ABSTRACT

Title of dissertation: UNDERSTAND, DETECT, AND BLOCK MALWARE DISTRIBUTION FROM A GLOBAL VIEWPOINT

Bum Jun Kwon
Doctor of Philosophy, 2018

Dissertation directed by: Professor Tudor Dumitraș
Electrical and Computer Engineering

Malware still is a vital security threat. Adversaries continue to distribute various types of malicious programs to victims around the world. In this study, we try to understand the strategies the miscreants take to distribute malware, develop systems to detect malware delivery and explore the benefit of a transparent platform for blocking malware distribution in advance.

At the first part of the study, to understand the malware distribution, we conduct several measurements. We initiate the study by investigating the dynamics of malware delivery. We share several findings including the downloaders responsible for the malware delivery and the high ratio of signed malicious downloaders. We further look into the problem of signed malware. To successfully distribute malware, the attacker exploits weaknesses in the code-signing PKI, which falls into three categories: inadequate client-side protections, publisher-side key mismanagement, and CA-side verification failures. We propose an algorithm to identify malware that exploits those weaknesses and to classify to the corresponding weakness. Using the algorithm, We conduct a systematic study of the weaknesses of code-signing PKI on a large scale. Then, we move to the problem of revocation. Certificate revocation is the primary defense against the abuse in code-signing PKI. We identify the effective revocation process, which includes the discovery of compromised certificates, the

revocation date setting, and the dissemination of revocation information; moreover, we systematically measure the problems in the revocation process and new threats introduced by these problems.

For the next part, we explore two different approaches to detect the malware distribution. We study the executable files known as downloader Trojans or droppers, which are the core of the malware delivery techniques. The malware delivery networks instruct these downloaders across the Internet to access a set of DNS domain address to retrieve payloads. We first focus on the downloaded by relationship between a downloader and a payload recorded by different sensors and introduce the downloader graph abstraction. The downloader graph captures the download activities across end hosts and exposes large parts of the malware download activity, which may otherwise remain undetected, by connecting the dots. By combining telemetry from anti-virus and intrusion-prevention systems, we perform a large-scale analysis on 19 million downloader graphs from 5 million real hosts. The analysis revealed several strong indicators of malicious activity, such as the slow growth rate and the high diameter. Moreover, we observed that, besides the local indicators, taking into account the global properties boost the performance in distinguishing between malicious and benign download activity. For example, the file prevalence (i.e., the number of hosts a file appears on) and download patterns (e.g., number of files downloaded per domain) are different from malicious to benign download activities. Next, we target the silent delivery campaigns, which is the critical method for quickly delivering malware or potentially unwanted programs (PUPs) to a large number of hosts at scale. Such large-scale attacks require coordination activities among multiple hosts involved in malicious activity. We developed Beewolf, a system for detecting silent delivery campaigns from Internet-wide records of download events. We exploit the behavior of downloaders involved in campaigns for this system: they operate in lockstep to retrieve payloads. We utilize Beewolf to identify these locksteps in an unsupervised and deterministic fashion at scale. Moreover, the lockstep detection exposes the indirect relationships among the downloaders. We investigate the indirect relationships and present novel findings such as the overlap

between the malware and PUP ecosystem.

The two different studies revealed the problems caused by the opaque software distribution ecosystem and the importance of the global properties in detecting malware distribution. To address both of these findings, we propose a transparent platform for software distribution called Download Transparency. Transparency guarantees openness and accountability of the data, however, itself does not provide any security guarantees. Although there exists an anecdotal example showing the benefit of transparency, it is still not clear how beneficial it is to security. In the last part of this work, we explore the benefit of transparency in the domain of downloads. To measure the performance, we designed the participants and the policies they might take when utilizing the platform. We then simulate different policies with five years of download events and measure the block performance. The results suggest that the Download Transparency can help to block a significant part of the malware distribution before the community can flag it as malicious.

UNDERSTAND, DETECT, AND BLOCK MALWARE DISTRIBUTION FROM A GLOBAL VIEWPOINT

by

Bum Jun Kwon

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Advisory Committee:
Professor Tudor Dumitraș, Chair/Advisor
Professor Michael Hicks, Dean's Representative
Professor Joseph JaJa
Professor Charalampos Papamanthou
Professor Dana Dachman-Soled
Doctor Jiyong Jang

In loving memory of my grandfather, Joongyoon Kwon (1922–2013).

Acknowledgments

It has been quite a long journey towards my Ph.D. degree. Since the year 2013, when I began my life as a graduate student, I encountered numerous people who owe my gratitude. I believe it was their help that made this possible.

I first would like to express my deepest appreciation for Dr. Tudor Dumitraș, my advisor. It was his guidance and support that made me through this. He introduced me to the area of security and the joy of research. He gave me the freedom to challenge the problems on my own and make decisions such as exploring opportunities in the industry. Also, I am thankful for all the great dinners and the parties, which lifted the difficulties of studying abroad. I believe I have to mention the free Nespresso! Thank you for that as well.

I am extremely grateful to the committee members, Dr. Joseph Jaja, Dr. Michael Hicks, Dr. Charalampos Papamanthou, Dr. Dana Dachman-Soled, and Dr. Jiyong Jang, for their valuable comments and time. It was my first project that I first met Dr. Joseph Jaja. I learned a lot from the collaboration, and it became the foundation of my studies. I would like to mention Dr. Jiyong Jang too. I enjoyed collaborating with him, which became my first publication in the field of security, and I am thankful for the valuable experiences he provided me as my mentor at IBM. Besides the committee members, I would like to say thank you to Dr. Amol Deshpande. His guidance has helped me gain a broader understanding in the area of graph analytics.

I would like to extend my sincere thanks to my collaborators Dr. Jayanta Mondal and Virinchi. And those from Symantec: Dr. Leyla Bilge and Dr. Christopher Gates. Thanks to your help I was able to enjoy the fruit of this study. Also, special thanks to my colleagues and friends in MC2. I feel fortunate to have such wonderful colleagues and friends. Especially, thank you Octavian and Ziyun, for being awesome lab mates. I will miss the drinks we had at Looneys and the conversations we had at MC2.

When you are a graduate student, you also have to deal with a lot of administrative issues. Thank you for the Umiacs and ECE staff members for their help with these matters. Special thanks to Dana Purcell, Emily Irwin, Maria Hoo, and Melanie Prange.

Now I would like to mention those who encouraged me outside of school, starting with my friends. Thank you for Seokbin, Doowon, Sanghyun, Jonggi, and Kyungjun for the meals we shared and the coffee breaks we had at the CS lounge. Also, thank you Kyungjin and Ahrong for the hosting! The parties and the fishing trips we had will always be in my memory. Seongsil, Taeyoung, Seokjin, Mijin,

and Sangwook, thank you for the meals, drinks, and all the other moments I did not mention here. It was a pleasure to meet all of you here in Maryland. I am very thankful to my friends from high school for greeting me every time I go back to Korea. Changhyun and Seongkwon thank you for visiting the US. Taehwang, Byunghyun, and Hyungjoo, thanks for the talks we had regardless the 13-14 hours of time difference.

This dissertation would never have been written without the love, support, and encouragement from my family. Thanks to my cousin Hyunah, it was good to have you here in Maryland. I would like to say thank you to my uncle Jinsoo too for always caring about me. I dedicate this work to the memory of my grandfather, Joongyoon. I always feel sorry that I could not be at his side when he passed away. I have saved the last word of acknowledgment to my parents Taewook and Jinsil and my brother Beomseo, for their love and support. I want to take this chance to say to them that I love you, which I am too shy to express in person.

Kwon, Bum Jun

October, 2019
Maryland, USA

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Malware is the important security threat, and adversaries continue to adopt various strategies to deliver various types of malware to the victims around the world. The well-known techniques they take are drive-by-download attacks [1], pay-per-install infrastructures [2], self-updating malware [3], or compromising the benign software distribution (e.g., abusing the code-signing certificates) [4]. In order to make the most out of the delivered malware, some malware has the functionality to deliver additional malware. The malware delivery networks instruct these downloaders across the Internet to access a set of DNS domain address to retrieve payloads. The payload delivery results in the form of coordinated waves, which often do not require any user intervention to avoid attracting attention. The payload may turn out to be another downloader, which results in a chain-of-download.

This dissertation consists of a series of studies regarding malware distribution. We first aim to understand the malware distribution, focusing on how they get delivered. Among the various distribution channels, we focus on the case where malware distributors exploit the code signing Public Key Infrastructure (PKI), which is one of the benign software distribution mechanisms. We currently depend on the code-signing PKI to set up trust in the software coming from the Internet. Where the

Certification Authorities (CAs) and the software publishers first construct the trust. The CA issue the certificate to the software publishers, vouching for their identity. With the issued certificate, the publishers sign the software they release. The users check the certificate and decide to trust the software or not. However, if adversaries can breach this chain of trust, it can threaten the end-host security. Anecdotal information suggests that an extensive scope of malicious programs may hold valid digital signatures, resulting from compromised certificates [5–8]. Moreover, we have not paid much attention to the effectiveness of its primary defense: certificate revocation. In case of the Web's PKI, many efforts have been made to measure the ecosystem, including the problems in revocation [9–12], and the vulnerabilities and abuse [13–15]. In contrast, both the threats in code-signing PKI and the effectiveness of revocation have not been measured systematically due to several challenges in the code-signing PKI. Whereas we can systemically discover potentially compromised certificates through network scanning [9–11] in the Web's PKI, we do not have a comprehensive list of code signing certificates and the corpus of signed samples, including malware, on end-hosts around the world. Such challenges make the code-signing PKI ecosystem opaque and might cause platform security protections to make incorrect assumptions about how critical properly validating the code-signing certificates and performing effective revocation are for end-host security.

In the second part of the dissertation, we develop systems that can detect malware distribution. The growing commoditization of the underground economy has given rise to malware delivery networks [2, 16]. These networks orchestrate *campaigns* to quickly deliver malware to a large number of hosts. Prior studies

generally focused on either understanding the properties of the global malware distribution networks or on investigating the various techniques that malware delivery networks utilize for the final step of the delivery process. For instance, they analyze the business models [2], their network-level behavior [17, 18], or their server-side infrastructure [19, 20]; or investigate drive-by-download attacks [1] (which exploit vulnerabilities in web browsers when users are surfing the Web), pay-per-install infrastructures [2] (which are paid services that distribute malware on behalf of their affiliates), or self-updating malware (e.g., worker bots from recent botnets [3]). Instead, we pay attention to the executable files known as downloader Trojans or dropper, which functionality is to deliver additional malware to the victim's machine. However, it is not trivial to distinguish between benign and malicious downloads based merely on their content and behavior because the act of downloading software components from the Internet is not a sign of inherently malicious intent. For example, many benign applications download legitimate installers and software updates. To overcome the challenge, we can connect the dots between the downloader and the payload. Such relationships and the generated graph can provide a better understanding of the malware distribution and expose the larger picture of the malware distribution. Then we can come up with the properties that lead to separation between benign and malicious software delivery. Moreover, by mining the graphs that reflect relationships among the artifacts of malware delivery, it is possible to identify the entire structure of the malware download activity, which may otherwise remain undetected. For instance, we can precisely characterize the relationships among downloaders and the domain names associated with each mal-

ware delivery campaigns, and the malware payloads disseminated. Furthermore, we can establish precise time bounds to identify correlated downloads. The graphs capturing these relationships can help us understand the malware delivery campaigns, provide new insights into the malware landscape, and expose fragile dependencies in the underground economy, leading to practical intervention strategies for disrupting the malware delivery process [21].

The digitally signed malware and the problems in the code signing PKI infers the opaque software distribution ecosystem as one of the primary cause. Also, we discussed the importance of understanding the structure of malware delivery and the global properties. It supports the introduction of transparency to the software distribution ecosystem as a promising direction. The anecdotal evidence suggests that deploying a transparency platform has brought a positive impact to security. A set of misissued certificates by Symantec's Thawte CA left a trace on the Certificate Transparency (CT) Log, which included a certificate issued for the domains google.com and www.google.com [22]. By introducing transparency to the domain of PKI, we now have a better opportunity to detect misbehaving CAs and the misissued certificates from the opaque ecosystem. However, besides the anecdotal case of CT, the benefit of transparency is yet not fully explored. It is still unclear whether we can benefit or not even with users with different policies concerning the utilization of the platform or with adversary influence. In the last part of the dissertation, we explore the adoption of transparency in the domain of software distribution and study the benefit it introduces regarding the blockage of the unknown malware delivery. Such transparency platform may consist of logs with events of software distribution

and the flagged malicious binaries, shared in a transparent manner (i.e., the data is open to the public, and we can verify the integrity). *Users* can submit the software distribution events (e.g., download events) to the log and utilize the log to make decisions whether to accept a download or not. The *security analyitics* may utilize the download events in the log and apply detection algorithms on the constructed downloader graph [23] with the globally aggregated features such as prevalence from the log. It can be used as a platform for *software publishers* to announce their software distribution in advance. Having such a platform, we can measure how much benefit the community can get under different policies. For example, users can have different policies in terms of *submission of download events* or *making download decisions* or *the threshold of accepting the analytic reports*. It could also be applied to the software publishers for deciding to opt-in or not to pre-announce the software distribution. We may evaluate the benefit from the aspect of how many malicious samples can be detected earlier after introducing transparency without losing too much performance regarding the accuracy of blocking malware.

Outline of the Introduction. In the introduction, we discuss the objectives and the contributions of the three topics regarding the malware distribution. We then illustrate the principle concepts including the code signing PKI and the revocation, downloaders, silent delivery campaigns, and transparency. We also provide an overview of the common data sources used throughout the studies.

1.1 Objectives and Contributions

The dissertation consists of three parts: understanding the malware distribution by measurement, detecting malware distribution, and blocking the malware distribution by having a transparent platform for software distribution.

1.1.1 Understanding Malware Distribution

In the first part of the work, we conduct several measurements regarding malware distribution. We initiate the study by investigating the dynamics of malware delivery. We share several findings including the downloaders responsible for the malware delivery and the high ratio of signed malicious downloaders. We further look into the problem of signed malware. To successfully distribute malware, the attacker exploits weaknesses in the code-signing PKI, which falls into three categories: inadequate client-side protections, publisher-side key mismanagement, and CA-side verification failures. We propose an algorithm to identify malware that exploits those weaknesses and to classify to the corresponding weakness. Using the algorithm, We conduct a systematic study of the weaknesses of code-signing PKI on a large scale. In summary, we make the following contributions:

- We propose a threat model that highlights three types of weaknesses in the code signing PKI.
- We develop techniques to address challenges specific to measuring the code signing certificate ecosystem.

- We conduct a systematic study of threats that breach the trust encoded in the Windows code-signing PKI.

Then, we move to the problem of revocation. Certificate revocation is the primary defense against the abuse in code-signing PKI. We identify the effective revocation process, which includes the discovery of compromised certificates, the revocation date setting, and the dissemination of revocation information; moreover, we systematically measure the problems in the revocation process and new threats introduced by these problems. In summary, we make the following contributions:

- We collect a large corpus of code signing certificates and the revocation information.
- We conduct the first end-to-end measurement of the code signing certificate revocation process.
- We use our data to estimate a lower bound on the number of compromised certificates.
- We highlight the problems in the three parts of the revocation process as well as new threats that result from those problems.

1.1.2 Detecting Malware Distribution

For the next part, we explore two different approaches to detect the malware distribution. We first focus on the downloaded by relationship between a downloader and a payload recorded by different sensors and introduce the downloader graph

abstraction. The downloader graph captures the download activities across end hosts and exposes large parts of the malware download activity, which may otherwise remain undetected, by connecting the dots. By combining telemetry from anti-virus and intrusion-prevention systems, we perform a large-scale analysis on 19 million downloader graphs from 5 million real hosts. The analysis revealed several strong indicators of malicious activity, such as the slow growth rate and the high diameter. Moreover, we observe that, besides the local indicators, taking into account the global properties boost the performance in distinguishing between malicious and benign download activity. For example, the file prevalence (i.e., the number of hosts a file appears on) and download patterns (e.g., number of files downloaded per domain) are different from malicious to benign download activities. In summary, we make the following contributions:

- We build a large data set of malicious and benign download activities on 5 million real hosts by reconstructing download events. We also build a ground truth of malicious and benign downloaders by combining three data sources.

- We propose a graph-based abstraction to model the download activity on end hosts, and perform a large measurement study to expose the differences in the growth patterns between benign and malicious downloader graphs.

- We use insights from our measurements to build a malware detection system, using machine learning on downloader graph features, and evaluate it using both internal and external performance metrics.

Next, we target on the silent delivery campaigns, which is the key method

for quickly delivering malware or potentially unwanted programs (PUPs) to a large number of hosts. Such large-scale attacks require coordinated activities among multiple hosts involved in malicious activity. We develop Beewolf, a system for detecting silent delivery campaigns from Internet-wide records of download events. We design Beewolf based on the key observation that campaigns frequently distributed payloads through downloaders in lockstep. Beewolf employs an algorithm that can effectively identify such locksteps in an unsupervised and deterministic fashion. By utilizing Beewolf, we study silent delivery campaigns at scale. The lockstep detection exposes the indirect relationships, not visible otherwise, among the downloaders across different machines that are caught in lockstep together. The investigation on the indirect relationships yields novel findings, e.g., malware distributed over benign software updates, a considerable overlap between the malware and PUP distribution ecosystems, and business relationships within these ecosystems, which may remain hidden. In summary, we present the following contributions:

- We conduct a systematic study of malware delivery campaigns and we report several new findings about the malware and PUP delivery ecosystems.
- We propose techniques for discovering silent delivery campaigns by detecting lockstep behavior in large collections of download events. These techniques are unsupervised and deterministic, as they do not require seed nodes and are not based on machine learning.
- We present a system, Beewolf, which implements these techniques, along with evidence-based optimizations that allow it to detect silent delivery campaigns

in a streaming fashion.

- We identify the parts of Beewolf where we can introduce parallelism, and evaluate the scalability of the parallel version of Beewolf.

1.1.3 Blocking Malware Distribution

The two different studies revealed the problems caused by the opaque software distribution ecosystem and the importance of the global properties in detecting malware distribution. To address both of these findings, we introduce a transparency platform called Download Transparency, which apply transparency in the download events. Transparency guarantees openness and accountability of the data, however, itself does not provide any security guarantees. Although there exists an anecdotal example showing the benefit of transparency, it is still not clear how beneficial it is to security. In the last part of the dissertation, we explore the benefit of transparency in the domain of downloads.

The Download Transparency consists of two transparency logs, where the download events and the flagged malicious binaries recorded in a public log backed by the Merkle tree. We utilize the platform to evaluate the benefit of transparency, which we define as (1) accurately blocking malware and (2) blocking unknown malware (i.e., before the community is aware of its maliciousness). For a much realistic evaluation, we design the participants and the policies they stick to when utilizing the platform. For instance, *users* can submit the download events to the log based on a *submission policy*. They may also use the log for decision making for downloads,

which is backed by different *download policies*. Another factor that may impact their choice is the *enforcement policy*, which is a threshold for accepting the analytic reports. In other words, to consider a binary as malware, how many analytic reports they need to see. The *security analytics* may post the malware they found to the log. Some may utilize the download events in the log to apply detection algorithms on the constructed downloader graph [23] with the globally aggregated features such as prevalence from the log. To prevent their distribution to be blocked and to provide the legitimate distribution to their user base, the *software publishers* can announce their software distribution in advance to the platform. Moreover, we do not neglect the fact that the software distribution is noisy (i.e., legitimate software download malware and malware deliver benign software). We identify the existing download events that can be interpreted as adversary influence, and evaluate the effect. We then introduce several methods that can mitigate specific adversarial submissions. In summary, we make the following contributions:

- We introduce Download Transparency, a platform for download event transparency.
- We measure the benefits of applying transparency under realistic policies and the existence of adversaries.
- We propose solutions that can mitigate adversary influence and discuss their effect.

1.2 Background Principles

1.2.1 Code Signing PKI

The code signing PKI serves as a mechanism to ensure trust in the unknown software from the Internet. We can first verify the integrity of a binary executable by comparing the hash of the file and the signed hash in the digital signature. Moreover, the chain of trust constructed from the certificate authority (CA) to the publisher provides authentication of the software. We give a brief description of the code signing process, the code signing in Windows platforms, and the trusted time stamping, which is a unique property of code-signing PKI in this chapter.

Code signing process. To issue a code signing certificate, the software publisher has to ask a Certificate Authority (CA). The CA performs a vetting process on the publisher to confirm the identity. Once the identity is proven, the CA issues the code signing certificate based on the X.509 v3 certificate standard [24] to the publisher. The publisher then utilizes the certificate to produce digitally signed software. First, the hash value of the binary is computed. Then using the code-signing private key, we digitally sign the hash value. Finally, we append the digital signature to the binary as well as the code-signing certificate.

Microsoft Authenticode. The Windows platforms have Authenticode [25] as the standard of code-signing. Authenticode is designed for Windows portable executables (PE) e.g., executable (.exe), dynamically loaded library (.dll), cabinet (.cab), ActiveX control (.ctl, .ocx), and catalog (.cat), based on the the Public Key Cryp-

tography Standard (PKCS) #7 [26]. The certificate has to contain either the X.509 code signing certificate or TSA certificate chains, a digital signature, and a hash value of the PE file, with no encrypted data.

Trusted Timestapming. The *trusted timestamping* is a unique property of the code signing PKI, which guarantees that the owner (legitimate or not) of the key signed the binary at a specific date and time. Whereas the trust in the web expires with the certificate expiration, in code signing, we employ the trusted timestamp for extending the trust in the signed software even after the certificate expiration date by allowing the software writers to obtain the signed timestamp from the Time Stamping Authority (TSA) during the code signing process.

1.2.2 Revocation Process

To protect the PKI ecosystem from the abuse of code-signing, we have "Certificate revocation" as the primary defense mechanism. CAs should revoke a certificate if it falls into either one of these reasons: the private key associated with a certificate is made public, the entity behind the certificate becomes untrusted, the certificate is used to sign malware even if the source is unknown, or if the CAs issue a certificate erroneously [4]. The revocation process consists of three roles: (1) promptly discovering compromised certificates, (2) performing an effective revocation of the certificate, and (3) disseminating the revocation information.

Discovery of potentially compromised certificates. Once notified from either internally or externally (e.g., from Anti-virus companies) about the abuse, the CAs,

who have issued the certificates, should promptly investigate and revoke the abused certificates. In other words, the "revocation delay" ($t_p - t_d$), which is the time passed from the initial discovery (t_d) and the time when revocation information is posted to the public (i.e., *revocation publication date* (t_p)), should be as short as possible. Figure 1.1 depicts the case where the discovery happened after the expiration (t_e). As the code-signing certificate supports trusted timestamping, which extends the validity of the code even after the expiration date, the revocation should be performed even for the expired certificates.

Setting the revocation date. Once the CAs confirm the abuse, they have to decide the *effective revocation date* (t_r) either in collaboration with the certificate owners or independently. The effective revocation date determines which binaries will be impacted. As shown in Figure 1.1, any binary signed after t_r, regardless of the trusted timestamp, become invalid. However, the ones signed before t_r remains valid. Furthermore, if the binary is trusted timestamp, it remains valid even after expiration, even it has a revoked certificate.

Dissemination of revocation information. The final step is disseminating the revocation information to the users. While the prior two steps required several actors in the process, CAs are solely responsible for this part of the revocation process. The two dominant methods to disseminate certificate revocation information are (1) Certificate Revocation List (CRL) [24] and (2) Online Certificate Status Protocol (OCSP) [27].

- *CRLs* contain the revocation information (certificate serial numbers, (effective)

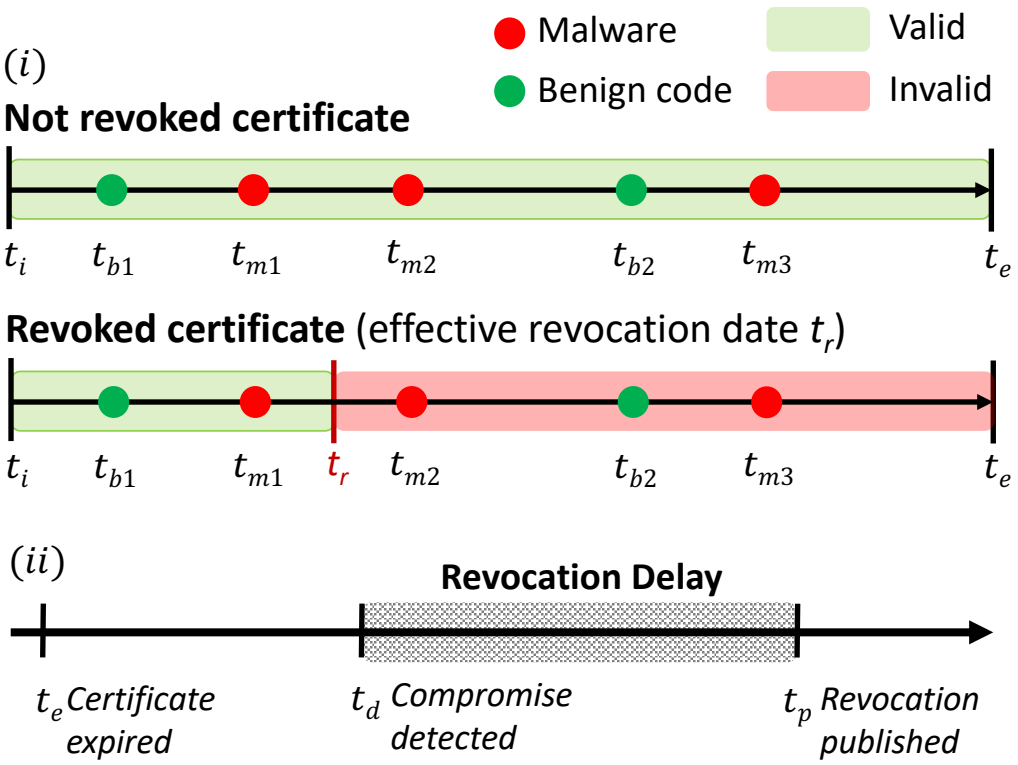

Figure 1.1: An example of (i) an *effective revocation date* (t_r) that determines the validity of signed malware and (ii) a *revocation delay* (t_p - t_d) (t_i: issue date, t_e: expiration date, t_r: effective revocation date, t_b: signing date of a benign program, t_m: signing date of malware, t_d: detection date, and t_p: revocation publication date). When an effective revocation date is set at t_r, the malware signed at t_{m1} validates continuously as it was signed before t_r.

revocation date, revocation reason) of the revoked certificates. Each CRL is updated based on their CA's policy; for example, they can publish when they have a newly revoked certificate or at a specific time of day or a day of a month. The CA has to specify the location of the CRL at CRL Distribution Point (CDP) of the X.509 certificate. Clients have to periodically download the entire CRL (not just recent changes) to check the latest revocations.

