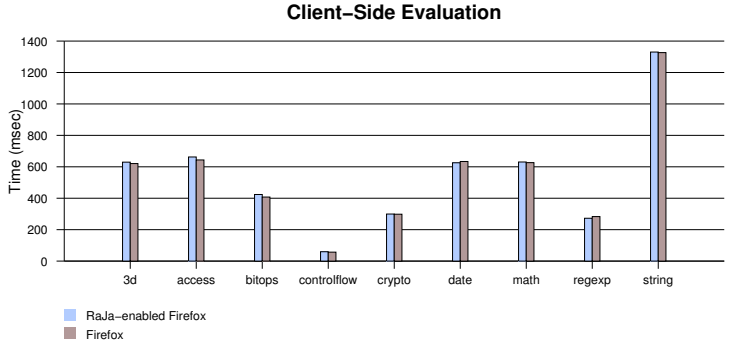


Figure 2.14: Client-side evaluation for a RaJa-enabled web browser using the SunSpider benchmarks.

fied SpiderMonkey engine, which de-randomizes each JavaScript identifier while parsing. We expect this overhead to be relatively small compared to the authentic SpiderMonkey engine, since it involves a string comparison, relatively to the size of the token, for every identifier parsed.

In order to evaluate the client-side overhead we use the SunSpider [39] suite, which includes a collection of JavaScript benchmarks. These tests are meant to stress the JavaScript engine of a web browser. The benchmarking is carried-out using JavaScript which in the RaJa case is also randomized and cannot be accounted. However, the code involved for accounting the benchmarks is significantly lesser than the code of the tests and thus the results are slightly deviate from the real overhead. We plot the benchmark results for all seven different families of tests in Figure 2.14. As expected, the RaJa -enabled web browser is slightly slower than the original Firefox.

Attack Coverage. We use the repository hosted by XXSed.com [70] which includes a few thousands of XSS vulnerable web pages. This repository has been also used for evaluation in other works [87, 50]. The majority of these exploits refers to XSS reflection attacks, excluding some `<iframe>` injections. We evaluate RaJaagainst all web sites that have not fixed the vulnerability since the exploit's publication, 1,401 in total. More precisely,

we use the exact methodology presented in [50]. As expected, all 1,401 exploits are successfully detected and prevented from executing.

2.3.5 Lessons Learned

In this work we present RaJa, which applies randomization directly to JavaScript. We modify a popular JavaScript engine to carefully modify all JavaScript identifiers and leave all JavaScript literals, expressions, reserved words and JavaScript specific constructs intact. We further augment the engine to recognize tokens identifying the existence of a third-party programming language. This is driven by two observations: First, JavaScript usually mixes with one server-side scripting language, like PHP, with well defined starting and ending delimiters; second, Server-side scripting elements when mixed up with JavaScript source act as JavaScript identifiers or literals in the majority of the cases. To verify these speculations we deploy RaJa in four popular web applications. RaJa fails to randomize 9.5% of identified JavaScript in approximately half a million lines of code, mixed up with JavaScript, PHP and markup. A carefully manual inspection of the failed cases suggests that failures are due to coding idioms that can be grouped in *five* specific practices. Moreover, these coding practices can be substituted with alternative ones.

2.4 Combining Static and Dynamic Analysis for the Detection of Malicious Documents

A more detailed description of the work can be found in [110].

2.4.1 Background

PDF, created by Adobe Systems, has become the de facto file format for the distribution of printable documents. A file adhering to the PDF specification has four main sections: a one-line header with the version number of the PDF specification; the main body of the document, which consists of objects such as text, images, fonts, annotations, or even other embedded files; a cross-reference table with the offsets of the objects within the file; and finally, a trailer for quick access to the cross-reference table and other special objects.

Besides static data, PDF objects can also contain code written in JavaScript. This allows document authors to incorporate sophisticated features such as form validation, multimedia content, or even communication with external systems and applications. Unfortunately, attackers can also take advantage of the versatility offered by JavaScript for the exploitation of arbitrary code execution vulnerabilities in the PDF viewer application.

Through JavaScript, the attacker can achieve two crucial goals: trigger the vulnerable code, and divert the execution to code of his choice. Depending on the vulnerability, the first is achieved by calling the vulnerable API function or otherwise setting up the necessary conditions. Then, through heap-spraying [67] or other memory manipulation techniques, the flow of control is transferred to the embedded shellcode, which carries out the final step of the attack, e.g., dumping on disk and then launching an embedded malware executable.

Besides exploiting some vulnerability in the PDF viewer, attackers have exploited advanced PDF features such as the `/Launch` option, which automatically launches an embedded executable, or the `/URI` and `/GoTo` options [71], which can open external resources form the same host or the Internet. Although in both cases the application first asks for user authorization, such features are quite hazardous, and after the public exposure of their security implications, they were promptly mitigated.

2.4.2 Design and Implementation

The mere presence of JavaScript code in a PDF file is not an indication of malicious intent, even if the code has been highly obfuscated. Besides hindering malicious code analysis, code obfuscation is legitimately used for preventing reverse engineering of proprietary applications. To be resilient against highly obfuscated code, MDScan analyzes any embedded code by actually running it on a JavaScript interpreter. During execution, if some form of shellcode is revealed in the address space of the JavaScript interpreter, then the input document is flagged as malicious.

Document scanning in MDScan consists mainly of two phases. In the first phase, MDScan analyzes the input file and reconstructs the logical structure of the document by extracting all identified objects, including objects that contain JavaScript code. In the second phase, any JavaScript code found in the document is executed on an instrumented JavaScript interpreter, which at runtime can detect the presence of embedded shellcode. The overall design of MDScan is presented in Fig. 4.15. Below, we describe its main components and the details of our detection method.

2.4.2.1 Document Analysis

Upon reading the input document, MDScan analyzes its structure and extracts all identified objects, which are then organized in a hierarchical structure. The complexity and ambiguities [105, 102, 115] of the PDF specification make this process a non-trivial task. In addition, most PDF viewers, including Adobe Reader, attempt to render even malformed documents, and generally do not strictly follow the PDF specification. This gives attackers

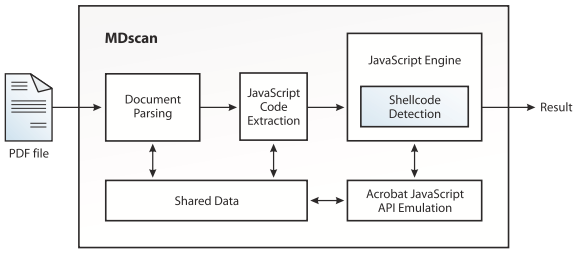


Figure 2.15: Overall architecture of MDScan.

even more room to hinder document analysis, by taking advantage of these intricacies to obfuscate the structure of malicious PDF files.

2.4.2.1.1 File Parsing File parsing begins with the extraction of all objects found in the body of the document, including objects that have been deliberately left out from the cross-reference table. In fact, the cross-reference table can be omitted altogether, as in the document shown in Fig. 2.16, along with other required (according to the PDF specification) elements, such as the `endobj`, `endstream`, and `%%EOF` keywords. Attackers can also use seemingly incorrect but actually valid keywords, such as `objend` instead of `endobj`. In general, the parser is resilient on parsing errors, and attempts to extract as much information in the best manner possible, in accordance with the behavior of the most popular PDF viewers.

After all objects have been identified, the parser proceeds to a normalization step that neutralizes any further obfuscations, and extracts semantic information about each identified object. Probably the most common object-level obfuscation technique is the use of filters to transform the object stream data and conceal the embedded JavaScript code. The PDF format supports many different filters for the decompression of arbitrary data (`/FlateDecode`, `/LZWDecode`, `/RunLengthDecode`), the decompression of images (`/JBIG2Decode`, `/CCITTFaxDecode`, `/DCTDecode`, `/JPXDecode`), or the decoding of arbitrary 8-bit data that have been encoded as ASCII text (`/ASCIIHexDecode`, `/ASCII85Decode`).

For instance, a typical use of these filters is to compress an image using JBIG2 compression, and then encode the compressed data using an ASCII hexadecimal representation. In practice, attackers can combine any number of filters to conceal the embedded malicious javascript code. Special care is taken for the correct handling of filter abbreviations such as the use of `/Fl` in place of `/FlateDecode`. Although it is straightforward to extract the encoded data by undoing each transformation, stream compression is very effective against simple detection methods such as pattern matching.

Another important aspect of the object normalization step deals with keywords that have been encoded using an ASCII hexadecimal represen-

```
1  1 0 obj <<
2    /Type /Catalog
3    /Pages 1 0 R
4    /OpenAction <<
5      /S /JavaScript
6      /JS (app.alert({cMsg: 'Hello!'});)
7    >>
8  >>
9  endobj
10
11 2 0 obj <<
12   /Title (Malicious Document)
13 >>
14
15 trailer <<
16   /Root 1 0 R
17   /Info 2 0 R
18 >>
```

Figure 2.16: A malformed PDF document that is rendered normally by Adobe Reader.

tation. The PDF format allows the arbitrary use of hexadecimal numbers in place of ASCII characters in keywords, as shown in Fig. 2.17. Similarly, strings in objects can be represented using various other encodings, such as octal or hexadecimal representations with flexible character whitespace requirements [105]. Finally, version 1.5 of the PDF specification introduced the concept of *object streams*, which contain a sequence of PDF objects. Sophisticated attackers have been using this feature for deeper concealment of PDF objects that contain malicious code by wrapping them inside object streams. MDScan handles object streams by identifying objects with a `/Type` key that has the value `/ObjStm` in the object's dictionary.

2.4.2.1.2 Emulation of the JavaScript for Acrobat API Adobe Reader provides an extensive API that allows authors to create feature-rich documents with a wide range of functionality. The JavaScript for Acrobat API is accessible as a set of JavaScript extensions that provide document-specific objects, properties, and methods. Unfortunately, attackers can take advantage of this versatile API to obfuscate further their malicious documents. This can be achieved by embedding parts of the JavaScript code, or actual data on which it depends, into objects or elements that are accessible only through the Acrobat API. The malicious code can then retrieve its missing parts or access any hidden data through the Acrobat API, and continue its execution.

For instance, some malicious PDFs use the `info` property of the Adobe JavaScript Document Object Model (DOM) to store parts of code or data, as shown in Fig. 2.17. The `info` property provides access to document meta-

```
1  ..JUNKDATA..%PDF-x.y..JUNKDATA..
2  1 0 obj <</tYpE
3  /C#61t#41log /#50#61#67#65#73 1 0 R
4  /Open#41ction<<
5    /S/JavaScript/JS(eval(
6              this.\
7  info.author);)>>>>
8  ..JUNKDATA..
9
10 6 0 obj <<
11   /Title <4D61 6C   69636
12           96F757320446F63756D656E 74>
13   /Author(app.al\145rt(
14           {cMsg: 'Hell\157!'});)
15 >>
16
17 trailer<</Root 1 0 R/Info 6 0 R>>
18
19 ..JUNKDATA..
```

Figure 2.17: An obfuscated version of the document shown in Fig. 2.16 that is still rendered normally by Adobe Reader.

data such as the document title, author, copyright notice, and so on. Other objects that can hold data supplied by the attacker include annotations, XML specifications for embedded forms [59], or even the document pages themselves. For example, the script can read the actual words of a page using the `getPageNthWord` function.

It is clear from the above that the proper execution of the code embedded in a malicious PDF file requires an environment that provides the functionality offered by the Acrobat API. Unfortunately, standalone JavaScript engines such as SpiderMonkey [37], which is the engine used in MDScan, do not support this API and are not aware of the Adobe JavaScript DOM, since both are proprietary. In MDScan, we resolve this issue by augmenting the JavaScript engine with our own implementation of the DOM parts and the API calls that are most frequently used in malicious PDF documents. We have followed an incremental approach, adding more functions according to the ones found in the samples that we have encountered so far. After the completion of the data parsing and normalization steps, MDScan analyzes the identified objects and reconstructs the hierarchical structure of the DOM objects needed for the emulation of the implemented API calls.

2.4.2.1.3 JavaScript Code Extraction After all extracted objects have been analyzed, we need to identify the objects that contain JavaScript code, and reconstruct the entire code image that will be fed to the JavaScript engine for execution. According to the PDF specification, objects that contain JavaScript code are denoted by the keyword `/JS`. The code can be located

either in the object itself, or in some other object linked to the parent object through an indirect reference (or a chain of indirectly linked objects).

At this point, we aim at recovering only the initial JavaScript code that is set to run automatically when the document is opened. A common practice of malicious PDF authors is to scatter this code across many objects with the aim to hinder detection and analysis. However, no matter into how many objects the code has been split, in order for the original code to be executable when the document is opened, the respective objects (or their associated parent objects) should all have been marked as containing JavaScript code using the `/JS` key. Any parts of the code that have been concealed into other non-code PDF objects are not relevant at this stage, since they will be retrieved at runtime through the appropriate API calls.

Having located the objects with the initial code to be executed, a crucial next step is to identify the entry point of the code. This can be achieved by looking for objects with specific declarations that denote immediate execution of the object's content, such as `/OpenAction`, `/AA`, `/Names`, and others [43]. The code of these objects is placed at the very bottom of the whole reconstructed code, so that it follows any previous function or variable declarations.

Another important aspect of the code extraction phase is the order in which the code fragments are arranged before being loaded on the JavaScript engine. For example, an attacker can place a statement that assigns a variable with the string representation of the shellcode in a PDF object, and access that variable from code located in another object. If the code of the second object (variable access) precedes the code of the first object (variable definition), the JavaScript interpreter will issue a reference error. In most cases, the correct order of the code chunks can be inferred from the inherent ordering of the PDF objects in the file, and the chains of indirect references. However, we also use some additional heuristics to identify any use-before-declaration conditions, and reorder the respective code chunks appropriately.

2.4.2.2 Code Execution and Shellcode Detection

Having extracted the embedded code, MDScan proceeds into the dynamic analysis phase, in which the code is executed on a JavaScript interpreter. In most malicious PDF files, the goal of the JavaScript code is to trigger a vulnerability in the PDF viewer, and divert the normal execution flow to the embedded shellcode. The shellcode can be initially concealed using multiple layers of encryption or transformations, such as UTF-encoded characters, `eval` chains, mapping tables, or other complex custom schemes. However, during execution, its actual binary code will eventually be revealed into a contiguous buffer referenced through a JavaScript string variable [98, 67].

Strings in JavaScript are immutable, and thus a modification to an existing string results to the allocation of a new memory buffer. This allows us to detect a PDF document that contains malicious JavaScript code by scanning each newly created string for the presence of shellcode—a benign document would never execute JavaScript code that carries any form of shellcode. To that end, we have instrumented SpiderMonkey to scan the memory area of each allocated string using Nemu [97], a shellcode detector based on binary code emulation. The runtime heuristics of Nemu can identify the most widely used types of Windows shellcode, including egg-hunt shellcode [97], which has been widely used in malicious PDFs [118].

Our approach is analogous to the one used by Egele et al. [67] for the detection of drive-by download attacks. In their system, the JavaScript engine of Mozilla Firefox has been instrumented to detect the presence of shellcode during the execution of malicious scripts embedded in rogue web pages. Unfortunately, we cannot directly modify the JavaScript engine of Adobe Reader since its source code is not available. An alternative approach would be to intercept the routines of the memory allocator used by Acrobat Reader through library interposition, and scan each newly allocated buffer, similarly to the design of Nozzle [98]. An advantage of this technique is that it eliminates the need for custom document parsing, data and code extraction, and emulation of the JavaScript for Acrobat API.

However, we have designed MDScan with the aim to be used as a standalone PDF scanner, and not as a protection enhancement for existing PDF viewers. This allows MDScan to be easily embedded as an additional detection component in existing intrusion detection systems, virus scanners, or proxy servers. In contrast, a detector integrated with the actual PDF viewer application, in the same spirit as the above browser-embedded systems [98, 67], cannot be easily used as a standalone component. Indeed, this would at least require a fully-blown virtual machine running Windows to host the instrumented viewer, and the viewer should be restarted for every input file. In fact, this design is being used by malicious code analysis systems like CWSandbox [114, 75], which can provide a detailed analysis of the actions and OS-wide side effects of malicious PDF files.

2.4.3 Experimental Evaluation

In this section we present the results of the experimental evaluation of our prototype implementation. First, we evaluate the detection effectiveness of MDScan using real PDF samples. We, then, evaluate the overall processing throughput, as well as the individual overhead of each analysis phase. For our experiments, we used a diverse set of 197 malicious documents gathered from public malware repositories and malicious websites [1, 2, 3, 21], as well as from individual sources. The above set also includes nine samples generated using the nine different PDF exploit modules of the Metasploit

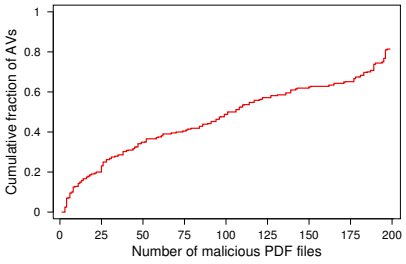


Figure 2.18: Cumulative fraction of the virus scanners of VirusTotal that detected a set of 197 malicious PDF samples.

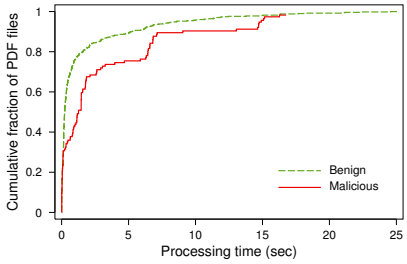


Figure 2.19: Cumulative distribution of the processing time for malicious and benign PDF samples.

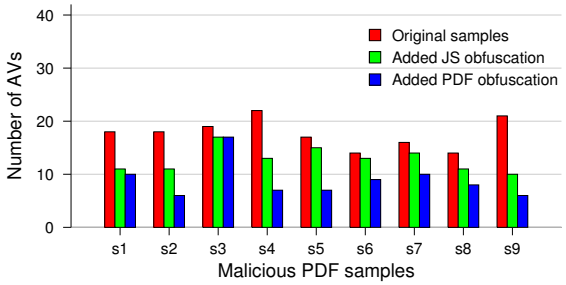


Figure 2.20: Number of virus scanners (out of 41) of VirusTotal that detected obfuscated versions of malicious PDF files generated with Metasploit.

Framework [4]. We also used a set of 2,000 randomly chosen benign PDF files that we found on Google.

2.4.3.1 Detection Effectiveness

We began our evaluation by testing the detection effectiveness of MDScan using real malicious PDF samples. From the 197 malicious files, MDScan successfully detected 176 (89%). From the files that were not detected, 15 did not attempt to exploit any arbitrary code execution vulnerability, but relied other features such as `/Launch` and `/URI`, as discussed in Sec. 2.4.1. We plan to extend the PDF parsing module to detect these types of attacks by checking the extracted objects for the relevant keywords. The remaining six samples were not detected due to faults during the parsing phase, which we have been investigating.

For comparison, we submitted all samples to VirusTotal [41] and retrieved the results from 41 antivirus engines (AVs), which we have plotted in Fig. 2.18. About half of the samples were detected only by half or less of the AVs, while 24 samples were detected by 20% or less. Even for the most detectable samples, there were about 20% or more of the AVs that did

not detect them. We also submitted all samples to Wepawet [72, 60], which reported 119 files as malicious, 16 as suspicious, 58 as benign, while four resulted to error.

To test further the effectiveness of existing antivirus systems against PDF threats, we created variants of the nine Metasploit samples by applying additional obfuscation techniques. The first derived set was generated by obfuscating the JavaScript code of the original samples using a publicly available code obfuscator [5]. The second set was generated from the previous one by removing any PDF filter encodings from the objects that contained JavaScript code. We, then, treated the exposed JavaScript code in each object as a string, and encoded it using its hexadecimal representation.

As shown in Fig. 2.20, in most cases the original Metasploit samples were detected by less than half of the AVs. The additional obfuscation applied in the samples of the other two sets reduced the detection rate significantly, with only ten or less of the AVs detecting the malicious files in the third set. The only exception is the sample number three, which is detected by almost the same number of AVs irrespectively of the applied obfuscation, due to the inclusion of encrypted mediabox objects that were not altered by our modifications. MDScan successfully detected all 27 samples.

Finally, we tested MDScan for false positives using the set of benign files. After verifying that all 2,000 files were reported as benign by all AVs of VirusTotal, we scanned them using MDScan, which did not misclassify any of them.

2.4.3.2 Runtime Performance

We measured the processing time of MDScan for both malicious and benign samples. We repeated each experiment ten times and report the average values. Figure 2.19 shows the distribution of the processing time for all samples in our two datasets, and Fig. 2.21 shows the breakdown of the average scanning time for the two sets. As expected, most of the processing time for malicious PDF files is spent on the emulation of the JavaScript code, which is a much more CPU-intensive operation compared to file parsing and code extraction. The average processing time for malicious inputs is just less than three seconds, with about half of the files being scanned in less than one second.

The average processing time for benign PDF files is 1.5s, with about 80% of the files being scanned in less than one second. In contrast to the malicious files, the amount of time spent on code execution is negligible, since only a small fraction of files contain JavaScript code. Instead, due to the very large size of some of the files, the time spent on parsing and analysis of the PDF objects in each file is significant.

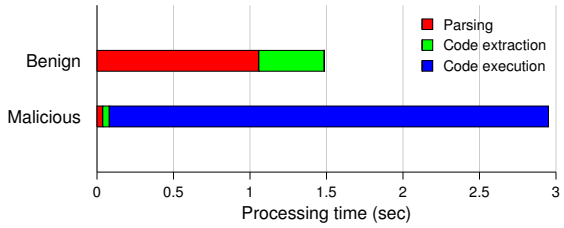


Figure 2.21: Average processing time for malicious and benign samples.

2.4.4 Lessons Learned

Malicious PDF files remain an important threat, requiring effective and robust detection mechanisms. As we have demonstrated, the effectiveness of existing antivirus systems against malicious PDF files is quite modest, given that in most cases the samples were well known and quite old, and at the same time is highly affected by the application of simple obfuscation techniques. MDScan is not affected by JavaScript code obfuscation, and is robust against most of the known obfuscation techniques based on intricacies of the PDF format specification. At the same time, it does not rely on any specific vulnerability or exploit features, which allows the detection of previously unknown threats. Combined with its standalone design, we believe that these features make MDScan an effective detection component for larger network or host-level attack detection systems. However, due to its emulation of the JavaScript for Acrobat API, MDScan will probably need to be combined with VM-based analysis systems in case PDF threats start to employ more advanced or diverse API calls.

3 Attacks on Smart and Mobile Devices

3.1 Introduction

In this chapter, we present our work related to attacks on smart devices like smartphones or tablet PCs. In Section 3.2, we present techniques that can be used in order to evade Android runtime analysis through emulation detection and we evaluate these techniques in different Android dynamic analysis tools. The work we conducted raises important questions about the effectiveness of existing analysis systems for Android malware. For this reason, we propose a number of possible countermeasures for improving the resistance of dynamic analysis tools for Android malware against VM detection evasion. Then, in Section 3.3, we proceed to present a threat scenario that targets WebView apps and show its practical applicability in a case study of over 287,000 Android apps. In a nutshell, the benefit of a better user-experience that webview provides comes at the cost of serious security implications. In case of a server compromise of just a single, vulnerable app, the consequences can be severe: For example, we found that seemingly harmless, simple apps can exceed 500,000 installs. The resulting multiplication effect is enormous: by compromising only one server, the attacker gains access to a huge number of mobile devices. Consequently, WebView's JavaScript support should be used with extreme caution

3.2 Evading Dynamic Analysis of Android Devices

For more information, an extended version of this work can be found in [93].

Category	Type	small Examples
Static	Pre-initialized static info	IMEI has a fixed value
Dynamic	Dynamic info does not change	sensors produce same values
Hypervisor	VM instruction emulation	Native code runs differently

Table 3.1: A summary of the main types of VM detection heuristics.

3.2.1 Anti-analysis Techniques

Anti-analysis techniques that can be employed by Android apps to evade detection can be classified in three categories: (a) *static heuristics*, based on static information always initialized to fixed values in the emulated environment, (b) *dynamic heuristics*, based on observing unrealistic behavior of various sensors, and (c) *hypervisor heuristics*, based on incomplete emulation of the actual hardware. Table 3.1 provides a summary of all categories, along with some representative examples.

3.2.2 Static Heuristics

The *static* set includes heuristics that can be used for detecting emulated environments by checking the presence and the content of unique device identifiers, such as the serial number (device ID), the current build version, or the layout of the routing table.

Device ID. Each smartphone contains an IMEI (International Mobile Station Equipment Identity), which is a unique number identifying it in the GSM network. The IMEI has already been used by malicious Android apps to hinder analysis by malware detection tools running on emulators [6]. Another mobile device identifier is the IMSI (International Mobile Subscriber Identity), which is associated with the SIM card found in the phone. Our simplest evasion heuristics are based on checking these identifiers, e.g., whether the IMEI is equal to `null`, which is true for the default configuration of Android Emulator. We will refer to this kind of heuristics using the abbreviation **idH**.

Current build. Another way to identify emulated environments is by inspecting information related to the current build, as extracted from system properties. For instance, the Android SDK provides the public class `Build`, which contains fields such as `PRODUCT`, `MODEL`, and `HARDWARE`, that can be examined in order to detect if an application is running on an emulator. For example, a default Android image on an emulator has the `PRODUCT` and `MODEL` fields set to `google_sdk`, and the `HARDWARE` field set to `goldfish`. We have implemented a number of heuristics based on this kind of checks, to which we refer as **buildH**.

Routing table. An emulated Android device by default runs behind a virtual router within the 10.0.2/24 address space, isolated from the host machine's network. The emulated network interface is configured with the IP address

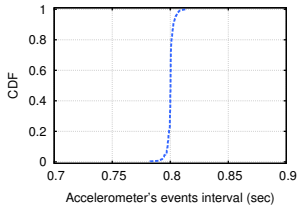


Figure 3.1: CDF accelerometers' events intervals in an Emulator.

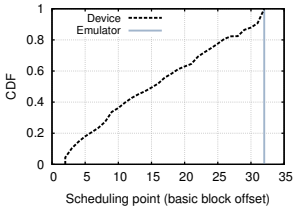


Figure 3.2: CDF of scheduling events in device and Emulator.

10.0.2.15. The configured gateway and DNS servers have also specific values. We use these networking properties as another detection heuristic. Specifically, the heuristic checks listening sockets and established connections (through `/proc/net/tcp`), and attempts to find a port number associated with addresses 10.0.2.15 and 0.0.0.0, as an indication of an emulated environment. We refer to this heuristic as **netH**.