- *OCSP* was introduced to resolve the network overhead problems of CRL. Clients can query an OCSP server for a specific certificate, which helps mitigate the

network overhead at the server as well as clients. Authority Information Access (AIA), an extension field in an X.509 certificate specifies OCSP point for each certificate.

The TLS CAs are typically not responsible for providing the revocation status of expired certificates. The code signing CAs, however, must maintain and provide the revocation information of all certificates that they have issued including expired certificates due to the trusted timestamp [28, 29].

1.2.3 Overview of Downloaders

The ubiquitous Internet environment introduces many benefits for the software delivery process including customized software installation and efficient update mechanisms. It is often the case that applications use a dedicated component for determining which software components need to be installed or updated, and for downloading them from remote servers. We call this particular component the *downloader*, and both legitimate and malicious software use it to distribute their software.

It is unclear when downloaders first appeared, but this probably happened in the early 2000s. Symantec's generic Downloader.Trojan AV definition is dated April 4, 2002. A search in Google Scholar for "Downloader Trojan" doesn't return anything until 2004. We can find an early form of downloaders in the multi-phase malware. When the multi-phase malware lands to a system, it installs a small executable—called *droppers* or *downloader trojans*. Then the dropper or the

downloader trojan downloads additional components, which we call the *payload*. One well-known example of such multi-phase malware was the Sobig email worm, discovered in 2003. The downloader component was responsible for downloading additional files to set up spam relay servers on the infected machines. The downloader also had a self-updating functionality [30]. Some adapted the auto-update function against to the defenses deployed by the security community. We can find such functionality in many modern botnets [31, 32].

Then it came the emergence of the general-purpose droppers. The Bredolab trojan [33] used a general-purpose dropper which was configured to propagate different malware families. The *pay-per-install* (PPI) infrastructures employ the dropper on the client-side, to deliver arbitrary executables on thousands of hosts. Recent studies on these droppers [2] used in PPIs found that they are even configured to select the targets based on the properties of the victims (e.g., their geographical locations). The need to maintain connectivity with the droppers has lead to the creation of server-side infrastructure, which includes C&C servers for managing their malware, exploit servers to distribute the malware, payment servers for monetization, and redirectors for anonymity. These server-side systems are often designed to be resilient to takedown efforts [2,19,34] (e.g., peer-to-peer, DGA, bulletproof hosting), which allow droppers installed on end hosts continue downloading malware for long periods of time, often exceeding two years [35].

The malware distributors also made a large effort to employ sophisticated techniques to disseminate malicious payloads to a large number of hosts. These techniques include drive-by-downloads [1], social engineering and search engine poi-

soning [36]. Some even provide this functionality as a service [2]. These sophisticated downloaders represent only a fraction of the current population of downloaders; simpler and older forms of downloaders also continue to operate [35].

The act of downloading other software components without informing the user is not particularly malicious behavior. Benign applications download and install software updates silently [37,38], while other applications come in the form of bundles (e.g., InstallQ, Softronic), which include third-party software, as part of their monetization strategy. Additionally, malware authors may try to infiltrate bundles and ad-supported software, which may otherwise be benign, in order to extend their reach. Due to these reasons, it is difficult to distinguish between benign and malicious downloaders based only on their content and behavior. In consequence, the propagation of malware often relies on the downloaders.

1.2.4 Silent Delivery Campaigns

As we described in Chapter 1.2.3, malware delivery networks employ various methods to install their downloaders, e.g., drive-by-download exploits, social engineering, affiliate programs [2]. Through the connection established between the C&C server and the downloader, the malware delivery networks push the payloads to the victim hosts. The payloads include various software including malware and potentially unwanted programs (PUP), which are submitted by the clients of these delivery networks. These coordinated waves of payload delivery to the victims often do not require any user intervention to avoid detection. We define these coordi-

nated waves as *silent delivery campaigns* from its similarity with the silent updating mechanisms, which we can find in many benign software updates [37, 39]. We can further label these campaigns depending on the payloads they distribute *malware delivery campaigns*, which deliver malware such as trojan horses, bots, keystroke loggers, or *PUP delivery campaigns*, which drop PUPs such as adware, spyware and even additional droppers.

With the help of various efforts made by the security community (e.g., DNS domain blacklist), malicious silent delivery campaigns exhibit a high domain churn, while the benign campaigns keep access the same server-side infrastructure. Such frequent change in the domains is a critical behavior that can distinguish malicious and benign campaigns.

1.2.5 Transparency

Transparency is getting attention these days as a solution for solving the problems due to the opaque ecosystem. Including the cryptocurrency, e.g., the famous Bitcoin, we can find the introduction of transparency in various areas of security. Another well-known implementation is Certificate Transparency (CT), which came out as a countermeasure for the misbehavior in the Web's PKI. Such misbehavior was hard to detect due to the challenges in observing the certificates issued in the wild.

Then, how can transparency be achieved? The transparency implementations in security guarantee two properties: openness and accountability. Openness implies

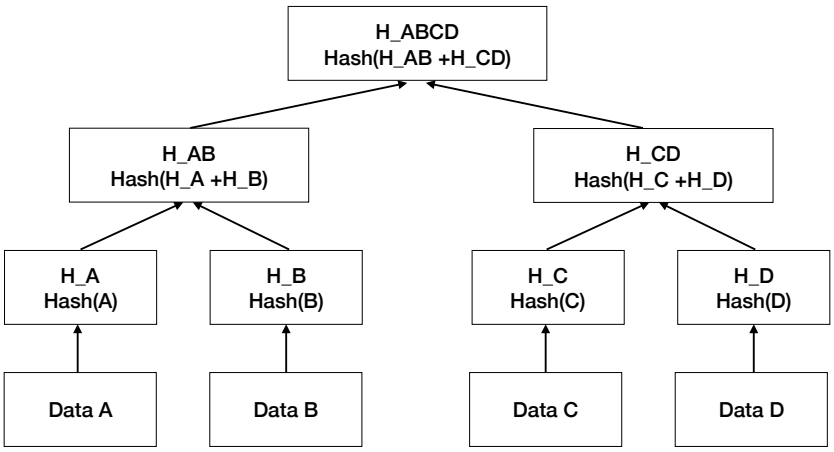

Figure 1.2: Merkle tree.

the provisioning of the data. For example, everyone can see the certificates recorded in the CT log. The next property, accountability indicates a third party can monitor the data for any malicious attempts to tamper the data. In summary, a platform is transparent if anyone can freely monitor the log and confirm the no malicious actor tampered with the data. To guarantee that someone did not tamper with the data, transparency platforms such as CT utilize cryptographic approaches, e.g., Merkle tree.

Merkle Tree is a cryptographic hash tree where the leaf node of the tree has the data, and other nodes are cryptographic hashes. As shown in Figure 1.2, the data are kept in the leaf nodes. Each of the leaf nodes has a parent node which is the computed cryptographic hash of the data. The tree is constructed by paring the leaves and hashing the concatenation. For example, the hash of data A (H_A)

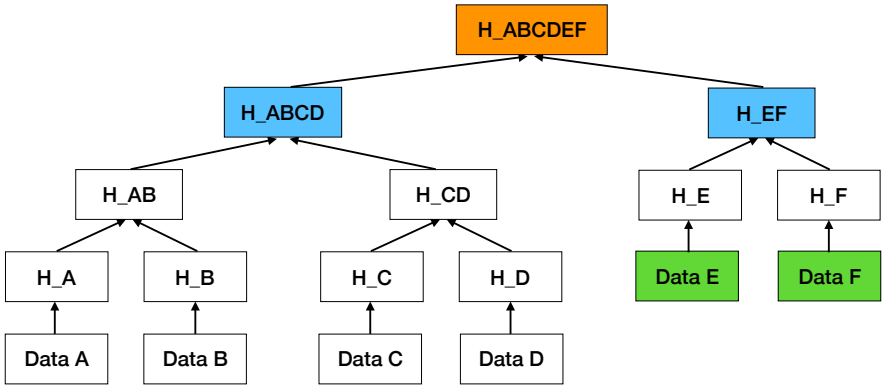

Figure 1.3: Consistency proof.

and the hash of data B (H_B) are pared, and the concatenation ($H\_A + H\_B$) is hashed resulting in node H_AB. This process continues until a single hash node remains, which becomes the head of the tree (H_ABCD). Merkle tree provides useful features for accountability.

Consistency proof. The Merkle tree supports an efficient way of verifying the consistency of two different versions of the log. The two logs are consistent, if and only if the later version contains every data in the old version in the same order. Such property implies the old version is not compromised and the log has never been branched. We can verify the consistency can by checking if the old version is the sub-tree of the new version and we can reconstruct the new version by appending the new data elements to the old version. We try to explain the process by figure 1.3.

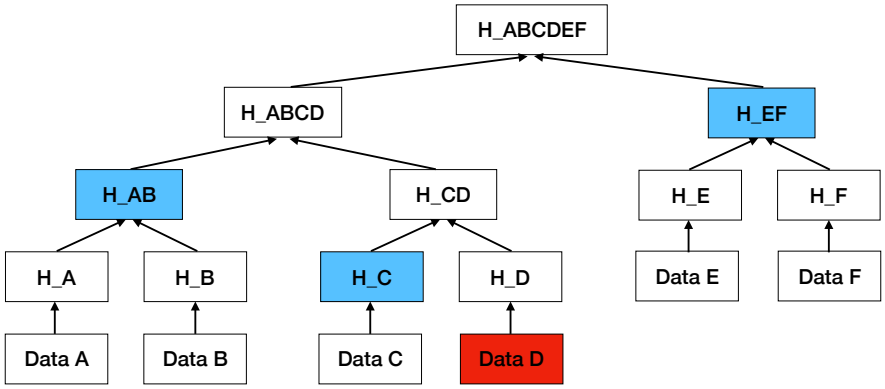

Figure 1.4: Inclusion proof.

Let us assume the old version whose head is H_ABCD, and we append Data E and Data F in the new version. In this case, the log returns the old head H_ABCD and node H_EF as the consistency proof. The old head H_ABCD shows the Data A, B, C, and D in the old version of the tree has not been modified and the order is preserved. The node H_EF can be used with H_ABCD to construct the head of the new version. If the constructed head is H_ABCDEF, then it can be guaranteed that the log is consistent.

Inclusion proof. Merkle tree also supports an efficient way of verifying if a specific data is in the log. The proof is the minimum set of hash nodes that are required to compute the intermediate nodes from the Data D leaf node to the tree head. Now we want to prove that Data D is in the log. The nodes we have to compute are H_D,

H_CD, H_ABCD and H_ABCDEF. We can compute H_D without any further information since we already know the Data D. To compute the other nodes H_CD, H_ABCD and H_ABCDEF, the following nodes are required: H_C, H_AB and H_EF. These three hash values are the minimum set of nodes that are necessary for computing the nodes between Data D and the tree head. Therefore, when we ask the Merkle tree to provide the inclusion proof of Data D H_C, H_AB and H_EF are returned.

1.3 Overview of the Data Sources

In this chapter, we discuss the primary data sources, which we continually use throughout the study.

1.3.1 Binaries Seen in the Wild

We introduce the core dataset of our study, where we extract information regarding the binaries seen in the wild. Symantec provides a platform called the Worldwide Intelligence Network Environment (WINE) [40], which consists of security telemetry collected on real hosts. Specifically, we use three WINE datasets: (1) *binary reputation*, (2) *intrusion prevention system telemetry*, and (3) *antivirus (AV) telemetry*. WINE keeps the data sets as SQL relations consisting of multiple columns.

Binary reputation. This dataset includes summarized information about all binary that appears on the Symantec's customers' machines. We collect the server-side

timestamp of the event, the unique identifier for the machine where the file is located, the name and SHA2 hash of the file, the SHA2 hash of the parent file (an archive including the file or the actual downloader), and the high-level information of the code signing certificate (e.g., issuer of the certificate and subject of the certificate).

Intrusion prevention system telemetry. The IPS telemetry dataset consists of the report of downloads of Portable Executable (PE) files over HTTP, which includes both malicious activities and the records that are not necessarily tied to any malicious activities on network streams. We extract the unique identifier of the host, the MD5 hash of the process initiated the network activity (*portal* in IPS jargon), the server-side timestamp and the URL from which the portal downloads the binary.

Anti-virus (AV) telemetry. The AV telemetry data contains information on the anti-virus signatures triggered on user machines. From this dataset, we collect the SHA256 hash of the binary that triggered the report and the name of the AV detection signature assigned to the binary.

1.3.2 Ground Truth Data

As a ground truth for known malicious and benign files, we utilize two different datasets: VirusTotal and National Software Reference Library (NSRL).

VirusTotal. VirusTotal [41] is a service that provides a private API (report API) for scanning files with up to 63 different anti-virus (AV) products, and for querying hashes to retrieve their previous analysis reports. We query VirusTotal for each

binary in the dataset to obtain its first-seen timestamp, the number of AV products that flagged the binary as malicious, the AV detection names assigned to it, the total number of AV products that scanned the binary, and the corresponding file signer information.

National Software Reference Library (NSRL). NSRL [42] is a project supported by the U.S. Department of Homeland Security, federal, state, and local law enforcement, and the National Institute of Standards and Technology (NIST). It provides a reference data set (RDS) of benign software, which is a collection of the hashes (mainly SHA1, MD5, and other metadata) of known, traceable software applications. It also maintains a list of benign software publishers.

1.4 Structure

The paper is organized as follows. In Chapter 2, we review the related works. In Chapter 3, we present the measurement studies on the malware distribution. In Chapter 4, we introduce the two techniques for detecting malware delivery. In Chapter 5, we discuss the download transparency in detail and present the benefits when introducing it to the current software distribution ecosystem. We conclude this study in Chapter 6.

1.5 Published Works and Copyrights

This dissertation extends the material from four papers by the author ©Bum Jun Kwon [4, 23, 43, 44]. Chapter 2 is based on the literature studies presented

in all four works [4, 23, 43, 44]. Chapter 3 presents the findings from measuring the code-signing PKI, which are based on the papers [4, 44]. Chapter 4 covers the materials from paper [23, 43], with some extended work from paper [43]. This introductory Chapter and the conclusion in Chapter 6 also include some material from each of these publications by the author [4, 23, 43, 44]. The definitive Version of Record of each papers were published in ©ACM CCS'15 `http://dx.doi.org/10.1145/2810103.2813724`, ©Internet Society NDSS'17 `http://dx.doi.org/10.14722/ndss.2017.23220`, ©ACM CCS'17 `https://doi.org/10.1145/3133956.3133958`, and ©USENIX Security'18 ISBN 978-1-931971-46-1.

Chapter 2: Related Work

In this Chapter, we provide the review of the works that are related to ours. We discuss the measurement studies in PKI, prior research on malware distribution, and transparency proposals in the domain of security.

2.1 PKI Measurement

We begin with reviewing the measurement studies in PKI including the HTTPS and TLS certificate ecosystem and the abuse in Authenticode and Android code signing.

Measurements of the TLS certificate ecosystem. The introduction of network scanners such as ZMap [45], which can scan the entire IPv4 address space, enabled the studies on the TLS certificates and HTTPS ecosystems. For example, two measurement studies [10, 11] have investigated the impact of *Heartbleed* and the reaction to it. These studies revealed that the majority of the compromised certificates were not revoked even after the incident. There have been several additional research on the TLS certificate revocation. Liu et al. [12] found that a large number of revoked certificates are still in use. Kumar et al. [46] measured the mismanagement of OCSP and CRLs in the Web's PKI.

Code signing abuse. Prior works studied the abuse in code signing in Windows Authenticode and Android. *Sophos* reported their findings regarding the signed malicious Windows PE files collected from 2008 to 2010 [47]. Whereas we utilize a more recent data, and we also propose a threat model that accentuates the weaknesses in the code signing PKI. Kotzias et al. and Alrawi et al [48, 49]. investigated the Windows PE samples and found that most of the signed samples are potentially unwanted programs (PUP), while we focus on the signed malware. The measurement in Android code signing [50] reported security threats such as the signature-based permissions caused by the key reuse.

2.2 Malware Delivery and Detection

Behavior of downloaders. In Chapter 3.1 and 4.1, we analyze the behavior of malicious downloaders *in the wild*. Moreover, we analyze the properties of the malicious and benign downloader graphs to determine the features that are a strong indication of malicious behavior. We then train a classifier for malware detection using information extracted from the downloader graphs. Whereas, prior research on the behavior of downloaders [2, 35, 51] focused on executing malware droppers in *lab environments*. They typically focused for short periods of time (e.g., up to one hour), in order to observe the communication protocols they employ and to *milk* their server-side infrastructures (i.e., to download the payloads). For example, Caballero et al. [2] described pay-per-install infrastructures, which distributed malware on behalf of their affiliates. They analyzed their structure and business model

by milking the server-side infrastructures. Follow-up work reported a longer investigation showing that downloaders might remain active for over 2 years [35], and characterized several malware distribution operations [51]. However, less attention has been given to the client side behavior or downloaders.

Malware distribution. In Chapter 4, we conduct two studies on detecting malware distribution. We discuss the prior works on malware distribution and compare with our work.

We can find various prior works on malware distribution. These studies focused on characterizing the properties of the malware delivery and the distribution infrastructure. Provos et al. [1] described drive-by-download attacks caused by exploiting vulnerabilities in web browsers. They analyzed the tree-like structure of redirection paths leading to the main malware distribution sites. Additionally, they identified several mechanisms to inject malicious web content into popular pages (e.g., comments on blogs or syndication in Ad serving networks). Li et al. [19] conducted a study on URL redirection graphs. By analyzing those graphs, they identified a set of topologically dedicated malicious hosts (e.g., Traffic Distribution Systems). Perdisci et al. [52] analyzed the malicious HTTP traffic traces and studied the structural similarities between them. Xu et al. [20] fingerprinted several types of malicious servers, such as exploit servers (for malware distribution through drive-by downloads), C&C servers (for command and control), redirection servers (anonymity), and payment servers (for monetization).

Several proposed techniques exist for detecting malware download events by

leveraging these studies. Cova et al. [53] analyzed the rogue anti-virus campaigns by investigating the malicious domains involved in the distribution. Based on the analysis, they further presented an attack attribution method with feature-based clustering. Vadrevu et al. [54] and Invernizzi et al. [55] introduced systems for detecting malware download activities in the network traffic. Zhang et al. [56] employed an unsupervised technique to identify the group of related servers that are likely to be involved in the same malware campaign.

Our studies differ from these works as follows. First, we focus on the client side of malware distribution networks and reconstruct the downloaded by relationship. Second, we detect malware delivery campaigns by utilizing an unsupervised technique based on graph patterns. Lastly, while prior works generally rely on the malicious domains for attribution, we exploit the code signing behavior of the downloaders.

Graph-based attack detection. We then review the works where the attacks are detected by utilizing graphs. Chau et al. [57] and Tamersoy et al. [58] proposed techniques for assigning *reputation* scores to the nodes (i.e., executable files). They set the reputation by performing belief propagation on the host-file graph. Graph analytics have also been employed for analyzing function call graphs in malware samples [59–61], spamming operations [62, 63], and vote gaming attacks [64]. The Oddball approach [65] extracting features from the k-hop neighborhoods of every node. They analyze the patterns in the features and investigate the outliers.

We also propose graphs based approaches for detecting malware distribution

in Chapter 4. In contrast to these prior works, we analyze the malware delivery activities on the client side. Explicitly, in the downloader graph analytics, we construct graphs that encode semantic relationship between files (i.e., a file downloads another file). Such graphs can provide more profound insights into malware distribution activities. Similarly, we maintain a graph that captures the *accessed by* relationship between the downloader and domain. We apply a lockstep behavior detection to identify the clusters of downloaders and domains involved in the same campaign.

2.3 Malicious Campaigns and Detection

In Chapter 4.2, we conduct a study on detecting silent delivery campaigns. Here we present some relevant prior works on detecting malicious campaigns.

Spam campaigns. Spam is one of the well-studied types of attack, where we can find campaigns. We observe several studies on measuring and analyzing spam across different domains, such as email [66, 67] and social media [68, 69]. Prior work on social media campaigns developed machine learning techniques to characterize these campaigns, which are often domain specific features that cannot transfer outside the domain. Whereas, the lockstep detection algorithm utilized in our study has broad applicability.

Detecting coordinated activities. The core of our detection system is the frequent pattern tree based lockstep detection. We discuss prior systems for detection coordinated activities, which utilize different methods. CopyCatch [70] detects

locksteps by analyzing the connectivity between users and pages through the *likes* relationship. We give a comparison with our system in Chapter 4.2.5.1 and the limitations of this algorithm when applied to the detection of silent delivery campaigns. Besides the CopyCatch, most of the works utilize outlier detection to identify suspicious nodes [71] or suspicious edges [72]. SynchroTrap [73] proposes a malicious account detection system in the context of social networks to uncover malicious accounts and campaigns. They cluster users based on the Jaccard similarity of their behavior. Whereas, our system detects malicious campaigns which correspond to near bipartite cores.

2.4 Transparency in Security

PKI. The current web PKI relies on the trust in the CA (certificate authorities), which is vulnerable to man in the middle (MITM) attacks, i.e., compromised CAs. Moreover, the assumption of honesty does not scale up very well because it is hard for a user to check the trustfulness of the hundreds of CAs. In order to address these issues, several prior works propose publicizing the issued certificates as a solution. Sovereign key [74] is a long-term key that is used for cross-signing the TLS key. These keys are kept in an append-only data structure called the timeline server, and the clients have to check if the domain has a sovereign key; if it has a sovereign key, then the client has to verify if the public key is cross-signed correctly. Google's Certificate Transparency [75] aims to make the certificate issuance transparent. Certificate Transparency also uses the public append-only log, which is implemented using a

Merkle tree. By making the certificate issuance transparent, i.e., record all certificate chains seen in the wild to the log, the misissued or maliciously issued certificates can be detected efficiently.

The Merkle tree data structure used in the previous studies were not suitable for managing revocation. However, there exist several pieces of research that take into account certificate revocation. The google revocation transparency [76] proposed the sparse Merkle tree as the data structure for revocation. The accountable key infrastructure (AKI) [77] suggest a new public-key validation infrastructure, which uses Integrity Log Server where an Integrity Tree keeps all the registered certificates transparently. The keys are in lexicographical order in the tree and only the currently valid certificates are kept, which is a design decision that takes into account revocation checking. Attack Resilient Public-Key Infrastructure (ARPKI) [78], which is an improvement of AKI, also use the similar transparency data structure. Certificate Issuance and Revocation Transparency (CIRT) [79] use two Merkle trees, one in chronological order and one in lexicographical order. The former is used to record all the issued certificates as in the CT, and the latter is used to record the currently valid certificates, which provides an efficient way to check revocation. Distributed Transparent Key Infrastructure (DTKI) [80] is an extension of the CIRT with more considerations on the distributed setting.

Binary. There has been a couple of proposals for making binaries transparent. CT for binary codes [81] is an extension of Certificate Transparency, accepting the binary codes to the log. Repository of signed code (ROSCO) [82] propose to publicize the

signed by the relationship between a binary and the certificate that is used to sign the binary. The Mozilla binary transparency [83] propose to utilize the current Certificate Transparency for publicizing the integrity of the software update.

Cryptocurrency. The heart of cryptocurrency, e.g., BitCoin, is in the transparency in transactions. Each of the blocks that consists of the blockchain contains the set of transactions in an append-only log.

Chapter 3: Understanding Malware Distribution

3.1 Dynamics of Malware Delivery

In this chapter, we provide insights into how malware is delivered to end hosts.

Web browsers, updaters and instant messengers. We initiate the investigation by focusing on the programs responsible for most downloads in our download activity dataset. The well-known programs, which appear in the NSRL dataset and considered as benign, place on the top of the list. We identify that these programs correspond to the browsers, software updates, and Skype (an instant messaging program), by looking at their publisher name in their code-signing certificate and the product name on their PE metadata.

Benign Programs Dropping Malware. Then, we measure which programs are involved in delivering the malicious droppers. For this analysis, we considered a binary as malicious if more than 30% of the AV engines flagged it as malicious on the VirusTotal reports. We identified that 94.8% of the malicious droppers are downloaded using the top-3 Web browsers.

We observe multiple benign programs that drop malware besides the browsers, listed in Table 3.1. These include three Windows process, `EXPLORER.EXE`, `CSRSS.EXE`

Downloader file name	Payloads
CSRSS.EXE	14801
EXPLORER.EXE	1717
JAVA.EXE	892
DAP.EXE	749
OPERAUPGRADER.EXE	584
SVCHOST.EXE	547
WMPLAYER.EXE	247
IDMAN.EXE	237
CBSIDLM-CBSI145-SUBTITLESSYNCH-ORG-10445104.EXE	209
MODELMANAGERSTANDALONE.EXE	187
KMPLAYER.EXE	140
JAVAW.EXE	105

Table 3.1: Top benign downloaders dropping malware.

and `SVCHOST.EXE`. In the case of `CSRSS.EXE`, the ten most frequently downloaded executables are adware programs (detected by several AV products); 8 out of 10 are from Mindspark Interactive Network. For `EXPLORER.EXE`, the ten most frequently downloaded executables are adware programs from Conduit, Mindspark, Funweb, and Somoto. In the case of `SVCHOST.EXE` most of top 10 payloads are generic trojan droppers. Executables downloaded from `JAVA.EXE` are specific trojans (Zlob, Genome, Qbot, Zbot, Mufanom, Cycbot, Gbot and FakeAV), and a hack tool (passview). `DAP.EXE` and `IDMAN.EXE` are downloader managers, and the top 5 executables they drop are products signed by Mindspark. For `JAVAW.EXE`, Bitcoin mining executables made the top of the list. Another Java related process, `MODELMANAGERSTANDALONE.EXE`, is also used for dropping trojans. In many of these cases it is difficult to pinpoint the exact program that is responsible for delivering malware; for example, `SVCHOST.EXE` is a process that can contain a variety of Windows services, while `JAVA.EXE` is an interpreter that runs Java programs.

The result suggests the fact that various types of malware often infiltrate benign software ecosystems to remain undetected. Which eventually makes the software delivery ecosystem noisy. It may imply that the resulting download graph may be a mixed with both benign and malicious. This finding motivates our decision of using the sub-graph rooted in each downloader (i.e., influence graph), instead of the full graph, in the study of detecting malware distribution based on graphs in Chapter 4.1.

Signed Malicious Downloaders. We share another surprising finding from the analysis of download activities. Almost 22.4% of the malicious downloaders among the total 67,609 malicious downloaders have a valid digital signature. Including the ones with invalid signatures, the ratio goes up to 55.5%. Softonic International, Amonetize Ltd, InstallX, Mindspark Interactive Network, and SecureInstall rank as the top-5 publishers with downloaders having valid signatures. These downloaders are known to distribute third-party software. Especially, Amonetize is known to be a pay-per-install provider [84]. The purpose of this behavior seems to be an effort of evading detection, mimicking the benign software distribution, which is usually carrying digital signatures. We discuss the problem in details in the next chapter.

3.2 Measuring the Code Signing Abuse

Goal. The goal in this work is to measure breach-of-trust in the Windows code signing PKI. An adversary can steal the private key from the legitimate software developer or fool the verification process of the CA to issue a certificate with a false

identity or exploit the vulnerability in the code-signing checking on the client side. We aim to perform a real-world measurement of the prevalence of these threats and the mechanisms for breaching the trust.

Non-goals. The non-goals include fully characterizing the code signing ecosystems, analyzing certificates issued legitimately to real publishers, or developing new techniques for authenticating executable programs.

We start the discussion by a description of the threat model (Chapter 3.2.1), followed by the challenges of measuring the problem in Chapter 3.2.2. We then present the methods of the measurement (Chapter 3.2.3) and the abuse detection algorithm that we developed to overcome those challenges (Chapter 3.2.4). Finally, we discuss the findings from the measurement in Chapter 3.2.5.

3.2.1 Threat Model

We can list the goal of an adversary as (1) distribute and install malware on end-user machines, and (2) conceal its identity. The adversary with these goals will digitally sign malware, which may help to evade AV detection and exploit the client side security policies such as the User Account Control (UAC) and Windows SmartScreen. For the second goal, the adversary shall not use its true identity to acquire a code-signing certificate. The prior works on PUP found that PUP publishers often use their actual identity to get a legitimate code-signing certificate [43, 48, 85, 86]. Instead, our adversary, who is interested in delivering malware instead of PUP, tries to exploit the weaknesses in the code-signing PKI to sign mal-

ware. These weaknesses fall into three categories: inadequate client-side protections, publisher-side key mismanagement, and CA-side verification failures.

Inadequate client-side protections. Windows operating systems can verify code-signing signatures, and when the program needs elevated privilege, the UAC presents the information of the certificate to the user. However, if the user allows granting the privilege, no further enforcement is done from the OS. Such policy may be a problem when the user accidentally grants privilege to a suspicious publisher or a program with an invalid certificate. As a supplement, Anti-virus engines may block these suspicious programs. However, the certificate verification may have been implemented differently, which may result in varied interpretations of the validity of the certificate among AV engines.

Publisher-side key mismanagement. The private key of the code-signing certificate needs to be secure. However, if the adversary penetrates the development machines involved in the signing process, it can (1) steal the private key that corresponds to the publisher's certificate or (2) inject their malware to those machines to sign it.

We define the former as a certificate is *stolen*. There are several reported cases of potentially stolen private keys. Stuxnet and Duqu 2.0 [5, 8] are advanced pieces of malware that carried valid digital signatures. Certificates belonging to legitimate companies located in Taiwan were utilized to generate these digital signatures.

As the second case, adversaries may also *infect developer machines* and sign their code without the victim's consent. The community first encountered the

W32/Induc.A [87] in 2009. The malware infects files required by the *Delphi* compilers. In consequence, the compiler involved in the development process automatically signed the malicious code with a valid certificate, and the signed malware got distributed as a legitimate software package.

CA-side verification failures. The chain of trust starts from CAs properly verifying the publisher's identity before issuing a code-signing certificate under the provided identity. However, if the CA fails the verification, it results in the misissuance of the certificate to the false identities.

We define two types of methods the adversaries take to exploit the CA's verification process. The first case is called the *Identity theft*, which is a masquerade attack pretending as a reputable company. A successful attack will grant a code-signing certificate with that company's identity to the attacker. In January 2001, there was a report of a successful masquerade attack, which resulted in Verisign issuing two code signing certificates to an adversary who claimed to be an employee of Microsoft [88]. However, the attack is not limited to large software publishers. Perhaps it may be easier to target companies that are not involved in software development, since the victims may not expect such abuse of their identity.

Instead of stealing an existing identity, the adversary may take advantage of *Shell companies*. It may be easier for the adversaries regarding the preparation of the identity materials because they have full control of the information used for registering the company. A downside of this method would be in a low reputation, both in the defense mechanisms like SmartScreen and the users. However, the

adversary nevertheless gets a malware with valid digital signatures, which may give a better chance of delivery.

3.2.2 Challenges for Measuring Code Signing Abuse

The challenges in measuring the abuse in code signing certificate are collecting the digitally signed binaries and distinguishing between the abuse cases. While in TLS, it is now possible to get a comprehensive list of certificates by scanning the IP spaces, there exists no easy way to collect a large enough corpus of certificates used in the wild. Even for those that are collected, capturing the binaries carrying the certificate is another difficulty. Moreover, it is challenging to classify the abused certificates to its abuse types, e.g., stolen, identity theft, shell company, due to the lack of ground truth about the type of abuse. Only a few cases have been reported and identified as a specific type of abuse, for instance, the Duqu and Stuxnet.

3.2.3 Measurement Methods

We take two steps in large to identify the breaches of trust in the Windows code signing PKI. Initially, we collect the certificates used to sign malware in the wild. Then, we classify the certificates by using an algorithm for distinguishing among the three types of threats introduced in Chapter 3.2.1.

3.2.3.1 Data Sources

We identify the hashes of signed malware samples, and the corresponding publishers and CAs, from Symantec's WINE dataset, we collect detailed certificate information from VirusTotal, and we assess the publishers using OpenCorporates and HerdProtect.

Worldwide Intelligence Network Environment (WINE). We infer the download activities on 5 million end-hosts described in Chapter 1.3.1. From the WINE data set, we query the (1) *anti-virus (AV) telemetry* and (2) *binary reputation*. We extract about 70,293,533 unique hashes from the AV telemetry.

There are 587,992,001 unique binaries from the binary reputation dataset. However, we cannot directly infer to the WINE dataset to distinguish between files with different certificates in case they share the same publisher. It is due to the high granularity of the information given: WINE does not provide more detailed information about the certificate such as its serial number.

VirusTotal. To get a much fine granularity of information regarding the code-signing certificates, we use the reports from VirusTotal in Chapter 1.3.2.

OpenCorporates. OpenCorporates [89] maintains the largest open database of businesses around the world, providing information on over 100 million companies. We use this database to determine if the publisher is a legitimate company.

HerdProtect. We utilize another source providing information about the publisher, using *HerdProtect* [90]. We collect the following information for each publisher in the dataset: whether the publisher is a known PUP distributor, the profile

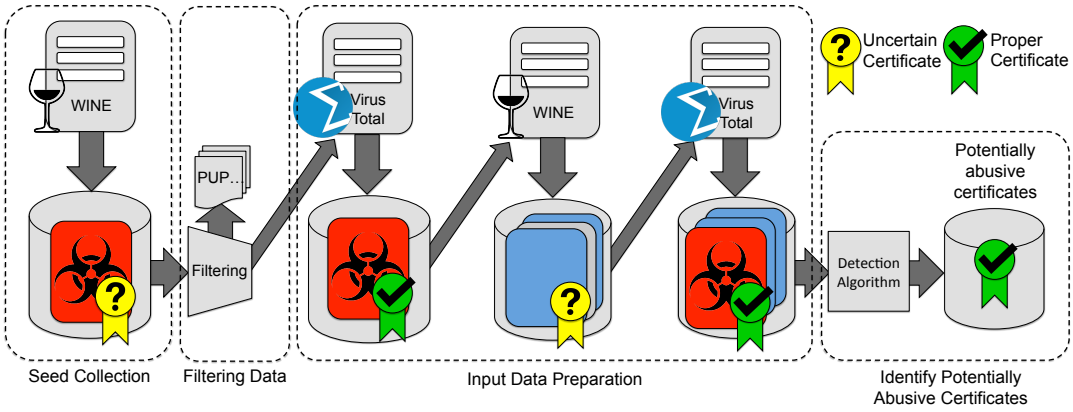

Figure 3.1: Data analysis pipeline.

of the company (e.g., location, business type), and the reported list of certificates associated with the publisher.

3.2.3.2 Binary Labeling

Malware. In this chapter, we describe how we label a binary as either malware or potentially unwanted programs (PUP) or benign. Initially, we define c_{mal}, which indicates the number of anti-virus products that flagged the binary as malicious in the VirusTotal report. We say a binary is suspicious (i.e., either malware or PUP), if $c_{mal} \geq 20$. To discriminate PUP from malware, we inspect the detection names given by these AV products and compute r_{pup}, which is the ratio of PUP detection names [43]. We consider malware to have both $c_{mal} \geq 20$ and $r_{pup} \leq 10\%$.

Benign programs. To consider a binary as benign, it has to be $c_{mal} = 0$ and has a valid signature.

3.2.3.3 System Overview

Pipeline overview. By utilizing the data sources listed above, we design a data collection and analysis pipeline as depicted in Figure 3.1. The pipeline is a sequence of four processes: seed collection, data filtration, input data preparation and identifying potentially abusive certificates.

- *Seed collection.* This step of the pipeline creates a list of SHA256 file hashes that are likely malicious and has a digital signature. We start by collecting a set of malware samples reported in AV telemetry data set in WINE, in their SHA256 hash value. To extend the dataset, we supplement the list with additional hashes of known malicious binaries from an external source (Symantec). To narrow them down to the ones that are likely digitally signed, we join the list of hashes to the binary reputation schema and get the high-level information of the certificate.

- *Filtering data.* What we need is the digitally signed malware. However, the list from the previous step may still contain Potentially Unwanted Programs (PUPs) and false positives. We filter out PUPs through a conservative process. We first filter out the previously known PUP publishers in prior work [43, 48, 85, 86] and the publishers listed in *HerdProtect* as a PUP provider. Then, we pick ten binary samples for each publisher and filter out the publisher if at least one of them is determined to be a PUP. To determine whether a binary is a PUP or not, we follow the method described in Chapter 3.2.3.2. For the

publishers with a large number of binaries, such as Microsoft or Anti-virus companies, we sample the files that have a bad reputation in the Symantec ground truth.

- *Input data preparation.* We now query VirusTotal to get detailed information on the code signing certificate and the AV detection results of these filtered hashes. We consider a binary as a signed malware if it satisfies the two condition: (1) it is duly signed, and (2) it is labeled as malware according to the rules in Chapter 3.2.3.2. At this stage, we encountered several malicious samples with malformed signatures. These correspond to the evidence of inadequate client-side protections, which we analyze in Chapter 3.2.5, However, we do not take them to the next of the pipeline.

  Then we use the <publisher, CA> pair of the signed malware, to query the binary reputation schema in WINE. The query returns us a set of binaries that are likely benign and share the same certificate with the signed malware. These binaries are also queried to VirusTotal to get additional information. Then, we determine benign files as described in Chapter 3.2.3.2.

- *Identify potentially abusive certificates.* As the final step of the pipeline, we classify them to the abuse type using the algorithm described in Chapter 3.2.4.

3.2.4 Abuse Detection Algorithm

The proposed algorithm distinguish code signing certificates to its abuse cases. We group the signed binaries by their certificate. The groups may consist of either

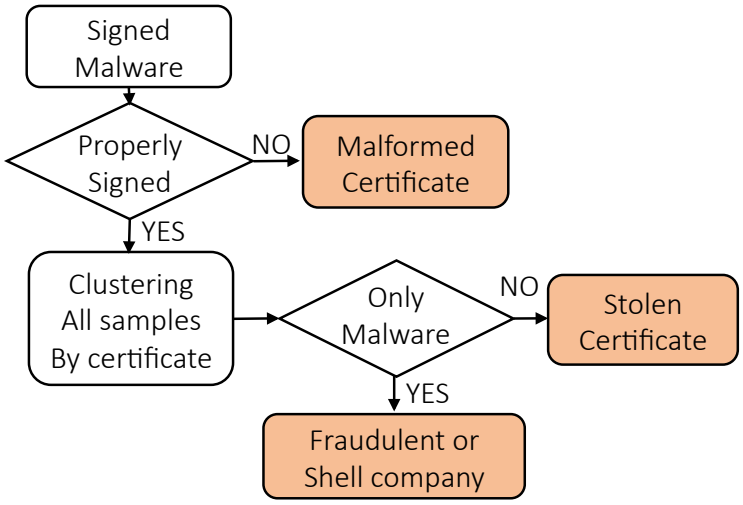

Figure 3.2: Flowchart of the abuse detection algorithm.

(1) only malware or (2) both malware and benign binaries. Since there are binaries that we were not able to label due to the lack of information, we consult additional data sources to extend the ground truth. We query *HerdProtect* for more samples signed with the same certificate. Moreover, we visit the publishers' website to get information about the publishers (e.g., legitimate company or involved in software development). With the additional information, we use the method illustrated in Figure 3.2 to detect the type of abuse.

Stolen certificates. As described in Chapter 3.2.1, in this case, the certificate is issued initially to the legitimate publisher and then leaked to the adversary. Intuitively, it is likely to see both benign and malicious programs in this case of abuse. We further obtain the trusted timestamps to reenact the timeline of the abuse.

Identity theft & shell companies. We assume for this case, where the malware writers have sole possession of the private key, they have no motivation of signing and distributing benign software, which will result in a group of only malicious

samples.

To distinguish between identity theft and shell companies, we use the information of the publisher acquired from *OpenCorporates* and *HerdProtect*. If the publisher appears in *OpenCorporates* and the company address in the X.509 certificate matches the information in the profile, we suspect that the company is a victim of the masquerade attack. If not listed, we assume it is a shell company.

Verification. To verify the results, we manually analyze the signing time-line of the binaries in the certificate, and we contact the publishers and CAs to confirm the findings.

3.2.5 Measurement Results

3.2.5.1 Summary of the Input Data

We provide a summary of essential numbers out of the analysis pipeline in Chapter 3.2.3.3. We have 153,853 samples in the seed set that are suspicious (i.e., not yet defined as malicious based on our labeling method but from AV telemetry) and has a full certificate chain. To apply the abuse detection algorithm from Chapter 3.2.4, we search for potentially benign samples signed with the certificates in the seed set. It gives us a total of 415,377 from VT.

We examine the validity of the certificates from the seed set referring to the *verified* message in the VT reports. VT verifies the certificate using the `sigcheck` tool provided by Microsoft [91]. Table 3.2 shows a breakdown of the result. The 325 samples are detected as malware in a total of 153,853 signed samples. Of these

Cert.	Desc. (Error code)	Total	Malware
Properly	Valid	130,053	109
	Revoked (0x800b010c)	4,276	43
	Expired (0x800b0101)	17,330	37
	Total	151,659	189
Malformed	Bad Digest (0x80096010)	1,880	101
	Others	81	0
	Parsing Error	233	35
	Total	2,194	136
Total		153,853	325

Table 3.2: Property of the certificates. Others include unverified time stamping certificates or signature, distrusted root authority, etc.

325 signed malware, 58.2% samples have valid digital signatures while 41.8% have malformed certificates. Most (74.3%) improperly signed malware results from bad digests. We provide a detailed explanation of these malformed signatures in the next chapter. Surprisingly, more than a half of all properly signed samples (57.7%) are still valid at the time of this work was done.

3.2.5.2 Malformed Digital Signatures

We encountered 101 samples which signature and the authentihash do not match. Such mismatch results in an Authenticode error code "0x80096010", which appears when a digital signature and a certificate is just copied from one file to another. There is no need for an adversary to acquire a private key to produce a sample with the malformed signature. Therefore, this does not correspond to a breach of trust in the publisher or the CA. However, the high prevalence of these signatures (31.1% of the signed malware), suggests the need for a further investigation of the intention: it may help malware bypass client-side protections. Therefore, we con-

duct experiments to see how the malformed signatures affect client-side protections provided by browsers, operating systems and anti-virus products.

Browser protections. We test the browser protections including Safe Browsing in Chrome and SmartScreen in Microsoft IE9. To prepare a test sample, we copied a legitimate certificate and signature to a benign and simple calculator, which does not ask for elevated privilege. We checked if the browsers block this sample at the time of download. Both Safe Browsing and SmartScreen blocked our sample. However, when the file extension is removed (`.exe`), the browsers do not block the download.

Operating system protections. We tested how windows react to an executable with a malformed signature. We attached a certificate to an installer that triggers UAC at execution time, regardless of where the file originated (i.e., from the Web or not). UAC displays a message saying that the file is from an unknown source, which does not differ from the unsigned binaries. Windows only warn the user and do not block the execution of the binary with malformed signature. Which means, no enforcement is done by the OS, once the user chooses to ignore the warnings.

Anti-virus protections. Now we check if malformed signatures affect the AV detection. We prepare five test samples, which are recently reported unsigned ransomware samples with high detection, i.e., 56–58 AV detection from VirusTotal. For the certificates to append, we prepared two certificates and signature from different publishers. To give a better chance for the AV engines, we set another criterion: the certificate and signature has been employed as a malformed signature in the wild.

nProtect	8	F-Prot	4	Symantec	2	Sophos	2
Tencent	8	CrowdStrike	4	TrendMicro-HouseCall	2	SentinelOne	2
Paloalto	8	ClamAV	4	Avira	2	VBA32	2
AegisLab	7	VIPRE	4	Microsoft	2	Zillya	1
TheHacker	6	AVware	4	Fortinet	2	Qihoo-360	1
CAT-QuickHeal	6	Ikarus	4	ViRobot	2	Kaspersky	1
Comodo	6	Bkav	3	K7GW	2	ZoneAlarm	1
Rising	5	TrendMicro	3	K7AntiVirus	2		
Cyren	4	Malwarebytes	2	NANO-Antivirus	2		

Table 3.3: Bogus digest detection (AV and the number of detection fail).

For each sample, we create two sample with a different malformed certificate, which results in total ten samples.

Then, we test the change in the detection of these ten samples compared to the detection before appending the certificate and signature. The result shows that this simple attack disturbs many anti-virus products from detecting the malware. Table 3.3 lists the 34 AVs fail to detect the same sample after appending the signature. The malformed signatures reduced the VirusTotal detection rate r_{mal} by 20.7% on average. We have reported the issue to the antivirus companies. One of them confirmed that their product fails to check the signatures properly and plans to fix the issue. A second company gave us a confirmation but did not provide details.

Summary. In summary, we explored three different protection mechanisms on the client side. The browser blocks the binaries with bogus certificates but has ways to evade, the OS provides the minimum level of protection with no enforcement, and many AVs have the incorrect implementation of Authenticode signature checks. The current state gives malware authors the opportunity to evade detection with a simple and inexpensive method.

Stolen		Identify Theft		Shell Company	
Issuer	Count	Issuer	Count	Issuer	Count
Thawte	27	Thawte	8	Wosign	2
VeriSign	24	Comodo	4	DigiCert	1
Comodo	8	VeriSign	4	USERTrust	1
USERTrust	2	eBiz Networks	3	GlobalSign	1
Certum	2	USERTrust	1		
Others	9	Others	2		
Total	72 (64.9%)	Total	22 (19.8%)	Total	5 (4.5%)

Table 3.4: Type of abuse and the top 5 frequent CAs.

3.2.5.3 Measuring the Abuse Factors

We utilize the algorithm from Section 3.2.4 to distinguish the 111 certificates by to their type of abuse.

Publisher-side key mismanagement. Among 111 certificates, 75 certificates are carried by both malware and benign software. Since these are likely to be compromised certificates, we examined if they are revoked. 13.3% of certificates are explicitly revoked. However, a more significant number of certificates (50, 66.7%) are still valid at the time of this study. To further distinguish the certificates, we manually investigated the signed malware samples.

- ***Stolen certificate.*** Most of the certificates (72) are stolen and used to sign malware. One of the certificates were carried by the *Stuxnet* malware, which is known to have been signed with a compromised certificate [5]. In our dataset, it has the *Realtek Semiconductor Corp.* certificate issued by Verisign. Also, an Australian department's private key is found, which is used for signing *autoit* malware.

- ***Infected developer machines.*** Three certificates are identified, which are carried by *W32/Induc.A* malware that infects only Delphi developer machines.

CA-side verification failure. The algorithm suggests that 27 certificates are issued to malicious producers due to this weakness. We perform a manual investigation to distinguish between identity theft and shell companies. We first check if the publisher is legitimate by searching for the publisher on the Internet or in *openCorporates*. 22 certificates masqueraded legitimate companies and 5 certificates exploited the verification using shell company information.

Revocation. We investigate the revocation practice in the field. By category, 15.3%, 40.9%, and 80.0% of the certificates were revoked for stolen, identity theft, and the shell company, respectively. Interestingly, the revocation rate was significantly less for the stolen certificates compared to the other abuse types. The observation of the low revocation rate motivates our next study on the effectiveness of the code signing certificate revocation.

3.3 Measuring the Code Signing Certificate Revocation

There are various reasons for certificate revocation, and it is unclear what an effective revocation process is. However, when the private code-signing key has been used to sign malware, it requires a prompt and precise revocation [28]. Therefore, this study focus on the effective revocation in case of the certificate abuse.

Goals. The goal of this study is to measure the problems in the revocation process systemically, (1) promptly discovering compromised certificates, (2) revoking the

compromised certificates effectively, and (3) disseminating the revocation information, and new threats introduced by these problems.

Non-goals. We do not aim a full characterization of (1) CA's internal infrastructure fully problems, (2) their internal revocation policies, and (3) the internal revocation checking policies of Windows.

We initiate the study by illustrating the research questions for the effective revocation process (Chapter 3.3.1), followed by the challenges of measuring the problem in Chapter 3.3.2. We then present the methods of the measurement (Chapter 3.3.3). The following chapters present the measurement results of each step of revocation starting from the discovery to the dissemination (Chapter 3.3.4–3.3.6).

3.3.1 Effectiveness of Revocation Process

In this chapter, we introduce four research questions that have to be answered to check the effectiveness of the revocation process in code-signing.

Q1. How many certificates are being used to sign malware? The revocation process starts with the discovery of compromised certificates. We initiate the investigation by estimating the magnitude of the current threat that should be the target of the revocation process. The estimation should also provide how much coverage our community has for the compromised certificates.

Q2. How prompt is the revocation process? Once a notification of abuse is received, CAs have to begin investigating the reports within 24 hours and revoke the compromised certificates and publish the revocation information within seven days

or get reasonable cause from the owner of the certificate to delay [28]. However, the requirements in code-signing [28] do not explicitly state, who is responsible for discovering the abused certificates. Such ambiguity may result in a delay between the initial evidence of compromise (t_d) and the revocation published date (t_p).

Q3. Are effective revocation dates set properly? Two strategies are used when setting the *effective revocation date* (t_r): *hard revocation* where $t_r = t_i$, and *soft revocation* where $t_i < t_r \leq t_e$. Hard revocation has the advantage that all malicious signed files are untrusted, but it lets benign files to become invalid as well. To mitigate the impact to the benign files, CAs and publishers can try soft revocation. If we set the precisely effective revocation date, it can make the malware invalid and save a large number of benign programs. However, if we fail to set the date correctly, then signed malware (i.e., malware signed before the date, $t_m < t_r$) may still exist and continue to be trusted as the example shown in Figure 1.1.

Q4. Is revocation information served properly? The last question is to determine if the revocation information reaches the users rightly and the users are protected even in the circumstances where they cannot retrieve the information of revocation. The code-signing requirement [28] states that we should consider a binary unsigned when it is not possible to check the revocation status. However, if the client-side platform (e.g., Windows), who check the validity of the certificate, applies a *soft-fail* policy in checking the revocation, the binary will be considered as valid when the information is not accessible. Provided that, if we fail to serve the revocation information correctly, then signed malware could remain valid even after

the revocation has happened.

3.3.2 Challenges for Measuring Code-signing Certificate Revocation

The two major challenge in this study is in the (1) visibility and (2) timing. As we discussed in the study of code signing abuse in Chapter 3.2, *Visibility* is an issue in the measurement of code-signing certificates. Because it is difficult to identify all the certificates that are actively being used in the wild. To address this challenge, we combine various data sources and try to obtain as wide a view of the ecosystem as possible. A unique challenge in studying revocation is in the *Timing*. This is due to the *revocation publication date* (t_p). In the CRL, we can only see the *effective revocation date* (t_r). To record the *revocation publication date* (t_p) of the revoked certificate, we have to monitor every update that happens in the CRL and the precise timing of the update.

3.3.3 Data Collection

In this chapter, we describe the methodology of the data collection that can overcome the challenge and how we measure the process of the code signing certificate revocation.

3.3.3.1 Fundamental Data (`D1` – `D2`)

The code signing certificates are the fundamental data for this research. They contain the revocation distribution points (CRLs and OCSP points) and other infor-

	Malsign	***Malcert***	***Symantec***	***WINE***	**Total***
PKCS #7	2,171	801,995	149,840	11,108	965,114
CS certs.**	2,106	1,121	145,411	1,137	145,582
CRL URLs	55	60	403	49	413
OCSP URLs	24	24	130	16	131

Table 3.5: Summary of the fundamental data. (*: total number of unique data, **: CS stands for code signing – some certificates have parsing errors.)

mation that are necessary to monitor the revocation process. This chapter presents how we collect the code signing certificates and the revocation information from them. Table 3.5 shows the breakdown of the fundamental data.