3.2.3 Dynamic Heuristics

Mobile phones are equipped with a variety of sensors, including an accelerometer, gyroscope, GPS, gravity sensor, etc. Essentially, these sensors output values based on information collected from the environment, and therefore simulating them realistically is a challenging task. The existence of sensors is a key difference between smartphones and conventional computing systems. The increasing number of sensors on smartphones presents new opportunities for the identification of actual mobile devices, and thus for the differentiation and detection of emulators. For instance, there are studies focused on smartphone fingerprinting based on sensor flaws and imperfections [7, 63]. Such fingerprinting approaches can be leveraged for the detection of emulated environments.

By default, the Android emulator cannot simulate device movements; this can be achieved through additional sensor simulators [8]. Current builds of the Android Emulator also support partially or not at all simulation of other types of sensors. In our testing of the available simulated sensors, we found that they generated the same value at equal time intervals equal in average to 0.8 second with negligible standard deviation (equals to 0.003043). The CDF of the intervals between accelerometers' events as observed in an Android Emulator running for a couple of minutes is shown in Figure 3.1. We found that the CDF for the rest of the sensors in Android Emulator follows a similar pattern. We implemented our sensor-based heuristics by taking advantage of the `SensorManager` [9] class of the Android API. We developed an Android Activity that attempts to register a sensor listener to monitor its output values using the following approach. First, we try to register a sensor listener. If the registration fails, then the execution proba-

bly takes place in an emulated environment (except in the case of an actual device that does not support the specific sensor). Otherwise, if sensor registration is successful, then we check the `onSensorChanged` callback method, which is called when sensor values change. If the sensor values or time intervals observed are the same between consecutive calls of this method, then we assume that the app is running on an emulated environment and we unregister the sensor listener. We implemented this sensor-based heuristic for the accelerometer (**accelH**), magnetic field (**magnFH**), rotation vector (**rotVecH**), proximity (**proximH**), and gyroscope (**gyrosH**) sensors.

3.2.4 Hypervisor Heuristics

Identifying QEMU scheduling. Our first hypervisor heuristic is related to QEMU scheduling [85], and the fact that QEMU does not update the virtual program counter (PC) at every instruction execution for performance reasons. As translated instructions are executed natively, and increasing the *virtual* PC needs an additional instruction, it is sufficient and faster to increase the virtual PC only when executing instructions that break linear execution, such as branch instructions. This means that if a scheduling event occurs during the execution of a basic block, it would be impossible for the virtual PC to be reconstructed. For this reason, scheduling in a QEMU environment happens only *after* the execution of a basic block, and never within its execution.

A proof of concept QEMU Binary Translation (BT) detection technique based on a histogram of the scheduling addresses of a thread has already been implemented [85]. In a non-emulated environment, a large set of various scheduling points will be observed, as scheduling can happen at an arbitrary time, whereas in an emulated environment, only a specific scheduling point is expected to be seen, at the beginning of a basic block, as scheduling happens after the execution of a complete basic block. We have implemented this technique and used it in our experiments as an extra heuristic, abbreviated as **BTdetectH**. In Figure 3.2, we show the different behaviors of scheduling points by running this detection heuristic on an Android Emulator and on a real device.

Identifying QEMU execution using self-modifying code. As a second heuristic, we implemented a novel QEMU detection technique (we call **xFlowH**) based on the fact that QEMU tracks modifications on code pages. The technique is based on incurring variations in execution flows between an emulator and a real device through self-modifying code.

`ARM` processors include two different caches, one for instructions accesses (I-Cache) and one for data accesses (D-Cache) [55]. Harvard architectures (like `ARM`) do not ensure coherence between I-Cache and D-Cache. Therefore, the CPU may execute an old (possibly invalid) piece of code after a newly one has already been written in main memory. This issue can be

resolved by enforcing consistency between the two caches, which can be achieved through two operations: (a) *cleaning* main memory, so as the newly written code lying in the D-Cache to be moved into main memory; and (b) invalidating the I-Cache so that it can be repopulated with the new content of the main memory. In native Android code, this can be done through the `cacheflush` function, which carries out the above operations through a system call.

We implemented an example of self-modifying (native) code that uses a memory segment with write and execute rights which is overwritten several times, in a loop, with the content (code) of two different functions, `f1` and `f2`, alternately. After each code patch, we run the code of this segment, which in turn runs either `f1` or `f2`. These are two simple functions which both append their name at the end of a global string variable, so that the function call sequence can be inferred. To achieve an alternating call sequence, we have to synchronize the caches through a `cacheflush` call as described previously.

We ran this code, along with the extra calls for cache synchronization after each patch, on a mobile device and on the emulator, with the same results—each execution produced a consistent function call sequence as determined by the loop. Then, we performed the same experiment, but this time excluded the `cacheflush` calls. As expected, on the mobile device we observed a random call sequence for each run. As the caches are not synchronized before each call, the I-Cache may contain stale instructions because it is not explicitly invalidated. Interestingly, we found that this does not happen on the emulator. Instead, the call sequence was exactly the same as in the first case, when the caches were consistent before each function call. This behavior is expected, as QEMU discards its translated block for the previous version of the code, and re-translates the newly generated code, as it tracks modifications on code pages and ensures that the generated code always matches the target instructions in memory [36].

3.2.5 Implementation

We have implemented the heuristics described in Section 3.2.1 using the Android SDK. For **BTdetectH** and **xFlowH**, we used the Java Native Interface (JNI) to invoke the native code that implements the functionality of each heuristic. We developed a simple Android application (test app) that runs our heuristics in the background, and for each one collects information about its effectiveness. The collected data is sent to an HTTP server to be stored in a local database. Moreover, we incorporated our heuristics in a set of well known Android malicious apps. For this purpose, we used Smali/Baksmali [10] along with Apktool [11], which we used for disassembling and reassembling process. The incorporation of our heuristics in malicious apps was done by patching the Smali Dalvik bytecode (generated by the

Family	Package name	Heuristic	Description
BadNews	ru.blogspot. playsib.savageknife	magnFH	Data extrusion
BaseBridge	com.keji.unclear	accelH	Root exploit
Bgserv	com.android. vending.sectool.v1	netH	Bot activity
DroidDream	com.droiddream. bowlingtime	gyrosH	Root expoit
DroidKungFu	com.atools.cuttherope	rotVecH	Root exploit
FakeSMS Installer	net.mwkekdsf	proximH	SMS trojan
Geinimi	com.sgg.sp	buildH	Bot avtivity
Zsone	com.mj.iCalendar	idH	SMS trojan
JiFake	android.packageinstaller	BTdetectH	SMS trojan
Fakemart	com.android.blackmarket	xFlowH	SMS trojan

Table 3.2: Malware samples used for our study.

disassembly process) with the Smali code of each heuristic, which was previously extracted from our developed test app. Each malicious app was modified to carry one of the implemented heuristics, as listed in Table 3.2. At first, we ran each original sample as is, both on the emulator and on a real device, and observed through the `logcat` command of Android logging system the initial spawned Android activities and services. Afterwards, we patched these components with one of the heuristics, which, depending on the detection result, decides whether to continue the execution of the component or not.

We tested the repackaged applications both on multiple emulators and actual devices to make sure that the malicious behavior is triggered only when the execution happens on a real device. Note that apart from the above changes in the produced code of malicious apps, no other additions are needed in any other parts of the APK files, except for the following cases. The idH heuristic requires the `READ_PHONE_STATE` permission explicitly declared in the Android Manifest file, to be able to retrieve information about the phone state. For the BTdetectH and xFlowH heuristics, a new folder named `lib` needs to be created inside the top level directory containing the desired native code in the form of shared libraries.

3.2.6 Experimental Evaluation

3.2.7 Data and Tools

Malware Samples. We patched a number of well known Android malicious apps with the code of our detection techniques using the process described in Section 3.2.5. We used 10 samples from different malware families with distinct capabilities, including root exploits, sensitive information leakage, SMS trojans, and so on. All tested samples are publicly available and are part of the Contagio Minidump [21]. Table 3.2 provides a summary of the set of malware samples used, along with the heuristics used in each case.

Dynamic Analysis Services. The Android dynamic analysis services and tools used in our evaluation are listed in Table 3.3. We used both standalone

Type	Tool	Web Page
Offline	DroidBox	http://code.google.com/p/droidbox/
	DroidScope	http://code.google.com/p/decaf-platform/wiki/DroidScope
	TaintDroid	http://appanalysis.org/
Online	Andrubis	http://anubis.iseclab.org/
	SandDroid	http://sanddroid.xjtu.edu.cn/
	ApkScan	http://apkscan.nviso.be/
	VisualThreat	http://www.visualthreat.com/
	Tracedroid	http://tracedroid.few.vu.nl/
	CopperDroid	http://copperdroid.isg.rhul.ac.uk/copperdroid/
	APK Analyzer	http://www.apk-analyzer.net/
	ForeSafe	http://www.foresafe.com/
	Mobile Sandbox	http://mobilesandbox.org/

Table 3.3: Android analysis tools and services used in our study.

tools available for download and local use, as well as online tools which analyze submitted samples online.

We used three popular open source Android app analysis tools: DroidBox [107], DroidScope [117], and TaintDroid [68]. All three tools execute Android applications in a virualized environment and produce analysis reports. DroidBox offers information about about incoming/outgoing traffic, read/write operations, services invoked, circumvented permissions, SMS sent, phone calls made, etc. DroidScope performs API-level as well as OS-level profiling of Android apps and provides insight about information leakage. TaintDroid is capable of performing system-wide information flow tracking from multiple sources of sensitive data in an efficient way.

In addition to standalone tools, we also used publicly available online services that dynamically analyze Android apps, briefly described below. Andrubis [113] performs both static and dynamic analysis on unwanted Android apps. SandDroid performs permission/component analysis as well as malware detection/classification analysis. ApkScan provides information including file accesses, network connections, phone calls, SMS sent, information leakage, and cryptographic activities. VisualThreat provides information spanning from network activity and data leakage to malware family detection through API correlation. TraceDroid emulates some actions, when analyzing apps, such as user interaction, incoming calls, SMS messages, which can reveal malicious intents. CopperDroid [99] is built on top of QEMU and performs out-of-the-box dynamic analysis (both OS-level and API-level) of Android malware. APK Analyzer, ForeSafe and Mobile Sandbox [104] are also dynamic analysis services, which inspect mobile malware by following similar approaches as the previously mentioned tools.

3.2.8 Methodology

To evaluate the effectiveness of our heuristics, we follow two different approaches. In the first approach we submit an application sample that incorporates *all* of the heuristics, as described in Section 3.2.5, to each online service of our set. During the analysis, the application tests the effectiveness of the heuristics by trying them all sequentially for inferring if the execution

	idH	buildH	netH	accelH	magnFH	rotVecH	proximH	gyrosH	BTdetectH	xFlowH
DroidBox	✓	✗	✗	✗	✗	✗	✗	✗	*JNI NS*	*JNI NS*
DroidScope	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
TaintDroid	✗	✗	✗	✗	✗	✗	✗	✗	*JNI NS*	*JNI NS*
Andrubis	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
SandDroid	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
ApkScan	✓	✗	✗	✗	✗	✗	✗	✗	*JNI NS*	*JNI NS*
VisualThreat	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Tracedroid	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
CopperDroid	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Apk Analyzer	✓	✓	✓	✗	✗	✗	✗	✗	*JNI NS*	*JNI NS*
ForeSafe	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Mobile Sandbox	✓	✗	✗	✗	✗	✗	✗	✗	*JNI NS*	*JNI NS*

Table 3.4: Resilience of Android dynamic analysis tools against VM evasion techniques. "✓" indicates *resilient*, "✗" indicates *vulnerable* and "*JNI NS*" indicates lack of support for JNI.

environment is emulated or not. For each heuristic tried, the app communicates the result to an HTTP server. After that, we are able to ascertain which of the heuristics were effective on identifying a virtualized environment and which were not.

In order to perform a more realistic analysis and demonstrate how such heuristics can complicate the work of various dynamic-analysis tools from drawing correct conclusions, we follow a different approach. We modify actual malicious apps so as to incorporate one of the heuristics, following the process described in Section 3.2.5, and we submit both the original sample and the repackaged one to each service. Then, we collect the generated reports and compare them to find differences in the observed behaviors.

3.2.9 Evasion Results

The results of our evaluation are summarized in Table 3.4. The first three rows correspond to standalone analysis tools, and the rest to online analysis services. We next describe our key findings based on the collected results.

Finding 1. *All the studied dynamic analysis tools and services are vulnerable to five or more heuristics.* As we can see in Table 3.4, all the tools we tested can be easily evaded using just one of the proposed heuristics. Specifically, only 6 out of the 12 tools are resilient to our simplest static heuristic **idH**, which checks the IMEI of the device. We also found that only one analysis service, Apk Analyzer, is resilient to all of the static heuristics. In addition, all of the services are vulnerable to all of the dynamic heuristics, which are based on using the sensors of the device, as well as the hypervisor heuristics that take advantage of QEMU code generation intricacies.

Finding 2. *All analysis tools failed to correctly infer the malicious behavior of the repackaged malware samples.* We compared all reports produced by all

of our studied tools manually (both offline and online ones) following our second evaluation approach, and observed that the results are consistent with those found by our first methodology.

Finding 3. *All of the studied online analysis services can be fingerprinted based on inferred information about their execution environment*. During our analysis, we did not receive any requests with results from three of the online analysis services (CopperDroid, Visual Threat, and APK Analyzer) when following the first approach. Apparently, those tools analyze apps in an environment with network connectivity disabled. Nonetheless, we observed that if an application attempted to write a file during the analysis process then the filename was reflected on the results page of all these tools.

Finding 4. *Only one of the studied tools provides information about VM evasion attempts.* We found, by analyzing the reports generated by each analysis tool, that only Apk Analyzer has the feature of detecting evasion behavior in the submitted samples, and generates relevant alerts in the generated reports (for example we observed the message: *"May tried to detect the virtual machine to hinder analysis"*). Moreover, it reports the ways used to find the evasion behavior. For example for the repackaged samples containing the **idH** and **buildH** heuristics, it mentions the VM artifact strings found in memory during the execution of the sample.

3.2.10 Countermeasures

Mobile malware with VM detection capabilities can pose a significant threat, since it is possible to evade detection and therefore nullify the work of dynamic analysis tools that run on emulators. Moreover, Google bouncer, which is the official tool used in order to detect malicious applications that are about to be published in Google Play, is based on dynamic analysis conducted in the default emulated environment of Android on top of QEMU. Google Bouncer is also vulnerable to environment detection techniques [12]. In this section we propose a set of defenses that can be applied in the current emulated environment of Android so as to make it more realistic.

Emulator Modifications. The Android Emulator can be modified easily in order to be resistant to our proposed static heuristics. Mobile device identifiers such as IMEI and IMSI which are used from our **idH** heuristic are all configurable in Android emulator. By looking at the Telephony Manager service in the Android Emulator's source code and through code analysis, one can find the place where the modem device is emulated, which is implemented as part of QEMU [13]. Thus the IMEI, IMSI, as well as other features can be modified to make the emulator resemble to a real device. The **buildH** heuristic that uses the information of the current build can be easily deceived by modifying the build properties loaded by Android Emulator. These properties are defined in the `build.prop` file of Android Emulator's

source code. Finally, the default network properties of Android Emulator can be modified to provide protection against **netH** as Apk Analyzer does.

Realistic Sensor Event Simulation. Our second set of heuristics, the dynamic heuristics, are based on the variety of sensors that a mobile device supports. As already mentioned, Android Emulator supports trivially detectable sensor simulation, with sensor events occurring at precise intervals and the produced values not changing between consecutive events. All the dynamic heuristics (**accelH**, **magnFH**, **rotVecH**, **proximH** and **gyrosH**) are based on this behavior. In order to make all dynamic heuristics ineffective, better sensor simulation is required. Nonetheless, realistic simulation of such hardware components is challenging and requires in-depth knowledge of both the range of values that these devices can produce, as well as realistic user interaction patterns. In this context, external software simulators [8] could be used or record-and-replay approaches [73], in order to simulate sensor data at real time as an additional component of the Android Emulator.

Accurate Binary Translation. Binary translation used by QEMU, on which the **BTdetectH** heuristic is based, is an essential operation of the Android Emulator, in which each `ARM` instruction is translated into the `x86` equivalent in order to be able to run on the `x86`-based host machine. **BTdetectH** is based on a fundamental operation of the Android Emulator, as already discussed in Section 3.2.1, and is not trivial to change it. One way to remedy this issue is by making the binary translation process of QEMU more accurate to the real execution used in the CPU of a device. That is, the virtual program counter has to be always updated after an instruction, as happens when instructions are getting executed in a CPU. Thus, this requires revision and expansion of the current binary translation operation in QEMU. On the other hand, this approach would end up producing a higher execution overhead, making QEMU, and consequently the Android Emulator, easily detectable (e.g., by comparing the different execution times that could be observed in an emulator and in a real device for specific operations).

Hardware-Assisted Virtualization. Another way to cope with the above issue is to use hardware-assisted virtualization, which is based on architectural support that facilitates building a virtual machine monitor and allows guest OSes to run in isolation. For example, the upcoming hardware-assisted `ARM` virtualization technology [20] can be used to avoid the process of binary translation and the problems associated with it. By replacing instruction emulation (QEMU) with such technology, **BTdetectH** and **xFlowH** heuristics which are based on VM intricacies would become ineffective.

Hybrid Application Execution. Furthermore, another solution to our hypervisor heuristics would be to use real mobile devices to execute applications that contain native code. Both these two heuristics (**BTdetectH** and **xFlowH**) require native code execution in order to act. Hybrid application

execution would be the most secure and efficient way for a dynamic analysis tool against all the suggested evasion heuristics. That is, application bytecode can run in a patched version of Android Emulator shielded with all protection measures described above; when an application is attempting to load and run native code, then the execution of the native code can be forwarded and take place on a real device.

3.2.11 Lessons Learned

In this work, we explored how dynamic analysis can be evaded by malicious apps targeting the Android platform. We implemented and tested heuristics of increasing sophistication by incorporating them in actual malware samples, which attempt to hide their presence when analyzed in an emulated environment. We tested all re-packaged malware samples with standalone analysis tools and publicly available scanning services, and monitored their behavior. There was no service or tool that could not be evaded by at least some of the tested heuristics. The work we conducted raises important questions about the effectiveness of existing analysis systems for Android malware. For this reason, we proposed a number of possible countermeasures for improving the resistance of dynamic analysis tools for Android malware against VM detection evasion.

3.3 A Study of WebView-related vulnerabilities in Mobile Applications

For more information there exists an slightly extended version of this work [90].

3.3.1 Background

With the rise of Web 2.0 and its technologies, the web shifted from static to dynamic content, enabling the advent of social networks and peaking in the current state of web apps that strive to rival their full-blown desktop counterparts. Parallel to this development, another sector enjoys undiminished growth: smartphones and their mobile device siblings, i.e., tablets. Inevitably accompanied by these trends is the fact that web content consumption shifts from desktop computers to mobile devices.

On mobile devices, end-users expect functionality to be delivered as a standalone app. In order to make the life for developers easier, all major mobile platforms, such as Android, iOS, Windows Phone and Blackberry introduced *WebView*. WebView is essentially a browser-library that enables developers to deliver web content, or even a whole web application as part

of their smartphone client app. It is geared towards ease of use: fetching and displaying web content is a matter of a single method invocation. Using WebView, the developers do not need to re-implement and maintain their web application for every single platform. In addition, updates are distributed instantaneously and without requiring any user interaction: the developer just needs to change the content delivered by the web server.

While a pure browser-based solution would feature the same benefits, the main advantage of choosing WebView is the streamlined integration of device functionality. By making persistent storage, access to the short message service and other functionality available to the web application, the resulting apps are both flexible like web applications and powerful like ordinary applications. Typically, the developer exposes the needed APIs via a JavaScript-interface that can then be accessed from within web application JavaScript code.

The security implications of this feature are obvious: by providing a direct bridge between web content and the operating system, WebView punches a hole in the browser sandbox containment. If an attacker manages to serve malicious content to a WebView-enabled app, she will have access to all APIs that have been exposed via JavaScript.

Previous work in this area is scarce, Luo et al. [84] pick up attack vectors on WebView (as does [45]), but do not delve into the actual exploitation of apps. Bhavani [52] discusses an orthogonal problem on how a malicious app may harm a benign web page via WebView. Finally, Fahl et al. reveal orthogonal security problems in Android's SSL handling [69].

In this work, we discuss two realistic threat scenarios that target WebView. We continue by presenting case studies on apps that we have successfully exploited. Based on the insights of the case studies, we conducted an analysis of over 287k Android apps to check for WebView-related vulnerabilities.

3.3.2 Threat Scenario

A fundamental requirement for exploiting a WebView app is to gain control over the web content that is requested by the app. To access the exposed APIs, the attacker needs to inject JavaScript code that is subsequently executed by the app. Depending on time and location of the manipulation, we can distinguish between two possibilities:

Server compromise. If the attacker manages to manipulate the content stored on the server, the attack leverage is very high, since every single installation of the targeted app will be affected. The server compromise can be achieved by arbitrary means, as long as parts of the web content can be manipulated – a typical example being a stored cross-site scripting or SQL injection attack (see Figure 3.3(a)). A great advantage of this attack vector

is that the attacker does not need to take encryption into account, as the server will take care of it.

Traffic compromise. While compromising a tightly secured server might prove difficult, manipulating the traffic on its way can be an equally capable alternative. In a typical man-in-the-middle (MITM) attack, the adversary injects the malicious code in transmitted HTML or JavaScript (see Figure 3.3(b)).

With mobile devices, a typical MITM attack intercepts the WiFi traffic. This can be achieved by setting up a roque WiFi access point that lures victims into connecting to them blindly. For example, the Jasager firmware of the WiFi pineapple [47] will respond to any WiFi SSID scan request and impersonate the requested network in the following.

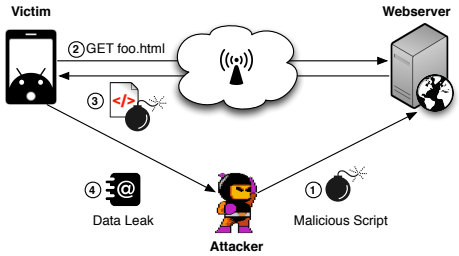
Obviously, while the MITM attack works well with plaintext, end-to-end encryption, such as HTTPS, is an issue. Since our scenario does not include direct control over the device or the app code itself, a MITM attack will only work if the app does not check certificate origins. In this case, the attacker can establish two encrypted channels, one to the web content server and one to the app, using a self-signed certificate.

Once the means to inject JavaScript code has been established, the actual exploit can be crafted. Its design depends on both the targeted app and platform.

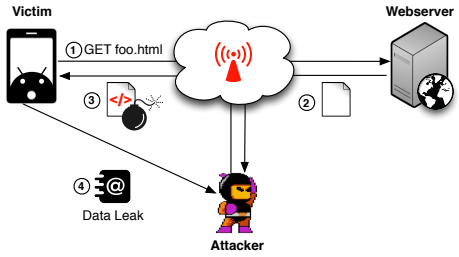
On Android, APIs can be exposed as a whole: after an invocation of `WebView.addJavascriptInterface (<object>, <js_object_name>)`, the native Java object will be available through JavaScript via the provided name. The only information an attacker requires from the app in this case is the JavaScript object name. Once determined, the latter opens up vast possibilities: Via reflection the attacker can create objects and invoke their methods as long as the app has requested the corresponding Android permissions. An even more drastic example would be to use Java's `HttpClient` to download a binary executable that then runs a root exploit (e.g. rage against the cage [44]) to escalate its privileges and circumvent the permission system altogether. Naturally, such an attack would have to cope with different devices and versions to be effective.

If an app is built using the Cordova [19] framework or its predecessor, Phonegap, exploitation is even easier. Cordova is a convenience layer that sits above WebView and provides certain JavaScript interfaces, e.g. access to contacts or the camera out of the box. In addition, it always registers a JavaScript interface object called `_cordovaNative` that can be leveraged as described above.

On iOS, the attacker's possibilities are more limited, as iOS' WebView implementation does not include a "native" JavaScript bridge. Instead, most apps implement their own bridging techniques. However, a generic Cordova



(a) Server compromise.



(b) Traffic compromise.

Figure 3.3: Example of an attacker compromising (a) the server or (b) the traffic to steal a victim's address book.

exploit can, for instance, always read the contact list by accessing Cordova's `navigator.contacts` object.

Generally, attacks that target frameworks such as Cordova are both app- and platform independent as long as they stick to the facilities supported by the framework and provided that the app is granted the corresponding permissions.

While usually suited to thwart traffic compromise, the role of end-to-end encryption in Android's WebView implementation is twisted. By default, only certificates signed by a trusted certificate authority are accepted. To accept self-signed certificates, developers have to overwrite the `onReceivedSslError` method of the WebView client.

3.3.3 Case Study

For our case study we manually analyzed and exploited four representative apps that use WebView. As a test setup we had our mobile devices connect to our own WiFi hotspot that rerouted all traffic through the mitmproxy [34].

Take Weather. This is a photo sharing app with the idea of combining weather reports on certain geographical locations with up-to-date pictures taken by the app's users. It is built based on Cordova and available for both Android and iOS. The network communication consists of JSON encoded information on the supported locations as well as the terms of use in

HTML format and a JavaScript that dynamically fetches CSS style information. Since the traffic is transmitted unencrypted using plain HTTP, we can easily inject our malicious JavaScript code. On both Android and iOS we were able to access the address book, location information and the call log. On Android we could also access other Java objects via the reflection attack described above.

Most Wanted. This app displays information on the most wanted criminals and terrorists of the United States. The requested permissions include access to camera and geolocation in order to be able to submit tips. WebView is used to directly display HTML content fetched from `http://mobileweb.cdc.nicusa.com/most_wanted_web/`. It adds a JavaScript interface to allow HTML elements to change the displayed content via a native Java object. Since the data is transmitted in plain text, it is easy to inject malicious JavaScript embedded in a `<script>` tag.

Nature Wallpaper. Who would expect harm from an app that displays nature wallpapers, has excellent ratings and features over 500,000 installations? The problem with Nature Wallpaper is that it uses a JavaScript interface to set, download and manage favorite wallpapers. Again, the traffic is unencrypted and malicious JavaScript can thus be easily injected. Since the app has the permission to access persistent storage, downloading (and executing) further malicious content would be possible.

Jiepang. This Chinese location-based social networking app offers a ”check in” service similar to Foursquare and has excellent ratings as well as over 100,000 installs. In contrast to the previous applications, the traffic is partially encrypted. However, it overwrites the default WebView SSL error handler and opens the door for attackers: Its custom implementation of the `onReceivedSslError` does not perform any error handling and simply calls `handler.proceed()`, thus accepting any certificate and loading a page without notifying the user. This circumstance and the use of a JavaScript interface exposes the app to the traffic compromise threat scenario through a (MITM) attack even in spite of the use of HTTPS. The app itself has the permissions to access persistent storage and install packages, which again would allow downloading and executing further malicious code.