Code signing certificates (`D1`). As there exist no large corpus of code-signing certificates, we use multiple public datasets from prior research [40, 48, 49] and a proprietary repository of binary samples to find code signing certificates. We utilize the following datasets:

- *Malsign.* Kotzias et al. [48] evaluated signed malicious PE files, and they publicly released the 2,171 leaf code signing certificates used to sign the PE files.
- *Malcert.* Alrawi et al. [49] examined 3.3 million samples collected from a commercial feed of a private company, and they shared 801,995 signed PE samples. The reason for the substantial reduction from PKCS #7 to CS certs for Malcert in Table 3.5 is that most of the PKCS #7 files were duplicate code signing certificates used to sign binaries with different hashes.
- *Symantec data set.* Symantec has an internal repository of binary files, from which they extracted a sample of 149,840 PKCS #7 files for analysis.
- *Samples from WINE [40] and VirusTotal.* To get additional code signing certifi-

CA	Leaf Certificates	
Verisign	44,014	(30.23%)
Thawte	26,884	(18.47%)
Comodo	24,780	(17.02%)
GlobalSign	12,079	(8.30%)
Symantec	8,913	(6.12%)
DigiCert	8,300	(5.70%)
Go Daddy	7,376	(5.07%)
WoSign	3,796	(2.61%)
Certum	1,874	(1.29%)
StartCom	1,830	(1.26%)
Other	4,281	(2.94%)
Total	145,582	(100%)

Table 3.6: Top 10 code signing certificate authorities. The top 10 CAs account for 97% of the certificates in our data set (`D1`).

cates, we also select around 300 PE files for each CA from WINE (c.f., chapter 3.3.3.3) and download the samples from VirusTotal using the download API; we accumulate 11,108 PE samples. The details of VirusTotal are explained in Chapter 3.3.3.3.

A PKCS #7 [26] file consists of code signing certificate chains, TSA certificate chains, a signature, and a hash value of a PE file. We can extract the PKCS #7 file from each dataset, except for the *Malsign* data set that provides only leaf code signing certificates. We initially extract the leaf certificate from each PKCS #7. Then, we narrow down the data to only code signing certificates using the keyword of "Code Signing" in the *extendedKeyUsage* extension field. When we only take the unique leaf code signing certificates, we get 145,582 certificates that CAs legitimately issued. Table 3.6 shows the number of code signing certificates for the top-ten most popular CAs in the dataset (`D1`).

We utilize the D1 dataset for investigating (1) the certificates without CRL and OCSP (Section 3.3.6.2), (2) the inconsistent responses from CRLs and OCSP (Section 3.3.6.2), and (3) the unknown or unauthorized responses from OCSP (Section 3.3.6.2).

Revocation information (D2). The CRLs and OCSP points (URLs) can be found at *CRLDistributionPoints* and *AuthorityInfoAccess* extensions respectively. We extract the CRLs and OCSP points from the unique leaf code signing certificates maintained in the previous dataset. A majority of the certificates (137,027, 94.1%) contain both CRL and OCSP point; the remainder are 7,794 (5.3%) that has only CRL and 98 (0.06%) certificates only OCSP points are specified. We observe a total of 413 unique CRLs, however, to filter out the CRLs with mixed usage, we manually search *Censys.io* for each CRL and filter out CRLs used for other purposes. This results in 215 CRLs that are dedicated in code-signing. We observed 131 unique OCSP points.

We utilize the D2 dataset to examine the problems in (1) the effective revocation date setting (Section 3.3.5), (2) the transient certificates in CRLs (Section 3.3.6.2), and (3) the no longer updated CRLs (Section 3.3.6.2).

3.3.3.2 Revocation Publication Date List (D3)

It is challenging to obtain the date when the revocation is posted since the CRL does not contain the information. Therefore, we devise a system, called *revocation publication date collection system* that monitors the revoked serial numbers once a

day from the collected CRL dataset (D2) in order to record the *revocation publication date (t_p)* when the CRL or OCSP servers post the certificate. During the study of Apr. 16th, 2017 to Sept. 10th, 2017, we observed 2,617 different certificates added to the CRLs. This D3 data set is used to examine the revocation delay ($t_p - t_d$)(Section 3.3.4.2).

3.3.3.3 Binary Sample Information (D4 – D6)

In order to answer some of the research questions, it is necessary to have not only the code-signing certificates but also the binary that carry them. One example could be the effective revocation date setting. Whether the date is set correctly or not can only be answered by looking at the signed binaries. Specifically, we need to see the signing date of the malware that are carrying these certificates. Therefore, we collect information about the signed binaries from three data sets: WINE, Symantec, and VirusTotal.

Worldwide Intelligence Network Environment (WINE) (D4). WINE, as described in Chapter 1.3.1, provide multiple telemetry data from users around the world. We find the necessary information for this study from the binary reputation data that contains metadata of binary files seen on endpoints. WINE only provides the high-level information of the certificate and detailed information of the certificate (e.g., a serial number of the certificate, CRL) is not provided. It does not have the actual binary as well. This D4 data set is used to examine the problems in revocation date setting (Section 3.3.5).

Symantec metadata telemetry (D5). Besides the WINE dataset, Symantec provided additional telemetry data that correspond to the revoked certificates observed by the *revocation publication date collection system*. The data contains the metadata of the binaries signed by the 2,617 revoked code signing certificates. The information comes from a more recent period (from Jan. 1st, 2016 to Sept. 10th, 2017) and consists of the serial number of the signing certificates, the SHA256 hash of the binary, the first seen timestamp. It enables us to obtain information about the revoked certificates we observed (D3) during the time of this study. In addition to the provided metadata, Symantec also shared us the ground truth for identifying malware among these signed binaries. With the ground truth, we can identify the abused certificates and the signed malware. This D5 data set is used to estimate malware signing certificates in the wild (Section 3.3.4.1), and to examine the revocation delay (Section 3.3.4.2).

VirusTotal (D6). We utilize the VirusTotal [41] private API to find additional information that we need in some part of the study. VirusTotal provided us three APIs for the measurement, including the report-API mentioned in Chapter 1.3.2.

Besides the report-API, we utilize the VirusTotal Hunting [92]. It allows users to apply rule-based matching (YARA [93] rule) on the incoming submissions, which can help researchers find a specific type of malware. We write a YARA rule that targets a binary both signed and flagged by at least 10 AV engines. From each collected submissions, we extract the SHA256 hash of the binary, the first submission date, and the serial number of the leaf code signing certificate. The data collection

began on Apr. 18th, 2017. The extracted data set is used in the estimation of malware signing certificates (Section 3.3.4.1), and to examine the revocation delay (Section 3.3.4.2).

We also use the VirusTotal download API to download the actual binary of a given hash when necessary (e.g., to collect the certificate to extract the CRL/OCSP information).

3.3.3.4 CRL/OCSP Reachability History (D7)

To see if the CAs are correctly serving the revocation information, we monitor the following information.

CRL reachability history. We check the reachability of the CRLs daily from Aug. 10th, 2017 to Sept. 10th, 2017. In case a CRL is unreachable, we record the timestamp and the reason of failure to the log. This D7 data set is used for measuring the unreachability of CRLs (Section 3.3.6.1).

OCSP reachability history. We also develop an OCSP reachability checker. The checker tests the reachability of each OCSPs every 30 minutes. It does not simply ping the domain, but it queries each OCSP points with the certificates that contain the OCSP point over the OCSP protocol using *Openssl*. If the OCSP is not reachable, we log the timestamp and the reasons for failure. It has been running with 131 unique OCSP points from Aug. 10th, 2017 to Sept. 10th, 2017. This D7 data set is used for measuring the unreachability of OCSP points (Section 3.3.6.1).

3.3.4 Discovery of Potentially Compromised Certificates

We begin the measurement by investigating the first step of the revocation: discovering the potentially compromised certificate. Here, we try to answer the two questions: *Q1. How many certificates are used to sign malware in the wild?* and *Q2. After the discovery of signed malware, how promptly is the corresponding certificate revoked?*.

3.3.4.1 Estimation of the Abused Certificates

It is challenging to understand how much coverage the community has in discovering the certificates used in malware since there exists no official repository for code signing certificates and the signed binaries. We address this problem by employing an estimation technique called the capture-recapture analysis [94].

Capture-recapture population estimation. This technique is widely utilized for measuring wildlife populations. The goal of capture-recapture is to estimate the size N of a population that is hard to observe entirety. In our case, N is the number of abused certificates. In order to make an estimate, we have to draw two separate samples from the population. The first sampling results in the capture of n_1 subjects. These subjects are marked and released in the wild. The second sampling results in the capture of n_2 subjects, among which p has been re-captured. In other words, p is the size of the intersection of the two samples. A population

estimation $\hat{N}$ can then be made as:

$$\hat{N} = \frac{n_1 n_2}{p} \tag{3.1}$$

Assumptions and interpretation. To estimate the total number of potentially compromised certificates, we apply the capture-recapture technique to the malware signing certificates from two different data sets: Symantec telemetry (`D5`, n_1) and VirusTotal (`D6`, n_2). We consider that each data set is a sample of the total population potentially compromised certificates.

However, capture-recapture makes three assumptions about the population and the sampling process that we may have violated in this case. First, the subjects in the population should be *homogeneous*, which means they have an equal chance of being captured. However, the certificate population does not meet this property. For example, a certificate used on a common software will appear much frequently in the dataset. Second, the samples drawn should be *independent* to each other. However, we know that security companies often share malware feeds with each other, which raises the probability of recapture for the potentially compromised certificates captured in the first sample. Third, the population should be *closed*. In other words, the size of the population should not fluctuate due to the birth and death of its members during the investigation. However, this property does not hold either, since the certificate issuance and revocation happens daily.

We take several mitigation strategies to address these issues. First, we estimate $\hat{N}$ separately for each day to address the last issue. We consider the birth of the

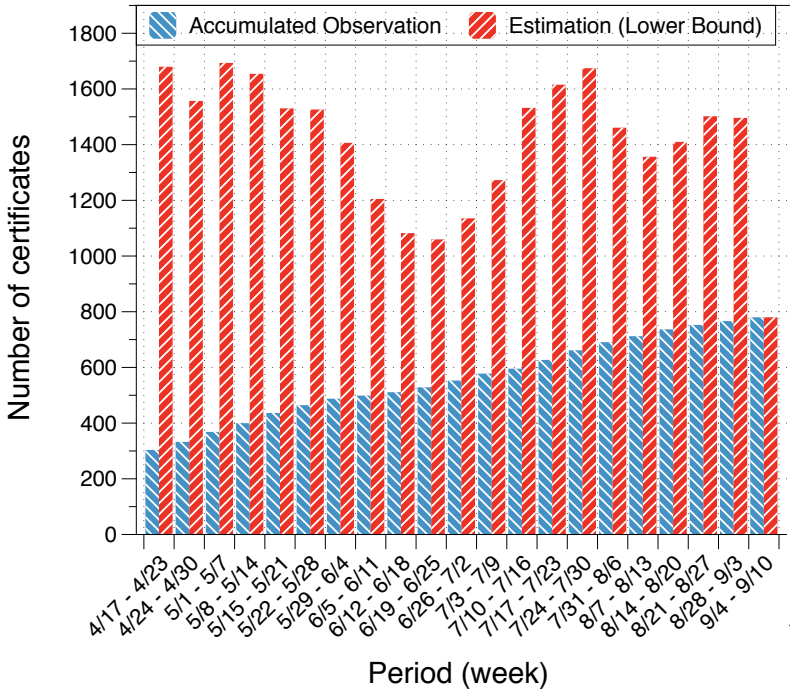

Figure 3.3: Trend in malware signing certificates (capture-recapture estimation as red and observed number as blue) over time.

certificate either at the first seen timestamp in the Symantec telemetry or the first submission date for VirusTotal. The certificate is considered to be dead at its revocation publication date (t_p). The population of interest is approximately closed within each day because CRLs are updated daily. The other two violations result in an underestimation for the actual population of potentially compromised certificates. For example, the certificates with low prevalence may not occur in either sample population. Similarly, dependencies between the two data sets would lead to an increase of the intersection p. The former leads to the decrease in n_1 or n_2, and the latter may increase p, which decreases the size of the estimation. Therefore, we interpret the estimation as the *lower bound* of the actual population.

Results. Figure 3.3 shows the average of our daily estimations $\hat{N}$, for each week dur-

ing our measurement period. We also compare these estimations with the number of potentially compromised certificates that we observe. The observed population is the union of the sets of certificates observed daily from the Symantec telemetry (D5) and VirusTotal (D6). Excluding the last week (9/4–9/10), the result of the estimation shows that at least 1,004–1,786 code signing certificates were used to sign malware in the wild and had not been revoked by the date of the estimation, which is 2.74× larger than the observed number of certificates on average. Such results suggest that even a major security company like Symantec and an information aggregator like VirusTotal may not observe a large portion of the potentially compromised certificates.

3.3.4.2 Revocation Delay

The remaining question in this step of the revocation process is *Q2. After the discovery of signed malware, how promptly is the corresponding certificate revoked?*.

We need two dates to determine the revocation delay ($t_p - t_d$): (1) the revocation publication date (t_p) and (2) the discovery date of the signed malware (t_d). The revocation publication date is collected by the *revocation publication date collection system* (D3) for 2,617 code signing certificates between Apr. 16th, 2017 and Sept. 10th, 2017. To identify t_d, we take the following steps. We begin by collecting the hashes for the binaries that carry the revoked certificates (D3) from the Symantec metadata telemetry (D5). We were able to retrieve 468 (17.9%) revoked certificates and 146,286 hashes by this step. Then, we utilize VirusTotal (D6) to get an analysis

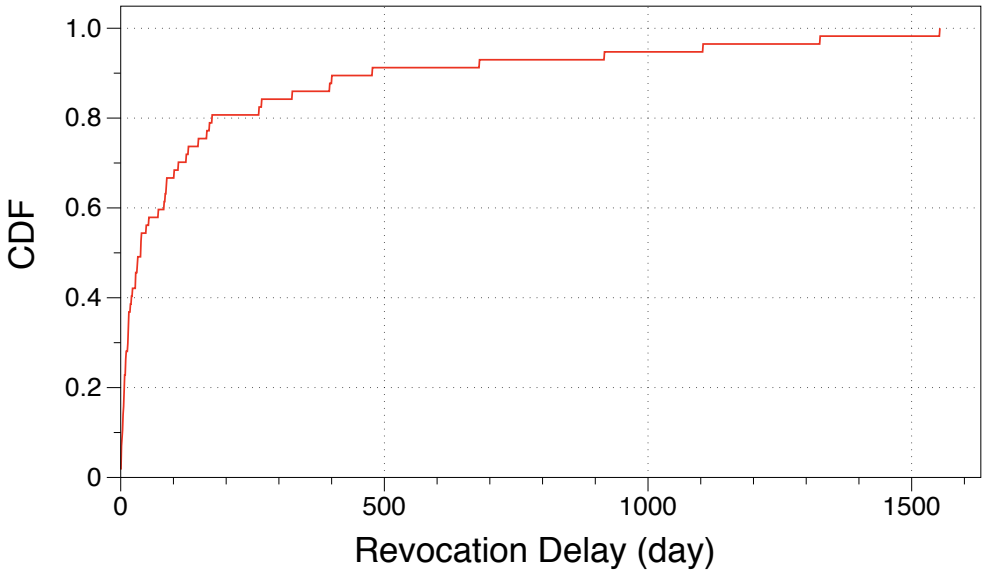

Figure 3.4: Revocation delays between the dates on which the malware signed with compromised certificates and the dates on which CAs revoke the compromised certificate.

report of the binary. As a result, we get additional information for 19,053 unique samples and 254 unique certificates used to sign the samples. For each certificate, we use the earliest detection date of a signed malware sample as the discovery date (t_d).

Results. Figure 3.4 shows a cumulative distribution of the revocation delay. We observe the average delay of 171.4 days (5.6 months). The result implies that users remain exposed to the signed malware over five months, on average, even after the community discovers the malware.

3.3.5 Problems in Effective Revocation Date Setting

Once discovering the potentially compromised certificates, the CA must determine a *effective revocation date*. In this chapter, we explore the answer to the

	$< t_i$	$= t_i$	$\leq t_e$	$> t_e$	Total
Comodo	0	426	1,437	17	1,880
Thawte	0	74	1,055	39	1,168
Go Daddy	2	14	672	18	706
Verisign	2	59	430	51	542
Digicert	1	161	323	3	488
Starfield	0	3	153	2	158
Symantec	0	33	89	1	123
Wosign	0	57	17	0	74
Startcom	0	0	47	0	47
Certum	0	1	9	0	10
Other	0	96	117	1	214
Total	5	924	4,349	132	5,410

Table 3.7: Effective revocation date setting policy for top 10 CAs (t_i: issue date, t_e: expiration date).

research question: *Q3. Are effective revocation dates set properly?*. As described in Chapter 3.3.1, CAs must set revocation dates when revoking the certificates that they have issued. CAs can set t_r (effective revocation date) to any date between t_i (issue date) and t_e (expiration date). As the trust in a signed binary relies on the effective revocation date, a general strategy a CA tries is to set t_r (effective revocation date) close to the oldest date on which the certificate signed malware ($min(t_m)$). However, if the t_r is set in a wrong date, i.e., after some t_m, then it can cause security problems.

Ineffective revocation date setting. We observe 5,410 (3.7% out of 145,582 certificates) explicitly revoked certificates. Then we investigate the effective revocation date setting for these revoked certificates. Table 3.7 shows the breakdown of the effective revocation date.

As mentioned in Chapter 3.3.1 a wrong effective revocation date setting may

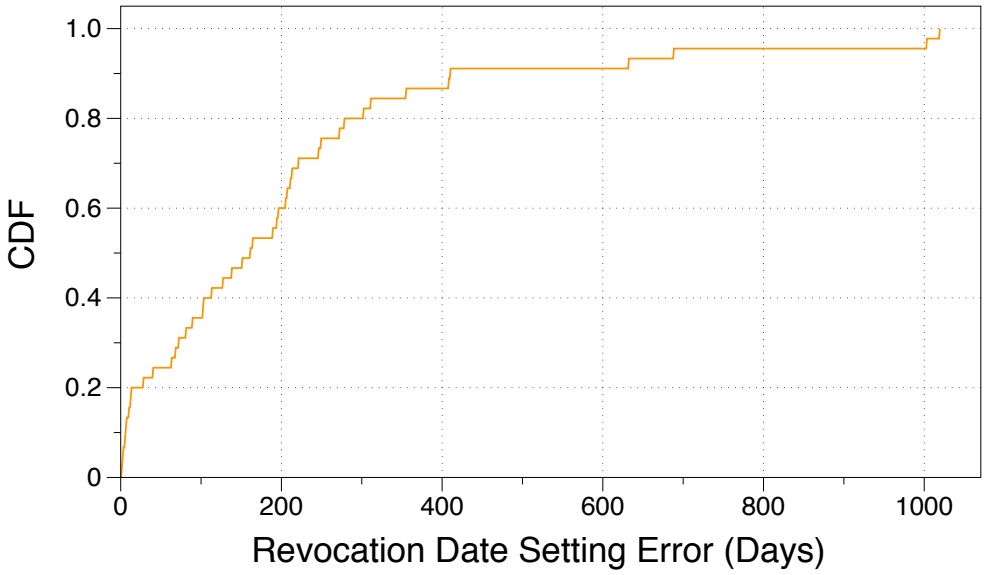

Figure 3.5: CDF of the revocation date setting error ($t_r - min(t_m)$): difference between the effective revocation date and the first malware signing date of a certificate.

result in the survival of signed malware even after a certificate revocation. We can find one prominent case of the wrong effective revocation date setting in Table 3.7; a lot of CAs have set the effective revocation dates after the certificate expiration date. In this case, the revocation has no better impact than the expiration. All the time-stamped binaries remain valid even after the revocation of the certificate.

A not very obvious case is setting the effective revocation date after the date a malware was signed. We conduct the following process to measure how many certificates corresponds to this issue and how many malware remain valid due to this ineffective revocation date setting. We first query VirusTotal with the 12,351,946 signed hashes from WINE (`D4`) and retrieve the signing date. Only 4,729,023 (38.3%) samples have *sigcheck* information in its VirusTotal report, which are signed with 45,613 unique certificates. We then obtain the effective revocation date by querying the CRLs we have (`D2`). For the certificates in the CRL (revoked), we retrieve the

	Comodo	Thawte	Go Daddy	Verisign	Digicert	Starfield	Symantec	WoSign	Startcom	Certum
Ineffective revocation date	●	●	●	●	●	○	○	○	○	○
Certs. without CRLs and OCSP points	○	●	○	○	○	○	○	○	○	○
Unreachable OCSP or CRLs points	●	○	○	●	○	○	○	○	●	●
Inconsistent responses from CRLs and OCSP	○	○	●	○	○	●	○	○	○	○
Unknown or Unauthorized OCSP response	○	○	○	○	○	○	○	○	○	●
Transient certs. in CRLs	●	○	○	○	●	○	○	○	○	●

● = Issues found, ○ = Issues not found

Table 3.8: Mismanagement issues found across the top 10 CAs.

effective revocation date (t_r). At the end of the process, we have 1,022 revoked certificates.

The analysis of these revoked certificates revealed that 45 (5.1%) certificates have a wrong effective revocation date (t_r). The affected CAs are summarized in Table 3.8. Additionally, we measure how many digitally signed malware remain valid as the effect of the ineffective revocation date. As a first step, we use AVClass [95] to label malware using the VirusTotal reports. For the signed binaries identified as malware, we extract the signed date t_m. We keep monitor if there is a signed malware with $t_m < t_r$, in case we encounter such sample, we say an error is in the effective revocation date setting, and the malware remains valid.

Result. We identified 250 malware (5.3% out of the 4,716 malware) signed with the 45 revoked certificates and remain valid. Such a result indicates clients remain vulnerable to the signed malware. They may execute or install the still-valid malware because the OS may not trigger any warnings for clients even though it carries a revoked certificate.

3.3.6 Dissemination of Revocation Information

The last step of the effective revocation process is to disseminate the revocation information to the clients properly. As discussed in Chapter 3.3.1, when combined with the soft-fail policy, the clients may be vulnerable if the revocation information cannot be retrieved. We examine the security problems in this step and try to answer the last research question *Q4. Is the revocation information served adequately?*.

3.3.6.1 Unavailable Revocation Information

In the code signing PKI, CAs must maintain the revocation information even beyond the expiration date. Moreover, taking into account the soft-fail policy, the revocation information has to be always available. However, we identified several cases when the revocation status information for a certificate is not available for clients. The cases and affected CAs are summarized in Table 3.8.

Certificates without CRL and OCSP. We observe that 788 (0.5% out of 145,582) certificates has neither CRLs nor OCSP points. Such a result implies that clients have no means to check the revocation status for these certificates.

Unreachable CRLs and OCSP server. We then examine how many CRLs and OCSP points were unreachable during the observation period (Apr. 16th, 2017-Sept. 10th, 2017). In total 55 CRLs were unreachable at least in one day. Among them, there were 13 CRLs that we could never reach. Of the 13 CRLs, 5 (38.4%) CRLs are unreachable due to HTTP 404 Not Found Error, which means the CA has removed the CRL from the address, but a server still exists. In the OCSP's

case, we experienced 15 OCSP URLs operated by eight CAs (AOL, Verisign, Comodo, StartSSL, WoSign, GlobalTrustFinder, Certum, and GlobalSign) that were unreachable.

3.3.6.2 Mismanagement in CRLs and OCSPs

We also encountered some mismanagement issues while observing the CRLs and OCSPs during the period from Apr. 16th, 2017 to Sept. 10th, 2017. Table 3.8 depicts the affected CAs and the corresponding issues.

No longer updated CRLs. According to the code-signing minimum requirement [28], CRLs should be re-issued at least once a week, and the next update timestamp at the *nextUpdate* field should be less than ten days from *thisUpdate* field. We examined if this guideline is well kept in practice. Of 215 CRLs, 57 CRLs had no change in the *nextUpdate* timestamps, which implies they show no evidence of being updated during the observation period.

Transient certificates in CRLs. Recall that code signing CAs must maintain and provide the revocation status information of all certificates even after expiration due to time stamping. However, 278 certificates were removed from 18 CRLs.

Inconsistent responses from CRLs and OCSP. We expect that the state in the CRL and OCSP to be consistent. However, we encountered 19 certificates having inconsistent responses from CRLs and OCSP from our dataset (`D1`); the certificates are valid according to the OCSP, but the CRLs indicate they are revoked.

Unknown or unauthorized responses from OCSP. The OCSP server can re-

turn three status regarding a certificate; *good*, *revoked*, and *unknown* [27]. The *unknown* state is returned when the server is unaware of the status of the certificate. In our dataset (`D1`), the three OCSP servers (Certum, Shanghai Electronic CA, and LuxTrust) respond with *unknown* for their 669 certificates; most of the certificates (658, 98%) are issued by Certum. We may receive an error message from the OCSP servers, which message is among *malformedRequest*, *internalError*, *tryLater*, *sigRequired*, and *unauthorized*. In our dataset (`D1`), we observe that 2,129 certificates (1.5% out of 145,582) return the *unauthorized* error message; most certificates (1,515, 71.2%) came from Go Daddy. To figure out whether client or server-side causes the problem, we check the revocation status of the certificates through OCSP using different tools: OpenSSL and Windows *SignTool*. We receive the *unauthorized* error for both tools, which indicates that this problem likely resides on the server-side. Which implies either (1) they are not granted to access the revocation records for the certificate, or (2) they removed the revocation records of expired certificates. Since the Windows platform also checks the CRLs when they encounter these responses, it may not result in security issues. However, it is another evidence of the improper maintenance of the OCSP servers by the CAs.

Chapter 4: Detecting Malware Distribution

4.1 Downloader Graph Analytics

In this chapter, we present a systematic analysis of downloaders in the wild. Primarily, we focus on the relationship between the downloaders and the payloads they distribute. We can represent such a relationship as a graph abstraction, where the nodes are the downloader and the payload with an edge connecting the two.

Goals. We set two goals for this study. The *first goal* is to explore the differences between the benign and malicious download activities, by analyzing the graphs constructed on the download relationship. We leverage the insights gained from the analysis to develop a novel method to discriminate against malicious downloaders from benign, which is our *second goal*. The overall performance of malware detectors could improve by propagating the detection down the graph and achieving an earlier warning for malicious download activities. We note that the proposed method complements existing malware detectors by connecting the dots between the downloader and payloads.

Non-goals. We state the non-goals that are outside of the scope of this study: (1) the analysis of the server-side infrastructure and network-level behavior of malware

delivery networks, (2) the attribution of the malware distribution campaigns, (3) improving the network security. As we mentioned above, we instead focus on complementing the approaches based on server-side infrastructures [19,20] and network-level behaviors [17,18].

We start the discussion by a formal description of the *downloader graph* abstraction (Chapter 4.1.1), followed by the summary of the data (Chapter 4.1.2) and how we construct the downloader graphs from the collected data (Chapter 4.1.3). We then share the insights regarding the malicious IGs in Chapter 4.1.4 and design features based on the properties of these malicious IGs in Chapter 4.1.5. Finally, we build the classifier and evaluate its performance in Chapter 4.1.6.

4.1.1 Downloader Graph

We introduce two graph abstractions that captures the *downloaded by* relationship between the downloader and the payload: (i) the *downloader graph* (DG) and (ii) the *influence graph* (IG). We begin with the DG, which is a directed graph defined for each host machines. Each node in the graph represents a file, and an edge from f_a to f_b indicates that f_a has downloaded f_b, which all occur on the same host machine. An IG is a subgraph of the DG, where the *root* corresponds to a downloader. *Influence graph* captures the *impact* of the *root downloader* by depicting the chain of downloads originated from the root.

We present an example of a DG and two IGs (in looped dotted lines) in Figure 4.1. The node can have an attribute such as the filename (e.g., IExplore.exe)

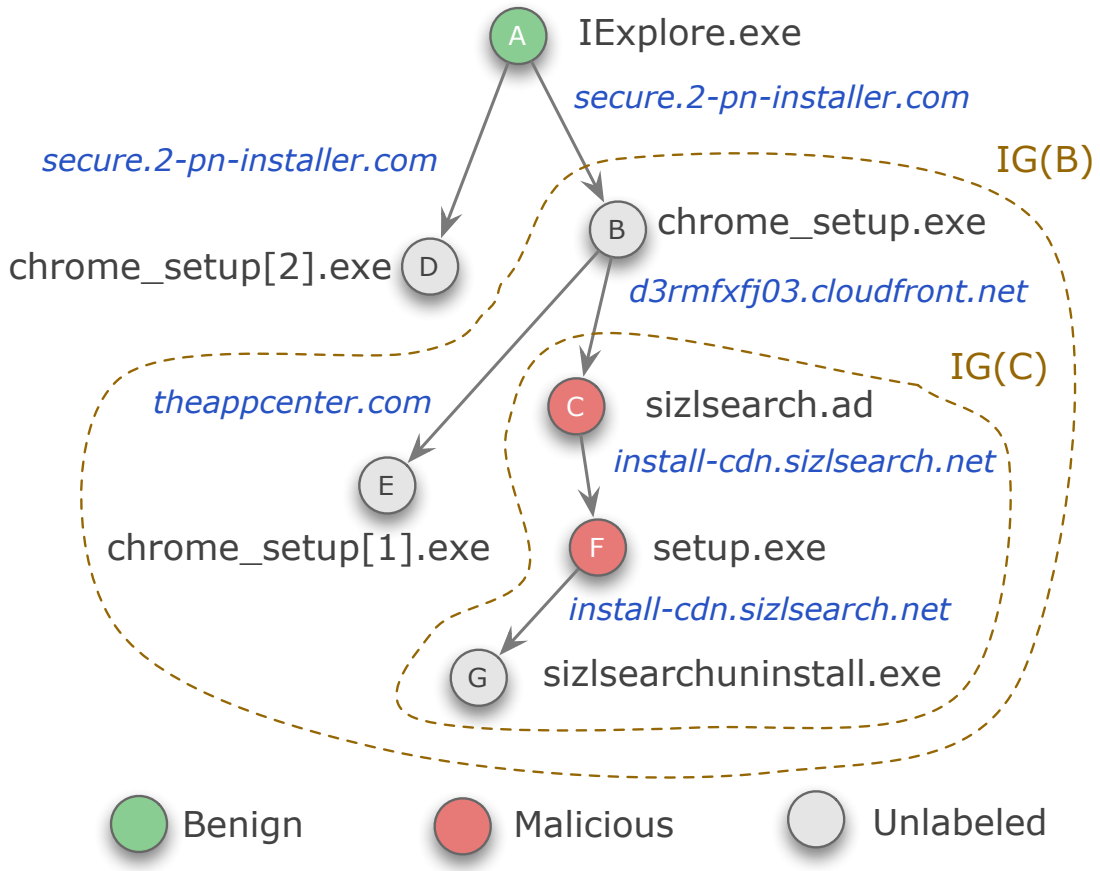

Figure 4.1: Example of a *real* downloader graph and the influence graphs of two selected downloaders.

and each edge has the DNS domain that hosts the payload (e.g., secure.2-pn-installer.com). We observe multiple downloader graphs per machine since they could be disconnected. We also encounter some influence graphs nested in another influence graph, which implies the influence graph of a malicious downloader can be a part of the influence graph of a benign downloader, which we observed in Chapter 3.1.

Formal definition of the downloader graph and influence graph abstraction. Let $\mathcal{V}$ be the set of all files in the dataset, and $\mathcal{M}$ to be the set of all host machines. A downloader graph $\mathcal{G}_i$ for machine $\mathcal{M}_i$ is defined as $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i, \alpha, \beta)$,

where:

- $\mathcal{V}_i \subseteq \mathcal{V}$ denotes the downloader and the downloaded executables presented in machine $\mathcal{M}_i$. We can observe an executable file across multiple machines.
- $\mathcal{E}_i \subseteq \mathcal{V}_i$ X $\mathcal{V}_i$ denotes a set of *directed edges* that represents the *downloaded by* semantics between executables.
- α denotes a set of properties regarding the nodes. These are (a) machine-dependent (unique across all machines), and (b) machine-independent (unique to a host machine) properties.
- β denotes a set of edge properties.

We then define influence graph. The influence graph $\mathcal{I}_g(d_{\mathcal{M}_i})$ is the subgraph of $\mathcal{G}_i$, which is reachable from a *download root* d in machine $\mathcal{M}_i$. The *download root* d has to be a downloader, which has downloaded at least one payload.

4.1.2 Data Summary

Download activities. We utilize the binary telemetry data on 5 million end-hosts described in Chapter 1.3.1. Specifically, we focus on the events that can reconstruct the download. We extract the downloaders and payloads from the binary reputation and the IPS telemetry dataset. From the first dataset we extract information about 24 million distinct files, including 0.4 million parent files. We extract information about 0.5 million portals from the next dataset. We explain how we use these pieces of information to reconstruct the download relationship in Chapter 4.1.3. Table 4.1 summarizes the data.

Influence graphs	19 million
Files (*graph nodes*)	24 million
Total Downloaders	0.9 million
Benign downloaders	87,906
Malicious downloaders	67,609
Download events (*graph edges*)	50.5 million
Domains accessed	0.5 million
Hosts	5 million

Table 4.1: Summary of the data sets and ground truth.

Ground truth. We collect the ground truth for labeling a file. These are a set of a large number of known-malicious and known-benign files from VirusTotal, the National Software Reference Library (NSRL), and an additional data set received from Symantec. The details regarding VirusTotal and NSRL are given in Chapter 1.3.

We label a file as malicious/benign as follows:

- *Malicious.* We compute the percentage r_{mal} of products that flagged the binary as malicious based on the VirusTotal report. We consider that a file is malicious if $r_{mal} \geq 30\%$.
- *Benign.* We treat all the downloaders with matching hashes in NSRL as benign. We use the RDS 2.47 version released in December 2014.

Due to the limitations imposed by the VirusTotal API, we are unable to query all the 24 million file hashes at the time of this study. We, therefore, complement the ground truth with an additional ground truth maintained by Symantec. This step increases the coverage of the ground truth.

After combining all these sources of ground truth, we identify 87,906 benign and 67,609 malicious downloaders; the rest of the downloaders remain unlabeled.

4.1.3 Constructing DGs and Labeling IGs

We begin by constructing the DGs for each machine using the data described in Chapter 4.1.2. Then we extract the IGs from each possible *root* of the DGs and label the IG as *benign*, *malicious*, or *unlabeled* using the ground truth data for the individual downloaders.

Constructing downloader graphs. We utilize the two telemetry datasets from WINE to construct the downloader graphs. At first, we connect the portals in IPS and the downloaded files in the binary reputation dataset by matching the HTTP records. We extract the names of the downloaded files from the URL column of the IPS telemetry data set (example entry:`http://somedomain.com/file_name.exe`); this corresponds to the name of the file created on a disk. 95% of the URLs from which users downloaded PE files include the name of the file downloaded. We then search for these files in the binary reputation dataset, which reports file creation events and includes the corresponding SHA2 hashes. The report should match both the filename and source domain. Also, the timestamp should be within ± 2 days[1] from the IPS event timestamp. If there exists a record that meets the conditions, we create a graph node for the file and add an incoming edge from the portal reported in the IPS event. When we match the filename, we employ an approximate file name matching algorithm by computing the edit distance between file names and by accepting pairs with distance below three as matches. Such a method allows us to handle differences in the file name caused by duplicate downloads (e.g., `setup.exe`

[1]We look two days taking into account the transmission delays and different submission orders between the two data sets.

and `setup[1].exe`).

In addition to the above method, we utilize the parent information in the binary reputation dataset to generate the edge. If there exists a parent with the source domain in the record, we consider it as a download event. Therefore, we add a directed edge from the parent to the file. The observation motivated the step that many of the parents recorded in the binary reputation dataset are the same as the portals in the IPS telemetry. Note that we extract the domain name from the URL and add it as an attribute to the edge.

During the downloader graph construction, we add the following properties to each node in the graph:

- *Number of outgoing edges:* This property captures the download volume of a downloader.
- *Number of incoming edges:* This property captures how often an executable is downloaded on a host.
- *Time interval between a node and its out-neighbors:* This property captures how quickly, on an average, a downloader tends to download other executables after it landed on a host.
- *File score (based on digital signatures):* We assign a score based on the digital signatures of the downloaders, in the range 0–3. Specifically, we set the score utilizing the following information: (1) signature, (2) publisher information, and (3) reputation of the certification authorities. We give a score 0 to the executables without a digital signature or a VirusTotal report. Otherwise, we assign score 1

to the executable if it contains information about the publisher. We increment the score by 1 if the certificate is from the following CAs Comodo, VeriSign, Go Daddy, GlobalSign, and DigiCert. Finally, if the signature is valid, we also add 1 to the score.

- *Number of out-neighbors with score 0 or 1:* This property represents a rudimentary quantification of the malicious intent of a downloader.

We add the following properties to each edge in the graph:

- *URL has IP instead of domain:* denotes if the corresponding download URL had a domain name or an IP address.
- *URL is Localhost:* denotes if the corresponding download URL was relative to a localhost address.
- *URL is in Alexa top 1 million:* denotes if the download URL is in Alexa [96] top 1 million websites.

We also extract some aggregated properties, which will be leveraged to derive some features of influence graphs:

- *File prevalence (FP):* For every file, we count the number of hosts it appears on, in the binary reputation data set.
- *Number of unique downloaders accessing given URL (UDPL):* For every URL (domain), we count the number of unique downloaders that used the domain to download new executables, aggregated across all machines.
- *Number of unique downloads from a given URL (UDFL):* For every URL (domain), we count the number of unique executables downloaded from the domain,

aggregated across all machines.

Extracting the influence graphs. Extracting the influence graph is identical to the subgraph extraction. We iterate the graph and extract all possible subgraphs, which results in the collection of influence graphs. However, we filter out the influence graphs with fewer than 3 nodes, which might not provide sufficient insight for analyzing the properties of downloader graphs.

Labeling influence graphs. We next label the influence graphs, using the benign and malicious labels of downloaders, determined as described in Chapter 4.1.2. We label only the IGs whose root downloader is known in ground truth. Specifically, we consider that an IG is malicious if its root downloader is labeled as malicious. On the other hand, we consider that an IG is benign if one of the following three conditions is true: (1) the root is in NSRL, (2) the digital signature of the root is from a well-known publisher and verified by VirusTotal, or (3) the root is $r_{mal} = 0$ and is benign in Symantec ground truth, and the next two are also true: (1) none of the other nodes in the IG have $r_{mal} > 0$, (2) none of the influence graph nodes is labeled as malicious in Symantec ground truth. This results in 14,918,097 benign and 274,126 malicious influence graphs.

4.1.4 Properties of Malicious Influence Graphs

We now analyze the properties of influence graphs of the benign or malicious downloader and identify properties that have power for discriminating malware distribution. We report the most reliable indicators of malicious activity from our

findings.

Large diameter IGs are mostly malicious. The diameter of influence graphs (the maximum length of the shortest path between two nodes) ranges between 2–5. Figure 4.2(a) shows the distribution. A graph with diameter 2 (a single downloader with multiple payloads) is equally likely to be benign or malicious. However, when the diameter is 3 and above a high percentage (84%) of graphs are malicious. More importantly, almost 12% of the malicious influence graphs have a diameter of 3 and larger. These findings are consistent with prior observations of pricing arbitrage in the underground economy [2], where PPI providers distribute their droppers through opposing PPI infrastructures.

IGs with slow growth rates are mostly malicious. Figure 4.2(b) shows the distribution of the *average inter-download time* (AIT) of the influence graphs. We define AIT to be the average amount of time taken to grow by one node. Almost 88% of the influence graphs that grow slowly (AIT $>$ 1.5 days/node) are malicious. The ratio of malicious graphs further increases with slower growth rates. We also observe that over 65% of the malicious influence graphs have AIT $>$ 1.5 days/node. Such a result suggests that successful malware campaigns, which can evade detection for a long time, tend to deliver their payloads slowly.

URL access patterns vary across subclasses of malicious/benign downloaders. Figure 4.2(c) shows the distribution of the average number of distinct downloaders accessing an Internet domain. We compute this number by determining the set of source domains for the nodes in an influence graph and by taking the

average of the number of downloaders accessing them, across all the hosts. Even though the distribution does not distinctly separate malicious and benign behavior, we found some interesting patterns. The large number of IGs with between 1,100–1,200 downloaders per domain, towards the right side of the plot, is mostly caused by *adware*. In fact, three adware programs, `LUCKYLEAP.EXE`, `LINKSWIFT.EXE`, and `BATBROWSER.EXE` comprise 26% of the IGs in that distribution bucket. Such observation suggests that the organizations behind these programs have resilient server-side infrastructures (and the domains used to host the adware do not have to change very often) and that they frequently re-pack their droppers (in order to evade detection on the client side). Figure 4.2(d) zooms in on the left side of the same distribution. Most of the benign IGs have up to 10 downloaders per domain. However, we also identified several malicious IGs in this bucket; the top-3 are fake antivirus programs (`EASYVACCINESETUP.EXE`, `LITEVACCINESETUP.EXE`, and `BOAN- DEFENDERSETUP.EXE`), which access Korean domains and seem to be part of the same campaign. Windows Update IGs have between 60–70 downloaders per domain in the data set.

Malware tend to download fewer files per domain. Figure 4.2(e) shows the distribution of the average number files downloaded from the domains accessed from an influence graph. The figure also illustrates the diversity of malicious behavior. Most of the malicious droppers that download a large number of files per domain correspond to adware. For example, 40% of the IGs in the 4000–5000 files-per-domain histogram bucket (the tallest malicious bar) correspond to three adware

programs (`LUCKYLEAP.EXE`, `LINKSWIFT.EXE`, and `BATBROWSER.EXE`). Apart from the adware, most of the other malicious droppers (around 40% of all malware) download 1–5 files per domain, as they have to move to new domains after the old ones are blacklisted. Another interesting observation is that benign downloaders also exhibit various behaviors. For example, the influence graphs of Apple Software Update are also in the 4000–5000 histogram bucket, while DivXInstaller's influence graphs download around 10000 files per domain.

4.1.5 Feature Engineering

We design the features based on the properties of the influence graphs. We can organize the features into five *semantic categories*: internal dynamics, life cycle, properties of downloaders, properties of domains, and globally aggregated behavior. In addition to what these features imply, we provide the high-level *intuition* for why we expect they would help classify malicious and benign graphs.

We can distinguish the features into two *feature types*, depending on how we compute the features. Local features (LF) use information contained in the influence graph. We also compute Global features (GF) for each influence graph; however, they use properties aggregated across all the hosts. An example of a GF is the Average Distinct File Affinity, illustrated in Figure 4.2(e), which reflects the *tendency of an influence graph* to download files from domains that are known to serve a large number of distinct files. Table 4.2 provides the list of features and their descriptions.

We quantify the worth of each feature in terms from distinguishing benign and malicious influence graphs using the *gain ratio* [97] with 10-fold cross-validation. We select this metric because it is known to be more robust than alternative metrics, such as the information gain or the Gini index, when the features differ greatly for their range [98]. Table 4.3 shows a summary of the top-10 features in descending order of their gain ratio. The most useful features are the average file prevalence and the features illustrated in Figures 4.2(c)–(e). We emphasize that, because we compute the features per influence graph, the power of these global features helps to classify all the downloaders in the graph. For example, a dropper that normally evades detection because it is present on many hosts, but that always downloads different files, will likely be classified as malicious because of the low average prevalence of the files in its influence graph.

4.1.6 Malware Classification

While in the previous chapter we identify several features that indicate malicious download activity, none of these features, taken individually, are sufficient for detecting most of the malware in the dataset. We, therefore, build a malware detection system that employs *supervised machine* learning techniques for automatically selecting the best combination of features to separate the malicious and benign influence graphs. Specifically, we train a random-forest classifier using influence graphs labeled as described in Chapter 4.1.3. We test the classifier using both *internal* (cross-fold validation and early detection) and *external* (VirusTotal results for some

of the unlabeled samples predicted to be malicious) performance metrics.

Handling data skew. The ground data consists of 274,126 malicious and 14,918,097 benign influence graphs. Training a classifier on such a skewed data set may bias the model toward the abundant class (the benign IGs) and may focus on tuning the less-effective features, i.e., the ones that do not contribute to the classification but rather model noise in the dataset. We address this problem through *stratified sampling*: we sample the abundant and rare classes separately to select approximately equal numbers of IGs for the training set. In practice, there is no need to exclude any examples from the rare class, and some examples from the abundant class can be identified easily and filtered to reduce the variability within that class. We, therefore, filter out all the Web browsers from the benign set of IGs as the set of files they download is not predictable and includes both benign and malicious executables. We identify the top-3 browsers by searching the digital signatures for the following <publisher, product> pairs: <Microsoft Corporation, Internet Explorer>, <Google Inc, Google Chrome>, <Mozilla Corporation, Firefox>; we also check that the file name contains the keywords `chrome`, `firefox` or `explore`.

We keep all the malicious graphs. We then downsample the remaining set of benign graphs (sampling uniformly at random). We manually examined the properties of several random samples created in this manner and observed that they closely match the properties of the abundant class. The balanced training set consists of 43,668 malicious and 44,546 benign influence graphs.

4.1.6.1 Internal Validation of the Classifier

In this chapter, we evaluate the performance of the RFC classifier in three ways: (1) using 10-fold cross-validation on the balanced training dataset, (2) using all the labeled data as a test set, and (3) computing the detection lead time, compared to the anti-virus products employed by VirusTotal.

10-fold cross validation. We choose $N_t = 40$ and $N_f = \log_2(\#\text{ of features}) + 1$, for the experiments. These parameters correspond to the number of decisions trees (N_t) and the number of features per decision tree (N_f). We use the `RandomForestClassifier` module from Python's `scikit-learn` package for the experiment. We run a 10-fold cross validation on the balanced set. We use all the features described on Table 4.2 for training the classifier. The classification result is reported at 1.0% FP rate, which achieves 96.0% TP rate[2].

We list the results with a default `scikit-learn` threshold in the first row of Table 4.4, labeled "All Features".

Feature evaluation. While in Table 4.3 lists the most useful features for the classifier; we also want to know how using different feature combinations would affect the performance of the classifier. Starting from only using FI features from Table 4.2, we combine other feature categories one by one (FL, FD, FU, then FA) and evaluate how the classification performance increases. The result is shown as a Receiver Operating Characteristic (ROC) [99] plot in Figure 4.3, where the X-axis is the false positive rate and the Y-axis is the true positive rate. ROC plots show

[2] $TP_{rate} = \frac{TP}{TP+FN}, FP_{rate} = \frac{FP}{FP+TN}, F_1 = 2 * \frac{Precision*Recall}{Precision+Recall}$

the TP/FP trade-off: the top-left corner corresponds to a perfect classifier, which never makes mistakes. We obtain the curves for the different feature combinations by varying internal thresholds of the random forest classifier. There are large jumps in performance at two points, first is when we add the FL features and then when we add the FA features. We believe the classifier is capturing the insights discussed in Chapter 4.1.4, such as the slow growth rate and specific numbers of distinct downloaders accessing a domain for the malicious influence graphs. At FP rate 1%, the corresponding TP rates are 19.9%, 13.4%, 3%, 17.6%, 28.5% and 96.0%. Interestingly, only using FI rate was performing better than adding FL and FS at this point. Such a result is consistent with the observation that almost 12% of the malicious IGs have diameter 3 and more and 84% of the IGs are malicious at diameter 3 and beyond (Section 4.1.4). At FP rate 3%, the numbers for TP rate are 24.7%, 33.5%, 42.1%, 59.6%, 98.7%. At 10%, TP rates are 41.7%, 64.5%, 71.2%, 84.8%, and 99.8%.

Early detection. We also evaluate how early can we detect malicious executables that are previously unknown. We define "early detection" as "we can flag unknown executables as malicious before their first submission to VirusTotal". We approximate the date when malware samples become known to the security community using the VirusTotal first-submission time. We estimate the detection time in three ways: (1) an executable is detected at its earliest timestamp in the TP set (Lower bound), (2) an executable is detected at the timestamp when the last node was added to the newest influence graph it resides as a node (Upper bound), and (3) the

average timestamp of the executables in the TP set (Average).

We apply random forest with 10 fold cross-validation on the balanced labeled dataset using all 58 features. The outcome of 10 fold cross validation is TP=42,683, FP=822, FN=985, and TN=43,724, regarding the number of influence graphs. For distinct executables in the TP set excluding the ones in the FN set, we compared the first seen timestamps in VirusTotal and the lower-bound/upper-bound/average detection timestamps. Among 31,104 distinct executables that are in TP set but not in FN set, 20,452 files had scanning records in VirusTotal. Among them, 17,462 executables had at least one detection in VirusTotal ($r_{mal} > 0$), and 10,323 executables had detection rates over 30% ($r_{mal} \geq 30$). Table 4.5 lists the results. For $r_{mal} > 0$, the time difference between the VirusTotal first-seen timestamp and the lower bound detection timestamp is 20.91 days on average, and we can detect 6,515 executables earlier than VirusTotal. The upper bound is 23.73 after VirusTotal, with 3,344 executables detected early. On average, we can detect malware 9.24 days before the first VirusTotal detection. Interestingly, for $r_{mal} \geq 30$, the malware are detected on average 25.24 days earlier than VirusTotal. We further investigate this trend in Figure 4.4, where we plot the portion of executables that we are able to detect early, in the average detection scenario, against the VirusTotal detection rate r_{mal}. Up to $r_{mal} = 80\%$, the early detection lead time increases with r_{mal}. Such a result suggests that executables that are more likely to be malicious present stronger indications of maliciousness in their downloader graphs, allowing the classifier to detect them earlier than current anti-virus products.

4.1.6.2 External Validation of the Classifier

In this chapter, we evaluate the performance of the RFC classifier by drawing three random samples from the *unlabeled IGs* and querying the corresponding file hashes in VirusTotal. Out of the 580,210 unlabeled IGs, the classifier identifies 116,787 as malicious. As discussed in Chapter 4.1.2, we were unable to query VirusTotal for all the file hashes, which leaves many of the leaf nodes in the graphs unlabeled. We draw three random samples, of approximately 3000 influence graphs each, from the set of unlabeled graphs and try to estimate the accuracy of the predictions by presenting the results of each set. We consider that we misclassified an IG labeled as malicious if none of the AV products in VirusTotal detect it as malicious. We consider that we misclassified an IG labeled as benign if its nodes were detected by more than 30% of the AV vendors, to account for the fact that AV products may also produce false positives in gray-area situations, such as benign executables that are sometimes involved in malware delivery.

Table 4.6 shows the results. On average 41.41% of the binaries that construct the IGs are known to be malicious also by other AV vendors, while only 0.53% of the binaries in benign IGs has a malicious label. Moreover, on average 78% of the IGs labeled as malicious have at least one internal node that AV products detect as malware, and only 1.58% of the IGs labeled as benign carry a malicious node.

4.2 Detection of Silent Delivery Campaigns

In this chapter, we present a system for detecting silent delivery campaigns from Internet-wide records of download events. The detection is able by identifying locksteps in an unsupervised and deterministic manner.

Goals. Lockstep detection is challenging when analyzing large volumes of data. For example, finding a biclique with the maximum number of edges is an NP-complete problem [100]. It is also not clear *a priori* how to parametrize lockstep detection in order to distinguish benign software dissemination from malware delivery. The *first goal* is to build an efficient and scalable system for detecting lockstep behavior. The system should be unsupervised, i.e., it should not require any prior information or seed nodes. The system should be able to operate in real time and to build the locksteps incrementally, as the stream of stars is collected and fed to the system. While we evaluate the system using telemetry collected worldwide, similar to data available to security companies, OS vendors, or ISPs, we also aim to lower the deployment bar for small enterprises. Specifically, the system should detect locksteps if the same campaign infects at least three victims. The *second goal* is to conduct a large-scale empirical study of silent delivery campaigns. These campaigns may deliver benign software, PUPs, malware or a combination of these payload types. We aim to illuminate the characteristics and differences among the campaigns conducted by various organizations and to expose the business relationships among these organizations. Finally, the *third goal* is to incorporate this domain knowledge into the lockstep detection system and to assess how well it can identify suspicious

activity, such as malware or PUP dissemination campaigns. Using external information about the maliciousness of downloaders and domains caught in locksteps, we aim to assess the true positive and false positive rates[3] of this detection system. We also aim to measure the lead detection time, compared to the existing sources.