3.3.4 Large Scale Evaluation

Motivated by the results of our small case study, we proceeded to the next level: To get a grip on how widespread vulnerable WebView apps are, we examined 287,512 Android apps that had been submitted to Anubis [18] from July 2012 to March 2013.

WebView usage. First, we statically analyzed how many samples of our dataset perform the necessary method invocations to allow for exploitation (see Table 3.5). Starting point is the `loadUrl` call, which fetches

Table 3.5: WebView usage

Method call	Samples	Percentage of all samples
`loadUrl`	166,751	58%
`setJavascriptEnabled`	158,042	55%
`addJavascriptInterface`	87,079	**30%**

web content from a given URL. As a next step, `setJavascriptEnabled` has to be called with the boolean value "true" in order to enable execution of JavaScript. To finally expose a native Java object via JavaScript, `addJavascriptInterface` must be called. While well above half of the apps in our dataset fetch web content using WebView, still some remarkable 30% use a Java to JavaScript bridge functionality in their app, making it vulnerable to attacks.

Table 3.6: Unencrypted HTTP app traffic

Traffic type	Samples	Percentage of samples with a JS interface
HTML	22,803	26.18%
JavaScript	11,870	14.63%
Total	23,048	26.47%

App traffic. In theory, WebView might be used to just render web content that is stored on the device, thus making injection of JavaScript code based on our threat scenario impossible. Therefore we also analyzed the traffic transmitted during dynamic analysis in Anubis. Table 3.6 shows the results on unencrypted HTTP traffic. If either HTML or JavaScript or both are contained in the traffic, it is highly likely that an injection attack would be successful. Note that the given numbers are a lower bound, as some apps might require complex user interaction (such as a login) before they can be used and transmit network traffic.

Since end-to-end encryption makes a MITM attack impossible, we also had a look at apps that use a custom implementation of the `onReceivedSslError` handler. Developers have to overwrite this method of the WebView client to accept self-signed certificates and as we have seen in the case study of Jiepang, custom implementations can be rather "simple". Table 3.7 shows that a considerable amount of samples implements a custom WebView certificate handling. To assess their complexity, we have disassembled every custom SSL handler. The result is rather shocking: over 60% of the implementations are "simple": without executing any conditional statement, they call `handler.proceed` right away.

Vulnerable apps. Based on the previous analysis results, we define an app as being vulnerable, if it implements a JavaScript bridge and either trans-

Table 3.7: WebView certificate handling

Certificate handling	Samples	Percentage of all samples
Custom SSL handling	10,175	3.54%
Simple SSL handler	6,208	2.16%

mits data unencrypted or via an SSL connection that will accept self-signed certificates. According to this definition, 27,731 samples (i.e. nearly 10% of the dataset) are vulnerable. However, not every vulnerable app is equally worth to be exploited: the gain of a successful exploitation is limited by what the app is allowed to do according to its permission set. Table 3.8 gives an overview on how many security critical permissions are granted to vulnerable apps. We have categorized the permissions into multiple groups based on which risks are associated with them.

An impressive 76% of the vulnerable samples request privacy critical permissions. Nearly 2,000 samples request the `SEND_SMS` permission that could be abused to generate revenue by sending messages to premium numbers. Finally over 60% of the samples have the necessary permission to store and run further malicious content.

Table 3.8: Permissions of vulnerable apps

Permission (group)	Samples	Percentage of vulnerable samples
RECEIVE_SMS	1,375	4.96%
READ_SMS	1,590	5.73%
WRITE_SMS	933	3.36%
SEND_SMS	1,981	7.14%
SMS permissions	3,124	11.27%
PROCESS_OUTGOING_CALLS	355	1.28%
CALL_PRIVILEGED	134	0.48%
PHONE_CALL	0	0%
Call permissions	382	1.38%
WRITE_EXTERNAL_STORAGE	16,711	60.26%
INSTALL_PACKAGES	1,241	4.48%
Installation permissions	16,727	60.32%
READ_PHONE_STATE	18,935	68.28%
READ_CONTACTS	3,304	11.91%
ACCESS_FINE_LOCATION	11,022	39.75%
ACCESS_COARSE_LOCATION	12,923	46.60%
Privacy permissions	21,197	76.44%

Libraries. By using third party libraries that employ WebView, developers may unintentionally make their apps susceptible to our threat scenario.

As we have already mentioned, frameworks such as Cordova and its predecessor Phonegap add a JavaScript bridge with a known object name per default. If an app uses unencrypted HTTP or self-signed certificates, it is thus immediately vulnerable to a generic exploit written for the framework it employs. In our dataset, 1,435 samples use Cordova and 3,881 use Phonegap. Among those, 1,111 samples (0.39%) are vulnerable according to our definition.

But Cordova and Phonegap are not the only examples of libraries that use WebView. Table 3.9 shows to which extent third-party libraries are used by the samples in our dataset. Most of the libraries are related to ad networks while some (e.g. Flurry) collect statistics to generate revenue. We list the top ten ad networks according to Appbrain [16] as well as the Flurry Analytics library, Greystripe and Jumptap from [103].

To assess whether they are safe according to our threat scenario, we have downloaded the current SDKs of all libraries and inspected their class files. Since a JavaScript interface is a precondition for the threat scenario on Android, we regard all libraries that do not make use of a JavaScript bridge, safe.

While with app development frameworks such as Cordova and Phonegap, the developer can still decide whether to use encryption and which resources to fetch, ad libraries function autonomously to a large extent. For example, Startapp receives the URL of the ad to click on via a JSON object. This URL is then directly used in a `loadUrl` call, which opens a JavaScript enabled WebView. The latter features a JavaScript interface named `startappwall`, whose corresponding Java object is used to report back to the library when the displayed ad is closed.

A full security audit would be necessary to evaluate whether the listed ad libraries that use a JavaScript bridge are truly safe. However, such an audit is out of the scope of this work.

3.3.5 Mitigation

Even if disabling JavaScript is not an option, a WebView-based app can still be hardened against attacks.

The obvious way to thwart traffic tampering is a complete end-to-end encryption. While many developers make use of HTTPS to secure their connections, they are generally reluctant to invest the money for a certificate issued by a trusted authority. Instead, they usually overwrite the default behavior of current WebView implementations and accept self-signed certificates. Consequently, these applications are prone to MITM attacks again if they do not employ countermeasures.

Such countermeasures could for example include origin checks that will drop requests that do not match a certain IP address or are not encoded

Table 3.9: Library usage

Library	Samples	Percentage of all samples	Safe?
Cordova	1,435	0.50%	-
Phonegap	3,381	1.35%	-
Admob	70,987	24.69%	✓
AirPush	13,462	4.68%	✓
Flurry	9,838	3.42%	✓
Millennial Media	8,663	3.01%	✓
MobClix	7,285	2.53%	?
LeadBolt	6,195	2.16%	?
InMobi	3,924	1.37%	?
Greystripe	1,787	0.62%	?
Chartboost	1,052	0.37%	✓
Jumptap	482	0.17%	?
Startapp	81	0.03%	?

using a predefined SSL certificate. Simple checks are usually implemented by overwriting the corresponding WebView handler methods [46].

On the operating system side, Android 4.2 has introduced a new annotation `@JavascriptInterface` that needs to be added to each method that is exposed via the JavaScript bridge. This effectively prevents reflection-based attacks. However, currently only 2.3% of all Android devices run version 4.2, with most devices still operating on Gingerbread [17].

To limit the harm that can be done once an app is actually exploited, the principle of least privilege should be followed, i.e. in the case of Android, only necessary permissions should be requested. Besides, Android WebView allows to separately turn off access to local storage through the JavaScript bridge.

3.3.6 Lessons Learned

In this work we have pointed out deficiencies in real-world apps that use WebView and analyzed over 287,000 samples based on our threat scenario. In a nutshell, the benefit of a better user-experience comes at the cost of serious security implications. In case of a server compromise of just a single, vulnerable app, the consequences can be severe: seemingly harmless, simple apps like the Nature Wallpaper in our case study can exceed 500,000 installs. The resulting multiplication effect is enormous: by compromising only one server, the attacker gains access to a huge number of mobile devices. Consequently, WebView's JavaScript support should be used with extreme caution. In order to keep app development with WebView easy,

developing a static code checking tool for WebView related vulnerabilities could be rewarding future work.

4 Attacks on Privacy

4.1 Introduction

This chapter provides our research work related with cybersecurity attacks on users' privacy. The different topics covered in this chapter contain the implementation of a framework for minimum information disclosure across third-party sites with social login interactions in order to solve the privacy issues that arise by this kind of login mechanisms 4.2. In this work, we identify and describe an increasing threat to the users' privacy: a threat which masquerades under the convenience of a single sign-on mechanism and gives third-party Web sites access to a user's personal information stored in social networks. We propose a new privacy-preserving framework for users to interact with single sign-on and OAuth-like platforms provided by social networks in their daily activities on the web. We implement a prototype of our framework as a browser extension for the Google Chrome browser. Our prototype supports the popular single sign-on mechanism "Facebook Connect" and can be easily extended to support others, such as "Sign in with Twitte". We evaluate our implementation and show that (i) it allows users to pre serve their privacy when signing on with third-party Web sites and (ii) it does not affect any open sessions they might have with other third-party Web sites that use the same single sign-on mechanisms. Then, we focus on the negative impact that social networking sites may have on privacy of their users and ways we can protect against these cyber threats in Section 4.3. In this work, we propose k-subscription, the first obfuscation-based approach to hide a user's interests in microblogging services. Our approach encourages users to follow k1 noise channels apart from each channel they want, so as to hide (i) their real interests in a set of k channels, and (ii) other users' in-terests in the microblogging service. To quantify the effectiveness of our approach, we introduce a new notion: the Disclosure Probability. This is the service's confidence that a user is interested in a specific channel. We present

an analytic evaluation of our approach and derive closed-form formulas for the disclosure probability. These formulas suggest that the disclosure probability can be made predictably small by fine-tuning the obfuscation level k. We evaluate k-subscription in a more realistic scenario using simulations, which are based on models derived from a real-world dataset with sensitive channels from Twitter. We implemented our system as a plug-in for the Chrome browser using Twitter as case study. We experimentally evaluate our prototype and show that it has minimal bandwidth requirements and negligible latency to browsing experience.

4.2 Minimizing Information Disclosure to Third Parties in Single Sign-On Platforms

There is also an extended version of this work with more details [81].

4.2.1 OAuth Protocol

The OAuth or Open Authentication protocol [35] provides a method for clients to access server resources on behalf of a resource owner. In practice, it is a secure way for end users to authorize third-party access to their server resources without sharing their credentials.

As an example, one could consider the usual case in which third-party sites require access to a user's e-mail account so that they can retrieve his contacts in order to enhance the user's experience in their own service. Traditionally, the user has to surrender his username and password to the third-party site so that it can log into his account and retrieve that information. Clearly, this entails the risk of the password being compromised. Using the OAuth protocol, the third party registers with the user's e-mail provider using a unique application identifier. For each user that the third-party requires access to his e-mail account, it redirects the user's browser to an authorization request page located under the e-mail provider's own domain, and appends the site's application identifier so that the provider is able to find out which site is asking for the authorization. That authorization request page, located in the e-mail provider's domain, validates the user's identity (e.g., using his account cookies or by prompting him to log in), and subsequently asks the user to allow or deny information access to the third-party site. If the user allows such access, the third-party site is able to use the e-mail provider's API to query for the specific user's e-mail contacts. At no point in this process does the user have to provide his password to the third-party site.

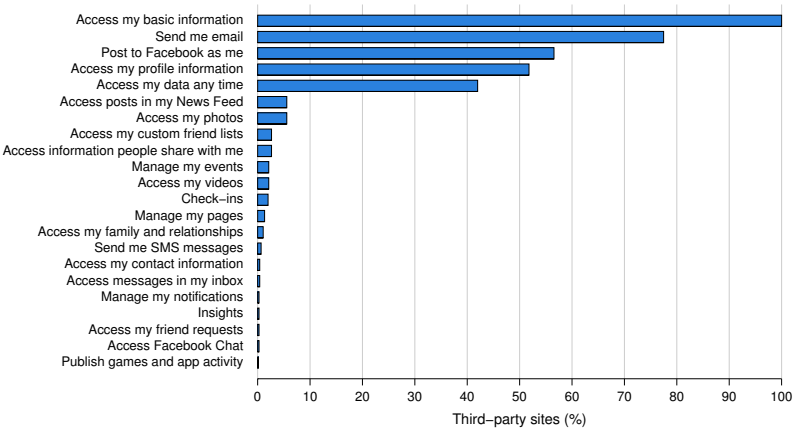


Figure 4.1: Distribution of requested permissions for a set of 755 websites that have integrated Facebook's single sign-on platform.

4.2.1.1 Facebook Authentication

Facebook's social login platform, known as Facebook Connect [24], is an extension to the OAuth protocol that allows third-party sites to authenticate users by gaining access to their Facebook identity. This is convenient for both sites and users; sites do not have to maintain their own accounting system, and users are able to skip yet another account registration and thereby avoid the associated overhead. A login with Facebook" button is embedded in a third-party Web site and, once clicked, directs the user's browser to a Facebook server where the user's cookies or credentials are validated. Upon successful identity validation, Facebook presents a "request for permission" dialog where the user is prompted to allow or deny the actions requested by the third-party Web site, for example, social plug-in interactions or access to various information in the user's social profile. However, the user is not able to modify or regulate the third-party Web site's requests, for instance to allow access to only a part of the information the site is requesting. If the user grants permissions to the site's request, Facebook will indefinitely honor API requests originating from that third-party site that conform to what the user has just agreed upon.

4.2.2 Social Login vs. User Privacy

To gain a better understanding of the type and extent of the permissions requested by third-party websites through the Facebook Connect mechanism, also known as "login with Facebook," we studied a random sample of 755 sites that have incorporated Facebook's social login platform. Figure 4.1 presents the frequency distribution of the different permissions requested by

these websites. A full list of the available permissions can be found through Facebook's developer page [23].

One may notice that all sites request access to a user's basic information. That is the minimum amount of private information a user must disclose, even if a third-party website does not really need all that information. According to its description, the basic information includes the "user id, name, profile picture, gender, age range, locale, networks, user ID, list of friends, and any other information they have made public."

Besides the basic profile information, the administrator of the third-party website may explicitly ask for additional permissions to access more user information or perform certain actions on behalf of the user. For instance, 77% of the studied sites request access to the user's e-mail address, 57% are able to post content on behalf of the user, and more than 42% require to be able to indefinitely access user information even when a user is not using the application.

Moreover, permission to manage Facebook notifications could enable malicious third-parties to hide the misuse of other permissions granted to them. What is more, access to direct messages sent or received and Facebook's real-time chat system, could seriously compromise a user's private communications. Finally, special consideration should be given to permissions that may result in real-world consequences for the user; the ability of a third-party to access information about the user's physical location (*"Check-ins"*) or send SMS messages, which may result in monetary charges.

We argue that in most of the cases the type of permissions and the amount of information requested from the user during social login are more than necessary. Even with benign third-parties, the more personal data being shared, the greater the damage in case of leaks either accidental or as a result of an attack. To give an example, one of the cases in our study is a music band which urges its fans to perform a social login when visiting its site. Although we could not confirm the presence of functionality dependent upon social login, we will give the site the benefit of the doubt. However, its requirements are over the top. It requests access to basic, contact and profile information, photos and videos, to the user's e-mail address and Facebook chat. Moreover, such access is requested even if the user is not using the site. Finally, it requests to ability to upload content to Facebook on behalf of the user, read and manage the user's events and reports on his physical location. Figure 4.2 is a screenshot of the social login dialog for that site. Its name has been anonymized. Access to all of the user's photos and videos is unjustified as is access to the user's private conversations. Furthermore, the ability to impersonate the user on Facebook is in no way restricted to purposes related to the nature of the third-party. Finally, managing all events and physical location information so it can for instance generate activity related to the band clearly demonstrates the need for fine-grained permissions. Ideally, the third-party would request access to photos tagged with

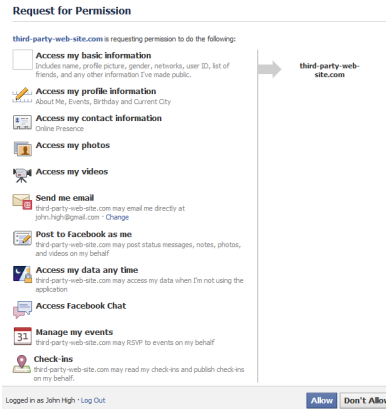


Figure 4.2: A website requesting an excessive amount of personal data.

a certain keyword related to the band, manage events and locations with specific prefixes in their names and add a "uploaded by X on behalf of user A" label to content uploaded on Facebook.

Overall, the above study confirms our intuition that the amount, type, and combination of permissions requested by third-party sites can seriously put users in a compromising position. At the same time, user reactions to a recent effort [38] that enables users to become aware of third-party applications and websites with access to their (private) social information, confirm the general request for improved control and better protection over the data one uploads to a social network. Facebook itself acknowledges the issue and, in a slight effort for remedy, offers users the option to anonymize the e-mail address they surrender to third-parties. This option is unfortunately opt-in and enabled by default in rare occasions driven by abuse-related heuristics.

Motivated by this issue, in the rest of this work we present the design and implementation of a framework for minimum information disclosure across third-parties in social login platforms.

4.2.3 Design

The modus operandi we assume in our approach is the following:

1. The user browses the Web having opened several tabs in her browser.

2. Then, the user logs in her ordinary Facebook account so as to interact with friends and colleagues.

3. While browsing at some other tab of the same browser, the user encounters a third-party website asking her to log in with her Facebook

credentials. At this point in time, our system kicks in and establishes a new and separate downgraded session with Facebook for that cross-site interaction. That session is tied to a stripped-down version of her account which reveals little, if any at all, personal information. Now:

(a) The user may choose to follow our "advice" and log in with this downgraded Facebook session. Let there be noted that this stripped-down mode does not affect the browsing experience of the user in the tabs opened at step 2 above: the user remains logged in with her normal Facebook account in the tabs of step 2, while in the tabs of this step she logs in with the stripped-down version of her account. Effectively, the user maintains two sessions with Facebook:

   i. One session logged in with her normal Facebook account, and

   ii. One session logged in with the stripped-down version of her account.

(b) Alternatively, the user may want to override our system's logic and log in with her normal Facebook account revealing her personal information; in that cases she performs a "sudo" on that particular cross-site interaction with Facebook and elevate the by-default downgraded web session.

In the description of our system we assume the use of Facebook Connect [24, 106], however our mechanisms can be extended to cover other social login platforms as well.

Figure 4.3 shows the architecture of our system. To understand our approach we will first describe in Figure 4.3 how an ordinary web browser manages session state. We see that the browser uses a default session store (Session Store [0] (default)) which stores all relevant state information, including cookies. Thus, when the user logs into Facebook (or any other site for that matter) using her ordinary Facebook account, the browser stores the relevant cookie in this default session store. When the browser tries to access Facebook from another tab (Tab 3 in the figure), the cookie is retrieved from the default session store and the page is accessed using the same state as before.

In our design, we extend this architecture by including more than one session stores. Indeed, in Figure 4.3 (bottom left) we have added "Session Store [1]" which stores all relevant information, including cookies, for the stripped-down Facebook session. This gives us the opportunity to enable users to surf the Web using two distinct and isolated sessions with Facebook at the same time: a session tied to the "normal" account is enabled in Tab 1 while a stripped-down session is in effect in Tab 3. To select the appropriate account, our system (IMF) intercepts all URL accesses and checks their HTTP

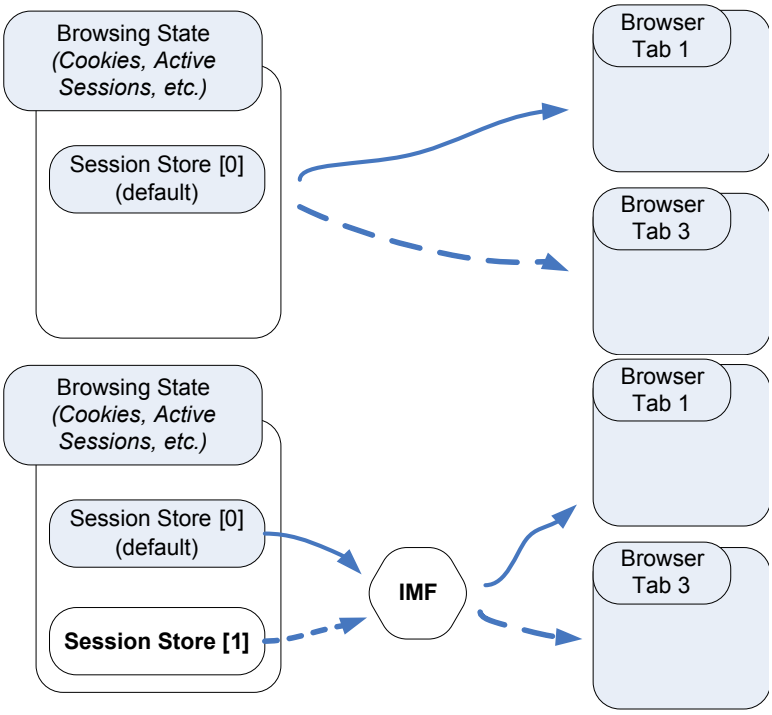


Figure 4.3: Typical communication of session state to loaded pages *v.s.* the one followed by *SudoWeb*

referrer field. If the URL points to Facebook Connect but the HTTP referrer field belongs to a different domain name, then our system suspects that this is probably an attempt from a third-party website to authenticate the user with her Facebook credentials.

Therefore, as it stands inline between the loading page and the browser's state store(s), it supplies the appropriate state (from Session Store [1]) for the stripped-down Facebook session to be employed. This is an implicit privacy suggestion towards the user. If the user disagrees, she may choose to authenticate with her ordinary Facebook account, in which case, Tab 3 will receive all cookies from Session Store [0].

We consider the proposed concept as analogous to privilege separation in operating systems, i.e., different accounts with different privileges, such as root and user accounts. Our design can scale and evolve so that it accommodates different privacy-preserving scenarios in interaction with third-party websites.

Figure 4.4 shows the modules of our system. Initially, in the upper left corner, the user browses ordinary web pages (Web Browser). When a new browser page (i.e., tab or window) is created (New Web Browser Page), the

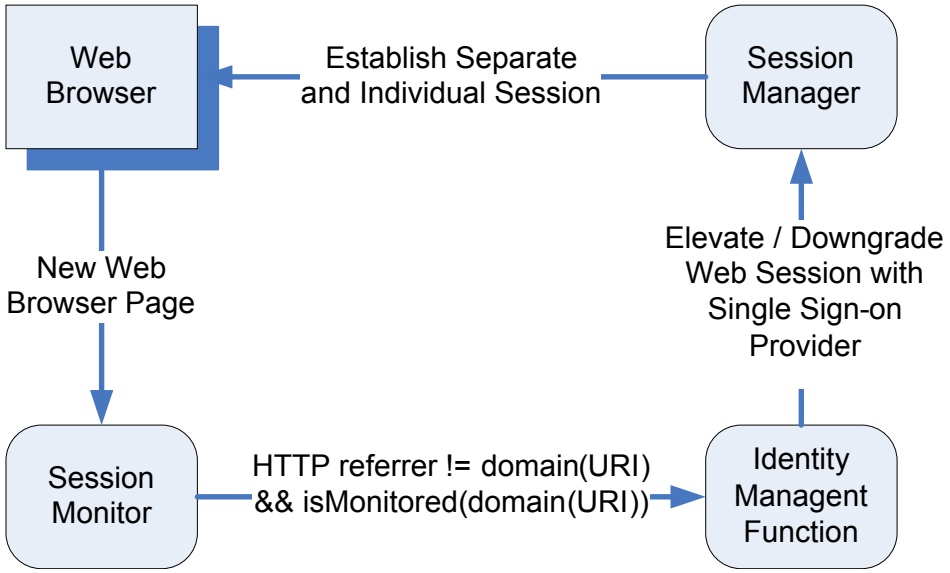


Figure 4.4: *SudoWeb* extension modules.

Session Monitor kicks in to find whether this is a social login attempt[1]. If (i) it is such an attempt (i.e., isMonitorred(domain(URI)) is TRUE) and (ii) the attempt is from a third-party website (i.e., HTTP referrer != domain(URI)) then our system calls the Identity Management Function (IMF) which employs a downgraded, stripped-down from all personal information, session for the user. From that point onwards, the Session Manager manages all the active sessions of the user, in some cases different sessions with different credentials for the same single sign-on domain. Figure 4.5 shows the workflow of our system in more detail.

4.2.4 Implementation

We have implemented our proposed architecture as a browser extension for the latest version of the Google Chrome browser[2] with support for the Facebook Connect social login platform. We find that, due to its popularity, our proof of concept implementation covers a great part of single sign-on interactions on the Web. Nevertheless, our browser extension can be seamlessly configured to support a greater variety of such cross-site social login interactions.

[1] *SudoWeb* keeps a list with all social login domains currently supported and thus monitored. If such a domain is monitored the isMonitorred(domain(URI)) function returns TRUE.

[2] As we take advantage of generic functionality in the extension-browser communication API, we find it feasible to also port the extension to Mozilla Firefox.

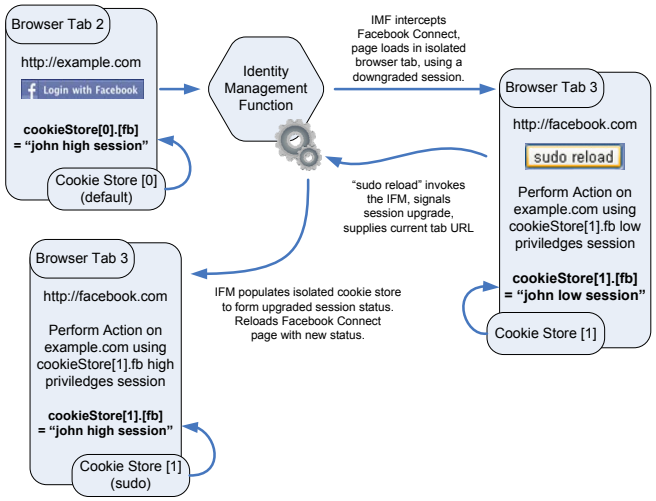


Figure 4.5: Outline of *SudoWeb*'s workflow.

4.2.4.1 *SudoWeb* Modules

Here we describe the modules that comprise our extension to the Google Chrome browser, in support of our proposed architecture.

Identity Management Function (IMF).

In the heart of the extension lies the logic module offering the identity management function or IMF. This function is responsible for detecting the possible need for elevating or downgrading a current session with a social login provider (here: Facebook).

Such a need is detected by identifying differences in the HTTP referrer domain and the URL domain of pages to be loaded. That is, when the user navigates away from a third-party website (identified by the HTTP referrer field) towards a social login website (we keep a configuration file with all social login sites supported), IMF steps in, instantiates a new, isolated and independent session store in the browser, and instructs the *session manager module* to initialize it. This allows the browser to receive state that establishes a downgraded or stripped-down session with the social login provider. Furthermore, it places a "sudo reload" HTML button on that page giving the user the opportunity to reload that page using an elevated session instead.

Session Monitor.

The Session Monitoring module plays a supporting role to the IMF. If one considers our extension as a black box, the session monitor stands at its input. It inspects new pages opening in the browser and looks for cases where the page URL belongs to a monitored domain of a social login provider (here: Facebook) but the page has been invoked through a different, third-party domain. It does so by comparing that URL with the HTTP referrer.

The referrer is an HTTP parameter supplied by the browser itself based on the URL of the parent tab or window that resulted in a child tab or window being spawned.

The session monitor notifies the IMF of such incidents and supplies the respective page URL. We should note that recent research has revealed that the HTTP referrer field in several cases can be empty or even spoofed [51, 86] undermining all mechanisms based on it. Although it is true at the network elements may remove or spoof the HTTP referrer field so that it will be invalid when it reaches the destination web server, our work with the HTTP referrer field is at the web *client* side, not at the web server side. That is, the HTTP referrer field is provided to *SudoWeb* by the web browser *before* it reaches any network elements which may remove it or spoof it.

Session Manager.

The Session Manager module also plays a supporting role to the IMF. If one considers our extension as a black box, the session manager stands at its output. Upon the installation of our extension, the session manager prompts the user of the web browser to fill in his ordinary social login (here: Facebook) account username and password, as well as his stripped-down one that is to be used for the downgraded integration with third-party websites.

The session manager maintains in store the necessary state, e.g., cookies, required to establish the two distinct sessions with the social login provider and is responsible for populating the browser's cookie store once instructed by the IMF. As a result, it stands at the output of our extension and between the browser's session store and the rendered pages that reside in tabs or windows. It affects the state upon which a resulting page rely on.

Our extension takes advantage of the incognito mode in Google Chrome to launch a separate browser process with isolated cookie store and session state so that when the session manager pushes the new state in the cookie store, the user is not logged off of the existing elevated session (here: with Facebook) that may be actively used in a different browser window.

Operation and Interaction of *SudoWeb* Modules.

In the spirit of the use case presented at the beginning of Section 4.2.3, a user of *SudoWeb* will configure the *session manager* once, continue browsing the Web, and eventually come across a third-party that wishes to interact with his Facebook identity via social login.

The configuration of the *session manager* provides the necessary information to seamlessly alternate between downgraded and elevated privileges with one or more social login providers. Figure 4.6 presents an example screenshot of such a configuration for Facebook, where the user declares the two identities that will be used for that purpose; two buttons are used to indicate to the *session manager* that the current session the user has with Facebook is to be treated either as a primary (elevated) or secondary (downgraded) session. The user logs in on Facebook with one identity, assigns one of the two characterizations, is then briefly logged out of that identity so

Figure 4.6: Screenshot of the configuration for the *session manager module*.

that he can repeat the process for the second identity, and finally his original session with Facebook is restored, making the entire process minimally disrupting. At the bottom of the configuration page, a session test, retrieving the names of the two identities from Facebook in real time, demonstrates how two parallel Facebook sessions can be seamlessly maintained.

After it has been configured, *SudoWeb* monitors the web browser for social login events. Upon the user reaching a third-party page and clicking on the "login with Facebook" button, our system kicks in:

1. The *session monitor* detects the launch of a new Facebook page from a page under the domain of the third-party website. The *session monitor* notifies the *IMF* module of our extension and so the page launch is intercepted and loaded in an incognito window, i.e., an isolated browser process with a separate and individual session store.

2. The *IMF* coordinates with the *session manager module* so that this isolated environment is populated with the necessary state for a downgraded Facebook session to exist.

The entire process happens in an instant and the user is presented with a browser window similar to figure 4.7. In this figure, we have used `third-party-web-site.com` as the name of the third part website which wants to authenticate the user using her Facebook account. We see that in addition to authenticating the user, the third-party website asks for permission to (i) send email to the user, (ii) post on the user's wall, (iii) access the user's data any time, and (iv) access the user's profile information. Although Facebook enables users to "Allow" or "Don't Allow" access to this information (bottom right corner), if the user chooses not not allow this

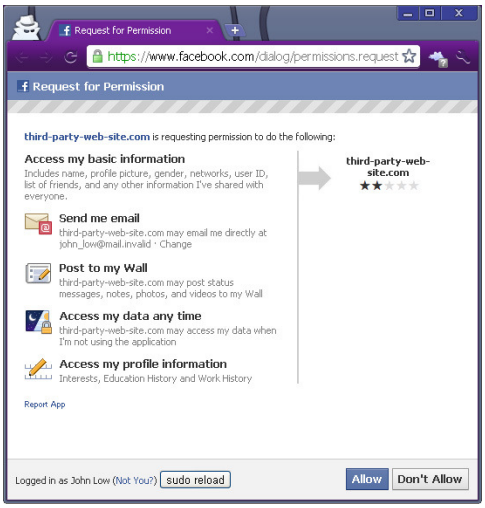


Figure 4.7: Screenshot of a Facebook "Request for Permission" page.

access, the entire authentication session is over and the user will not gain access to the content of `third-party-web-site.com`.

Having intercepted this third-party authentication operation, *SudoWeb* brings the stripped-down account (i.e., John Low) forward, on behalf of the user. Therefore if the user chooses at this point to allow access to his information by the third-party site, only a small subset of his actual information will be surrendered. Note that a "sudo reload" button has been placed at the bottom of the page, allowing the user to elevate this session to the one tied to his actual, or a more privileged, Facebook identity.

The Facebook session with which the user was surfing prior to engaging in this cross-site Facebook interaction remains intact in the other open browser windows since, as mentioned earlier, we take advantage of the browser's incognito mode to initiate an isolated session store in which we manage the escalation and de-escalation of user sessions. All the user has to do is close this new window to return to his previous surfing activity.

4.2.5 Lessons Learned

Recent results suggest that hundreds of thousands of Web sites have already employed single sign-on mechanisms provided by social networks such as Facebook and Twitter. Unfortunately, this convenient authentication usually comes bundled (i) with a request to the user's personal information, as well as (ii) the request to act upon a users social network on behalf of the user, e.g., for advertisement. Unfortunately, the user can not deny these requests, if she wants to proceed with the authentication. In this work, we explore this problem and propose a framework to enable users to authenticate on third-party Web sites using single sign-on mechanisms provided by popular social

networks while protecting their privacy; we propose that users surf the Web using downgraded sessions with the single sign-on platform, i.e., stripped from excessive or personal information and with a limited set of privileged actions. Thereby, by default, all interactions with thir d-party Websites take place under that privacy umbrella. On occasion, users may explicitly elevate that session on-the-fly to a more privileged or information-rich state to facilitate their needs when appropriate. We have implemented our framework in the Chrome browser with current support for the popular single sign-on mechanism Facebook Connect. Our results suggest that our framework is able to intercept attempts for third-party Web site authentication and handle them in a way to protect the user's privacy, while not affecting other ongoing Web sessions that the user may concurrently have.

Availability

The source code of *SudoWeb* is available at https://code.google.com/p/sudoweb/.

4.3 Privacy-preserving Microblogging Browsing through Obfuscation

For more information, there exists an extended version of this work [92].

4.3.1 System Design

4.3.1.1 Threat Model

We assume the existence of a microblogging service where users are able to follow individual channels. A channel can be the account of a physical person, of an entity such as a corporation, of a news site, of a politician's office, and so on. Additionally, we assume that the microblogging service is capable of recording the users' interests by observing which channels each user follows. The information about the users' interests, which is property of the microblogging service, could be later sold to advertisers [100], and could be used for a variety of purposes, all of which are beyond the control of individual users [79]. We view this capability of the microblogging service as a potential concern for the users' privacy, and we would like to develop mechanisms that hide the users' real interests from the microblogging service.

In this work we assume an "honest but curious" microblogging service. In this aspect, the microblogging service may try to find the user's interest based on the channels the user is following, but it will not try to "cheat" by actively interfering with the process users are employing to protect their

Notation	Explanation
S	: Set of sensitive channels that can be followed
C	: Sensitive channel
U	: Number of all users in the system
U_C	: Number of users actually interested in channel C
U_{R_C}	: Number of users following channel C at random
p_C	: Popularity of channel C ($p_C = U_C/U$)
P_C	: Probability that a user following C is interested in C
N	: Number of sensitive channels a user is interested in
k	: Obfuscation level (per channel)

Table 4.1: Summary of Notation

privacy, or try to gain more information than what a user is willing, or required, to give. For example, the microblogging service will not create fake channels or fake users in order to break the anonymity of ordinary users. We think that this "honest but curious" model is reasonable in practice, as popular microblogging services have a reputation they do not want to jeopardize by becoming hostile against their own users. Therefore, we expect such microblogging services to only try to passively gain knowledge based on data given by their users.

4.3.1.2 Our Approach: k-subscription

Table 4.1 summarizes the notation we use throughout this section. Assume that user A is interested in following channel C, which deals with a sensitive issue, such as a medical condition. If user A follows only this channel, the microblogging service would easily figure out that A is interested in this medical issue. In this work we propose *k-subscription*: a system that makes sure that the microblogging service is not able to pinpoint A's interests with reasonable accuracy. To do so, *k-subscription* follows an obfuscation-based approach, which advocates that along with each channel C the user is interested to follow, she should also follow $k-1$ other channels (called "noise" channels). The number of noise channels is such that the microblogging service will not be able to determine A's interest with high probability, and will not be able to identify the actual set of users interested in each specific channel. All the noise channels are randomly chosen from a set S of "sensitive" channels. Note that A's real interests are also members of S.

4.3.1.3 Uniform Sampling

When a user wants to follow channel C, *k-subscription* encourages the user to follow $k-1$ other channels as well (say C_1, C_2, ..., C_{k-1}). In this way, the microblogging service will not know whether the user is actually interested in channel C or one of the C_1, C_2, ..., C_{k-1}. In our first algorithm, *k-*

subscription-UNIF, these channels are chosen randomly with uniform probability from S. Algorithm 1 presents the pseudocode for *k-subscription*-UNIF.

Algorithm 1 *k-subscription*-UNIF: Choose noise channels uniformly from the set S

$F = \emptyset$; {initialize the set of channels to follow}
for ($i = 1$; $i \leq k - 1$; $i++$) **do**
  C_i = randomly select a channel from set S ;
  $S = S \setminus C_i$; {remove C_i from S}
  $F = F \cup C_i$; {add C_i in the set of channels to follow}
end for
$F = F \cup C$; {add C in the set of channels to follow}
Follow all Channels in F in a random order ;

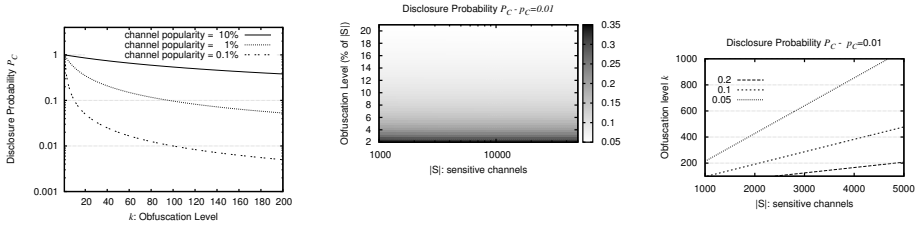
k-subscription-UNIF is a naive but powerful approach for obfuscation and we use it as a basic principle for our method. However, this approach leads to some practical problems. In case that not all channels enjoy the same popularity, then uniformly sampling from S may result in higher disclosure probability for the more popular channels. Thus, we discuss an improved version in the next section.

4.3.1.4 Proportional Sampling

A user following a popular channel (say C) along with several unpopular ones has a higher probability of being interested in C than in the rest of them. Capitalizing on this knowledge, the microblogging service has a better chance of finding those users who follow popular channels. To mitigate this issue, we propose *k-subscription*-PROP that sample channels from set S according to their popularity. Assume that U_C is the number of followers of channel C and $U_S = \sum_{\forall C \in S} U_C$ is the number of followers of all channels in S. Thus, instead of sampling all channels with probability $1/|S|$, we sample channel C with probability U_C/U_S. On Twitter, the popularity of a channel can be inferred by the number of users following the respective account. In other microblogging services similar metrics are available to determine the popularity of a channel. *k-subscription* with proportional sampling for adding noise does not affect the respective channel popularity.

4.3.1.5 Following Multiple Channels

Users may be interested in following more than one sensitive channels. Using *k-subscription*, users just need to select $k - 1$ other noise channels to follow for *each* channel C they are interested in. Therefore, a user interested in following N channels will result in following $k \times N$ channels in total. However, it is very likely that a user will be interested in N sensitive



(a) P_C as a function of k for different channel popularities.

(b) [P_C as a function of $|S|$ and k, shown as a percentage of $|S|$, for channel popularity 1%.

(c) [P_C as a function of $|S|$ and k, for channel popularity 1%.

Figure 4.8: Disclosure Probability P_C of *k-subscription*-UNIF as a function of the obfuscation level k and the size of S.

channels that are *semantically related*. This case may significantly increase the disclosure probability. Indeed, the microblogging service can easily find the correlated channels: it will get all the channels a user is following, classify them into semantic categories, and identify the sets of channels that are semantically related. If there is only one set of related channels, it is more probable that the user actually follows them, and the remaining unrelated channels are the selected noise.

One way to address this issue could be the following: whenever a user is interested in N related channels, the $(k-1) \times N$ noise channels could be selected in N-tuple groups, so that each N-tuple consists of N related noise channels. However, this approach has a certain limitation: the microblogging service and *k-subscription* may use different similarity metrics to identify related channels. For instance, the microblogging service may use a more fine-grained similarity metric to find out the actual related channels.

Fortunately, *k-subscription* is able to protect users' interests even when they are interested in multiple semantically related channels. Although a user will actually follow the set of N related sensitive channels she is interested in, which can be identified by the microblogging service, there will be a significant number of other users that also follow the same set of N related channels due to random noise channel selections, i.e., without being interested in them. This is due to the increased random selections when users are interested in multiple channels. Thus, the microblogging service will not be able to know which of the users following all these N related channels are actually interested in them.

4.3.2 Analytical Evaluation

4.3.2.1 Analysis of k-subscription-UNIF

The disclosure probability P_C is the probability (as it can be calculated by the microblogging service) that a user who follows channel C is really interested in C. nce along with channel C a user follows $k-1$ other channels as well, the disclosure probability is $P_C = 1/k$. However, for large values of k (i.e., in cases where the user wants to add a lot of noise) the microblogging service has a more effective way to increase its certainty about the interest of a user in a particular channel C. It knows that the U_C users who are interested in channel C actually follow it. At the same time, however, there are $U - U_C$ other users that are *not* interested in C, who may have randomly included C among their noise channels. The probability of C being included in the set of channels followed randomly by a user interested in a channel *different* than C is bounded by $1-(1-1/|S|)^{k-1}$, as this user will select $k-1$ channels randomly from S, which also includes C. Therefore, the average number of the $U - U_C$ users not interested in C that will follow C randomly as noise (U_{R_C}) are less than $(U - U_C) \times (1 - (1 - 1/|S|)^{k-1})$. So, the ratio of users following C who are really interested in C is less than

$$\frac{U_C}{U_C + (U - U_C) \times (1 - (1 - 1/|S|)^{k-1})}$$

Since the microblogging service does not know who of the users in U_C are interested in the channel C, it can only assume that all users following C are interested in C. The probability of a user following C actually being interested in C (denoted as P_C) is given by:

$$\begin{aligned} P_C &< \max(1/k, \frac{U_C}{U_C + (U - U_C) \times (1 - (1 - 1/|S|)^{k-1})}) \Rightarrow \\ P_C &< \max(1/k, \frac{p_C}{p_C + (1 - p_C) \times (1 - (1 - 1/|S|)^{k-1})}) \end{aligned} \tag{4.1}$$

where p_C is the channel's popularity. We see that the total number of users U and the number of users U_C interested in channel C do not affect the disclosure probability. Instead, the channel's popularity p_C, the parameter k, and the total number of channels $|S|$, are the key factors that affect the disclosure probability.

Estimating the Disclosure Probability.

First we arbitrarily fix $|S|$ to 1,000 channels. Figure 4.8(a) shows how the disclosure probability P_C changes as a function of k (the level of obfuscation). We see that when the popularity p_C of a channel is rather high (i.e., 10%), then it is difficult to obfuscate it with the *k-subscription*-UNIF approach. Indeed, when as many as 10% of the users are interested in channel C, then it would take a significant percentage of the rest 90% to

include channel C among their noise channels, which is very difficult to achieve. However, when popularity is around 1%, then it is much easier to obfuscate it using this approach. Indeed, for $k = 100$ the disclosure probability is as low as 0.1, which means that the microblogging service can not state with confidence larger than 10% that a user A who follows channel C is really interested in C. Fortunately, when the popularity of C is even lower (i.e., around 0.1%), the disclosure probability becomes 0.01 for obfuscation levels as low as 100. That is, the microblogging service can not say with confidence higher than 0.01 that a user who follows C is really interested in C.

In our next figure we explore how the disclosure probability changes as a function of the number of sensitive channels $|S|$ and the obfuscation level k. That is, if we double $|S|$ how should we increase k so as to have the same disclosure probability? Figure 4.8(b) shows a plot of the P_C as a function of $|S|$ and k. Note that the obfuscation level k in the y-axis is shown *not* as an absolute number but as a percentage of the number of sensitive channels. From this figure we clearly see that lines of the same color run horizontally. Horizontal lines mean that the value is the same for constant y (obfuscation level as a percentage of $|S|$). This implies that as long as the obfuscation level is a constant percentage of the size of the set of sensitive channels $|S|$, the disclosure probability remains constant. To put it simply, if we double the number of sensitive channels, we need to double the obfuscation level k (in absolute numbers) in order to keep the disclosure probability constant.

To explore the relation between $|S|$ and k even further, Figure 4.8(c) shows the iso-probability contours of the disclosure probability P_C as a function of the number of sensitive channels (i.e., $|S|$ on x axis) and the obfuscation level (i.e., k on the y axis). We plot the contours for probabilities 0.05, 0.1, and 0.2. Interestingly, we see that the iso-probability contours appear as straight lines, clearly implying an almost linear relation between $|S|$ and k. That is, doubling $|S|$ would require a twice as high k in order to keep the probability of disclosure to the same level. Similarly, if we can afford to double the obfuscation level, we can provide the same disclosure probability for twice as many sensitive channels. Or, equivalently, if we are forced to half the obfuscation level, we can still provide the same disclosure probability but only for half as many sensitive channels. These results are very encouraging in the sense that we can still achieve the levels of P_C we are comfortable with, even when we are forced to use small obfuscation levels.

Figure 4.9 shows the impact that the number of sensitive channels $|S|$ and the channel popularity p_C have on the disclosure probability. The obfuscation level k is set to 100. We see that the iso-probability contours are almost straight anti-diagonal lines indicating an almost inversely proportional relation between $|S|$ and p_C.

Finding a Reasonable Size for S. We now analyze how large S should be and how we can influence it. One can easily see that the larger S is, the

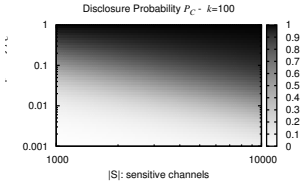


Figure 4.9: Disclosure Probability P_C of *k-subscription*-UNIF as a function of the size of S and channel popularity p_C.

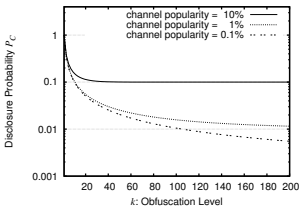


Figure 4.10: Disclosure Probability P_C of *k-subscription*-PROP as a function of the obfuscation level k.

(a) $|S|$=1,000, channel popularity=0.1%. (b) $|S|$=1,000, channel popularity=1%]. (c) $|S|$=2,000, channel popularity=1%.

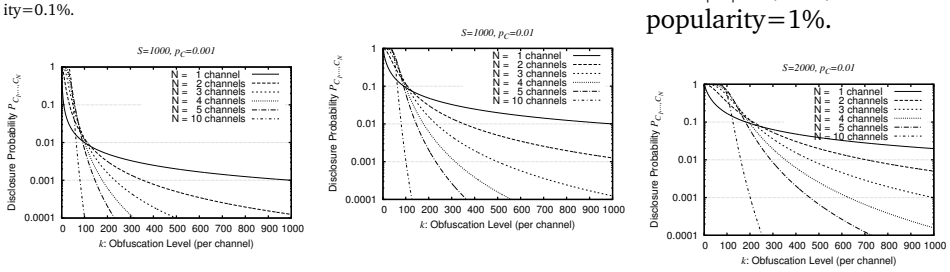


Figure 4.11: Disclosure Probability as a function of the obfuscation level k.

higher the disclosure probability will be (for a constant obfuscation level k). Therefore, we do not want the set S to be very large. On the other hand, a very small S would easily give away a user's true interests, and limit the users' choice for sensitive channels. Indeed, if S contains two channels, say C_1 and C_2, if a user follows C_1 (no matter whether she follows C_2 or not), the microblogging service will be able to conclude with probability 1/2 that the user is interested in C_1. In the same spirit, if S contains n members, the microblogging service will be able to conclude with probability at least $1/n$ that the user is interested in the channel she follows. As a result, S should be large enough so that the probability $1/n$ is small enough, so as not to be useful for the microblogging service. In the absence of any other information, the microblogging service is able to conclude with probability U_C/U that a random user is interested in channel C. Therefore, the set S should be large enough so that $1/|S| < U_C/U$. We can estimate U_C from external sources, or based on the number of followers of channel C.

4.3.2.2 Analysis of k-subscription-PROP

Let us now derive closed formulas for *k-subscription*-PROP. The probability of C being included in the set of channels followed randomly by a user interested in a *different* channel than C is bounded by $1-(1-U_C/U)^{k-1}$, as

this user will choose at random $k-1$ channels from S, and the probability of choosing C at each individual choice is U_C/U. The average number of users not interested in C who will follow C randomly as noise (U_{R_C}) are less than $(U - U_C) \times (1 - (1 - (U_C/U))^{k-1})$. So, the ratio of users who follow C and are interested in C is less than

$$\frac{U_C}{U_C + (U - U_C) \times (1 - (1 - U_C/U)^{k-1})}$$

Since the microblogging service does not know who are the U_C users interested in C, it can assume that all users following C are interested in C. The probability of a user following C being interested in C (i.e., the disclosure probability) is bounded as follows:

$$\begin{aligned} P_C < \max(1/k, \frac{U_C}{U_C + (U - U_C) \times (1 - (1 - U_C/U)^{k-1})}) \Rightarrow \\ P_C > \max(1/k, \frac{p_C}{p_C + (1 - p_C) \times (1 - (1 - p_C)^{k-1})}) \end{aligned} \tag{4.2}$$

where p_C is the channel's popularity. We observe that instead of the total number of users U and the number of users U_C that follow the channel C, the disclosure probability is affected only by channel's popularity p_C, number of channels S, and obfuscation level k.

Figure 4.10 shows how disclosure probability changes with the level of obfuscation k. We see that our *k-subscription*-PROP approach is able to efficiently hide popular channels. Indeed, for a popularity of about 10%, it is able to reach a disclosure probability of 0.1 using only $k = 40$. When the popularity is 1%, even small obfuscation levels such as $k = 50$ can lead to disclosure probability as low as 0.02, which is very encouraging. One can notice in Figure 4.10 that as k increases for 10% popularity, the disclosure probability tends to flatten out and in no case drops below 0.1. The reason is that for a channel with 10% popularity, the disclosure probability can never fall below 10% no matter how large the obfuscation level we use is. There is a simple explanation for this: without taking any channel-following information into account, the microblogging service knows that 10% of the population is interested in channel C. Hence, the microblogging service can safely assume that a user is interested in C with probability 0.1.

4.3.2.3 Analysis for Multiple Channels

A user may want to follow more than one sensitive channels, which may be semantically related to each other. For simplicity, we assume that all users U are interested in exactly N sensitive channels from S. Each user that is actually interested to follow the N channels $C_1, ..., C_N$ will also follow $(k - 1) \times N$ noise channels based on random choices from S. Besides the $U_{C_1,...,C_N}$ users that are interested in these N channels, there will be some

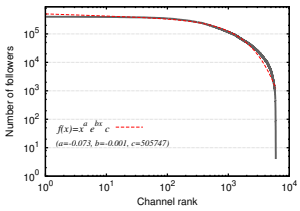


Figure 4.12: Distribution of the sensitive channels popularity.

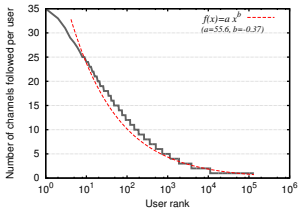


Figure 4.13: Distribution of the number of sensitive channels followed by a user.

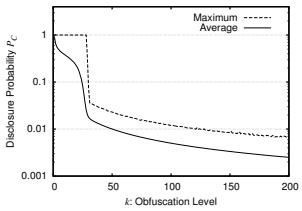


Figure 4.14: Disclosure probability as a function of k using realistic simulations.

other users that will follow the same set of N channels without being interested in all of them, due to random noise channel selections. These users contribute to hide the actual interests of the $U_{C_1,...,C_N}$ users.

Since the microblogging service does not know the users who are actually interested in channels $C_1, ..., C_N$, it can only assume that all users following these channels are interested in them. The disclosure probability $P_{C_1,...,C_N}$ that a user following all these N channels is actually interested in them is equal to:

$$\frac{U_{C_1,...,C_N}}{U_{C_1,...,C_N} + (U - U_{C_1,...,C_N}) \times \binom{|S|-N}{(k-1)N-N} / \binom{|S|}{(k-1)N}} = \frac{p_{C_1,...,C_N}}{p_{C_1,...,C_N} + (1 - p_{C_1,...,C_N}) \times \binom{|S|-N}{(k-1)N-N} / \binom{|S|}{(k-1)N}} \tag{4.3}$$

where $\binom{|S|-N}{(k-1)N-N} / \binom{|S|}{(k-1)N}$ is the probability that a user selects randomly these N channels from the set S with $(k-1) \times N$ random choices when using the *k-subscription*-UNIF approach. We estimate this probability with a hypergeometric distribution, where all the successes N in the population S should be drawn with $(k-1) \times N$ attempts. Since we assume that all channels have the same popularity, the *k-subscription*-PROP approach has exactly the same behavior with *k-subscription*-UNIF in this analysis. $p_{C_1,...,C_N}$ is the popularity of the N-tuple of sensitive channels, i.e., the percentage of users actually interested in all these $C_1, ..., C_N$ channels.