Non-goals. We do not aim to detect all possible malware delivery vectors, e.g., download instructions hard coded into the droppers, malware, and PUPs distributed through software bundles, vulnerability exploits, or other mechanisms that do not involve remotely controlling a group of downloaders. The campaigns do not aim to capture the end-to-end attack kill chain and do not include activities performed by the payloads on the hosts. Finally, the system should detect silent delivery campaigns in a deterministic manner, without using machine learning.

We initiate the study with the formal definition of the lockstep behavior in Chapter 4.2.1. We then describe the dataset in Chapter 4.2.2 and present the system in Chapter 4.2.3. We present the findings regarding the silent delivery campaigns in Chapter 4.2.4. Finally, we evaluate the performance of running the system in streams of data in serial mode (Chapter 4.2.5) and check the parallel scalability in Chapter 4.2.6.

4.2.1 Lockstep Detection

The coordinated waves of payload distribution by the silent delivery campaigns form a *lockstep behavior*. The lockstep behavior depicts the synchronized activity

[3]We cannot estimate the false negative rate because the ground truth about malware delivery campaigns is lacking. An undetected malicious downloader may be either a false negative or a dropper not controlled remotely.

among a group of downloaders (or DNS domains) and a set of DNS domains (or downloaders). To retrieve payloads, these group of downloaders (or domains) access (are accessed by) the same set of domains (downloaders) within a limited time window. The access pattern implies that these coordinated downloads do not involve user intervention and do not experience random delays. Additionally, it is required to observe several repeated coordinated downloads to detect a lockstep behavior, which means *a lockstep may correspond to one or several delivery campaigns that use the same infrastructure.* Therefore, the lockstep behavior is a strong signal of the silent delivery campaign.

Formal definition of lockstep behavior. We can define the lockstep behavior as a graph mining problem on a bipartite graph $G = (U, V, E)$ where U and V are a disjoint set of nodes corresponding to *left hand nodes* and *right hand nodes*, respectively. We can place an edge $e \in E$ between two nodes belonging to different sets but not nodes from the same set. Each edge has the time $t_{i,j}$ at which an edge is formed between node $i \in U$ and node $j \in V$ as an attribute.

Now we define a *star*. First, let $U' \subseteq U$ and $V' \subseteq V$. Then a *star* $[U', j, \Delta t, \delta t]$ on U' and some central node $j \in V'$ can be defined as follows:

$$| U' | \geq 2 \tag{4.1}$$

$$(\max_i t_{i,j} - \min_i t_{i,j}) \leq \Delta t \; \forall i \in U' \tag{4.2}$$

The above equations state two conditions a star should meet: it should have at least 2 left-hand nodes and the all the edges in a star should be formed within Δt.

A $[U', V', \Delta t, \delta t]$ in $G(U, V, E)$ is a *lockstep* if it satisfies the following constraints:

$$\mid U' \mid > 2 \tag{4.3}$$

$$\exists V_i^{'} \subseteq V' \ \ \forall i \ \text{ s.t. } \ \mid V_i^{'} \mid > 2 \ \text{ and } \ \mid V_i^{'} \mid \geq \alpha \mid V' \mid \tag{4.4}$$

$$(i, j) \in E \ \ \forall i \in U', \ j \in V_i^{'} \tag{4.5}$$

The above equations specify that lockstep consists of more than 2 nodes each from U and V and that lockstep is a nearly complete subgraph induced by these nodes. If $\alpha = 1.0$, this subgraph is a *complete biclique*, while for any value $\alpha_{min} \leq \alpha < 1$, it is defined as a *near-biclique*. So the structure-wise definition of a *lockstep* has been made: it is a near or complete biclique. It also has to satisfy the following temporal conditions. For the temporal constraints, we predefine Δt and δt and 2 distinct stars defined as $j, j' \in V_{i'}$. Then the temporal property of the lockstep could be written as:

$$(\max_i t_{i,j} - \min_i t_{i,j}) \leq \Delta t \ \forall i \in U' \tag{4.6}$$

$$(\max_i t_{i,j'} - \min_i t_{i,j'}) \leq \Delta t \ \forall i \in U' \tag{4.7}$$

$$| \max_i t_{i,j'} - \max_i t_{i,j} | \geq \delta t \tag{4.8}$$

The equations ensure that lockstep contains at least two stars and there exists a gap of at least δt between the stars. We stated the two stars j, j' as distinct since if the same star occurs in multiple timestamps, we consider it only once inside lockstep.

Lockstep behavior for detecting silent delivery campagins. Now we formulate the problem of detecting silent delivery campaigns through their lockstep behavior. The nodes of the bipartite graph corresponding to downloaders and domains. There would be an edge between a domain and a downloader in the bipartite graph if the downloader accessed the domain to drop a payload. The edge keeps the payload information as an attribute. We set two different topologies for the star. In order to detect the behaviors such as (1) campaign changing to a different domain after a C&C server takedown, (2) domains within the same campaign establishing a connection with a new version of the downloader. Therefore, a star can have (1) multiple downloaders accessing the same domain; or (2) multiple domains accessed by a single downloader.

Figure 4.5 depicts a real-world example of the lockstep behavior. At time $t = 0$, we observe a star with three downloaders accessing a domain. At $t = 3\delta t$, although we observe new stars, they do not correspond to lockstep as lockstep must contain

more than 2 domains and 2 downloaders according to the lockstep definition. Then, at $t = 6\delta t$, we observe a near-biclique, with $\alpha \geq 0.8$, which satisfies the definition of lockstep, and lockstep is detected. We can observe that lockstep corresponds to a series of campaigns. The lockstep consists of a set of stars across different time windows. We exploit the gap between these time windows, and define a *campaign* as follows. We consider the activities appearing in the time windows with a gap less than $n\delta t$ as a single campaign. If the gap is larger than $n\delta t$, we treat it as a different campaign.

4.2.2 Data Summary

Download activities. We utilize a large data set of download events collected in the previous study of the downloader graphs. We reconstruct these events from observations on end hosts. From this data, we utilize the SHA2 hash of the downloader and the downloaded file (payload), the source domain of the download, and the timestamp of the event. We focus on events from 2013, as the data set has good coverage for that year.

We exclude the downloads performed by Web browsers, which typically involve user actions. We identify the top 5 browsers in the dataset by searching the digital signatures for the following <publisher, product> pairs: <Microsoft Corporation, Internet Explorer>, <Google Inc, Chrome>, <Mozilla Corporation, Firefox>, <Apple Inc, Safari>, <Opera Software, Opera>. Table 4.7 summarizes the data after this filtering step.

Ground truth. While ground truth for malware delivery campaigns is currently unavailable, we collect ground truth about executables from multiple sources. We were able to retrieve VirusTotal reports for about 17% of the binaries from 2013.

In this study, we separate PUP from malware. Both malware and PUP are flagged by $r_{mal} \geq 30\%$. To separate them, we search the AV labels given to these samples for the following keywords: "adware", "not-a-virus", "not malicious", "potentially", "unwanted", "pup", "pua", "riskware", "toolbar", "grayware", "unwnt", and "adload" [101]. We define r_{pup} to be the percentage of AV labels that include one of these keywords.

A file is labeled as malware/pup/benign as follows:

- *Malware.* We consider that a binary is malware if $r_{mal} \geq 30\%$ and $r_{pup} \leq 10\%$.
- *PUP.* It is treated as PUP if $r_{pup} > 10\%$ and $r_{mal} \geq 30\%$.
- *Benign.* We consider benign all the executables where either (1) the hash matches or (2) the publisher matches and has a valid signature. We used the RDS 2.52 version released in December 2014.

We identify 1,228 malware samples and 15,350 PUPs through this process.

Information about publishers. To identify publishers engaged in the Pay-Per-Install (PPI) business [2], we utilize two lists of PPI providers from underground forums [102,103].For other types of publishers, we query the Reason Labs knowledge base [104]. This service provides details about the publisher, e.g., whether it is considered safe or if it uses its certificates to sign PUPs.

4.2.3 Detecting Lockstep Behaviors in Real-Time

In this chapter, we describe the design and implementation of Beewolf, which detects lockstep behavior in real-time. Beewolf can operate in two modes. In *offline mode*, the system analyzes the entire download events, with the aim of characterizing lockstep behaviors empirically. We utilize this mode in the experiment in Chapter 4.2.4.In *streaming mode*, Beewolf receives data incrementally and prunes the locksteps detected to focus on suspicious downloaders and domains. We evaluate this mode in Chapter 4.2.5.2. We implement Beewolf in Python, using the `NetworkX` [105] package to manipulate graphs.

As illustrated in Figure 4.6, Beewolf consists of a data analysis pipeline with four components: star detection, galaxy graph construction, frequent pattern (FP) tree construction, and lockstep detection. We also maintain a database with three tables: `download_events`, `stars`, and `locksteps`. The first step is to detect new *star patterns* as new download events appear. In the rest of the paper, we refer to the bipartite graph as "galaxy graph". The stars detected are updated incrementally in the galaxy graph. Further, we traverse the galaxy graph to build the FP tree which is an in-memory data structure to detect locksteps.

4.2.3.1 Whitelisting

As discussed in Chapter 4.2.2, we identify benign binaries using the NSRL data. We maintain a whitelist, which consists of these benign binaries. Before the main data analysis pipeline, we filter out the download events generated by

the benign downloaders listed in the whitelist. We do not expect this whitelist to be exhaustive—NSRL may not include all the legitimate downloaders—but this simple filtering step helps Beewolf to focus on the most suspicious campaigns and improves its performance. Moreover, while it is likely unfeasible to whitelist all benign software, only a few programs have a downloader functionality. The whitelist contains 6,996 downloaders.

4.2.3.2 Star Detection

Each row of the `download_events` table consists of a downloader (dlr), corresponding domain accessed (dom), the downloaded file ($payload$), and the timestamp when the download event occurred. We assign a unique identifier to each download event in the table and sort them in ascending time order. Conceptually, each download event corresponds to an edge in the galaxy graph, linking a node represented by dlr with a node represented by dom.

Given a moving time window of size Δt, We query the events that occurred within this time range. We utilize these series of download events to identify star patterns. The stars can be created in two ways, by starting from a downloader and aggregating the adjacent domains, or by starting from a domain and aggregating the adjacent downloaders. We assign a unique identifier to each new star, and record the associated events in the `stars` table. After generating all the stars within Δt, we slide the time window by δt and repeat the star detection process, until the end of the time window reaches the last event.

4.2.3.3 Galaxy Graph

Beewolf maintains the galaxy graph, which has two kinds of nodes: nodes that correspond to downloader programs and nodes that correspond to domains hosting payloads. We represent a node in the galaxy graph as $node_{gg}$. We explicitly maintain only 1 edge between a downloader and a domain. However, there can arise situations where a downloader accesses a domain at different times; we will discuss how to deal with this situation later in this chapter.

We update the galaxy graph incrementally, using the star patterns detected in the previous step. As explained earlier, there are 2 types of stars. We consider only one type of star and ignore the other while detecting and updating the stars to the galaxy graph; galaxy graph at any point contains only one type of stars. For simpler explanation, we discuss only the star type corresponding to multiple downloaders accessing the same domain; we can extend the same explanation when dealing with the other star type. Further, we present results corresponding to both star types, when dealt separately in Chapter 4.2.4.

When we detect a star, we add the central node (domain) and its adjacent nodes (downloaders accessing it) to the bipartite graph, and we create the corresponding edges. For each newly detected star, while adding the central node (domain) we also specify the star id (e.g., (2) dom_B), in order to separate it from the nodes corresponding to dom_B from different stars. When the new star is a superset of some existing star in the galaxy graph, we replace the existing star with it. If it is a subset of some existing star, Beewolf discards it from further processing.

4.2.3.4 FP tree

We traverse the galaxy graph, constructed in the previous step, to build a data structure called a Frequent Pattern (FP) tree. The FP tree was used successfully in other domains, for example, to design scalable algorithms for frequent pattern mining [106]. We employ the FP tree algorithm from [107]. Let us represent a node in the FP tree as $node_{fp}$. Given the galaxy graph $G = (U, V, E)$, the algorithm starts by sorting the adjacency list of V. The adjacency list is a representation of the galaxy graph and consists of the collection of neighbor lists for each $node_{gg} \in V$. We do the sorting in two rounds. In the first round, we sort each $node_{gg}$ $v \in V$ by their degree (the number of v's neighbors in U), in descending order. In the second round, we sort each list of neighbors. Specifically, we sort the neighbors u of v by their degree (the number of u's neighbors in V), also in descending order.

Once we finish the sorting, we start building the FP tree by creating a root $node_{fp}$ in the tree. For each neighbor u of v, we traverse the FP tree starting from the *root* and check if u is the child of the current $node_{fp}$. If this is the case, we set the current $node_{fp}$ as u and append v to its *visited list*. Otherwise, we first add u as the child of the current $node_{fp}$ and repeat the same process. We continue this process until we have checked all $node_{gg}$ $v's$ and their corresponding neighbors. Figure 4.7 illustrates the FP tree construction procedure given the galaxy graph as input.

Once we finish building the FP tree, we can traverse it to detect all the complete bicliques of the galaxy graph. However, the FP tree has some limitations :

(1) FP tree does not return near-bicliques and (2) FP tree misses part of complete bicliques when overlap exists at the left-hand nodes between a larger biclique and a smaller biclique. We address how we handle these limitations in the next chapter.

4.2.3.5 Lockstep Detection

After constructing the FP tree, we move to the lockstep detection phase. Each path downwards from the root to a $node_{fp}$ A in the FP tree indicates lockstep. The set of nodes along the path corresponds to the downloaders, and the visited list of A corresponds to the domains in the lockstep. For example, in Figure 4.7, $dlr_C \rightarrow dlr_B \rightarrow dlr_A$, the resulting lockstep will be [(dom_B, dom_A), (dlr_C, dlr_B, dlr_A)]. When identifying lockstep, we remove the star id from the domain nodes; however, we store the star ids along with the lockstep, so that we do not lose the download events that resulted from the lockstep behavior. We can observe in Figure 4.7 that some bicliques are not interesting; for example, when A is a child of the root (e.g., dlr_C), we get a star centered on A, and when A is a leaf (e.g., dlr_E), we get a star centered on the single domain from the visited list of A (e.g., dom_A). To avoid generating locksteps that are too small or that are a subset of larger lockstep, we filter out the locksteps that satisfy the following conditions: (1) the number of downloaders or the domains is either below 3; or (2) A has a child with the same visited list.

Partially missing locksteps. The FP tree captures most of the locksteps. However, it misses the small locksteps that share part of the left-hand elements with

the larger lockstep. In Figure 4.7, path $dlr_C \rightarrow dlr_D$ should have produced the lockstep of [(dom_B, dom_C), (dlr_C, dlr_D)]. However, because dlr_C and dlr_D are the part of the longer path $dlr_C \rightarrow dlr_B \rightarrow dlr_A \rightarrow dlr_D$, (2)$dom_B$ fails to visit the corresponding path. This phenomenon occurs at the nodes that appear in multiple paths, such as dlr_D and dlr_E in the example. The missing locksteps can be recovered by maintaining different *node versions*, for each path where the node appears, and by constructing a separate FP tree only on the stars that contain the node with multiple versions. To cover all the locksteps, we do this recursively until there is no node with multiple versions in the FP tree. However, considering the overhead due to the recursive computation and the chance that the near-biclique algorithm would help recover some of the partially missing locksteps as explained in the next paragraph, we only apply the FP tree construction once on each node with multiple versions without recursion.

Near-bicliques. The aim here is to detect locksteps even in cases where some edges are missing from the galaxy graph, e.g., the corresponding download events may not have been recorded for some reason. These missing edges could prevent some potential nodes from being added to the lockstep. Therefore we relax the lockstep definition and search for subgraphs that include a fraction $\alpha \geq \alpha_{min}$ of the edges that would form a biclique. We set α_{min} to 0.8 to accommodate for at most 1 missing edge in the smallest lockstep.

There could be many possible missing edges. We reduce the search space by exploiting the fact that the adjacent nodes in the FP tree have higher connectivity

than the other nodes, which implies that by introducing it into the lockstep will have fewer missing edges.

We point to the end node A of the path, which we want to extract the lockstep. We start by traversing the FP tree upwards, toward the root, until we reach a node B that has a larger visited list. We also count the number of hops ($missing_v$) required to reach B. We define the relative complement list as the difference between the visited list of B and that of A. We then add the relative complement list to the candidate list with $missing_v$ as an attribute. Next, we look at the children A. We add each child to the candidate list with the size of the difference between its visited list and $A's$ visited list as the attribute $missing_u$.

Once we get the candidate list, we sort it by the attribute in ascending order. Starting from the first node in the list, we add the node into the lockstep and calculate α which corresponds to the edge density within lockstep. We stop when α drops below α_{min}. We observed that, in practice, this heuristic is good enough, as the adjacent nodes in the FP tree are more likely to be connected to the lockstep than the other nodes.

4.2.3.6 Streaming Set-up

When using Beewolf in a streaming setting, we ingest the download event data in real time. Instead of triggering the system for each single data stream, we run the system by processing incoming data as a batch within a fixed period Δt. Except for the difference in how the data comes in the system, the rest of the process is

identical to that of the non-streaming setup. The star detection search for new stars from the batch data; the new stars are added to the galaxy graph; the FP tree is built from the galaxy graph, and the lockstep detection will find new locksteps.

4.2.3.7 Parallel Implementation

While Beewolf searches for locksteps for the nodes in the FP tree sequentially, the supplementation phase for finding partially missing locksteps can be done in parallel, since it consists of independent supplementation sub-processes (i.e., extracting locksteps for each node with multiple versions). We implement the parallel process of Beewolf using the `Multiprocessing` [108] Python package.

Algorithm. The input for the supplementation, a node with multiple versions and the stars containing the node $(u, stars)$, prepared during the FP tree construction, is fed to the worker. The worker then performs the entire process to extract locksteps from the input $stars$, in this case without the supplementation phase. The master then collects the extracted locksteps from all workers.

4.2.4 Silent Distribution Campaigns

We present a large-scale empirical study of silent delivery campaigns. As discussed in Chapters 4.2.3.2 and 4.2.3.3, we can track two types of stars in the galaxy graph: multiple downloaders accessing a domain ($type_{dlr:dom}$) and multiple domains accessing a downloader ($type_{dom:dlr}$). These two different star types result in different bicliques and capture different download activities. The difference derives

from the fact that the central nodes in the stars may be duplicated in the galaxy graph when we add new stars that emerge in later time windows. The resulting locksteps reflect different distribution strategies. $type_{dlr:dom}$ account for downloaders that are more stable than the domains. Conversely, $type_{dom:dlr}$ identify distribution networks where domains are more stable.

For the empirical analysis, we set a narrow time window, to detect download events that are remotely triggered and do not experience delays. More generally, we should choose a shorter time window than the typical reaction time of domain blacklists during the observation period. In consequence, we set the time window Δt to 3 days, and we use a sliding window δt of 3 days.[4]

Lockstep attribution. In general, it is challenging to precisely identify which organizations were controlling the download activities reflected in the locksteps we detect, as the domains may no longer be registered and the downloaders may no longer be active. However, we aim to make a coarse-grained distinction among the distribution campaigns for malware, PUP and benign software, to compare their properties and to assess their overlaps. To do so, we observe that 38.2% (3479 out of 9103) of the downloaders involved in locksteps are digitally signed, with valid X.509 certificates. We first analyze these signatures to determine the most frequent publisher in lockstep. We consider that a publisher is the *representative publisher (rep-pub)* of the lockstep, if it accounts for more than 50% of the signed downloaders in the lockstep. If we cannot identify a representative publisher, we

[4]During the observation period, domains delivering malware were blacklisted within 17 days on average [109].

set the lockstep's *rep-pub* to `mixed`. In this manner, we identify 335 *rep-pubs*. We investigate the top 50 *rep-pubs* from each lockstep type, and we manually categorize them into six different groups: potentially unwanted programs (PUP) [101], pay-per-install (PPI) [2], benign (BN), other, mixed, and unknown (UK). The first 4 groups inherit the label of the *rep-pub*, determined as discussed in Chapter 4.2.2. We place the `mixed` *rep-pubs* in a separate group. In some cases, we cannot identify the real publisher behind the lockstep, as the downloader is an archive extractor (Winzip). These correspond to the unknown group. Table 4.8 describes the distribution of these lockstep groups. While it can label some locksteps in this manner, we observe that most locksteps involve downloaders that are difficult to place in a specific category, as many locksteps have mixed rep-pubs.

We, therefore, perform a second labeling step, based on the payloads that the locksteps distribute. We distinguish between malware and PUP payloads with the method described in Chapter 4.2.2. We conduct the labeling in two steps. First, we label the downloaders by the payloads they distribute within the lockstep. We say a downloader is *malware downloader (MD)* if it distributes at least one malware.[5] Similarly, we label a downloader as *PUP downloader (PD)* if it downloads PUP payloads but no malware. A downloader is labeled as *Benign downloader (BD)* if it downloads a benign payload but no suspicious (malware, PUP) download. We place the rest as *unknown downloader (UD)*. As the next step, we label the locksteps. We label the locksteps that include at least one MD as *malware downloader lockstep*

[5]This is an aggressive labeling policy, as even benign downloaders may be tricked into downloading malware occasionally. However, this labeling produces a conservative estimate of the false positive rate (as discussed in Chapter 4.2, we do not aim to measure false negatives).

(MDL). Similarly, we label lockstep as *PUP downloader lockstep (PDL)* if it contains PDs but no MDs. We label the locksteps with no suspicious (MD, PD) downloader as *unknown downloader lockstep (UDL)*. Note that, as malware families sometimes evade detection for extended periods of time, not every UDLs correspond to benign download activities. Therefore, we try to identify the *benign downloader locksteps (BDL)* among the UDLs. Similar to the definition of MDLs and PDLs, the BDL should contain at least one BD.

We present the result of the labeling in Table 4.9. For both lockstep types, MDL occupy more than 80% of the total number of locksteps while benign are 4.82% and 2.48% for $type_{dlr:dom}$ and $type_{dom:dlr}$, respectively. The higher success rate in labeling with payloads, compared to labeling only with downloaders, reflects the security community's focus on detecting and labeling malware, rather than on understanding the client-side distribution infrastructure.

Identifying campaigns. As discussed in Chapter 4.2, we separate the campaigns within the lockstep by $n\delta t$. By setting $n = 3$, We identify 1,292,141/71,424/27,145/ 6,233 campaigns corresponding to MDL/PDL/BDL/UDL. On average there are 12.2/4.9/5.7/3.6 campaigns per lockstep for MDL/PDL/BDL.

4.2.4.1 Relationships Among Representative Publishers

The locksteps help determine the business relationships between rep-pubs and payloads among groups of rep-pubs. We focus on PPI and PUP providers, which distribute other executables intentionally. We collect PUP and PPIs from the top

10 rep-pubs with a high percentage of MDLs within their locksteps. Each of these rep-pubs conducted at least 40 campaigns. We also include the known PPI providers `Amonetize Ltd.`, `Conduit Ltd.` and `OutBrowse LTD` to this list.

We investigate which publishers frequently appear together in lockstep with the 13 rep-pubs. As the downloaders signed by these publishers simultaneously utilize the same server-side infrastructure, this likely reflects a relationship among the corresponding distribution networks. We also determine whether one of these downloaders was itself downloaded by one of the downloaders in the lockstep, which suggests a stronger business connection. We, therefore, term such relationship between the publishers as *partner*. For example, we observe such partnership relations among some PPI providers, e.g., `Outbrowse Ltd.` that frequently delivers downloaders from `Somoto Ltd.`. Additional frequent partners of `Somoto Ltd.` include `Mindad media Ltd.`, `IronInstall`, `betwlkx`, and `Multiply ROI`, which suggests a stable business relationship with these organizations.[6] The cases where we cannot establish a *downloaded-by* relationship among the downloaders in the lockstep may point to an organization that uses multiple code signing certificates to evade attribution or to relationships with a common third party. We term such relationship as *neighbor*. We illustrate some of these business relationships in Figure 4.8. The nodes are the publishers, and the edge between publishers indicate a business relationship, either *partner* or *neighbor*. The thickness of the edge indicates the frequency of that relationship.

To further illuminate this ecosystem, we employ a community detection algo-

[6]Several of these publishers attended the 2014 Affiliate Summit in Las Vegas [110].

rithm [111] to the graph illustrated in Figure 4.8(a). This algorithm identifies seven communities. Within each community, we determine the publisher with the highest betweenness centrality [112], which is the number of shortest paths between any two nodes that pass through the publisher. This graph centrality measure singles out a node that likely acts as a bridge between other nodes from each community.

- *Community #1: OutBrowse.* This community represents the advertisers or the affiliates of the Outbrowse PPI. The PUPs `Multiply ROI` and `Mindad media Ltd.` are frequently in lockstep with the rep-pub. The other publishers in this community represent variants of the rep-pub's certificate: `OutBrowse LTD` and `OutBrowse`.
- *Community #2: Somoto.* This community belongs to Somoto, which is also a PPI provider. Beside Somoto's certificates (`Somoto Ltd.` and `Somoto Israel`), this community includes 12 other publishers. `International News Network Limited`, a known PUP distributor, is tightly connected with the publishers in this community, suggesting a close relationship.
- *Community #3: raonmedia.* 22 publishers belong to this community. Three PUP publishers including `raonmedia`, `Pacifics Co.`, and `CIDA` showed high centrality in this community. All three publishers were located in Busan, Korea and the certificates were issued by Thawte, Inc., which suggests these publishers could belong to the same group.
- *Community #4: Sendori.* Although we see PPI `Conduit Ltd.` within the community, PUP `Sendori` has a higher centrality. At 77 publishers, this is the largest community. `Sendori` was is tightly connected to most of the publishers within

the group, which reflects an aggressive distribution strategy of this PUP.