We want to explore how the disclosure probability changes with the number of channels N that a user may be interested in. We assume that the users interested in N channels are $U_{C_1,...,C_N} = U_C/N$, i.e., they are reduced by N times. Note that we assume a hyperbolic decrease of the users as N increases, instead of an exponential decrease, because we believe that these N channels will be probably semantically related. We set the size of S to $1,000$ and $2,000$ sensitive channels and we assume that all channels have the same popularity p_C, which is set to 0.1% and 1%.

Figure 4.11 shows the disclosure probability as a function of k for different values of N, ranging from 1 up to 10. We see that as the number of channels N increases, the disclosure probability is increased for low values of k, but it decreases significantly for higher k. The increase for low k values is because the users interested in N channels are reduced just by N times, following a hyperbolic growth, while the probability of randomly selecting these N channels for the rest users is reduced significantly by following a hypergeometric distribution. Thus, it is unlikely for the rest users to follow all these N channels at random with low k values. This means that for users interested in many sensitive channels we need to use a higher k to achieve a low disclosure probability.

In contrast, for higher values of k, we see a significant reduction of the disclosure probability when users are interested in more channels. This is because the users interested in N sensitive channels follow $k \times N$ channels in total, so we have more random selections for higher N values. For instance, when $N = 5$ and $k = 200$, each user follows $1,000$ channels from S, i.e., all the existing channels in S when $|S| = 1,000$. The same happens in case of $N = 10$ and $k = 100$. When the size of S and channel popularity p_C increase, the disclosure probability increases respectively, according to Equation 4.3. However, a proper selection of a higher k value results in a much lower disclosure probability, as the users' interests can be efficiently hidden among the random selections of other users. Our experimental evaluation in Section 4.3.5 shows that the network bandwidth and latency when following few hundreds of noise channels are negligible, so our approach is able to protect the users' privacy even when they are interested in many sensitive channels that are probably semantically related.

4.3.3 Simulation-based Evaluation

To evaluate *k-subscription* in a more realistic setup, where users are interested in a different number of sensitive channels, and sensitive channels have different popularities, we built a realistic Monte Carlo simulator. The simulator assigns a random popularity p_C to each channel following a similar distribution to real-world sensitive channels. First, each user randomly selects the number of channels N that she is interested in following, based on a distribution similar to real-world users' selections. We assume that all N channels are semantically correlated. Then, the user selects these channels one-by-one at random, proportionally to channel's popularity. The noise selection is performed with *k-subscription*-PROP.

To simulate a realistic popularity distribution of sensitive channels, we selected a set S of $7,000$ sensitive channels using Twellow [40], a website that categorizes Twitter accounts according to their subject. The selected channels correspond to Twitter accounts dealing with health, political, religious, and other sensitive issues. We estimate the popularity of each channel

based on its number of followers, i.e., U_C. Figure 4.12 shows the distribution of sensitive channel popularity in our dataset. We see that this distribution can be approximated very well using a power law with exponential cutoff model. We use this approximation in the simulator to assign a popularity p_C in each channel. We also see that only a small percentage of the sensitive channels exhibit relatively high popularity, which increases the disclosure probability. In contrast, the majority of sensitive channels have low popularity, which results in low disclosure probability even for low values of k.

To simulate a realistic distribution of the number of sensitive channels N that each user is interested in, we used the same real-word dataset of sensitive channels. From the total $7,000$ channels, we used the Twitter API to collect the user IDs of the followers of 500 sensitive channels related to disability issues, and we measured the number of occurrences of each user ID. This is the number of channels belonging in S that each user in our dataset follows. In this analysis we found more than $530,000$ unique users. Figure 4.13 shows the respective distribution, which is approximated with a typical power law function. This approximation is used in our simulations for realistic user selections. Also, we observe that only 0.85% of the users follow more than 4 sensitive channels, while 91.65% of the users follow just one sensitive channel.

The simulator keeps two counters per each channel: the number of users that (i) select this channel as actual interest (U_C), and (ii) select this channel as noise (U_{R_C}). Before exiting, the simulator reports the disclosure probability per channel, which is $max(1/k, U_C/(U_C + U_{R_C})$. Additionally, it keeps two lists per user: (i) the channels she is interested in (C_i), and (ii) the channels she selects as noise (C_n). This way, the simulator reports the disclosure probability per user, based on the set of sensitive channels the user is interested in (C_i). This probability is equal to $max(1/k, U_{C_i}/(U_{C_i} + U_{R_{C_i}}))$, where U_{C_i} the number of users interested in C_i and $U_{R_{C_i}}$ the number of users that C_i is included within their C_n. This is because we assume that all channels in C_i are semantically correlated. Among all the disclosure probabilities reported per each user and each channel, the simulator reports the overall average and maximum disclosure probability. We repeat each simulation for 100 times and we use the average values.

We set $|S|$ to 1,000 channels, U to 1,000,000 users, and we vary k from 1 to 200. Figure 4.14 shows the average and maximum disclosure probability reported by the simulator as a function of k. We see that *k-subscription* achieves a low average disclosure probability over all channels and users, which decreases rapidly with k. However, we see that the maximum disclosure probability found for an individual user remains equal to 1 for low values of k up to $k = 30$. This is because there is at least one user interested in an N-tuple that no other user has selected among her random noise choices, especially for large values of N. However, as k increases, we see

that an increasing number of users tend to select a significant percentage of the channels in S as random choices, e.g., users with large value of N. As these users follow most of the channels in S, they tend to hide the actual users' interests, even for large and rare N-tuples, reducing effectively the maximum disclosure probability.

4.3.4 Implementation

To evaluate the feasibility and efficiency of *k-subscription* we have implemented a Twitter extension for the popular Chrome web browser. The extension uses Twitter API v1.1 and complies with the REST API Rate Limit. It is developed using Javascript and JQuery, Json2, OAuth and SHA-1 libraries.

Figure 4.15 shows the overall operation of *k-subscription* extension. Upon installation, users can follow Twitter accounts in exactly the same way, though Twitter's web interface or "Follow me" buttons in third-party web pages. To enhance user's privacy, *k-subscription* intercepts all follow requests and checks whether they correspond to sensitive channels contained in S. If so, the extension transparently subscribes the user to $k-1$ additional "noise" channels from S according to the *k-subscription*-PROP algorithm, where k can be configured by the user. These channels remain hidden and the user never interacts with them, providing exactly the same Twitter browsing experience as before. For this reason, the extension keeps a list of all "noise" channels and dynamically filters out the unsolicited tweets of these channels from user's feed. Other affected information, such as the number of channels followed, is adjusted appropriately by excluding the effect of the "noise" channels.

At the first run, the extension downloads the set S of sensitive channels used for obscuring user's selections. The set includes information about each channel and its number of followers to improve "noise" selection. The user can interfere with "noise" selection by proposing channels with predefined features such as language and country. Users can disable the effect of *k-subscription* on a follow request if they consider the related channel as non-sensitive. When a user unfollows a sensitive channel, the extension transparently removes its corresponding "noise" channels as well.

We envision that the set of sensitive channels S along with the project in general would be maintained by the broader community of users and/or Non-Governmental Organizations (NGO) that have a specific view towards protecting privacy. Hence, *S* can be seeded by an initial set of sensitive channels and further improved through human intervention and participation of the community. Similar privacy-concerned projects, such as Tor, enlist the help of volunteers to maintain and improve its networks of routers. We expect that similar approaches can be applied for *k-subscription*.

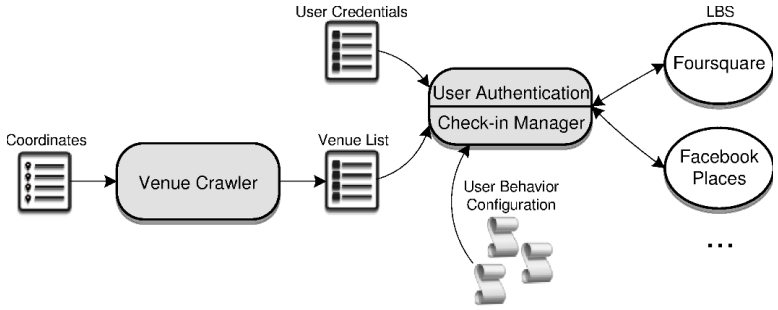


Figure 4.15: Overall operation of the *k-subscription* browser extension for Twitter.

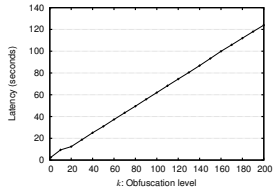


Figure 4.16: Time to follow a sensitive channel as a function of k.

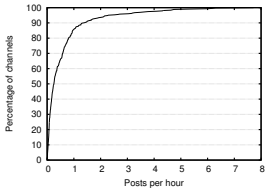


Figure 4.17: Number of tweets posted per channel per hour.

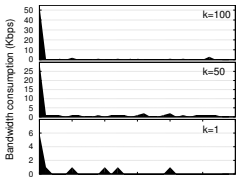


Figure 4.18: Bandwidth consumed for a user receiving tweets as a function of time.

4.3.5 Experimental Evaluation

4.3.5.1 How Much Does the Noise Cost?

In our next experiment we tried to quantify how much more traffic is generated by the noise channels. To do so, we measured the total number of tweets generated by all channels, divided by each channel's lifetime and found the average number of tweets per channel per unit of time. The CDF of this function is shown in Figure 4.17. We see that the median channel (y=50%) generates less than one tweet (actually 0.25 tweets) per hour while 93% of the channels generate less than two tweets per hour. Overall, we see that the extra traffic generated by the noise channels should be very small. Even adding 100 noise channels generates no more than 25 tweets per hour, a negligible amount of traffic by most standards.

The reader will notice that the maximum posting rate that we have observed is about 6 posts per hour (averaged over the entire lifetime of the channel). Published statistics [109] suggest that the most prolific twitter accounts post as much as one tweet update per minute. Such accounts usually belong to news stations or even to automated programs (bots). Given that each tweet corresponds to just few hundred of bytes transferred over the network, even in such cases the resulting network overhead will be relatively low.

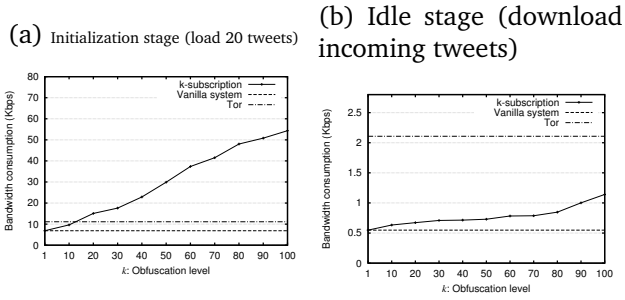


Figure 4.19: Bandwidth consumption with *k-subscription*, Tor and vanilla system.

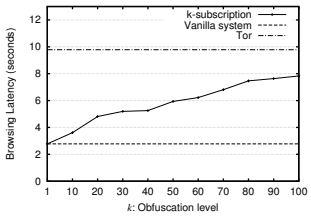


Figure 4.20: Browsing latency as a function of k when a user opens Twitter's main page.

4.3.5.2 Bandwidth Consumption

When a user follows k channels for each subscription, she downloads roughly k times more information than she actually needs. However, we would like to see if the bandwidth needed for these downloads can be sustained by a home DSL Internet connection or not, and the respective network overhead in terms of used bandwidth. Figure 4.18 shows the traffic load generated by our implementation over a 30-minute period for a user following one sensitive channel with $k = 100$, $k = 50$, and $k = 1$ (i.e, without using *k-subscription*). We notice that the bandwidth consumption even in case of $k = 100$ is reasonably low, usually less than 1.5 Kbps. We see that even in case of the vanilla system (see $k = 1$) the bandwidth consumption is not significantly lower than in high values of k. In all cases it is usually between 0.5 and 1.0 Kbps. By manually inspecting the traffic we found that most of the bandwidth is used to download information like images, trends and recommendations, which does not depend on the value of k. The bandwidth used for downloading the actual tweets, which increase with the value of k, was found to be a small percentage of the total bandwidth consumption.

We observe a large spike at the beginning of each experiment, when we have just opened the browser and loaded the Twitter page. For instance, bandwidth consumption reaches 54 Kbps for $k = 100$, 29 Kbps for $k = 50$, and 7 Kbps for the vanilla system at the first second. During this initialization stage Twitter downloads all the necessary content (widgets, scripts, CSS, profile images, *etc.*). At this stage, *k-subscription* downloads lot of tweets from all k channels. r this reason we observe a relatively higher spike as k increases. To improve browsing latency, we transparently increase the page size to receive more tweets.

Note that profile images are cacheable, so *k-subscription* downloads the additional images (depending on k) only once, without affecting the overall bandwidth consumption. After the initial spike in the first few seconds, we see constantly low bandwidth consumption, which correspond to the low rate of incoming tweets. The average consumption in this 30-minute

interval is 1.14 Kbps for $k = 100$, 0.71 Kbps for $k = 50$, and 0.54 Kbps for $k = 1$. Overall, we see that the total bandwidth consumed by *k-subscription* is not really an issue even if the user follows as many as 100 channels.

To evaluate the effect of the obfuscation level on bandwidth consumption while browsing Twitter with *k-subscription*, we plot in Figure 4.19 the bandwidth consumption as a function of k for two different stages: (i) when the user loads Twitter and downloads her timeline, which consists of the latest 20 tweets from the channels she is interested in, and (ii) when Twitter is idle and just receives new incoming tweets for 30 minutes. We see that the overhead is very low, even for large k, and can be easily handled by a home DSL or even a mobile connection. The bandwidth consumption is much lower in the idle stage, as expected, due to the low number of tweets per second, as shown in Figure 4.17. The increased bandwidth during initialization is because *k-subscription* asks for more tweets to display the default page of 20 tweets only from channels that user is interested in. However, the initialization lasts for just few seconds, e.g., 7.7 seconds for $k = 100$ and 2.8 seconds for the vanilla system. Thus, the increased bandwidth in Figure 4.19a, corresponds to short term spikes, while the low bandwidth in idle stage (see Figure 4.19b) corresponds to much longer periods, as the user keeps Twitter's page open in the browser.

In Figure 4.19, we also compare the bandwidth consumption of *k-subscription* with a Tor browser. Although Tor offers a completely different type of anonymity than *k-subscription*, it could be used with a fake account as a different approach to hide user's interests. Thus, we evaluate *k-subscription* using the performance of Tor with a fake Twitter account as a baseline case. We see that Tor adds an additional bandwidth overhead due to its data encapsulation. In particular, the average packet size of Twitter traffic over Tor is 789 bytes, when the vanilla Twitter traffic has an average packet size of 239 bytes. This is the main reason that during the idle stage the bandwidth consumption of Tor is quite higher than the consumption of *k-subscription*, e.g., two times higher when $k = 90$. During the initialization stage, Tor has a higher bandwidth consumption than *k-subscription* with values of k up to 10, and lower consumption when k exceeds 10. This is due to the increased number of tweets downloaded at startup by *k-subscription* with high k values to construct a full page of useful tweets. However, as the initialization stage lasts only for few seconds, compared with idle stage, *k-subscription* adds less overhead in terms of bandwidth.

When *k-subscription* compounds the user's timeline, it continues to download tweets in the background until it reaches a certain number, which is constant for each k value. This way, *k-subscription* avoids leaking any information that Twitter could analyze to find out the channels a user is interested in.

4.3.5.3 Browsing Latency

In our next experiment we set out to explore the latency that *k-subscription* imposes to user's browsing experience. We instrumented our browser extension to measure the latency from the time that a user asks for one or more tweets till the time that the browser actually displays the relative information in the page, excluding any tweets from noise channels. This latency includes the time spent in network for downloading tweets, as well as the time spent in the CPU for excluding the noise and rendering the page. Note that the user's timeline is fully rendered when all the 20 tweets needed are received, despite the fact that more tweets are downloaded in the background to hide the actual user's interests.

Figure 4.20 shows the latency for displaying a page with *k-subscription* for several values of k when the user opens Twitter and loads her timeline. We see that the latency for downloading and displaying a full page with 20 tweets slightly increases with the number of noise channels, reaching to 7.7 seconds for $k = 100$ when without *k-subscription* (see $k = 1$) the latency to display the same page is 2.8 seconds. Therefore, a slight delay of less than 5 seconds is not expected to significantly affect the user's browsing experience, while, at the same time, it enhances her privacy. Selecting a smaller number of noise channels results in even lower latency. Note that this small delay is only observed at the initialization stage, due to the increased number of tweets needed to construct the user's actual timeline. When the browser remains open (idle stage) we do not observe any noticeable delay to render the incoming tweets, even at very high values of k. If an incoming tweet belongs to a noise channel we just drop it, else it is immediately given to user with no further delay. Thus, our approach does not impose any significant overhead to the browsing latency.

In Figure 4.20 we also compare the browsing latency of *k-subscription* and Tor. We see that Tor requires a much higher latency to display Twitter's page, close to 10 seconds. This is due to the longer path from user to Twitter through the anonymization network.

During the previous experiments we measured the CPU load of the browser, using the Linux's `time` utility. The CPU load was negligible for all values of k, always less than 1%, even for $k = 100$. Thus, our *k-subscription* browser extension does not add any considerable CPU overhead to the system.

4.3.6 Lessons Learned

Although microblogging services enable users to have timely access to their information needs through a publish-subscribe model, this creates major privacy concerns. As users declare all channels they are interested in following, the microblogging service is able to gather all their interests, including

possible privacy-sensitive domains. To remedy this situation, we propose k-subscription: an obfuscation-based approach that encourages the users to follow k1 additional “nois” channels for each channel they are really interested in following. We present a detailed analysis of our approach and show that by fine-tuning the k parameter we are able to reduce the confidence that the microblogging service has in knowing which channels each user is really interesting in. We have developed a prototype implementation as an extension for the Chrome browser using Twitter as case study. Our experimental evaluation shows that users may easily follow hundreds of noise channels with minimal run-time overhead when they receive news they are interested in. We believe that as an ever-increasing number of users turn to microblogging services for their daily information needs, privacy concerns will continue to escalate, and solutions such as k-subscription will become increasingly more important.

5 Attacks on Social Networks

5.1 Introduction

This chapter provides our research work related with cybersecurity attacks on social networks and their users' privacy. The different topics covered in this chapter contain attacks to social networks ranging from how spammers can exploit social networks to serve their malicious intentions (Section 5.2), or other forms of cyber-attack like attacks against authentication mechanisms (Section 5.3) to social network profile cloning attacks (Section 5.4). In Section 5.2, In this work, we demonstrate that social networks are an enormous and ever expanding pool of information that can be used as a stepping stone for personalized phishing campaigns. We demonstrate that even by retrieving the most basic information, i.e., the name of the user, we are able to harvest millions of email addresses. We present two different approaches to harvesting; *blind harvesting* that aims to gather as many email addresses as possible, querying for names retrieved from OSNs in the Google search engine, and *targeted harvesting* that aims to gather email addresses and correlate them to personal information publicly available on social networking sites. In Section 5.3, we systematize and expand previous work, which pointed out (i) the feasibility of recognizing people's faces using Facebook photos, and (ii) the theoretical issues with face-based SA. This systematization allows us to describe an attack that breaks Facebook's SA mechanism, while retaining the assumptions of their threat model. We present our black-box security analysis of Facebook's SA mechanism and point out its weaknesses (and implementation flaws) when employed as the second factor of a two-factor authentication scheme. We design and implement an automated, modular system that leverages face detection and recognition to break Facebook's SA efficiently. We, thus, show the feasibility of such an attack in large-scale. Moreover, we show that publicly-available face recognition services offer a very accessible and precise alternative to

building a custom face recognition system. We discuss how Facebook's SA scheme should be modified so that users can trust it as a second authentication factor. Finally, in Section 5.4, In this work, we propose a tool that automatically seeks and identifies cloned profiles in social networks. The key concept behind its logic is that it employs user-specific (or user-identifying) information, collected from the user's original social network profile to locate similar profiles across social networks. Any returned results, depending on how rare the common profile information is considered to be, are deemed suspicious and further inspection is performed. Finally, the user is presented with a list of possible profile clones and a score indicating their degree of similarity with his own profile. The contributions of this work are the following. We design and implement a tool that can detect cloned profiles in social domains, and conduct a case study in LinkedIn. We present a study of the information publicly available on a large number of social network profiles we collected.

5.2 Using Social Networks to Harvest Email Addresses

This work is described in detail in its extended version [95].

5.2.1 Harvesting email addresses

In this section we give a brief overview of the current methodologies used by spammers to harvest email addresses.

Web crawling. Email addresses of users are posted in various places on the Web. Personal web pages, blogs and forums are such examples. By crawling the web attackers can gather thousands of email addresses. However, this methodology suffers from low scalability as web crawling is a very time-consuming and bandwidth-demanding process.

Crawling archive sites. Attackers can narrow down their crawling to sites they know contain thousands of email addresses. For example, the Mailing List Archives site [33] hosts archives for thousands of computer-related mailing lists. The obfuscation used to prevent crawlers from extracting addresses is very simple to bypass, as addresses are written in the form "username () domain ! top-level-domain".

Malware. Attackers can instrument their malware code to collect addresses from the email clients of infected users or their instant messaging clients. Given the widespread use of email clients and popularity of instant messaging networks, this technique provides good scalability.

Malicious sites. Attackers can lure users to sites and request for their email addresses in exchange for providing porn content and warez sites can offer access to movies and software provided that the user "registers" with their service.

Dictionary attacks. One can form email addresses by taking words from a dictionary. For example, the spammer can concatenate "john" with the domain "hotmail.com" to form the email address "john@hotmail.com". Dictionary attacks can be classified into one of two types: blind attacks and search-based ones. Blind attacks try to guess email addresses by random concatenation of dictionary words and popular email domains. In this case, the attacker would send spam to "john@hotmail.com" without any knowledge of the validity of the email address. This approach is not efficient and is limited to the dictionary size. Search-based attacks make use of Web search engines to validate the addresses acquired by the dictionary concatenation. The attacker now searches for "john@hotmail.com" and parses the results for email addresses. This approach is more efficient as it can return more addresses than expected. As an example, searching for "john@hotmail.com" can also lead to "other.john@hotmail.com" and "john@hotmail.de".

In this work, we describe a new approach on how attackers can use information from social networks to perform more advanced search-based dictionary attacks. Instead of using words from a dictionary, an attacker can crawl popular social networks and use the collected user names or pseudonames as search keywords. This approach has two major advantages. First, it scales with the growth rate of social networks. While dictionaries are limited to few hundred thousand terms, the number of user names and pseudonames that can be found in social networks is in the order of hundreds of millions. Second, information from social networks can be used for personalizing spam campaigns. For example, attackers can use the full names of users in order to construct more convincing spam emails. We describe our approach in more detail in Section 5.2.2.

5.2.2 Using Social Networks to harvest email addresses

Social networks provide a plethora of personal information. Users upload reports from their daily activities, political and religious status, events they have or will attend, photos, comments for other users and many more. Once a user has managed to become a friend with someone, he can extract various pieces of information that can be used for illegal purposes.

Even though social networking sites cannot protect users from other malicious users that want to harvest personal information through social engineering tricks, they protect email addresses from automated harvesting. Before we describe how to use social networks as harvesting engines, we present the defensive measures taken by two popular social networking sites, Facebook and Twitter. Facebook does not reveal a user's email address to any other user that is not in his friend list. In case the harvester is in the list, the user's email address is presented as a GIF image to prevent automated extraction. Twitter, on the other hand, does not reveal a user's

email address in any form. However, the personal information that is revealed includes the user's name, personal web page, location and a short bio description.

We identify and outline two different strategies that spammers may follow depending on the type of spam campaigns they wish to promote. First, we have spammers that propagate emails that contain advertisements for various products. This type of spammer will follow the *blind harvesting* approach which is the technique that will result in gathering as many email addresses as possible. Second, we have spammers that use spam emails to propagate scams, such as phishing campaigns. This type of spammer will use the *targeted harvesting* technique that returns a much smaller number of results, but harvests information that can be used to craft very convincing personalized emails.

5.2.2.1 Blind harvesting

This technique aims to blindly harvest as many email addresses as possible in an efficient manner. The spammer does not care for personal information but simply wishes to gather email addresses. As shown by our results in Section 5.2.3.1, using social networks in conjunction with search engines is the most efficient method to harvest large numbers of email addresses.

We follow the same approach for both Facebook and Twitter to harvest email addresses. We initially crawl both networks to find names. As the structure and properties of the Facebook and Twitter networks differ, we have implemented two different crawlers for extracting names. One might use the Facebook search utility to search for and harvest names. However a far more efficient way is to use Facebook fan pages. Users become fans of an artist or an activity. One can freely browse all the names of a fan page. For example, the fan pages of Madonna and Shakira (popular pop artists) have 1.3 and 1.7 million fans respectively, while Barack Obama has 8.8 million. Any attacker can visit a popular fan page, and will immediately have access to millions of names. In the case of Twitter, we started from one initial account and then crawled the accounts the user follows, then the accounts they follow and so on. As we were interested only in the users' names and nicknames and not the actual tweets, this simple crawling is effective and fast for harvesting names.

Once the names have been harvested, they are used as terms in a search engine query. We used the Google search engine to locate email addresses. For each search term we query 8 different combinations ("term@hotmail.com", "term", "term@msn.com", "term@windowslive.com", "term@", "term at", "term@gmail.com", "term@yahoo.com") and for each query we retrieve the first 50 results. For scalability and efficiency reasons we do not open the URLs returned by the search engine. Instead, we parse the two-line summary provided in the results, for email addresses. This re-

sults in us missing a number of email addresses that may not be returned in the summary, however we remove a large overhead of having to parse the whole page. Our parser takes into account the various techniques used to hide email addresses from web crawlers, such as "username [at] domain".

5.2.2.2 Targeted harvesting

Attackers that rely on spam messages to propagate phishing schemes, can craft personalized phishing emails that are far more efficient than traditional techniques, by using personal information publicly available in social networks. Even though the blind harvesting technique can collect millions of email addresses efficiently, it presents a low probability of having these addresses matched to the name of their owners. The targeted harvesting approach links names to email addresses with a high probability, if not, absolute certainty. Furthermore, it enables the gathering of additional information that can render a targeted message much more convincing. Depending on the attack and the amount of personal information the attacker wants to collect, we describe three different methodologies for targeted harvesting.