- *Community #5: Amonetize.* This group represents `Amonetize Ltd.` and several PUPs. In particular, `Shetef Solutions & Consulting (1998) Ltd.` is known to be the advertiser [113] of Amonetize.
- *Communities #6 & #7.* These communities are small and include the `InstallX` PPI and the `Wajam` PUP.

These results suggest that the partner and neighbor relationships can expose organizations that utilize distinct code signing certificates for different activities, e.g., PPI and PUP. Additionally, the graph communities capture close relationships among the publishers, such as delivery networks that rent the server-side infrastructure from a third party or publishers that engage in aggressive distribution campaigns using multiple providers. The graph also includes instant messengers and file sharing software, which are likely involved in locksteps resulting from spam campaigns.

4.2.4.2 Malware and PUP Delivery Ecosystems

Downloaders that appear in locksteps with different labels provide the opportunity to analyze the overlap of different software distribution ecosystems. 36.7% of the downloaders (3,345 out of 9,103) are present in both MDLs and PDLs. These downloaders are associated with 7,635 and 6,886 of $type_{dlr:dom}$ and $type_{dom:dlr}$ PDLs, which account for 97.8% and 99.8% of all the PDLs. 100 of these downloaders dropped payloads known to be malicious, while the other ones downloaded other files in lock-

step with the malware droppers. The PUP publishers from Figure 4.8 distributed 13 trojan families, including `vundo`, `pasta`, `symmi`, `crone`, `pahador`, `pecompact`, `scar`, `dapato`, `renum`, `jorik`, `fareit`, `llac`, and `kazy`. We also observed generic trojans, `induc` (virus), `zeroaccess` (botnet), `onescan` (fakeAV), `pincav` (keystroke logger), `dnschanger`, `startpage`, and several worms delivered through these publishers.

To further illuminate the connection of the malware and PUP delivery ecosystems, we compare the publishers from the locksteps to the ones from the Malsign blacklist of certificates used to sign PUP and malware payloads [101]. In this way, we identify 1,926 downloaders signed by 212 publishers from malsign, which were involved in 70,984 and 5,468 of MDLs and PDLs respectively. Such a result suggests that many publishers thought to belong to the PUP category are also involved in malware delivery. Considering that many of the unknown files in the data set may be malware samples (83% of the payloads not found in VirusTotal), the number of MDLs is likely higher.

These results contradict two recent studies [114, 115], which did not find substantial overlap between the malware and PUP delivery ecosystems. The key distinction is that these studies analyzed *direct download relationships* between publisher pairs, while lockstep detection allows us to identify *indirect relationships*, through the neighbor links discussed in Chapter 4.2.4.1. These indirect links can overcome evasive strategies such as certificate polymorphism or utilizing unsigned downloaders for malicious payloads. In particular, in Somoto's locksteps, 90.6% of the downloaders, on average, are either unsigned or have invalid certificates. We also observe several PUPs with over 50% ratio, including `Strongvault Online Storage LLC`,

`Save Valet`, and `LLC Mail.Ru`. Variations in experimental methods may further explain the different results. Thomas et al. [114] milk PPI downloaders on hosts located in the US, while our dataset includes hosts from 72 countries. Geographical targeting has been reported previously for PPI providers [2]. Additionally, their data set covers a different observation period. In contrast, Kotzias et al. [115] analyze data from WINE from a time span that largely overlaps with the observation period. However, they focus on 70 malware families, excluding for instance trojans that received generic labels from anti-virus vendors.

4.2.4.3 Properties of MDLs

We identify a total of 54,497 and 51,831 locksteps of $type_{dlr:dom}$ and $type_{dom:dlr}$, respectively, that download at least one malware. These MDLs come from 246 and 169 *rep-pub* each. In addition to the PPI and PUP delivery vectors discussed above, we observed that malware sometimes exploit the compromised software update channels to propagate. We identified a malware distribution campaign involving the `KMP Media Co.`, which is a legitimate media player. The campaign distributed trojan `dofoil`. The version of the media player involved in the MDL is 3.6.0.87, which is known to have a stack overflow vulnerability [116] that was exploited in the wild [117]. Additionally, while experimenting with larger values for Δt, we observed a Hewlett-Packard software updater deliver the `hexzone` ransomware.[7]

We observe several features that distinguish MDLs from other locksteps. Fig-

[7]We did not find evidence that HP's code signing certificate was compromised; it is more likely that the malware was able to infect the server-side infrastructure involved in software updates, which is consistent with prior reports of a trojan that was signed by HP after it infected the company's systems, but without having compromised any certificates [118].

ure 4.9 illustrates the approximate FP tree level where the MDLs reside. As each node in the FP tree corresponds to lockstep, the typical level of MDLs indicates the region of the FP tree where it is most likely to find evidence of malware distribution; which is an approximation, as we may add or subtract a level when computing near-bicliques, as described in Chapter 4.2.3.5. The median FP tree level where locksteps reside is 5, for both $type_{dlr:dom}$ and $type_{dom:dlr}$. In other words, the median number of downloaders in an MDL is 5, which is relatively close to the root of the FP tree, as malware delivery networks rely on only a few downloaders within a time window. This observation helps to improve the performance of Beewolf in streaming mode, as discussed in Chapter 4.2.5.2.

4.2.4.4 Detection Performance

While the previous chapter provides empirical insights into silent distribution campaigns, we now evaluate the effectiveness of Beewolf as a detection system. The aim is to detect suspicious activity, such as malware and PUP dissemination campaigns. We can use this information in several ways. The downloaders and domains caught in locksteps can help prioritize further analysis, e.g., to attribute the campaigns to publishers as we demonstrate in Chapter 4.2.4. We can combine with other techniques (e.g., DNS reputation systems [119, 120]) to detect a specific form of abuse (e.g., botnet activity). An enterprise may also block all downloads initiated remotely by unknown organizations; in this case, an initial whitelist with a few trusted publishers could be sufficient.

We use the locksteps labeled in Chapter 4.2.4 to validate the system: an MDL or PDL detection represents a *true positive*, while a BDL detection is a *false positive*. For the true positives, we compute the detection lead time, compared with the anti-virus products invoked by VirusTotal (for downloaders) and with three malware blacklists (for domains). We also analyze the causes of false positive detections. As the ground truth about malware distribution campaigns is lacking, we cannot estimate the false negative rate.

Experimental settings. We evaluate Beewolf in offline mode, and we build on the empirical insights to select the appropriate configuration parameters. We set $\Delta t = \delta t = 3$ days, to capture locksteps with a high domain churn.

Result. Table 4.9 lists the numbers of locksteps from each category. Overall, the benign locksteps (BDLs) represent 4.82% and 2.48% of $type_{dlr:dom}$ and $type_{dom:dlr}$ locksteps, respectively. We observe the highest fraction of BDLs among the mixed locksteps of $type_{dlr:dom}$, perhaps because malware and PUP creators utilize dedicated malicious infrastructures as well as generic downloaders, which may also distribute benign software. In contrast, PPI rep-pubs do not generate *any* BDL of $type_{dom:dlr}$ and only 4 BDLs of $type_{dlr:dom}$. Overall, the suspicious locksteps (MDL or PDL) account for 92.85% and 97.24% of all locksteps of $type_{dlr:dom}$ and $type_{dom:dlr}$, respectively.

Detection lead time. As Beewolf is content-agnostic (i.e., it does not analyze the downloader binaries or the Web content served by the URLs contacted), we evaluate how early it can detect suspicious downloaders or domains that are previously

unknown. We consider the downloaders submitted to VirusTotal in 2013 that have at least one detection record. We compare the time when Beewolf can detect these downloaders to the time of their first submission to VirusTotal. Because a downloader detected by Beewolf is active in the wild, and because VirusTotal invokes up to 63 AV products with updated virus definitions, we consider that a detection lead time illustrates the opportunity to identify previously unknown droppers. Lockstep emerges at the time when the second star is formed; we estimate the detection time of a downloader as the earliest detection timestamp among the locksteps that contain it. Figure 4.10 illustrates this comparison. The negative range represents a detection lead time, and the positive range corresponds to detection lag. We observe 1182 downloaders detected early and 213 downloaders detected late. The median detection lead time is 165 days. Among the late detections, 69 of the downloaders are detected <3 days late, which suggests that they may be detected early with a shorter Δt. In contrast, the detection lead time seems uniformly distributed, suggesting that Beewolf can detect both recent distribution campaigns as well as campaigns that have been operating for a while.

We also collect URLs blacklisted in 2013 from three publicly available sources [121–123]. These URLs correspond to 394 unique domains, of which 29 were present in the dataset. Among these 29 domains, we caught 14 domains in locksteps; the other 15 domains may represent false negatives, or they may correspond to malware dissemination techniques other than silent delivery campaigns. As for downloaders, we estimate the detection lead time for these 14 domains by comparing the lockstep detection dates with the blacklisting dates. Except for one domain that is detected

36 days later, 13 out of 14 domains are detected early, with a median lead detection time of 196 days.

4.2.5 System Performance

In this chapter, we present the performance of Beewolf. We evaluate the system in two ways: (i) comparison made with the alternative techniques, (ii) the performance at the streaming mode by feeding the download data in batches.

4.2.5.1 Comparison with Alternative Techniques

We compare the lockstep detection algorithm with two alternative techniques for detecting malicious campaigns: community detection algorithms [111,124], which have been explored extensively in the context of graph mining, and prior algorithm for detecting lockstep behavior [70].

Community detection. To compare lockstep detection and community detection algorithms, we construct a $type_{dlr:dom}$ bipartite graph with all the download events. We employ two popular community detection algorithms [111,124] based on optimizing modularity, i.e., maximizing the edges within each community and minimizing the edges between communities, and we compare them with the locksteps detected by Beewolf. We use the Python package `igraph` [125] to run these algorithms.

Table 4.10 shows the comparison of these algorithms with Beewolf. Most of the communities are very small (< 3 nodes). We observe that a large portion of the locksteps gets mapped to the larger communities. The number of locksteps/community

and the number of nodes/community reflect long tail distributions. We define the lockstep coverage as the fraction of locksteps that reside within a single community. We predominantly observe locksteps having large ($> 80\%$) coverage. Further, the number of unique rep-pubs per community is considerably large (10). Such a result suggests that most of the communities are mixed up with locksteps coming from different publishers, which makes it difficult to assign each community to a particular group logically. Community detection algorithms do not account for the timing of downloads, which makes it hard to pinpoint coordinated behavior between nodes.

Prior lockstep detection algorithm. We compare the locksteps detected by our algorithm to locksteps detected by the serial implementation of the CopyCatch [70] algorithm over one month (January 2013) of data. We reimplement CopyCatch, as the code is not available. There are qualitative differences between the proposed algorithm and CopyCatch. Firstly, the proposed algorithm is unsupervised. In contrast, CopyCatch requires seed domains corresponding to malicious domains and also times for all the domains at which some suspicious activity has occurred. Secondly, given a batch of data, we can detect all the locksteps within that batch; CopyCatch can detect one single lockstep, which depends on the seed. Thirdly, CopyCatch solves an optimization problem to detect locksteps, which makes it highly sensitive to the choice of seed domains and the times provided. Furthermore, this serial implementation of CopyCatch is not scalable for large lockstep sizes; we consider only small locksteps for comparison.

To make a fair comparison, we generate 470 locksteps using our algorithm over

the one-month data. Of these only 139 locksteps have a size less than ten which we consider for comparison. For each lockstep the algorithm detected, we provide CopyCatch the domains as seed nodes and the timestamp at which each domain was active in the lockstep as the seed times. The proposed algorithm generates 470 locksteps in 7.56 s, taking an average of 0.016 seconds per lockstep. In contrast, CopyCatch takes 600.9 s to generate 139 locksteps—an average of 4.32 s per lockstep detection. These results suggest that Beewolf shows promise for processing streaming data.

4.2.5.2 Streaming Performance

Experimental settings. We evaluate Beewolf in streaming mode by feeding the download data in batches. We set $\Delta t = \delta t = 3$ days, to capture locksteps with a high domain churn. In the lockstep detection phase, we filter out the FP tree level over 5, based on the observation that MDLs reside close to the root of the FP tree, and we measure the latency of lockstep detection. Each batch corresponds to a time window of $\Delta t = 3$ days. As we employ one year of data, we have 121 data points excluding the first batch in the experiment. For all 121 data points, we measure the elapsed time for each of the four phases in the data analysis (illustrated in Figure 4.7). We run the experiments on Amazon's Elastic Compute Cloud (Amazon EC2) [126]. We use one M4.4xlarge instance, which has a 16-core 2.4 GHz Intel Xeon E5-2676 v3 (Haswell) with 64 GB of memory. For this evaluation, we focus on $type_{dlr:dom}$ graphs.

Streaming performance. Figure 4.11(a) illustrates the growth of the data structures that Beewolf maintains. The plots have a logarithmic Y-axis, to compare both the number of new stars per batch and the cumulative number of nodes in the galaxy and the FP tree. On average, a batch contains 225,939 download events. Both the number of nodes in the galaxy graph and the FP tree grow linearly. In the end, the graph has 123,335 nodes and 637,814 edges. As the data grows, the cost for detecting lockstep also grows incrementally.

Figure 4.11(b) suggests that Beewolf's runtime is dominated by the lockstep detection phase, which accounts for 97.2% of the total runtime on average. The total runtime shows three growth patterns: a steep increase for the first 20 batches, a slower increase for most of the period, and another steep increase starting around batch 94–96. Each of these growth patterns is linear and follows a regression line with the coefficients shown in the figure. To further understand the latency of the lockstep detection step, recall that this phase consists of two parts: (1) lockstep detection on the main FP tree, (2) supplementation for finding partially missing locksteps (see Chapter 4.2.3.5). The near-biclique detection is done during lockstep detection, and it results in an overhead of at most 10 seconds. As shown in Figure 4.11(c), the first part is fast and requires at most 12 s. Most of the cost of lockstep detection comes from the supplementation effort, which induces the three phases of linear growth. In particular, the number of nodes that have multiple versions in the FP tree increases significantly around batch 94–96, which triggers the third growth pattern in the total runtime.

While Beewolf searches for these nodes sequentially, we note that this could

be done in parallel, as the supplementation sub-processes are independent of each other. To evaluate this potential optimization, we estimate the lockstep detection time with optimal parallelism. Assuming that enough computing resources are available for running all missing lockstep searches in parallel, the longest running supplementation will determine the cost of this part of the computation. We obtain the total cost of lockstep detection with optimal parallelism by adding this to the runtime of lockstep detection on the main FP tree. As shown in Figure 4.11(c), this cost is at most 19 seconds and shows a single pattern of slow linear growth. The supplementation phase is essential for detecting malicious locksteps: at the last batch, this phase contributes to 95% of the MDL detections and 91% of the PDL detections. These locksteps include 48.7% of the MDs and 80.6% of the PDs.

Overall, these results suggest that the cost of Beewolf's first two analysis steps is amortized over time, as we perform star detection only on the new batch of data and we maintain the galaxy graph incrementally. The FP tree construction algorithm is not incremental and requires traversing the entire graph, but we optimize this step by pruning the FP tree at level 5, as we typically do no observe MDLs below this level. Similarly, the lockstep detection requires traversing the whole FP tree and constructing version lists for its nodes, but we can optimize this step by performing the supplementation in parallel. The resulting runtime of Beewolf increases linearly with the size of the graph. The results suggest that maintaining one year of download events imposes reasonable resource and performance requirements, even if we execute lockstep detection every day.

4.2.6 Parallel Scalability

We evaluate the performance of Beewolf, in parallel, over a different number of processors (strong scalability) and at streaming mode (increasing the data size). We run our experiments on a 64-core 2.2 GHz AMD Opteron 6274 machine. In the lockstep detection phase, we filter out the FP tree level over 5, based on the observation that MDLs reside close to the root of the FP tree. For this evaluation, we focus on $type_{dlr:dom}$ graphs.

4.2.6.1 Strong Scalability

Experimental settings. We evaluate the strong scalability of Beewolf, in parallel. The idea is to measure the decrease in the running time of the algorithm by adding more computation resources while we fix the size of the data. We run the Beewolf in offline mode, where we have the supplementation phase done in parallel at the entire dataset and measure the total elapsed time from all four phases. We iterate the process by doubling the number of cores from 1 to 64.

Result. We depict the running time measured at a different number of cores in Figure 4.12(a). Each line represents the total elapsed time (total cost), the running time of the parallel processes only (star detection and supplementation), and the ideal scalability from the parallel processes. We also present the running time of parallel component separately. All data points are the average of 5 measurements, excluding the highest and lowest value. The total runtime and the total parallel component cost decreases as the number of core increases until it saturates around

64 cores. The measured performance is close to the ideal performance until 4–8 cores and converges afterward. We attribute this to the overhead of the parallelism.

4.2.6.2 Scaling with Data

Experimental settings. We evaluate parallel Beewolf in streaming mode by feeding the download data in batches. As a reminder, in streaming mode, only the supplementation phase is done in parallel. We have 121 data points with a time window of $\Delta t = 3$ days, which represents one year of download event data. For all 121 data points, we measure the elapsed time for each of the four phases in our data analysis (illustrated in Figure 4.7). We utilize 32 cores for this measurement.

Result. We illustrate the running time of Beewolf at figure 4.12(a). The total runtime shows three growth patterns: a steep increase for the first 20 batches, a slower increase for most of the period, and another steep increase starting around batch 94–96. Each of these growth patterns is linear. Figure 4.12(c) depicts the growth of the number of supplementation processes where we can find the similar three linear growth pattern. The correlation in the increase pattern between the two plots and the linear growth in the running time suggests that our algorithm is resilient to the increasing size of the dataset. Also, our results suggest that maintaining one year of download events imposes reasonable resource and performance requirements, even if we execute lockstep detection every day.

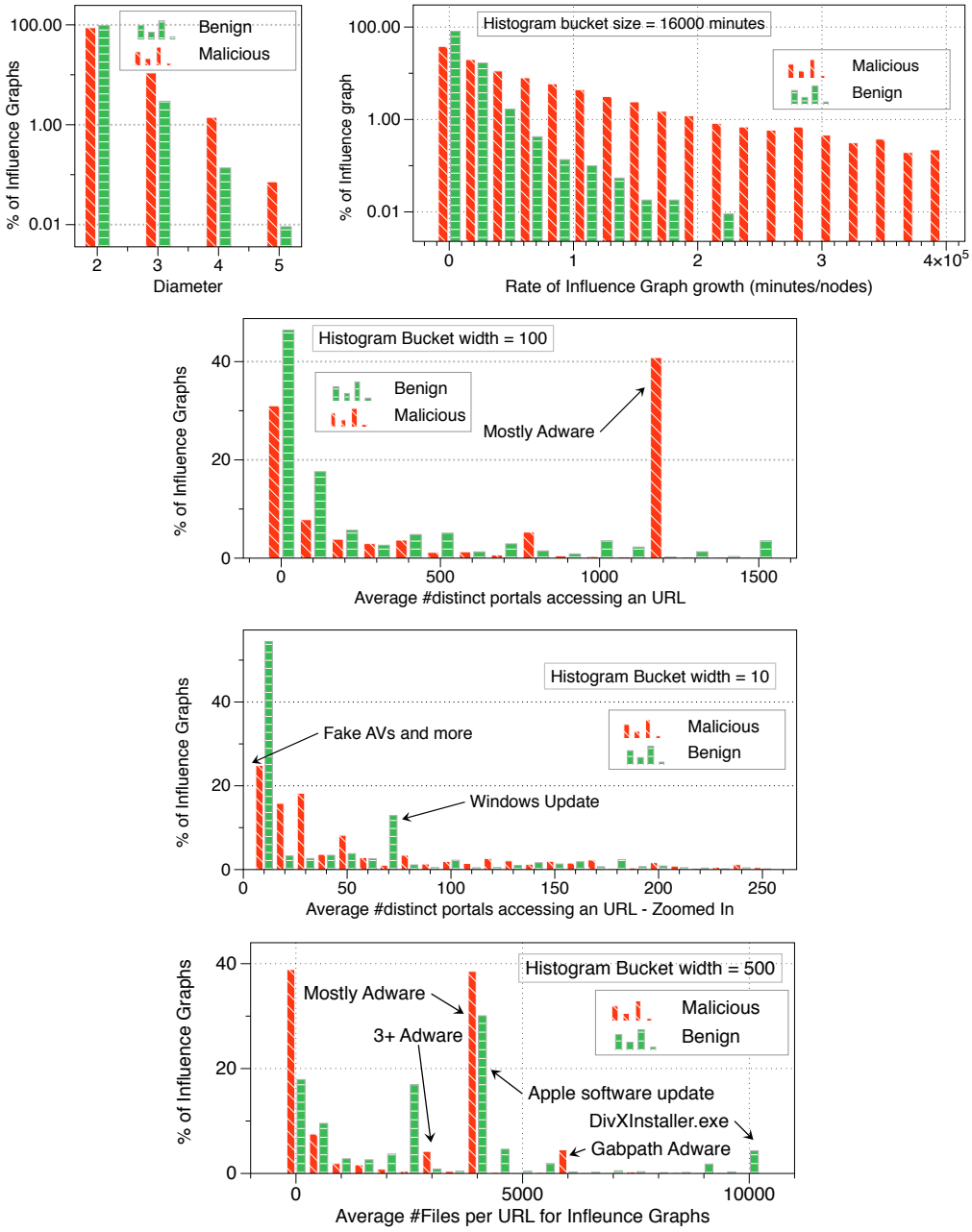

Figure 4.2: (a) Distribution of influence graph diameter (b) growth rate of influence graphs (expressed as the average inter-download time) (c) distribution of the average number distinct portals accessing the domains of an influence graph (d) zoomed in version of the previous plot (e) distribution of the number of executables downloaded from the source domains of an influence graph.

Categories	Name	Type	Explanation
Internal Dynamics (FI)	$\mathbf{f_1}$. Diameter	LF	Diameter capture a chain of download relations (e.g., $A \rightarrow B$, $B \rightarrow C$, and so on). High diameter could imply malicious behavior such as droppers or Pay-per-install ecosystem where there is an affiliate.
	$\mathbf{f_2}$. Clustering Coefficient	LF	Clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together, i.e., create triangles (e.g., (A $\rightarrow$ B, A $\rightarrow$ C, B $\rightarrow$ C) or (A $\rightarrow$ B, B $\rightarrow$ C, C $\rightarrow$ A)). Intuitively, I would expect the benign software to show low clustering as compared to malware.
	$\mathbf{f_3}$. Density	LF	A graph with high density means that the binaries are downloading each other actively, and there are binaries that are downloaded by multiple downloaders.
	$\mathbf{f_4}$. Total Download	LF	Total number of downloads made by the influence graph.
Domain Properties (FU)	$\mathbf{f_5}$. Number of Unique Domains	LF	Some malware droppers may access more domains, e.g. if they employ domain generation algorithms to bypass blacklists.
	$\mathbf{f_6}$. Domain Name Similarity	LF	I compute the similarity between all pairs of domains accessed from the graph: $$similarity = 1 - \frac{EditDistance(D_1, D_2)}{\min(length(D_1), length(D_2))}$$
	$\mathbf{f_7}$. Alexa top-1M	LF	Percentage of domains in the influence graph that appear in Alexa top 1 Million list
Downloader Score Properties (FD)	$\mathbf{f_8}$. Average Score	LF	Average score (based on signatures—see Section 4.1.3) of all the downloaders in an influence graph. Intuitively, even if the root has high score (signed malware) it might download low score downloaders, indicating that the root might be malicious.
	$\mathbf{f_9}$. Standard Deviation of the Score	LF	A malicious influence graph can achieve high average score by downloading known high score downloaders. Standard deviation of downloader score in the IG might be relatively robust in that regard.
Life Cycle (FL)	$\mathbf{f_{10}}$. Influence Graph Life Span	LF	The life span of an influence graph is defined as the time interval between the newest and oldest node. The life span of malicious IGs tends to be shorter, as A/V programs eventually start detecting the droppers.
	$\mathbf{f_{11}}$. Growth Rate	LF	Average inter-download time for the nodes in an influence graph ($= \frac{IG_{lifespan}}{\#Nodes}$). Malicious IG trying to remain stealthy tend to grow slowly, as shown in Figure 4.2(b).
	$\mathbf{f_{12}}$. Children Spread	LF	Average time difference between the root and the children of an influence graph.
	$\mathbf{f_{13}}$. Intra-children Time Spread	LF	Average difference in download timestamp among the children of the root. Intuitively, I would expect this value to be smaller for malware, as they tend to be more aggressive as downloaders.
Gloablly Aggregated Behavior (FA)	$\mathbf{f_{14}}$. Average File Prevalence	GF	Average FP (see Section 4.1.3) of the executables that appear in an influence graph. Benign binaries are expected to show high prevalence.
	$\mathbf{f_{15}}$. Average Distinct File Affinity	GF	Intuitively, this depicts whether an influence graph tends to prefer/avoid domains that download less/more distinct binaries. This feature is illustrated in Figure 4.2(e).
	$\mathbf{f_{16}}$. Average Distinct Dropper Affinity	GF	Intuitively, this depicts the bias of a dropper toward a specific set of domains in order download new binaries. This feature is illustrated in Figures 4.2(c)–(d).

Table 4.2: Feature categories and the high-level intuition behind some of the important features

Features	Gain Ratio	Avg. Rank
$\mathbf{f_{14}}$. Avg. File Prevalence	.16 ± 0	1.1 ± .3
$\mathbf{f_{15}}$. Avg. Distinct File Affinity	.16 ± 0	1.9 ± .3
$\mathbf{f_{16}}$. Avg. Distinct Dropper Affinity	.12 ± 0	3 ± 0
$\mathbf{f_{10}}$. IG Life Span	.1 ± 0	4.3 ± .64
$\mathbf{f_{7}}$. Alexa Top-1M	.1 ± 0	5 ± 0.5
$\mathbf{f_{11}}$. Growth Rate	.09 ± 0	5.7 ± 0.6
$\mathbf{f_{13}}$. Children Spread	.08 ± 0	7.4 ± .66
$\mathbf{f_{1}}$. Diameter	.06 ± 0	8.5 ± 1.1
$\mathbf{f_{12}}$. Intra-children Time Spread	.06 ± 0	11.4 ± 1.1

Table 4.3: Gain ratio of top 10 features of influence graphs.