Reverse lookup emails on Facebook. In the first case, we rely solely on the email-based search functionality of Facebook. Facebook allows users to search for other users based on their email address. We were surprised to find that even if the user has protected his email address through the privacy settings, and has made it visible only to him, his name will still show up in the search results when someone searches his email address. Only if the user disables his inclusion in public search results, we will not be able to find him using his email address. However, by default, Facebook includes users in search results. We collect names from highly populated Facebook fan pages and use the blind harvesting technique to search for email addresses using Google. We then search for the harvested email addresses in Facebook and obtain the results. This way we have a pair of a user's profile and his email address (and any other information that is public), the basic information needed for a personalized phishing email. We can augment the collected information of the matched users by inviting them to become our friends. Once a user has accepted, we now have access to all the information posted in his Facebook profile. Our results from a series of initial experiments showed that 30% of the random invitations were accepted.

A major advantage of this technique is that it does not only maps an email address to the owner's social profile, but also provides a technique for validating email addresses without the need of sending "probing" emails. When no profile is returned for a specific email address we cannot conclude if the email address is valid or not. However, when a user's profile is returned, we ascertain that the specific email is valid, since the user has entered it in his profile's contact information. Therefore, all the email addresses harvested using this technique are valid and eliminate the over-

head of sending spam emails to many email addresses that are not valid. This is another advantage for spammers, since by eliminating all the emails that would be sent to invalid addresses and reducing the overall volume of the spam emails they send, they may be able to evade spam detection systems [116] that rely on the collection of a large number of spam emails.

Nickname-based Email Harvesting. In the second case we aim to use information that is available on Twitter in order to narrow down the search space of our first technique and improve its efficiency. This is done by using the nickname information available on Twitter. Many people tend to create a nickname that they consistently use across different domains and email providers. Our method crawls Twitter and collects name and nickname pairs. We then query Google using the nickname as a search term and extract email addresses that are an exact match (for example, if the nickname was "john_doe_1" we would only extract emails of the form "john_doe_1@domain.com"). This provides an association between a name and one or more email addresses. Next, we use the harvested email addresses as terms in the email-based search functionality of Facebook, exactly as in the first technique. Using this approach, one has to check much fewer email addresses than the first technique and, additionally, the success rate is higher as Twitter users will probably also have a Facebook account. The innovation of this technique is that it combines disjoint sets of personal information publicly available on different social networks and can be fully automated.

Site-aware Harvesting. In the third case, we employ Google's Buzz [27], a recently launched social networking service. In a nutshell, Buzz is a Twitter-like social networking service (based on follower/followee relations), along with content feeds and integration with other Google services (Gmail, Google Reader, Picassa, YouTube etc.). Each Buzz user has a Google profile page that contains basic information about him and his follower/-followee relations. The Google profile page URL can either be based on the Google account username or a random long numeric identifier. The Google account username acts as a global identifier for all Google services, including the Gmail service. This means that if a user's Google profile URL includes his username and the user appears in the Buzz graph, then we automatically know his Gmail address. Thus, we can use the social graph of Buzz as a means to discover Gmail addresses. This approach has two major advantages. First, all harvested emails are valid. Second, and most important, for all collected email addresses we have the name of their owner, as we can extract it from the corresponding profile page. Moreover, since Buzz actually prompts the user to link and fetch content from other sites such as Twitter, Flickr, Google Reader, YouTube, FriendFeed and LinkedIn, the attacker can enrich the amount and type of information assembled and utilized for the targeted spam campaign. We crawl Buzz profiles, through the Buzz search feature, by looking up names collected from Facebook and extract the follower/followee relations, wherever it is feasible. Additionally,

references to unrelated profiles are returned by the search results as part of the indexed content. In the case where the user hides his relations, we are still able to process the profile contents, comprised of messages from and to other users. All names, that are rendered as clickable links to their respective profile pages, have their profile identifiers exposed. Even if Buzz decides to remove these links, effectively crippling the usability of the profile page, we could simply collect their names and look them up separately through the Buzz search feature.

5.2.3 Measurements

Here we evaluate the proposed email harvesting techniques described in detail in Section 5.2.2. Furthermore we compare our techniques with the currently used approaches described in Section 5.2.1. Finally, we perform a study regarding the use of harvested information in a spam campaign.

5.2.3.1 Blind Harvesting

We evaluate the use of our blind harvesting technique in comparison to current approaches. For obvious reasons we have omitted the malware and malicious site approaches from our comparison. Before proceeding to the analysis we first present and explain the comparison axes of our evaluation. We use three metrics:

- **Addresses-per-keyword ratio.** It is one of the most important metrics. A low ratio means that for each keyword queried the number of email addresses harvested is low. A high ratio means that the methodology can extract tens or hundreds of email addresses per keyword.
- **Traffic volume ratio.** Using search engines and sites for harvesting purposes requires downloading millions of pages. Downloading Gigabytes of data to harvest only a few email addresses decreases the scalability of the approach.
- **Automation.** Harvesting methodologies must be automated in order to be efficient. Although some approaches present high addresses-per-keyword ratio, they require manual intervention as they use information that does not expand and is located in multiple locations.

Address-per-keyword Ratio. Our first measurement evaluated the addresses-per-keyword ratio between our blind harvesting technique and four traditional harvesting methods: crawling archive sites, crawling the web for documents, a generic dictionary attack and a specialized dictionary attack. We crawled the MARC [33] and the W3C archive [42] sites to search for email addresses. For the document harvesting experiment, we

	Dataset	Unique emails	Ratio
Facebook Names	82,383	3,706,493	1:45
Twitter Names	87,334	2,012,391	1:23
Twitter Nicks	31,358	784,099	1:25
Dictionary	146,973	3,630,071	1:24.7
Surnames	23,300	2,200,225	1:94
Documents	680,973	445,678	1:0.65
MARC	438,722	5,265	1:0.012
W3C	376,641	330,436	1:0.87

Table 5.1: A detailed listing of the dataset size and the number of unique email addresses harvested for each technique.

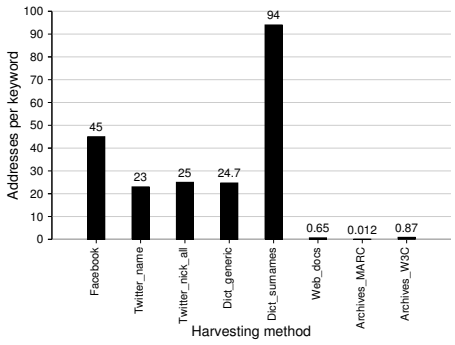


Figure 5.1: Ratio of unique email addresses per keyword for various email harvesting methodologies.

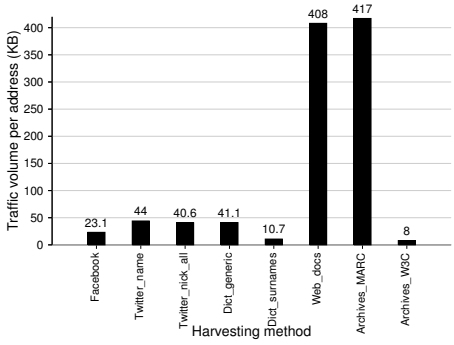


Figure 5.2: Ratio of traffic volume per email address for various harvesting methodologies

only retrieved MS Word, Excel, Powerpoint and PDF documents as a step to narrow down our search space. For the generic dictionary attacks, we used keywords from an English dictionary [31]. For the specialized dictionary attack we used the 23,300 most popular English surnames [26]. For our harvesting techniques we extracted user names from Facebook and Twitter as well as user "nicknames" from Twitter. In all the experiments, we extracted all email addresses from the Google query results, and additionally evaluated the case where email addresses were an exact match to the Twitter nicknames.

The results are summarized in Figure 5.1. In the case of Facebook, we extracted emails with a ratio of 1:45, i.e., we were able to harvest, on average, 45 unique email addresses per name queried. Using Twitter names, we achieved a ratio of 1:23, while a dataset of nicknames returned 25 addresses per query. The highest ratio observed was by the specialized version

of the dictionary attack, which yielded 94 addresses per keyword. In fact, this methodology was expected to harvest a larger number, as it follows a similar approach but takes the most popular English names. However, this method suffers from scalability issues as described later in this section. The generic dictionary attack, contrary to the specialized one, achieved a lower ratio of 1:24.7. Crawling the web for documents returned 0.65 addresses per file downloaded. Finally, in the case of archive site crawling, the ratio for MARC and W3C archives is 1:0.012 and 1:0.87 respectively, where the ratio is defined as addresses extracted per page fetched. The low ratio for crawling sites is due to the download of structure pages, which are pages without any email address that contain hyperlinks to pages deeper in the site hierarchy. In fact, 96.7% of the MARC pages were structure pages as this site is deeply nested. The W3C archive follows a more flat structure: 16.5% of the pages were structure pages. Ideally, if we exclude the structure pages, the ratios for the MARC and W3C archive become 1:0.4 and 1:1.05 respectively. Table 5.1 depicts the size of the aforementioned datasets, along with the count of harvested email addresses which produce the respected ratios.

Traffic Volume Ratio. Our second metric focuses on the cost per email address in Kbytes. The results are summarized in Figure 5.2. The traffic volume for the Facebook case is the number of names times the page size of Google results, that is 82,383 names times 130 Kbytes per Google result page times 8 (8 search combinations per name). The total traffic volume is around 79.8 Gbytes for approximately 23.1 Kbytes per email address. In the case we use names taken from Twitter, the ratio is 44 Kbytes per email address. When we use Twitter nicknames, the ratio drops down to 40.6 Kbytes per address. In the case of downloading office documents, the total volume of files was 181.6 Gbytes plus an additional 1.6 Gbytes for the Google queries, that is 408 Kbytes per email address. For the generic dictionary attack, we retrieved 142.3 Gbytes of search results which gives a ratio of 41.1 Kbytes per email address. For the specialized dictionary attack using popular surnames, we fetched 22.5 Gbytes of search results, that is a ratio of 10.7 Kbytes per email address. Finally, for the archive site crawling experiments, we downloaded 4.6 Gbytes, a ratio of 14.8 Kbytes per address in total. If we examine the two archive sites separately, the ratio for MARC is 417 Kbytes per address and for W3C is 8 Kbytes per address.

Automation. Our proposed harvesting technique is highly scalable. As we use information retrieved from social networks, our approach follows their growth rate. Therefore, our technique is fully automated as it expands, and no further manual intervention is needed for collecting more names that will be used as seeds. On the other hand, document crawling, generic dictionary attacks, and attacks based on surnames present very low scalability as the search terms are static, unlikely to change and have a limited dictionary size. Therefore, the process is semi-automated as customized crawlers have to be implemented for all new sites incorporated. Crawling

mailing list archives presents medium scalability as we extract information from communities that expand, but that are interested in specific topics and expand with a much slower rate than social networks. This technique is also a semi-automated process, as most of the sites follow their own format to depict email addresses, and the appropriate regular expressions have to be written by hand.

Overall, while the harvesting technique that uses surnames presents a higher ratio for keywords per email and a smaller cost, it is not the optimal and most efficient one as it relies on a finite and limited dictionary that does not expand. On the other hand, while the *blind harvesting* technique exhibits a lower ratio and slightly higher cost, it has the advantage of being scalable, as it follows the expansion rate of social networks. *In the long run, we consider this to be the optimal solution for large-scale efficient harvesting.*

5.2.3.2 Effectiveness of Targeted Harvesting

The second part of our evaluation focuses on our targeted harvesting techniques. Our experiment aims at measuring the effectiveness of these techniques for conducting personalized phishing campaigns. The results depict the percentage of names for which we can harvest at least one of their actual email addresses with each technique and therefore represent its effectiveness. We created two datasets containing randomly selected names from our databases. For reasons explained below, we selected names comprised solely of a first and last name, excluding middle names, dots or hyphens.

The first dataset contained 9000 names collected from a Facebook fan page. We used this dataset to evaluate our **first targeted harvesting technique**: for each name, we blindly harvested email addresses using the name as a search term in the Google engine and collected **any search results**. We then looked up the harvested email addresses using the Facebook search feature. If one or more profiles were returned, we checked whether any of them had a matching name with the one collected from the Facebook fan page and coupled with the email address in question. Overall, about 11.5% of unique names were associated with an email address that yielded a matching profile result from Facebook.

The second dataset was collected from crawling the Twitter network. For the **second targeted harvested technique** we wanted to measure the effectiveness of employing strict heuristics during the initial collection of email address through Google Search. For that matter, we included only **exact match results** of email addresses, i.e. only those whose prefix was identical to the Twitter username of the user being queried. Overall, using this strict Google search heuristic, we assembled 38986 <name,email> tuples, corresponding to 15627 unique names collected by our Twitter crawler. From those names, we selected 8,986 which did not contain middle names or special characters, just like in the first experiment. The reason for this

filtering lies on the straightforward verification heuristic we employed; for each email address coupled with a name, we looked it up using Facebook search and, from any profile results returned, considered a match only if the name was exactly the same as the one in the dataset. Therefore, entries with middle names or special characters, having a larger possibility of being written differently across disjoined social networks, were excluded. The addresses were grouped by the Twitter nickname that resulted in their discovery. From the 8,986 users, 3,588 (39.9%) returned a Facebook profile and 1,558 (17.8%) were an exact match. Thus, 43.4% of the names, that returned a profile, had a user name that was an exact match to the Twitter profile name. By using a fuzzy string matching approach we could improve the success percentage. It should be noted that names, that their harvested emails did not yield any Facebook results, may or may not be true positives of the targeted harvesting technique.

In comparison, the first and second methodologies, i.e., loose and strict collection of email address from Google search, may appear to be similarly effective with 11.5% and 17.8% of the names being a match. However, in the first case, a name is coupled with a much greater set of possible email addresses, requiring far more lookups in the Facebook than the second. In detail, in the first case, each name was coupled with an average of 104 email address, while, in the second case, only 4 address lookups took place for each name. Consequently, in the first case, 0.2% of email address returned a profile result with a matching name, while in the second case the effectiveness climbed to 7%.

In regards to the **Google Buzz** approach, we used 1705 names and 850 of the most common English words (such as book, chair etc.) as search terms. We gathered a total of 59,680 Google profile URLs. 40.5% of the Google profile URLs (24,206 profiles) included the users' Google username, also used by default as their email address prefix, while the rest of the profiles were assigned random identifiers. This means that for each search term we gather approximately 22 Google profile URLs and around 9 valid Gmail accounts. As mentioned in section 5.2.2, all email addresses extracted from the profile usernames are valid Gmail accounts.

5.2.3.3 Study of harvested personal info

In this section, we present a study based on the personal information publicly available on Facebook profiles harvested from our second *targeted harvesting* technique. As mentioned in Section 5.2.3.2, 1,558 unique names were associated with at least one email address which yielded an exact-match profile match in Facebook, thus verifying the initial <name,email> association made by the Twitter crawler. Some of those names had more than one email addresses providing matching profiles. We investigated

Label	Popularity
Current City	41.8% (667)
Hometown	38.8% (619)
Employers	24.9% (397)
College	24.5% (391)
High School	24.1% (385)
Relationship Status	21.0% (335)
Grad School	8.8% (140)
Birthday	3.9% (63)
Anniversary	3.4% (54)
Religious Views	2.5% (40)
Political Views	2.3% (36)

Table 5.2: Selected labels of personal information available on a Facebook profile page and their respective popularity among the matching profiles of the targeted harvesting evaluation.

Category	Frequency
TV/Cinema	50%
Music	24%
Activity/Sports	10%
City/Travel	11%
Various	3%
Technology	2%

Table 5.3: Content categorization of the 100 most frequent items in a Facebook profile page.

those cases and concluded that the profiles belonged to different people that shared the same name. Overall, 1,558 names led to 1,597 distinct profiles.

In Table 5.2, we present some selected labels of information, availabe on the Facebook profiles we harvested, which we consider to reveal personal information that can be exploited by attackers for targeted phishing attacks. For instance, one may use information about current employers or a person's studies to fake a workplace or college-related message. By adding such information, the email becomes more convincing and is therefore more likely to fool its recipient. For a full list of the categories, the reader may refer to the Appendix, at the end of this work.

Subsequently, we proceed to examine the content of the Facebook profile, i.e., the page elements. We select the top 100 that appear more frequently among our dataset and apply a manual categorization. Table 5.3 summarizes the results. One may observe that items related to TV and cinema are the most common. An attacker could lure victims by crafting phishing messages to include references to such popular content.

As shown by recent phishing campaigns [29], attackers use information regarding a victim's Facebook contacts, to impersonate their friends and trick them into giving them money. This type of attack could easily propagate to email phishing campaigns. To measure the feasibility of such attack, we calculate the percent of the harvested profiles which expore their respective friend lists. Overall, 72.6% of them, leak such information and the mean number of friends is 238.

5.2.4 Lessons Learned

In this work, we present how information, that is publicly available in social networking sites, can be used for harvesting email addresses and deploying personalized phishing campaigns. We argue that an inherent challenge of a social network is the visibility of its members. The mere participation of a user renders him a target for personalized. We present two different approaches to harvesting email addresses. Blind harvesting uses names collected from social networking sites and aims to collect as many email addresses as possible. Using this technique we were able to harvest millions of email addresses in an efficient fashion. Targeted harvesting aims to harvest email addresses that can be mapped to a name and publicly available information and, thus, greatly enhance the efficiency of a spam campaign. We present three such techniques. The first technique blindly harvests email addresses and uses Facebook to map them to a user name, with a success rate of 11.5%. By using information available in the Twitter network we are able to narrow the search space and accurately map 43.4% of the user profiles.

5.3 Breaking Facebook's Social Authentication

An extended version of this work can be found in [96].

5.3.1 Social Authentication

We first here describe the nature of Facebook's SA in terms of functionality and heuristics. We go beyond a general description and evaluate its behavior under real-world conditions. Facebook's SA was announced in January 2011 and, to the best of our knowledge, is the first instance of an authentication scheme based on the "who you know" rationale: A user's credentials are considered authentic only if the user can correctly identify his friends.

5.3.1.1 How Social Authentication Works

After the standard, password-based authentication, the user is presented with a sequence of 7 pages featuring authentication challenges. As shown

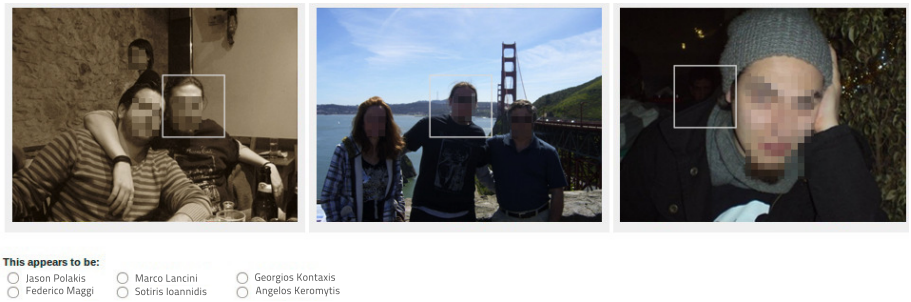


Figure 5.3: Example screenshot of the user interface of a Facebook SA page.

in Fig. 5.3, each challenge is comprised of 3 photos of an online friend; the names of 6 people from the user's social circle are listed and he has to select the one depicted. The user is allowed to fail in 2 challenges, or skip them, but must correctly identify the people in at least 5 to pass the SA test.

5.3.1.2 Requirements for Triggering

Based on our analysis, Facebook activates the SA only for the fraction of accounts that have enough friends with a sufficient amount of tagged photos that contain a human face.

Friend list. SA requires that the user to be protected has a reasonable number of friends. From our experiments we have concluded that, in the case of Facebook, a user must have at least 50 friends. To obtain this information, we created 11 distinct dummy profiles and increased the number of friends of these accounts on a daily basis, until we managed to trigger the SA (detailed in Section 5.3.6).

Tagged photos. The user's friend must be tagged (placed in a labeled frame) in an adequate number of photos. Keep in mind that since these are user-submitted tags, Facebook's dataset can get easily tainted. People often erroneously tag funny objects as their friends or publish photos with many friends tagged, several of whom may not actually be present in the photo.

Faces. SA tests must be solvable by humans within the 5 minute (circa) time window enforced by Facebook. We argue that Facebook employs a face detection algorithm to filter the dataset of tagged people to select photos with tagged faces. From our manual inspection of 127 instances of real SA tests (2,667 photos), we have noticed that Facebook's selection process is quite precise, despite some inaccuracies that lead to SA tests where some photos contain no face. Overall, 84% of these 2,667 photos contained at least one human-recognizable face, and about 80% of them contained at least one face such that an advanced face detection software can discern—in this test, we used `face.com`. To validate our argument on the use of face detection filtering, we repeated the same manual inspection on a different set of 3,486 photos drawn at random from our dataset of 16,141,426 photos

(detailed in Section 5.3.4). We then cropped these images around the tags; hence, we generated a SA dataset in the same manner that Facebook would if it naively relied only on people's tagging activity. Only 69% ($< 84\%$) of these photos contain at least one recognizable human face, thus the baseline number of faces per tag is lower in general than in the photos found in the real SA tests. This confirms our hypothesis that Facebook employs filtering procedures to make sure each SA test page shows the face of the person in question in at least one photo.

Triggering. Facebook triggers the SA when it detects a suspicious login attempt, according to a set of heuristics. Our experiments reveal that this happens when (i) the user logs in from a different geographical location, or (ii) uses a new device (e.g., computer or smartphone) for the first time to access his account.

5.3.1.3 Advantages and Shortcomings

The major difference from the traditional two-factor authentication mechanisms (e.g., confirmation codes sent via text message or OTP tokens) is that Facebook's SA is less cumbersome, especially because users have grown accustomed to tagging friends in photos. However, as presented recently by Kim et al. [80], designing a usable yet secure SA scheme is difficult in tightly-connected social graphs, not necessarily small in size, such as university networks.

Our evaluation suggests that SA carries additional implementation drawbacks. First of all, the number of friends can influence the applicability and the usability of SA. In particular, users with many friends may find it difficult to identify them, especially when there are loose or no actual relationships with such friends. A typical case is a celebrity or a public figure. Even normal users, with 190 friends on average[1], might be unable to identify photos of online contacts that they do not interact with regularly. Dunbar's number [65] suggests that humans can maintain a stable social relationship with at most 150 people. This limit indicates a potential obstacle in the usability of the current SA implementation, and should be taken into account in future designs.

Another parameter that influences the usability of SA is the number of photos that depict the actual user, or at least that contain objects that uniquely identify the particular user. As a matter of fact, feedback [77] from users clearly expresses their frustration when challenged by Facebook to identify inanimate objects that they or their friends have erroneously tagged for fun or as part of a contest which required them to do so.

Finally, in certain cases, Facebook currently presents users with the option to bypass the SA test by providing their date of birth. This constitutes

[1] https://www.facebook.com/notes/facebook-data-team/anatomy-of-facebook/10150388519243859

a major flaw in their security mechanism. Obtaining the victim's date of birth is trivial for an adversary, as users may reveal this information on their Facebook profile.

5.3.1.4 Threat Model and Known Attacks

Throughout this work we refer to the people inside a user's online social circle as friends. Friends have access to information used by the SA mechanism. Tightly-connected social circles where a user's friends are also friends with each other are the worst scenarios for SA, as potentially any member has enough information to solve the SA for any other user in the circle. However, Facebook designed SA as a protection mechanism against strangers, who have access to none or very little information. Under this threat model, strangers are unlikely to be able to solve an SA test. We argue that any stranger can position himself inside the victim's social circle, thereby gaining the information necessary to defeat the SA mechanism. Kim et al. [80] suggest that the progress made by face-recognition techniques may enable automated attacks against photo-based authentication mechanisms. At the same time, Dantone et al. [61] have demonstrated that social relationships can also be used to improve the accuracy of face recognition. Moreover, Acquisti et al. [48] went beyond the previous approach and presented a system that can associate names to faces and, thus, de-anonymize a person solely by using a picture of his or her face. Although no scientific experimentation on real-world data has been made to measure the weakness of SA, these studies suggest that the face-to-name relation, which is the security key behind SA, may be exploited further to demonstrate that the scheme is insecure. Our intuition that attackers can overcome the limitations of Facebook's perceived threat model has been the motivation behind this work.

5.3.1.5 Attack Surface Estimation

In our attack model, the attacker has compromised the user's credentials. This is not an unreasonable assumption; it is actually the reason behind the deployment of the SA. This can be accomplished in many ways (e.g., phishing, trojan horses, key logging, social engineering) depending on the adversary's skills and determination [64]. Statistically speaking, our initial investigation reveals that Facebook's current implementation results in 2 out of 3 photos of each SA page (84% of 3 is 2.523) with at least one face that a human can recognize. This makes SA tests solvable by humans. However, our investigation also reveals that about 80% of the photos found in SA tests contain at least one face that can be detected by face-detection software. This rationale makes us argue that an automated system can successfully pass the SA mechanism. To better understand the impact of our attack, we provide an empirical calculation of the probabilities of each phase of

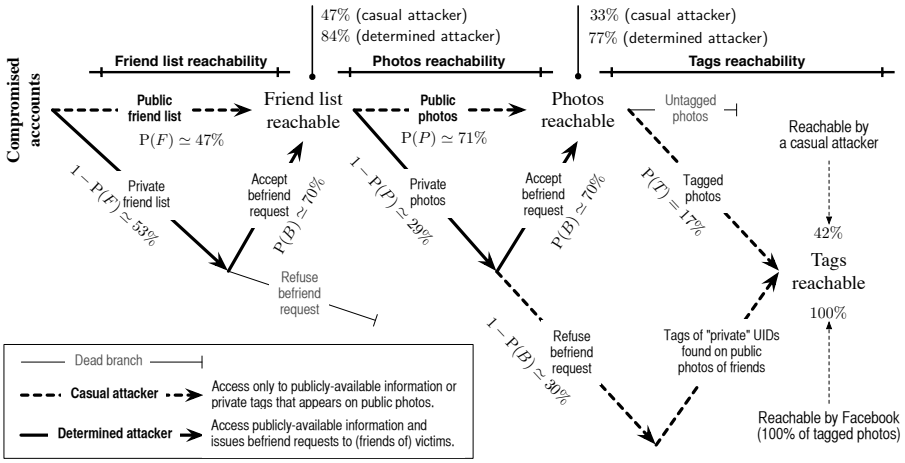


Figure 5.4: Attack tree to estimate the vulnerable Facebook population.

our attack. In other words, if an attacker has obtained the credentials of any Facebook user, what is the probability that he will be able to access the account? What is the probability if he also employs friend requests to access non-public information on profiles? To derive the portion of users susceptible to this threat, we built the attack tree of Fig. 5.4.

We distinguish between a casual and a determined attacker, where the former leverages publicly-accessible information from a victim's social graph whereas the latter actively attempts to gather additional private information through friendship requests.

Friends list. Initially, any attacker requires access to the victim's friends list. According to Dey et al. [62] $\mathrm{P}(F) = 47\%$ of the user's have their friends list public—as of March 2012. If that is not the case, a determined attacker can try to befriend his victim. Studies have shown [111, 53, 88, 54] that a very large fraction of users tends to accept friend requests and have reported percentages with a 60–90% chance of succeeding (in our analysis we use 70%, lower than what the most recent studies report). Therefore, he has a combined 84% chance of success so far, versus 47% for the casual attacker.

Photos. Ideally the attacker gains access to all the photos of all the friends of a victim. Then with a probability of 1 he can solve any SA test. In reality, he is able to access only a subset of the photos from all or a subset of the friends of a victim. Our study of 236,752 Facebook users revealed that $P(P) = 71\%$ of them exposed at least one public photo album. Again we assume that a determined attacker can try to befriend the friends of his victim to gain access to their private photos with a chance of $P(B) \simeq 70\%$ to succeed, which is a conservative average compared to previous studies. At the end of this step, the determined attacker has on average at least one photo for 77% of the friends of his victim while a casual attacker has that

for 33%. This is versus Facebook which has that for 100% of the friends with uploaded photos.

Tags. The next step is to extract labeled frames (tags) of people's faces from the above set of photos to compile $\langle \texttt{uid}, \texttt{face} \rangle$ tuples used by Facebook to generate SA tests and by the attacker to train facial models so as to respond to those tests. By analyzing $16,141,426$ photos from out dataset, corresponding to the 33% of friends' photos for the casual attacker, we found that 17% of these photos contain tags (hence usable for generating SA tests), yet only the 3% contain tags about the owner of the photo. This means that by crawling a profile and accessing its photos, it is more likely to get tags of friends of that profile than of that profile itself. The astute reader notices that Facebook also has to focus on that 17% of photos containing tags to generate SA tests: Facebook will utilize the 17% containing tags of all the photos uploaded by a user's friends and therefore generate SA tests based on 100% of the friends for whom tags are available, whereas an attacker usually has access to less than that. In the extreme case, having access to a single friend who has tagged photos of all the other friends of the target user (e.g., he is the "photographer" of the group), the attacker will acquire at least one tag of each friend of the user and will be able to train a face recognition system for 100% of the subjects that might appear in an SA test. In practice, by collecting the tags from the photos in our dataset we were able to gather $\langle \texttt{uid}, \texttt{face} \rangle$ tuples for 42% of the people in the friend lists of the respective users. Therefore, assuming that all of a user's friends have tagged photos of them on Facebook, a casual attacker is able to acquire this sensitive information for 42% of the tagged friends used by Facebook to generate SA tests. As we show in Section 5.3.6, with only that amount of data, we manage to automatically solve 22% of the real SA tests presented to us by Facebook, and gain a significant advantage for an additional 56% with answers to more than half the parts of each test. We cannot calculate the corresponding percentage for the determined attacker without crawling private phots. However, we simulate this scenario in Section 5.3.5 and find that we are able to pass the SA tests on average with as little as 10 faces per friend.

Faces. Finally, from the tagged photos, the attacker has to keep the photos that actually feature a human face and discard the rest—we can safely suppose that Facebook does the same, as discussed in Section 5.3.1.2. We found that 80% of the tagged photos in our dataset contain human faces that can be detected by face-detection software, and Facebook seems to follow the same practice; therefore, the advantage for either side is equal. Overall, our initial investigation reveals that up to 84% of Facebook users are exposed to the crawling of their friends and their photos. They are, thus, exposed to attacks against the information used to protect them through the SA mechanism. A casual attacker can access $\langle \texttt{uid}, \texttt{face} \rangle$ tuples of at least 42% of the tagged friends used to generate social authentication tests

for a given user. Such information is considered sensitive, known only to the user and the user's circle, and its secrecy provides the strength to this mechanism.

5.3.2 Breaking Social Authentication

Our approach applies to any photo-based SA mechanism and can be extended to cover other types of SA that rely on the proof of knowledge of "raw" information (e.g., biographies, activities, relationships and other information from the profiles of one's social circle). We focus on Facebook's SA, as it is the only widespread and publicly-available deployment of this type of social authentication. As explained in Section 5.3.2.1, our attack consists of three preparation steps (steps 1-3), which the attacker runs offline, and one execution step (step 4), which the attacker runs in real-time when presented with the SA test. Fig. 5.5 presents an overview of our system's design.

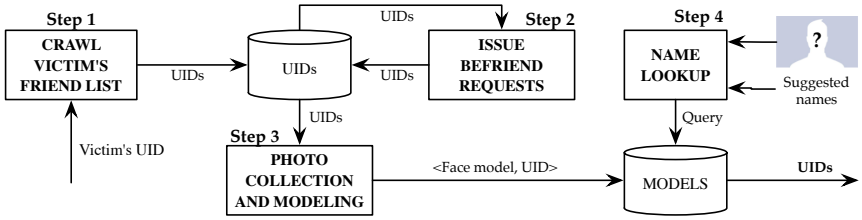


Figure 5.5: Overview of our automated SA-breaking system.

5.3.2.1 Implementation Details

Step 1: Crawling Friend List

Given the victim's UID, a crawler module retrieves the UIDs and names of the victim's friends and inserts them in our database. As discussed in Section 5.3.1.5, casual attackers can access the friend list when this is publicly available (47% of the users), whereas determined attackers can reach about 84% of the friend lists by issuing befriend requests. We implement the crawling procedures using Python's `urllib` HTTP library and regular expression matching to scrape Facebook pages and extract content. We store the retrieved data in a MongoDB database, a lightweight, distributed document-oriented storage suitable for large data collections, and keep the downloaded photos in its GridFS filesystem.

Step 2: Issuing Friend Requests

An attacker can use legitimate-looking, dummy profiles to send friend requests to all of the victim's friends. As shown in Fig. 5.4, this step can expand the attack surface by increasing the reachable photos. We implement

a procedure that issues befriend requests via the fake accounts we have created for our experimental evaluation (see Section 5.3.4). Even though we do not collect any private information or photos of these users for our experiments, we need an adequate number of friends in our accounts to be able to trigger the SA mechanism. We select users for our requests, based on the friends suggested by Facebook. Also, as shown by Irani et al. [76], to achieve a high ratio of accepted friend requests, we create profiles of attractive women and men with legitimate-looking photos[2] (i.e., avoiding the use of provocative or nudity photos). In addition, we inject some random profile activity (e.g., status messages, like activities). If Facebook triggers CAPTCHA challenges at some point, our system prompts a human operator to intervene. However, Bilge et al. [53] have demonstrated the use of automated systems against the CAPTCHA countermeasure. Moreover, to hinder spammers, Facebook limits the number of friend requests each profile is allowed to issue in a short period of time and enforces a "cooldown" period of two days on misbehavior. To overcome this obstacle and still have profiles with an adequate amount of friends, we spread our friend requests over a period of one week. We also noticed that for profiles that have education and employment information and send requests to people within these circles, Facebook enforces more relaxed thresholds and allowed us to send close to 100 requests in a single day. In addition, the method described by Irani et al.[76] allows an increase in the number of friends passively as opposed to requesting friendships explicitly.

Step 3: Photo Collection/Modeling

Photo collection We collect the URLs of all the photos contained in the albums of the target's friends using the same screen-scraping approach that we described in Step 5.3.2.1. We then feed the collected URLs into a simple module that does the actual download. This module stores in the database the metadata associated with each downloaded photo: URL, UID of the owner, tags and their coordinates (in pixels).

Face Extraction and Tag Matching We scan each downloaded photo to find faces. Specifically, we use a face detection classifier part of the OpenCV toolkit[3]. There are plenty of face detection techniques available in the literature more precise than the one that we decided to use. However, our goal is to show that face-based SA offers only a weak protection, because even with simple, off-the-shelf tools, an adversary can implement an automated attack that bypasses it.

Subsequently, we label each face with the UID of the nearest tag found in the adjacent 5%-radius area, calculated with the euclidean distance

[2] We selected photos from a database of models.

[3] http://opencv.itseez.com/

between the face's center and the tag's center. Unlabeled faces and tags with no face are useless, thus we discard them. We save the selected faces as grayscale images, one per face, resized to 130×130 pixels.

Facial Modeling We use the `sklearn` library[4] to construct a supervised classifier. We first preprocess each face via histogram equalization to ensure uniform contrast across all the samples. To make the classification job feasible with these many features (i.e., 130×130 matrices of integers), we project each matrix on the space defined by the 150 principal components (i.e., the "eigenfaces"). We tested K-nearest-neighbors (kNN), tree, and support-vector (with a radial-basis kernel) classifiers using a K-fold cross-validation technique. We found that support-vector classifiers (SVC) yield the highest accuracy, but are very expensive computationally. Therefore, we use kNN classifiers, with $k = 3$ as they provide a faster alternative to SVC with comparable accuracy.

Step 4: Name Lookup When Facebook challenges our system with a SA test, we submit the photos from the SA test to the classifier, which attempts to identify the depicted person and select the correct name. We detect the faces in each of the 7 photos of an SA page and extract the 150 principal components from each face's 130×130 matrix. Then, we use the classifier to predict the class (i.e., the UID) corresponding to each unknown face, if any. If, as in the case of Facebook, a list of suggested names (i.e., UIDs) is available, we narrow its scope to these names. Then, we query the classifier and select the outcome as the correct UID for each unknown face, choosing the UID that exhibits more consensus (i.e., more classifiers output that UID) or the highest average prediction confidence.

5.3.2.2 Face Recognition as a Service

Automatic face recognition is approaching the point of being ubiquitous: Web sites require it and users expect it. Therefore, we investigate whether we can employ advanced face recognition software offered as a cloud service. We select `face.com` which offers a face recognition platform for developers to build their applications on top of. Incidentally, `face.com` was recently acquired by Facebook[5]. The service exposes an API through which developers can supply a set of photos to use as training data and then query the service with a new unknown photo for the recognition of known individuals. The training data remains in the cloud. Developers can use up to two different namespaces (i.e., separate sets of training data) each one able

[4] http://scikit-learn.org
[5] http://face.com/blog/facebook-acquires-face-com/

to hold up to 1,000 users, where each user may be trained with a seemingly unbound number of photos. Usage of the API is limited to 5,000 requests an hour. Such a usage framework may be restrictive for building popular applications with thousands of users but it is more than fitting for the tasks of an adversary seeking to defeat photo-based social authentication. Assuming the free registration to the service, one may create a training set for up to 1,000 of a victim's friends (the max limit for Facebook is 5,000 although the average user has 190 friends). After that, one can register more free accounts or simply delete the training set when no longer necessary and reclaim the namespace for a new one. We develop a proof-of-concept module for our system that leverages the `face.com` API as an alternative, service-based implementation of steps 3 and 4 from Fig. 5.5. We submit the photos to the service via the `faces.detect` API call to identify any existing faces and determine whether they are good candidates for training the classifier. The next step is to label the good photos with the respective UIDs of their owners (`tags.save`). Finally we initiate the training on the provided data (`faces.train`) and once the process is complete we can begin our face recognition queries—the equivalent of step 4 from Fig. 5.5. Once the training phase is finished, the service is able to respond within a couple of seconds with a positive or negative face recognition decision through the `faces.recognize` call. We take advantage of the ability to limit the face matching to a group of uids from the training set and we do so for the suggested names provided by Facebook for each SA page.

5.3.3 Experimental Evaluation

Here we evaluate the nature of Facebook's SA mechanism and our efforts to build an automated SA solving system. We first assess the quality of our dataset of Facebook users (Section 5.3.4). We consider this a representative sample of the population of the online social network. We have not attempted to compromise or otherwise damage the users or their accounts. We collected our dataset as a casual attacker would do. Next we evaluate the accuracy and efficiency of our attack. In Section 5.3.5, we use simulation to play the role of a determined attacker, who has access to the majority of the victims' photos. In Section 5.3.6, we relax this assumption and test our attack as a casual attacker, who may lack some information (e.g., the victims may expose no photos to the public, there are no usable photos, no friend requests issued). More details on the capabilities of these two types of attacker are given in Section 5.3.1.5.

To perform our experiments we implemented custom face recognition software. This was done for two reasons. First, because we needed something very flexible to use, that allowed us to perform as many offline experiments as needed for the experiments of the determined attacker. Second, we wanted to show that even off-the-shelf algorithms were enough to break

	TOTAL	PUBLIC	PRIVATE
UIDs	236,752	167,359	69,393
Not tagged	116,164	73,003	43,161
Tagged	120,588	94,356	26,232
Mean tags per UID:		19.39	10.58
Tags[9]	2,107,032	1,829,485	277,547
Photos	16,141,426	16,141,426	(not collected)
Albums	805,930	805,930	(not collected)

Table 5.4: Summary of our collected dataset. The terms "public", and "private" are defined in Section 5.3.4.

the SA test, at least in ideal conditions. However, superior recognition algorithms exist, and we conducted exploratory experiments that showed that face.com, although less flexible than our custom solution, has much better accuracy. Therefore, we decided to use it in the most challenging conditions, that is to break SA tests under the hypothesis of the casual attacker.

5.3.4 Overall Dataset

Our dataset contains data about real Facebook users, including their UIDs, photos, tags, and friendship relationships, as summarized in Table 5.4. Through public crawling we collected data regarding 236,752 distinct Facebook users. 71% (167,359) of them have at least one publicly-accessible album. We refer to these users as public UIDs (or public users). The remaining 29% of UIDS (69,393) keep their albums private (i.e., private UIDs, or private users). We found that 38% of them (26,232 or 11% of the total users) are still reachable because their friends have tagged them in one of the photos in their own profile (to which we have access). We refer to these UIDs as semi-public UIDs (or semi-public users). Data about the remaining 62% of UIDs (43,161 or 18% of the total users) is not obtainable because these users keep their albums private, and their faces are not found in any of the public photos of their friends. The public UIDs lead us to 805,930 public albums, totaling 16,141,426 photos and 2,107,032 tags that point to 1,877,726 distinct UIDs. It is therefore evident that people exposing (or making otherwise available) their photos are not only revealing information about themselves but also about their friends. This presents a subtle threat against these friends who cannot control the leakage of their names and faces. Albeit this dataset only covers a very small portion of the immense

[9] On 11 April 2012, our crawler had collected 2,107,032 of such tags, although the crawler's queue contains 7,714,548 distinct tags.

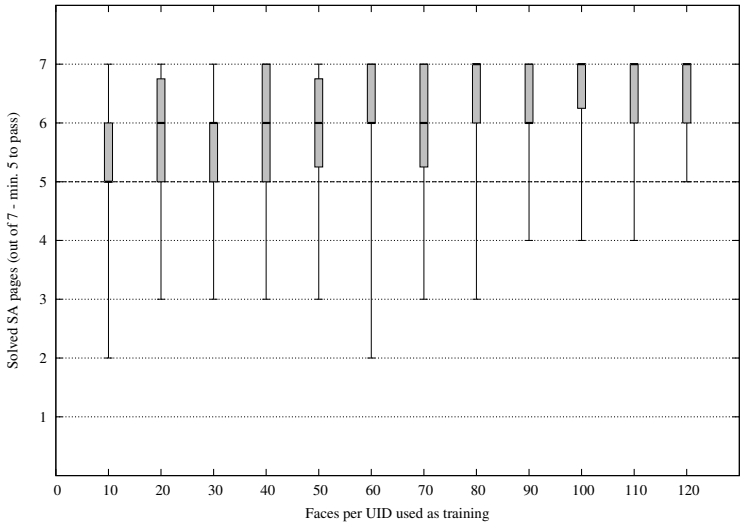


Figure 5.6: Successfully-passed tests as a function of the training-set size.

Facebook user base, we consider it adequate enough to carry out thorough evaluation experiments.

5.3.5 Breaking SA: Determined Attacker

The following experiment provides insight concerning the number of faces per user needed to train a classifier to successfully solve the SA tests. We create simulated SA tests using the following methodology. We train our system using a training set of $K = 10, 20, \ldots, 120$ faces per UID. We extract the faces automatically, without manual intervention, using face detection as described in Section 5.3.2.1. We then generate 30 SA tests. Each test contains 3 target photos per 7 pages showing the face of the same victim. The photos are selected randomly from the pool of public photos we have for each person, from which we exclude the ones used for the training. For each page and K we record the output of the name-lookup step (step 4), that is the prediction of the classifier as described in Section 5.3.2.1, and the CPU-time required. Fig. 5.6 shows the number of pages solved correctly out of 7, and Fig. 5.7 shows the CPU-time required to solve the full test (7 pages).

In order for an SA test to be solved successfully, Facebook requires that 5 out of 7 challenges are solved correctly. Our results show that our attack is always successful (i.e., at least 5 pages solved over 7) on average, even when a scarce number of faces is available. Clearly, having an ample training dataset such as $K > 100$ ensures a more robust outcome (i.e., 7 pages solved over 7). Thus, our attack is very accurate. As summarized in Fig. 5.7, our attack is also efficient because the time required for both "on

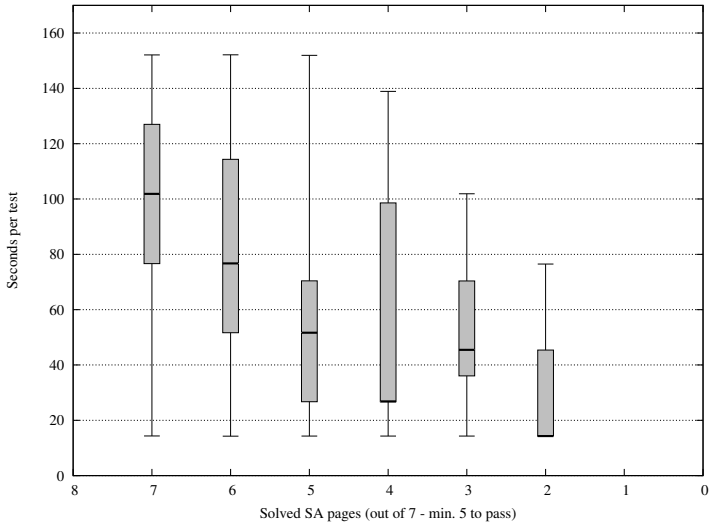


Figure 5.7: Time required to lookup photos from SA tests in the face recognition system.

the fly" training—on the K faces of the 6 suggested users—and testing remains within the 5-minute timeout imposed by Facebook to solve a SA test. An attacker may choose to implement the training phase offline using faces of all the victim's friends. This choice would be mandatory if Facebook—or any other Web site employing SA—decided to increase the number of suggested names, or remove them completely, such that "on the fly" training becomes too expensive.

5.3.6 Breaking SA: Casual Attacker

In the following experiment we assume the role of a casual attacker, with significantly more limited access to tag data for the training of a face recognition system. At the same time we attempt to solve real Facebook SA tests using the following methodology. We have created 11 dummy accounts that play the role of victims and populate them with actual Facebook users as friends and activity. Then, we employ a graphical Web browser scripted via Selenium[6] to log into these accounts in an automated fashion. To trigger the SA mechanism we employ Tor[7] which allows us to take advantage of the geographic dispersion of its exit nodes, thus appearing to be logging in from remote location in a very short time. By periodically selecting a different exit node, as well as modifying our user-agent identifier, we can arbitrarily trigger the SA mechanism. Once we are presented with an SA test, we it-

[6] http://seleniumhq.org
[7] http://www.torproject.org

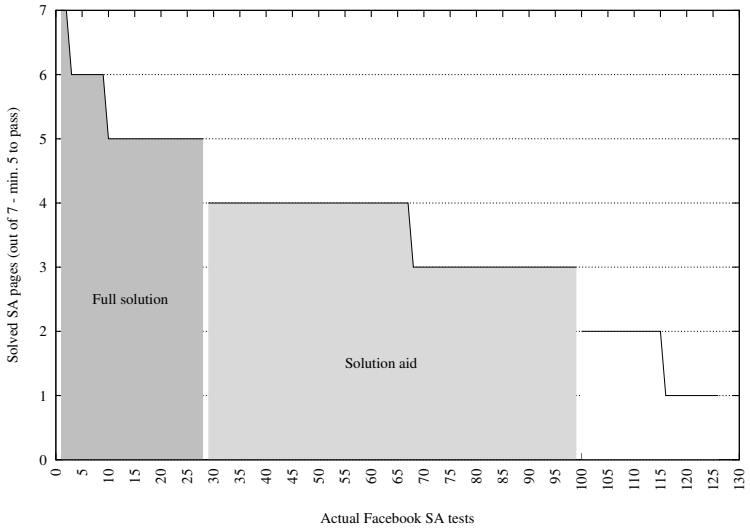


Figure 5.8: Efficiency of automated SA breaker against actual Facebook tests.

erate its pages and download the presented photos and suggested names, essentially taking a snapshot of the test for our experiments. We are, then, able to take the same test offline as many times necessary. Note that this is done for evaluation purposes and that the same system in production would take the test once and online. Overall, we collected 127 distinct SA tests.

We tried breaking the real SA tests using our module for face.com described in Section 5.3.2.2. Fig. 5.8 presents the outcome of the tests. Overall, we are able to solve 22% of the tests (28/127) with people recognized in 5–7 of the 7 test pages and significantly improve the power of an attacker for 56% of the tests (71/127) where people were recognized in 3–4 of the 7 test pages. At the same time, it took 44 seconds on average with a standard deviation of 4 seconds to process the photos for a complete test (21 photos). Note that the time allowed by Facebook is 300 seconds.

We further analyzed the photos from the pages of the SA tests that failed to produce any recognized individual. In about 25% of the photos, face.com was unable to detect a human face. We manually inspected these photos and confirmed that either a human was shown without the face being clearly visible or no human was present at all. We argue that humans will also have a hard time recognizing these individuals unless they are very close to them so that they can identify them by their clothes, posture or the event. Moreover, in 50% of the photos face.com, was able to detect a human face but marked it as unrecognizable. This indicates that it is either a poor quality photo (e.g., low light conditions, blurred) or the subject is wearing sunglasses or is turned away from the camera. Finally, in the remaining 25% of the pho-

tos, a face was detected but did not match any of the faces in our training set.

Overall, the accuracy of our automated SA breaker significantly aids an attacker in possession of a victim's password. A total stranger, the threat assumed by Facebook, would have to guess the correct individual for at least 5 of the 7 pages with 6 options per page to choose from. Therefore, the probability [8] of successfully solving an SA test with no other information is $O(10^{-4})$, assuming photos of the same user do not appear in different pages during the test. At the same time, we have managed to solve SA tests without guessing, using our system, in more than 22% of the tests and reduce the need to guess to only 1–2 (of the 5) pages for 56% of the tests, thus having a probability of $O(10^{-1})$ to $O(10^{-2})$ to solve those SA tests correctly. Overall in 78% of the real social authentication tests presented by Facebook we managed to either defeat the tests or offer a significant advantage in solving them.

After these experiments, we deleted all the photos collected from the real SA tests, as they could potentially belong to private albums of our accounts' friends, not publicly accessible otherwise.

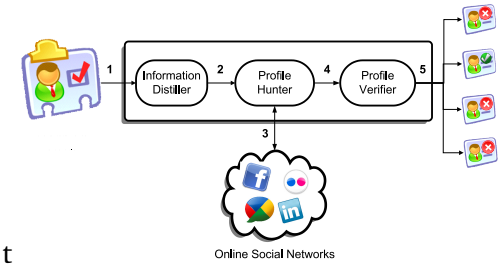
5.3.7 Lessons Learned

In this work we pointed out the security weaknesses of using social authentication as part of a two-factor authentication scheme, focusing on Facebook's deployment. We found that if an attacker manages to acquire the first factor (password), he can access, on average, 42% of the data used to generate the second factor, thus, gaining the ability to identify randomly selected photos of the victim's friends. Given that information, we managed to solve 22% of the real Facebook SA tests presented to us during our experiments and gain a significant advantage to an additional 56% of the tests with answers for more than half of pages of each test. We have designed an automated social authentication breaking system, to demonstrate the feasibility of carrying out large-scale attacks against social authentication with minimal effort on behalf of an attacker. Our experimental evaluation has shown that widely available face recognition software and services can be effectively utilized to break social authentication tests with high accuracy. Overall we argue that Facebook should reconsider its threat model and re-evaluate the security measures taken against it.

5.4 Detecting social network profile cloning

A more detailed version of our study is found in [83].

[8] Calculated using the binomial probability formula used to find probabilities for a series of Bernoulli trials.



t

Figure 5.9: Diagram of our system architecture.

5.4.1 Design

In this section we outline the design of our approach for detecting forged profiles across the Web. Our system is comprised of three main components and we describe the functionality of each one.

1. **Information Distiller**. This component is responsible for extracting information from the legitimate social network profile. Initially, it analyzes the user's profile and identifies which pieces of information on that profile could be regarded as rare or user-specific and may therefore be labeled as user-identifying terms. The information extracted from the profile is used to construct test queries in search engines and social network search services. The number (count) of results returned for each query is used as a heuristic and those pieces of information that stand out, having yielded significantly fewer results than the rest of the information on the user's profile, are taken into account by the distiller. Such pieces of information are labeled as user-identifying terms and used to create a user-record for our system along with the user's full name (as it appears in his profile). The record is passed on to the next system component that uses the information to detect other potential social network profiles of the user.

2. **Profile Hunter**. This component processes user-records and uses the user-identifying terms to locate social network profiles that may potentially belong to the user. Profiles are harvested from social-network-specific queries using each network's search mechanism that contain these terms and the user's real name. All the returned results are combined and a profile-record is created. Profile-records contain a link to the user's legitimate profile along with links to all the profiles returned in the results.

3. **Profile Verifier**. This component processes profile-records and extracts the information available in the harvested social profiles. Each profile is then examined in regards to its similarity to the user's original profile. A similarity score is calculated based on the common val-

ues of information fields. Furthermore, profile pictures are compared, as cloned profiles will use the victim's photo to look more legitimate. After all the harvested profiles have been compared to the legitimate one, the user is presented with a list of all the profiles along with a similarity score.

We can see a diagram of our system in Figure 5.9. In step (1) the Information Distiller extracts the user-identifying information from the legitimate social network profile. This is used to create a user-record which is passed on to the Profile Hunter in Step (2). Next, Profile Hunter searches online social networks for profiles using the information from the user-record in step (3). All returned profiles are inserted in a profile-record and passed on to the Profile Verifier in step (4). The Profile Verifier compares all the profiles from the profile-record to the original legitimate profile and calculates a similarity score based on the common values of certain fields. In step (5) the profiles are presented to the user, along with the similarity scores, and an indication of which profiles are most likely to be cloned.

5.4.2 Implementation

In this section we provide details of the proof-of-concept implementation of our approach. We use the social network LinkedIn [32] as the basis for developing our proposed design. LinkedIn is a business-oriented social networking site, hosting profiles for more than 70 million registered users and 1 million companies. As profiles are created mostly for professional reasons, users tend to make their profiles viewable by almost all other LinkedIn users, or at least all other users in the same network. Thus, an adversary can easily find a large amount of information for a specific user. For that matter, we consider it a good candidate for investigating the feasibility of an attack and developing our proposed detection tool.

5.4.2.1 Automated Profile Cloning Attacks

We investigate the feasibility of an automated profile cloning attack in LinkedIn. Bilge et al. [53] have demonstrated that scripted profile cloning is possible in Facebook, XING and the German sites StudiVZ and MeinVZ. In all these services but XING, CAPTCHAs were employed and CAPTCHA-breaking techniques were required. In the case of LinkedIn CAPTCHA mechanisms are not in place. The user is initially prompted for his real name, valid e-mail address and a password. This suffices for creating a provisionary account in the service, which needs to be verified by accessing a private URL, sent to the user via e-mail, and entering the account's password. Receiving such messages and completing the verification process is trivial to be scripted and therefore can be carried out without human intervention. To

address the need for different valid e-mail addresses, we have employed services such as 10MinuteMail [14] that provide disposable e-mail inbox accounts for a short period of time. Once the account has been verified, the user is asked to provide optional information that will populate his profile.

We have implemented the automated profile creation tool and all subsequent experiments detailed in this work rely on this tool and not manual input from a human. This was done to test its operation under real-world conditions. Let it be noted that all accounts created for the purposes of testing automated profile creation and carrying out subsequent experiments have been now removed from LinkedIn, and during their time of activity we did not interact with any other users of the service. Furthermore, due to ethical reasons, in the case where existing profiles were duplicated, they belonged to members of our lab, whose consent we had first acquired.

5.4.2.2 Detecting Forged Profiles

In this section we present the details of implementing our proposed detection design in Linkedin. We employ the cURL [22] command-line tool to handle HTTP communication with the service and implement the logic of the various components of our tool using Unix bash shell scripts.

1. **Information Distiller**. This component requires the credentials of the LinkedIn user, who wishes to check for clones of his profile information, as input. The component's output is a *user-record* which contains a group of keywords, corresponding to pieces of information from the user's profile, that individually or as a combination identify that profile. After logging in with the service, this component parses the HTML tags present in the user's profile to identify the different types of information present. Consequently, it employs the Advanced Search feature of LinkedIn to perform queries that aim to identify those keywords that yield fewer results that the rest [9]. Our goal is to use the minimum number of fields. If no results are returned, we include more fields in an incremental basis, according to the number of results they yield. In our prototype implementation, we identify the number of results returned for information corresponding to a person's present title, current and past company and education. We insert the person's name along with the other information in a record and provide that data to the next component.

2. **Profile Hunter**. This component employs the user-record, which contains a person's name and information identified as rare, to search LinkedIn for similar user profiles. We employ the service's Advanced

[9] Those that yield a number of results in the lowest order of magnitude or, in the worst case, the one with the least results.

Search feature to initially find out the number of returned matches and subsequently use the protected and, if available, public links to those profiles to create a *profile-record* which is passed on to the next component. The upper limit of 100 results per query is not a problem since at this point queries are designed to be quite specific and yield at least an order of magnitude less results, an assumption which has been validated during our tests.

3. **Profile Verifier**. This component receives a *profile-record* which is a list of HTTP links pointing to protected or public profiles that are returned when we search for user information similar to the original user. Subsequently, it accesses those profiles, uses the HTML tags of those pages to identify the different types of information and performs one to one string matching with the profile of the original user. This approach is generic and not limited to a specific social network, as the verifier can look for specific fields according to each network. In our prototype implementation, we also employ naive image comparison. We assume that the attacker will have copied the image from the original profile. We use the convert tool, part of the ImageMagick suite, to perform our comparisons. In detail, to discover how much image 'A' looks like image 'B', we calculate the absolute error count (i.e. number of different pixels) between them and then compare image 'A' with an image of random noise (i.e. random RGB pixels). The two error counts give the distance between 'A' and something completely random and the distance between 'A' and 'B'. This way we can infer how much 'A' and 'B' look alike. To correctly estimate the threshold of error that can be tolerated, we plan on conducting a study where images will be manipulated so as to differ from the original photo but remain recognizable. The component outputs a similarity score between the original profile and each of the other profiles.

5.4.3 Evaluation

In this section we evaluate the efficiency of our proposed approach for detecting forged social network profiles. First, we provide data from a study on LinkedIn regarding the amount of information exposed in public or protected [10] user profiles.

5.4.4 LinkedIn Study

In order to understand how much information is exposed in public profiles of LinkedIn users, we compiled three distinct datasets of profiles and studied their nature. The idea is that an adversary seeking to perform automated

[10] To view the profile information, a service account is required.

Trace Name	Description	Profiles
surnames	Popular 100 English names	11281
companies	Fortune 100 companies	9527
universities	Top 100 U.S. universities	8811

Table 5.5: Summary of data collected.

profile cloning, can create such datasets and copy their information. Here we study the type and amount of information available for such an attack.

Table 5.5 presents those three distinct datasets. To do so, we created a fake LinkedIn account, that contains no information at all, and used the service's search feature to locate profiles that matched our search terms. In the free version of the service, the number of search results is bound to 100 but one can slightly modify his queries to count as different searches and at the same time return complementary sets of results. In our case, we used three lists as search terms to retrieve user profiles; one with the most common English surnames, one with the top companies according to Fortune Magazine [25] and one with the top U.S. universities.

Each of the $\sim$30K search results returned a summary of the user's profile, which we consider adequate information to convincingly clone a profile. As we can see in table 5.6, almost one out of every three returned search results is public and contains the user's name, along with current location and current title or affiliation. These profiles are accessible by anyone on the web, without the need for a LinkedIn account. In detail, in the *surnames* dataset 89% of the profiles has a public presence on the web. On the other hand, for profiles collected from the *companies* and *universities* datasets, public presence is merely 2.3% and 1.6% respectively. The big discrepancy is probably due to the fact that users from the industry and academia use LinkedIn for professional purposes and therefore set their profiles as viewable by other LinkedIn users only.

Table 5.7 presents the core profile information in all the profiles that are publicly available. What is interesting is the fact that, besides the person's name, almost all public profiles carry information about the present location and relative industry. Additionally, about half of the profiles include a person's photo, current title or affiliation and education information.

In Table 5.8, we can see the information available in all the profiles that require a LinkedIn account for viewing. While the percentage of profiles from which we can access the user's photo is smaller compared to the public profiles, all the important information fields present a much higher availability. The fact that we cannot access the photos in many profiles is due to default privacy setting of LinkedIn where a user's photo is viewable only to other users from the same network. Nonetheless, an adversary could set his account to the specific network of the targeted victims in order to harvest the photo. Furthermore, all users reveal their location, and connections, and al-

	surnames	*companies*	*universities*
Public Name	90.5%	2.5%	2.0%
Public Profile	89%	2.3%	1.6%

Table 5.6: Exposure of user names and profile information.

	surnames	*companies*	*universities*
Photos	47%	59%	44%
Location	98%	99%	99%
Industry	85%	97%	98%
Current Status	70%	86%	72%
Education	53%	66%	82%
Past Status	42%	54%	63%
Website	36%	50%	39%
Activities / Societies	21%	22%	55%

Table 5.7: Information available in public LinkedIn profiles for each dataset.

most all their industry field. Most profiles from the surname dataset contain information regarding the user's current work status and education (86% and 70% respectively). The other datasets have an even larger percentage verifying the professional usage orientation of the users. Specifically, 99% of the profiles from the companies dataset contained information on current status and 92% revealed the user's education, and profiles from the universities dataset stated that information in 94% and 99% of the cases. Therefore, any user with a LinkedIn account can gain access to user-identifying information from profiles in the vast majority of cases.

A short study by Polakis et. al [94] concerning the type and amount of information publicly available in Facebook profiles, demonstrated a similar availability of personal information. While their results show a lower percentage of Facebook users sharing their information publicly, close to 25% of the users revealed their high school, college and employment affiliation, and over 40% revealed their current location.

As demonstrated from both of these studies, it is trivial for an adversary to gather information from social network accounts that will allow him to successfully clone user profiles. With the creation of a single fake account, an adversary can gain access to a plethora of details that we consider sufficient for deploying a very convincing impersonation attack. Even so, this information is also sufficient for the detection and matching of a duplicate profile from our tool.

5.4.4.1 Detection Efficiency

Initially, we evaluated our hypothesis that different pieces of information from a user profile yield a variable number of results when used as search

	surnames	*companies*	*universities*
Photos	22%	52%	26%
Location	100%	100%	100%
Industry	94%	100%	100%
Connections	100%	100%	100%
Current Status	86%	99%	94%
Education	70%	92%	99%
Past Status	58%	96%	95%
Twitter Username	13%	0%	1%
Websites	41%	2%	1%

Table 5.8: Information available in protected LinkedIn profiles.

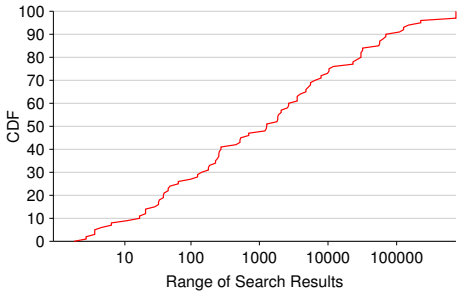


Figure 5.10: CDF of the range of search results returned for different pieces of information on a user profile.

terms, for instance in a social network's search engine. To do so, for each profile in our datasets, we extracted the values from different types of information and used them as search terms in the Advanced Search feature of the service. Next, we recorded the minimum and maximum number of results returned by any given term. Finally, we calculated the range (maximum - minimum) of search results for information on that profile. Figure 5.10 presents the CDF of the range of search results returned for each profile in our dataset. One may observe a median range value of $\sim$1000 and also that only 10% of profiles had a range of search results lower that 20. Overall, we can see that the majority of profiles exhibited diversity in the number of search results returned by different pieces of information, and by leveraging this can be uniquely identified by the carefully crafted queries of our system.

Next, we conducted a controlled experiment to test the efficiency of our tool. Due to obvious ethical reasons, we were not able to deploy a massive profile cloning attack in the wild. Thus, we selected a set of 10 existing LinkedIn profiles, that belong to members of our lab, and cloned them inside the same social network using the automated method described in 5.4.2.

We, then, employed our tool to try and find the duplicates. In total, we were able to detect all the profile clones without any false positives or negatives.

Finally, we used public user profiles as seeds into our system to try and detect existing duplicates inside LinkedIn. The Information Distiller produced user-records using information from current or past employment and education fields. Overall, we used 1,120 public profiles with 756 derived from the surnames dataset, the 224 public profiles from the companies dataset and the 140 public profiles from the universities dataset. The Profile Hunter component returned at least one clone for 7.5% of the user profiles (in 3 cases our tool discovered 2 cloned instances of the profile). Our prototype system relied on the exact matching of fields and did not employ our image comparison technique to detect cloned profiles. Furthermore, similarity scores were based on the number of fields that contained information on both profiles (in several cases, one profile had less fields that contained information). After manual inspection, we verified that all detected profiles pointed to the actual person and that the score produced by the Profile Verifier was accurate. We cannot be certain if those clones are the result of a malicious act or can be attributed to misconfiguration. Furthermore, our prototype may have missed cloned profiles where the attacker deliberately injected mistakes so as to avoid detection.

5.4.5 Lessons Learned

In this work, we propose a methodology for detecting social network profile cloning. We first present the design and prototype implementation of a tool that can be employed by users to investigate whether they have fallen victims to such an attack. The core idea behind our tool is to identify any information contained in a user's profile that can uniquely identify him. We evaluate our assumption regarding the effectiveness of such a tool and find that user profiles usually reveal information that is rare and, when combined with a name, can uniquely identify a profile and thereby any existing clones. In that light, we present the findings from a study regarding the type and amount of information exposed by social network users and conclude that the same user-identifying information which allows an attacker to clone a profile also assists us in identifying the clone. This is demonstrated by a test deployment of our tool, in which we search LinkedIn for duplicate profiles, and find that for 7% of the user profiles checked, we discover a duplicate profile in the same social network.

6 Conclusions

In this deliverable we gave an overview of the research conducted by the *SysSec* consortium during the project on the area of cyberattacks. This work deals with many of the properties of the research topics that we have identified in the *SysSec* Research Roadmaps [57, 58].

Briefly, in Chapter 2, we present our research work related to attacks on web applications and services. In the following Chapter 3, we introduce our work that deals with attacks on smart and mobile devices. Then, we present cyberattacks on users privacy in Chapter 4. Finally, in Chapter 5, we overview our research work related to cyber threats on social networks.

A significant part of our research on cyberattacks area was performed around social networks and privacy. This is because of the tremendous popularity that these systems have gained recently. Hundreds of millions of users are registered in social networking sites and regularly use their features, but, also expose sensitive information valuable for intruders. The stupendous popularity as well as the big amount of sensitive information that these sites expose, was the reason behind the fact that we focused more on social networks in our cyberattacks study.

Bibliography

[1] http://www.blade-defender.org/.

[2] http://www.malwaredomainlist.com/.

[3] http://www.offensivecomputing.net/.

[4] http://www.metasploit.com/.

[5] http://www.javascriptobfuscator.com/.

[6] http://vrt-blog.snort.org/2013/04/changing-imei-provider-model-and-phone.html.

[7] http://blog.sfgate.com/techchron/2013/10/10/stanford-researchers-discover-alarming-method-for-phone-tracking-fingerprinting-through-se

[8] http://code.google.com/p/openintents/wiki/SensorSimulator.

[9] http://developer.android.com/reference/android/hardware/SensorManager.html.

[10] http://code.google.com/p/smali/.

[11] http://code.google.com/p/android-apktool/.

[12] https://www.duosecurity.com/blog/dissecting-androids-bouncer.

[13] https://codepainters.wordpress.com/2009/12/11/android-imei-number-and-the-emulator/.

[14] 10 Minute Mail. http://10minutemail.com/.

[15] ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.0/programs/ab.html.

[16] Android ad networks. http://www.appbrain.com/stats/libraries/ad.

[17] Android Platform Distribution. http://developer.android.com/about/dashboards/index.html.

[18] Anubis. http://anubis.iseclab.org.

[19] Apache Cordova. http://cordova.apache.org.

[20] Arm: Virtualization extensions. http://www.arm.com/products/processors/technologies/virtualization-extensions.php.

[21] Contagio. http://contagiominidump.blogspot.com.

[22] cURL. http://curl.haxx.se/.

[23] Facebook Developers - Permissions. https://developers.facebook.com/docs/reference/api/permissions/.

[24] Facebook for Websites. https://developers.facebook.com/docs/guides/web/.

[25] Fortune magazine. http://money.cnn.com/magazines/fortune/.

[26] Genealogy data: Frequently occurring surnames. http://www.census.gov/genealogy/www/data/2000surnames/index.html.

[27] Google buzz. http://buzz.google.com/.

[28] Jeremiah Grossman - crossdomain.xml statistics. http://jeremiahgrossman.blogspot.com/2006/10/crossdomainxml-statistics.html.

[29] Latest Facebook Scam: Phishers Hit Up "Friends" for Cash. http://techcrunch.com/2009/01/20/latest-facebook-scam-phishers-hit-up-friends-for-cash/.

[30] LD_PRELOAD Feature. See man page of LD.SO(8).

[31] A lexical database for English. http://wordnet.princeton.edu/.

[32] LinkedIn. http://www.linkedin.com/.

[33] Mailing list archives. http://marc.info/.

[34] Man-in-the-middle proxy. http://mitmproxy.org.

[35] OAuth. http://oauth.net/.

[36] QEMU Internals. http://ellcc.org/ellcc/share/doc/qemu/qemu-tech.html.

[37] SpiderMonkey (JavaScript-C) Engine. http://www.mozilla.org/js/spidermonkey/.

[38] Start 2012 by Taking 2 Minutes to Clean Your Apps Permissions. http://mypermissions.org/.

[39] SunSpider JavaScript benchmark. http://www2.webkit.org/perf/sunspider-0.9/sunspider.html.

[40] Twellow Directory. http://www.twellow.com/categories/.

[41] VirusTotal. http://www.virustotal.com.

[42] W3C public mailing list archives. http://lists.w3.org/.

[43] 4 ways to die opening a PDF, 2009. http://esec-lab.sogeti.com/dotclear/index.php?post/2009/06/26/68-at-least-4-ways-to-die-opening-a-pdf.

[44] Rage against the cage. http://thesnkchrmr.wordpress.com/2011/03/24/rageagainstthecage/, March 2011.

[45] Abusing WebView JavaScript Bridges. http://50.56.33.56/blog/?p=314, December 2012.

[46] Adventures with Android WebViews. http://labs.mwrinfosecurity.com/blog/2012/04/23/adventures-with-android-webviews/, April 2012.

[47] WiFi Pineapple. http://hakshop.myshopify.com/products/wifi-pineapple, last accessed July 2013.

[48] Alessandro Acquisti, Ralph Gross, and Fred Stutzman. Faces of Facebook: How the largest real ID database in the world came to be. BlackHat USA, 2011, http://www.heinz.cmu.edu/~acquisti/face-recognition-study-FAQ/acquisti-faces-BLACKHAT-draft.pdf.

[49] Elias Athanasopoulos, Antonis Krithinakis, and Evangelos P. Markatos. An architecture for enforcing javascript randomization in web2.0 applications. In *Information Security Conference*, ISC, 2010.

[50] Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, and Evangelos P. Markatos. xJS: Practical XSS Prevention for Web Application Development. In *Proceedings of the 1st USENIX WebApps Conference*, Boston, US, June 2010.

[51] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[52] A B Bhavani. Cross-site Scripting Attacks on Android WebView. *International Journal of Computer Science and Network*, 2(2), 2013.

[53] Leyla Bilge, Thorsten Strufe, Davide Balzarotti, and Engin Kirda. All your contacts are belong to us: automated identity theft attacks on social networks. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 551–560. ACM, 2009.

[54] Yazan Boshmaf, Ildar Muslukhov, Konstantin Beznosov, and Matei Ripeanu. The socialbot network: when bots socialize for fame and money. In *Proceedings of the Annual Computer Security Applications Conference*. ACM, 2011.

[55] Bramley Jacob. Caches and Self-Modifying Code. http://community.arm.com/groups/processors/blog/2010/02/17/caches-and-self-modifying-code.

[56] The SysSec Consortium. Deliverable d4.1: First report on threats on the future internet and research roadmap, September 2011. http://www.syssec-project.eu/media/page-media/3/syssec-d4.1-future-threats-roadmap.pdf.

[57] The SysSec Consortium. SysSec D4.2: Second Report on Threats on the Future Internet and Research Roadmap, September 2012. http://syssec-project.eu/nNa#syssec-d4.2-future-threats-roadmap-2012.pdf.

[58] The SysSec Consortium. The Red Book: A Roadmap for Systems Security Research, September 2013. http://red-book.eu/.

[59] Marco Cova. Malicious PDF trick: XFA. http://www.cs.bham.ac.uk/~covam/blog/pdf/.

[60] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, 2010.

[61] M. Dantone, L. Bossard, T. Quack, and L. Van Gool. Augmented faces. In *Proceedings of the 13th IEEE International Workshop on Mobile Vision*. IEEE, 2011.

[62] Ratan Dey, Zubin Jelveh, and Keith Ross. Facebook users have become much more private: A large-scale study. In *Proceedings of the 4th IEEE International Workshop on Security and Social Networking*. IEEE, 2012.

[63] Sanorita Dey, Nirupam Roy, Wenyuan Xu, and Srihari Nelakuditi. Acm hotmobile 2013 poster: Leveraging imperfections of sensors for fingerprinting smartphones. *SIGMOBILE Mob. Comput. Commun. Rev.*, 17(3), November 2013.

[64] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, 2006.

[65] Robin Dunbar. *Grooming, Gossip, and the Evolution of Language*. Harvard University Press, 1998.

[66] E. ECMA. 357: ECMAScript for XML (E4X) Specification. *ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland*, 2004.

[67] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th international conference on Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, 2009.

[68] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2010.

[69] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, 2012.

[70] K. Fernandez and D. Pagkalos. XSSed.com. XSS (Cross-Site Scripting) information and vulnerable websites archive. http://www.xssed.com.

[71] Eric Filiol. New viral threats of PDF language. Black Hat Europe, March 2008.

[72] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Wepawet. http://wepawet.cs.ucsb.edu/.

[73] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE, 2013.

[74] Matthew Van Gundy and Hao Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 8-11, 2009.

[75] Thorsten Holz. Analyzing malicious pdf files, 2009. http://honeyblog.org/archives/12-Analyzing-Malicious-PDF-Files.html.

[76] Danesh Irani, Marco Balduzzi, Davide Balzarotti, Engin Kirda, and Calton Pu. Reverse social engineering attacks in online social networks. In *Proceedings of the 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2011.

[77] David Jacoby. Facebook Security Phishing Attack In The Wild. Retrieved on January 2012 from http://www.securelist.com/en/blog/208193325/Facebook_Security_Phishing_Attack_In_The_Wild.

[78] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[79] R. Jones, R. Kumar, B. Pang, and A. Tomkins. I Know What You Did Last Summer: Query Logs and User Privacy. 2007.

[80] Hyoungshick Kim, John Tang, and Ross Anderson. Social Authentication: Harder than it Looks. In *Proceedings of the 2012 Cryptography and Data Security conference*.

[81] G. Kontaxis, M. Polychronakis, and E. Markatos. SudoWeb: Minimizing Information Disclosure to Third Parties in Single Sign-On Platforms. *Information Security Conference*, 2011.

[82] Georgios Kontaxis, Demetris Antoniades, Iasonas Polakis, and Evangelos P. Markatos. An empirical study on the security of cross-domain policies in rich internet applications. In *The Fourth European Workshop on System Security*, EUROSEC, 2011.

[83] Georgios Kontaxis, Iasonas Polakis, Sotiris Ioannidis, and Evangelos P. Markatos. Detecting social network profile cloning. In *3rd IEEE International Workshop on SEcurity and SOCial Networking*, SESOC, 2011.

[84] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on WebView in the Android System. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.

[85] Felix Matenaar and Patrick Schulz. Detecting Android Sandboxes. http://dexlabs.org/blog/btdetect, August 2012.

[86] Mark Meiss, John Duncan, Bruno Gonçalves, José J. Ramasco, and Filippo Menczer. What's in a session: tracking individual behavior on the web. In *Proceedings of the 20th ACM conference on Hypertext and hypermedia*, 2009.

[87] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 8-11, 2009.

[88] Frank Nagle and Lisa Singh. Can friends be trusted? Exploring privacy in online social networks. In *Proceedings of the 2009 International Conference on Advances in Social Network Analysis and Mining*. IEEE, 2009.

[89] S. Nanda, L.C. Lam, and T. Chiueh. Dynamic Multi-Process Information Flow Tracking for Web Application Security. In *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*. ACM New York, NY, USA, 2007.

[90] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. A view to a kill: Webview exploitation. In *Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, Berkeley, CA, 2013. USENIX.

[91] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[92] Panagiotis Papadopoulos, Antonis Papadogiannakis, Michalis Polychronakis, Apostolis Zarras, Thorsten Holz, and Evangelos P. Markatos. k-subscription: Privacy-preserving microblogging browsing through obfuscation. In *The 29th Annual Computer Security Applications Conference*, ACSAC. ACM, 2013.

[93] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychroanki, and Sotiris Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of mobile malware. In *The 7th European Workshop on System Security*, EUROSEC, 2014.

[94] Iasonas Polakis, Georgios Kontaxis, Spiros Antonatos, Eleni Gessiou, Thanasis Petsas, and Evangelos P. Markatos. Using social networks to harvest email addresses. In *WPES '10: Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*.

[95] Iasonas Polakis, Georgios Kontaxis, Spiros Antonatos, Eleni Gessiou, Thanasis Petsas, and Evangelos P Markatos. Using social networks to harvest email addresses. In *The 9th Annual ACM Workshop on Privacy in the Electronic Society*, WPES, 2010.

[96] Iasonas Polakis, Marco Lancini, Georgios Kontaxis, Federico Maggi, Sotiris Ioannidis, Angelos D. Keromytis, and Stefano Zanero. All your face are belong to us: Breaking facebooks social authentication. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC, 2012.

[97] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, December 2010.

[98] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th USENIX Security Symposium*, August 2009.

[99] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6th European Workshop on System Security (EUROSEC)*, EUROSEC, 2013.

[100] RT. Privacy betrayed: Twitter sells multi-billion tweet archive. http://rt.com/news/twitter-sells-tweet-archive-529/.

[101] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 8-11, 2009.

[102] Karthik Selvaraj and Nino Fred Gutierres. The rise of PDF malware, 2010. http://www.symantec.com/connect/blogs/rise-pdf-malware.

[103] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[104] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC, 2013.

[105] Didier Stevens. Malicious PDF documents explained. *IEEE Security and Privacy*, 9(1):80–82, 2011.

[106] Brad Stone. Facebook aims to extend its reach across the web. *New York Times*, 2008.

[107] The Honeynet Project. Droidbox. https://code.google.com/p/droidbox/.

[108] The SysSec Consortium. Deliverable D7.1: Review of the state-of-the-art in cyberattacks, June 2011. www.syssec-project.eu/nNa#syssec-d7.1-SoA-Cyberattacks.pdf.

[109] Twitaholic. Top 100 Twitterholics based on Updates. http://twitaholic.com/top100/updates/.

[110] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the 4th European Workshop on System Security*, EUROSEC, 2011.

[111] Blase E. Ur and Vinod Ganapathy. Evaluating attack amplification in online social networks. In *Proceedings of the 2009 Web 2.0 Security and Privacy Workshop*.

[112] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[113] Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis: Android Malware Under The Magnifying Glass. Technical Report TR-ISECLAB-0414-001, Vienna University of Technology, 2014.

[114] Carsten Willems, Thorsten Holz, and Feliz Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.

[115] Julia Wolf. OMG WTF PDF. 27th Chaos Communication Congress (27C3), December 2010.

[116] Yinglian Xie, Fang Yu, Kannan Achan, Rina Panigrahy, and Geoff Hulten. Spamming botnets: signatures and characteristics. In *In SIGCOMM*, 2008.

[117] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[118] Bojan Zdrnja. Sophisticated, targeted malicious pdf documents exploiting cve-2009-4324, 2010. http://isc.sans.edu/diary.html?storyid=7867.