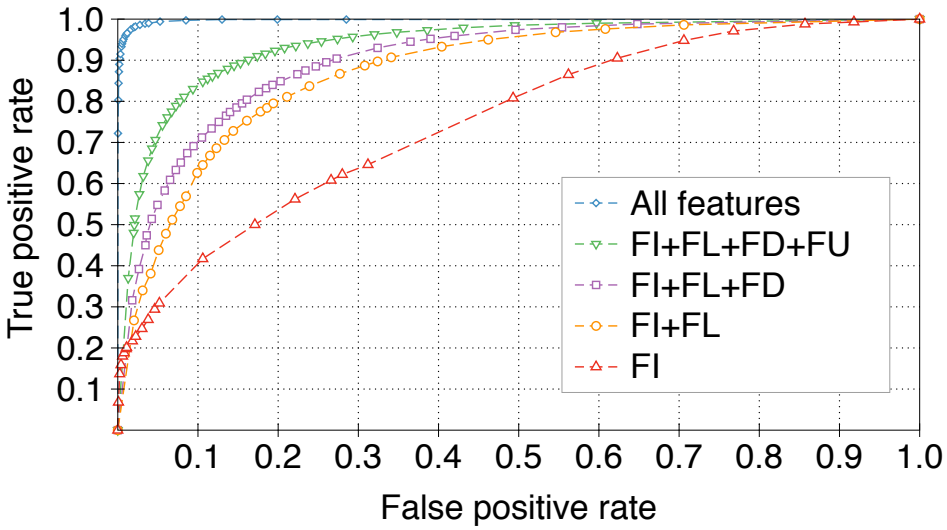

Figure 4.3: ROC curve for different feature groups

Algorithms	**TP rate**	**FP rate**	**F-score**	**ROC Area**
All Features	0.980	0.020	0.980	0.998
FI+FL+FD+FU	0.868	0.124	0.870	0.939
FI+FL+FD	0.831	0.184	0.823	0.902
FI+FL	0.811	0.211	0.801	0.876
Only FI	0.602	0.261	0.644	0.752

Table 4.4: Classifier performance on malicious class.

$r_{mal} > 0$	**Lower bound**	**Upper bound**	**Average**
Distinct Executables	6,515 (37.3%)	3,344 (19.2%)	4,871 (27.9%)
Early detection avg.	20.91	-23.73	9.24
$r_{mal} \geq 30$	**Lower bound**	**Upper bound**	**Average**
Distinct Executables	3,939 (38.2%)	2,041 (19.8%)	3,002 (29.1%)
Early detection avg.	35.86	-7.69	25.24

Table 4.5: Early detection.

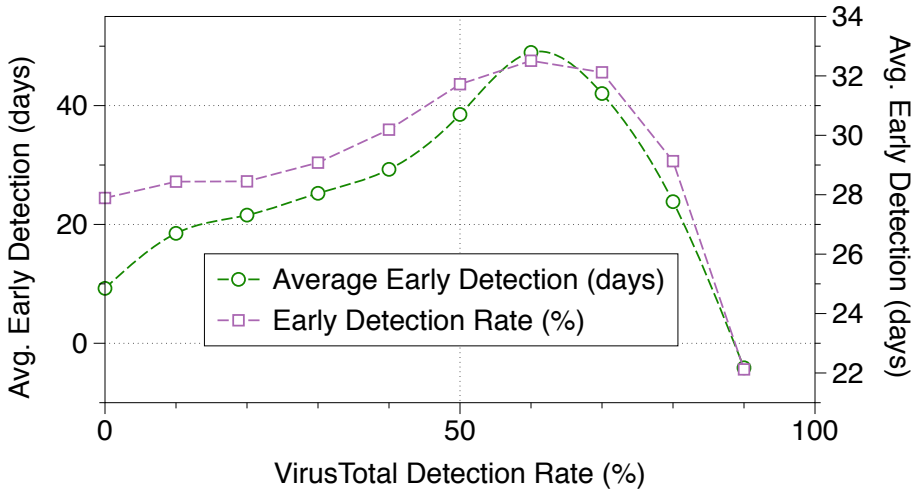

Figure 4.4: Detection rate vs. Early detection ratio / Early detection avg. (days)

#Run	Predictions	#IGs	$\#IG_{mal}$ by VT	$\#Binary_{mal}$ by VT
#1	Malicious	582	444 (76%)	1093(41%)
	Benign	2456	43 (1.7%)	60 (0.5%)
#2	Malicious	590	471 (80%)	1249 (43%)
	Benign	2454	30 (1.2%)	38 (0.3%)
#3	Malicious	592	466 (79%)	1041 (41%)
	Benign	2495	44 (1.7%)	67 (0.6%)

Table 4.6: Testing classifier on the unlabeled influence graphs.

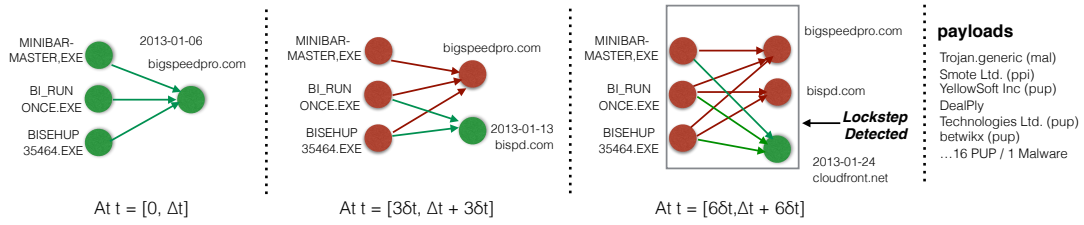

Figure 4.5: Lockstep illustration (Red color corresponds to existing nodes and edges. Green color corresponds to new nodes and edges which we receive in the data stream in an online fashion).

Lockstep Behaviors	127,495
$type_{dlr:dom}$	67,094
$type_{dom:dlr}$	60,401
Total Downloaders	83,088
Domains accessed	60,002
Download events	33.3 million
Total Payloads	0.7 million
Hosts	1.9 million

Table 4.7: Summary of the data sets of the year 2013.

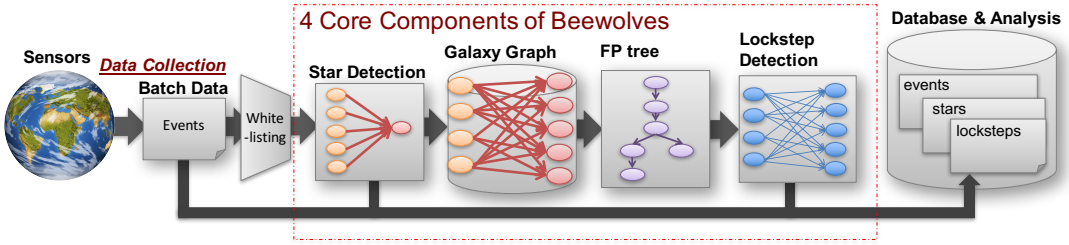

Figure 4.6: System architecture.

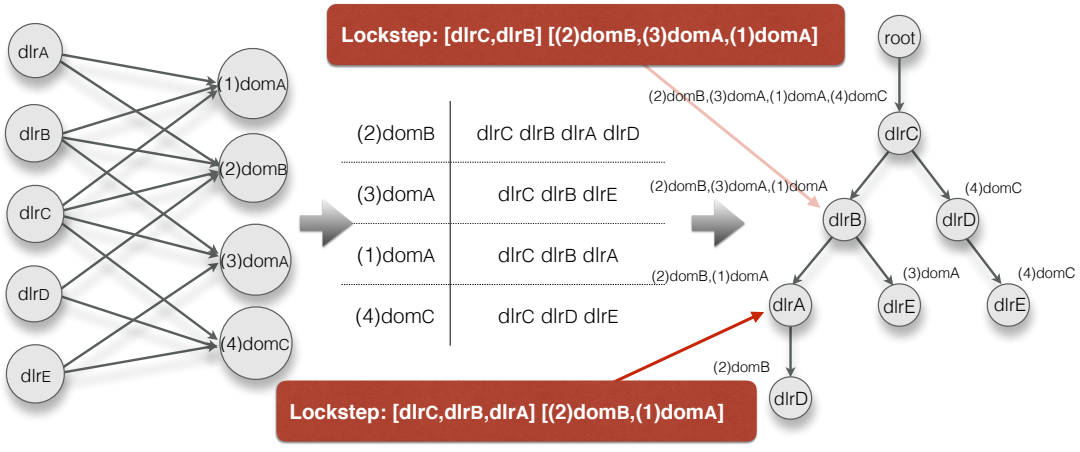

Figure 4.7: Example of FP tree construction: (a) Galaxy graph, (b) Sorted adjacency list, (c) FP tree.

Table 4.8: Lockstep group statistics.

	$type_{dlr:dom}$ (MDL/PDL/BDL/UDL)	$type_{dom:dlr}$ (MDL/PDL/BDL/UDL)
PUP	27,522 (26,764/501/109/148)	13,117 (11,902/1,202/6/7)
PPI	2,639 (2,137/498/4/0)	1,496 (1,164/332/0/0)
BN	3,939 (1,749/888/597/705)	2,021 (1,152/840/7/22)
Other	9,203 (8,041/1,053/58/51)	5,092 (3,479/1,580/8/25)
Mixed	20,766 (14,085/4,069/2,255/357)	36,594 (32,576/2,479/1,449/90)
UK	86 (86/0/0/0)	835 (808/27/0/0)

Table 4.9: Lockstep label statistics

	$type_{dlr:dom}$	$type_{dom:dlr}$
MDL	54,497 (81.22%)	51,831 (85.81%)
PDL	7,800 (11.63%)	6,901 (11.43%)
BDL	3,231 (4.82%)	1,500 (2.48%)
UDL	1,566 (2.33%)	169 (0.28%)

Table 4.10: Community detection and locksteps.

	FastGreedy [111]	Multilevel [124]
Number of Communities	6919	6439
Average #nodes/community	21	22
Median #nodes/community	2	2
Average #locksteps/community	2042	2387
Median #locksteps/community	7	31
Average Lockstep Coverage	89.7	85.9
Median Lockstep Coverage	91.67	87.5
Average #Unique rep-pubs/community	9	11

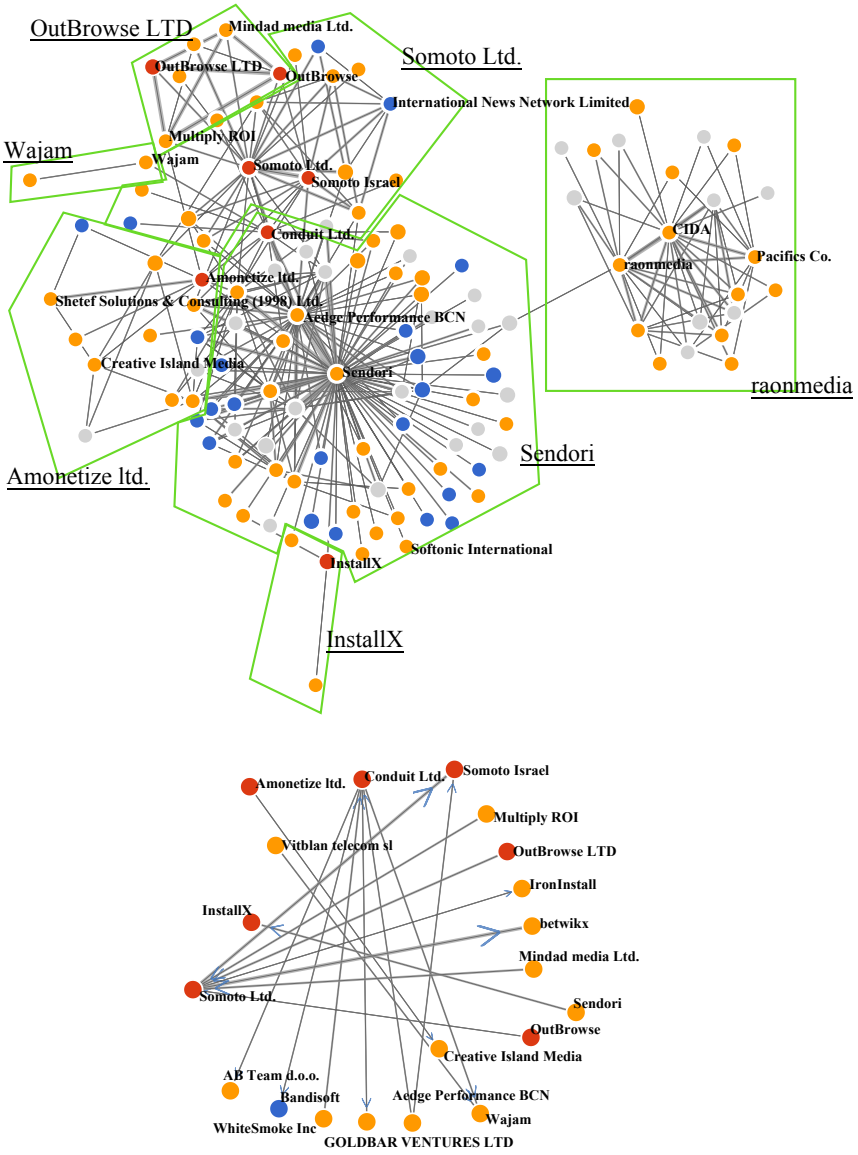

Figure 4.8: Business relationship: (a) both partner and neighbor, (b) partner relationship for PPIs. (node color red/orange/blue/gray corresponds to PPI/PUP/benign/other).

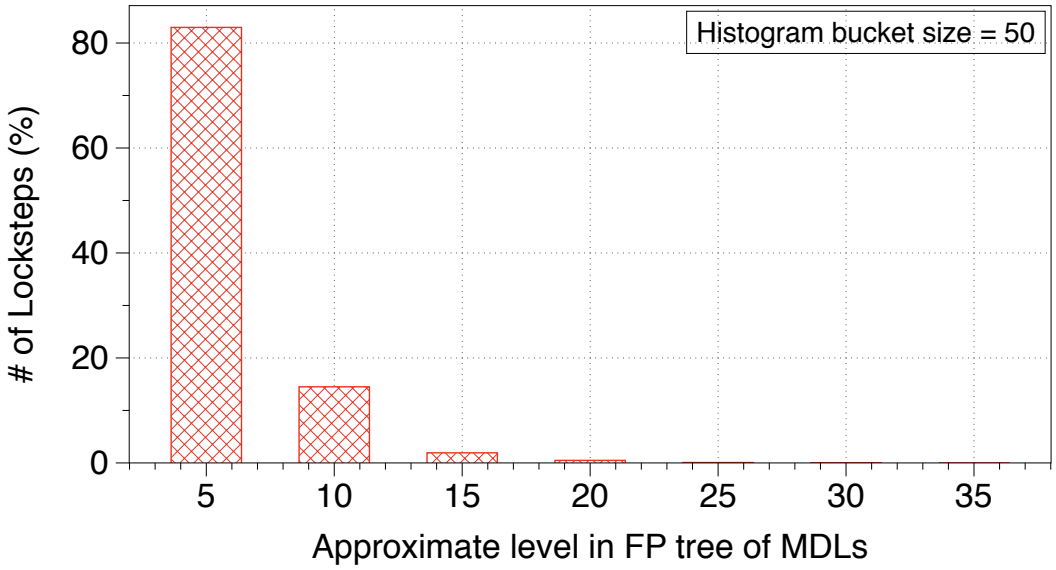

Figure 4.9: Approximate FP tree level of the MDLs.

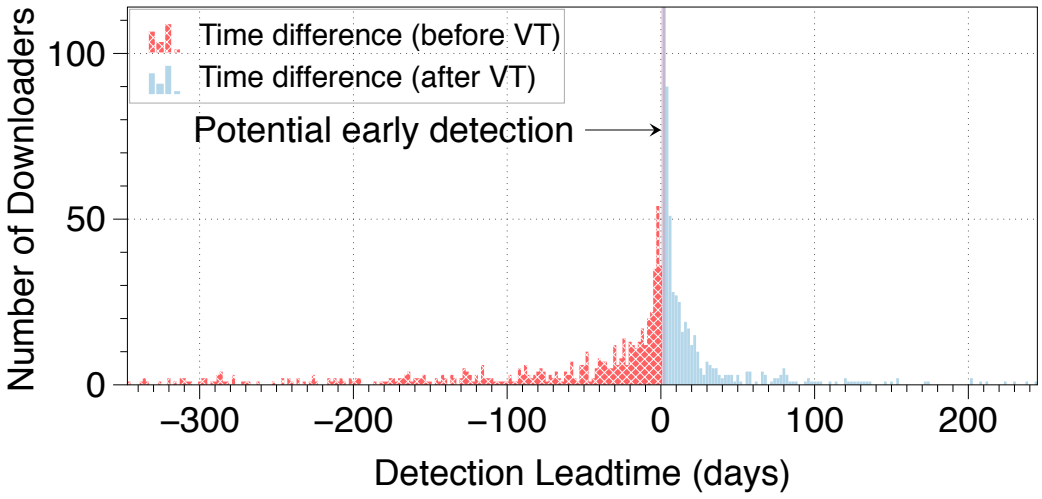

Figure 4.10: Detection lead time for MD/PDs.

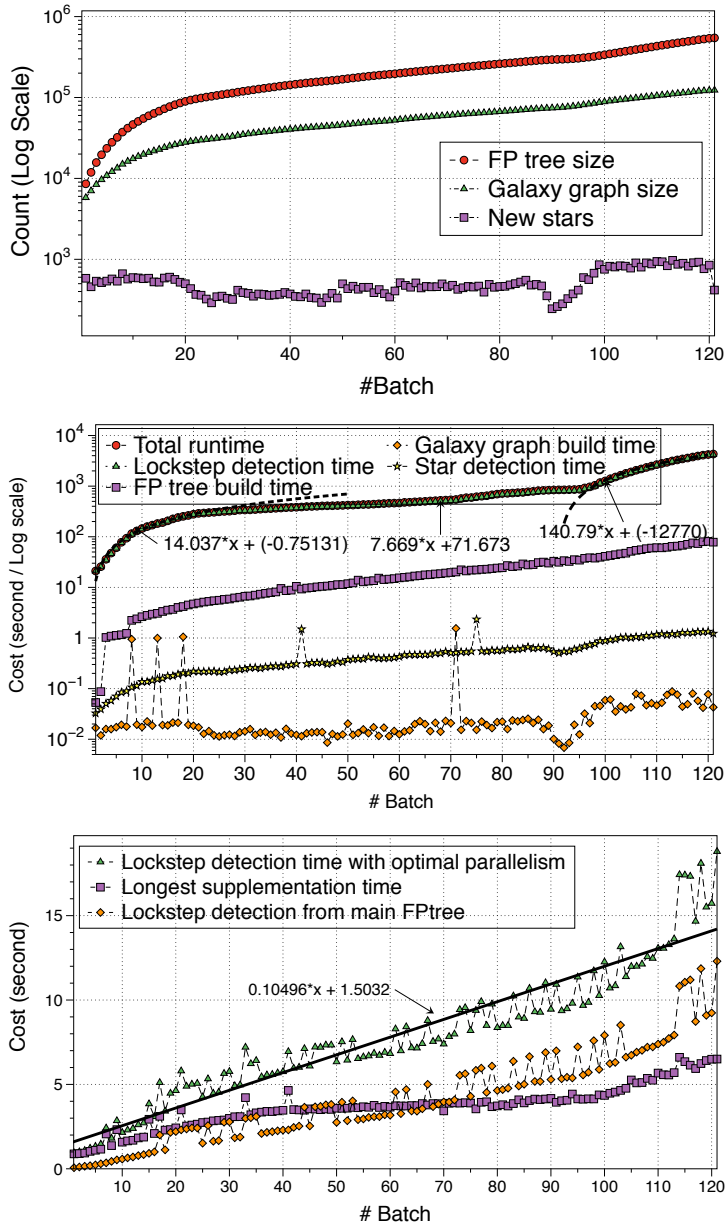

Figure 4.11: Streaming performance: (a) data growth, (b) running time of the streaming system, (c) estimated lockstep detection runtime with optimal parallelism.

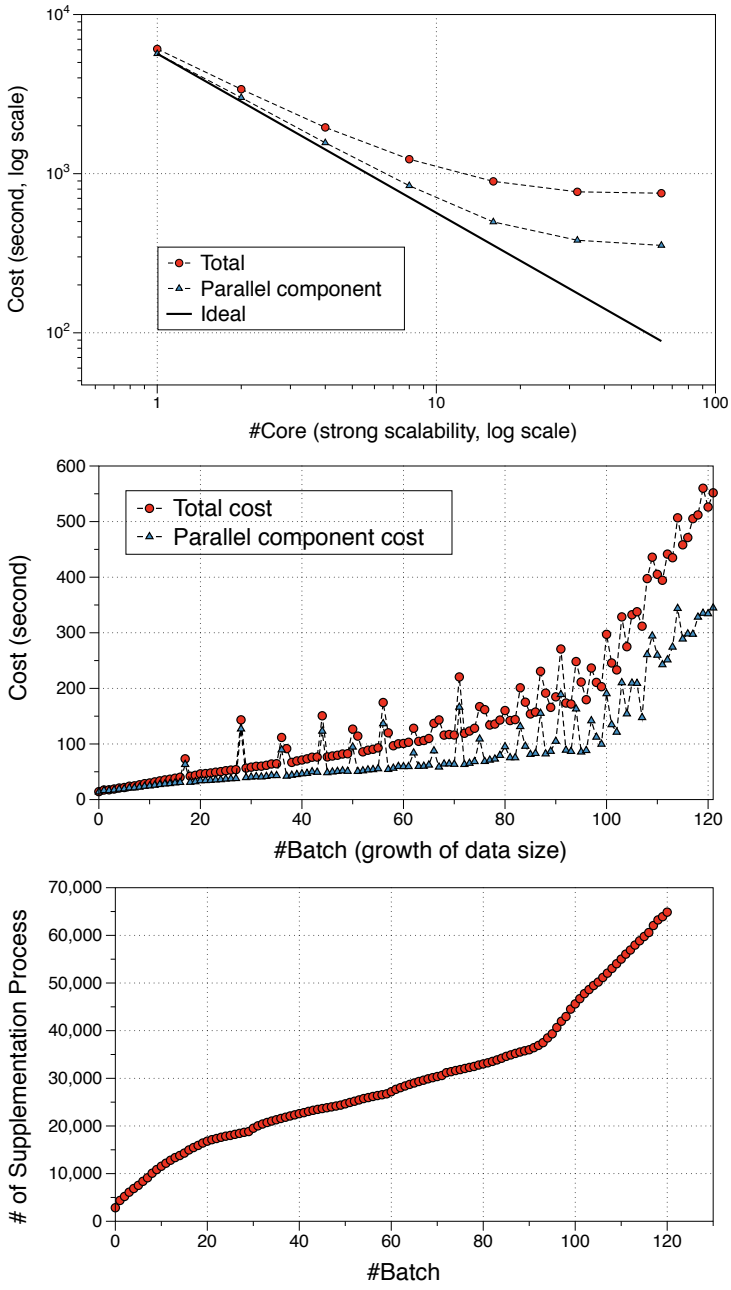

Figure 4.12: Scalability measurement: (a) strong scalability, (b) the running time (cost) over a increasing data size and (c) the number of supplementation processes at each batch.

Chapter 5: Blocking Malware Distribution

5.1 The Benefit of Transparency

Transparency guarantees openness and accountability of the data, however, itself does not give any guarantee as a security mechanism. It is still not clear how beneficial it is to security by introducing transparency to its domain. So far, we only have encountered the anecdotal example of the benefit, which is the case of the misbehaving CAs detected by investigating the Certificate Transparency log. In this paper, we explore the benefit of transparency in the domain of downloads.

The opaque software distribution. The opaque software distribution ecosystem is exposing users to security threats. When accepting an incoming software from the Internet, the users rely on the code-signing certificate. However, we reported the widespread abuse in the ecosystem (e.g., stolen private keys, certificates issued with false identities) in Chapter 3. According to our investigation, the revocation, which is the primary defense mechanism against such abuse, is also broken. The dominant factor in the fail of the ecosystem is the difficulty of precisely identifying the software and the compromised certificates. Moreover, malware often exploits the benign software to distribute their software [127]. Even the benign software

publishers do not sign all of their distributed software [4], which makes users more vulnerable in such attacks.

Download transparency. To remove the opaqueness of the software distribution, we introduce the Download Transparency. We have a platform, which logs the download events from the internet, submitted by the users. The software publishers may also publish the software they will distribute in the future in advance. These submitted records of download events are backed by the Merkle Tree, which gives the cryptographic guarantee that the records are not compromised since the submission.

Disturbing malware propagation. We can utilize the transparency log as a ground for making decisions of accepting an incoming software. Which may help in preventing the successful landing of malware, for instance, a user can only accept download which has been reported before. In this case, we may expect the repacking behavior to be less effective for malware because it has a high chance of not seen in the log. Moreover, it becomes hard for the malware writers, who exploit the benign software distribution by compromising their downloaders or certificates, to stick to the same strategy. Because the Download Transparency, force them to announce their distribution to the log, which means the exposure of their resources.

Adversary influence. Due to the nature of an open platform, Download Transparency is exposed to adversary influence. The attacks the adversary performs falls largely into two categories. The first type of attack is called "Evasion". The attacker knows how the users utilize the log (e.g., decline if not seen in the log) and the data used to make those decisions (e.g., downloads reported), and utilize the pieces of

information to deliver its software successfully. The attacker can make a different attempt of "Poison" the dataset. It can be submitting false information regarding the distribution or reports that could lead users to download malware.

Research questions. The Download Transparency has potential benefits as we have discussed above. However, it is still unclear if these expectations will be valid even under different conditions, such as the different user's policies in utilizing the platform. We also have to explore the effect of the adversary influence. Provided that, we state the following research questions:

- **Q1. Can we propose realistic policies for utilizing the Download Transparency?** Various parties join the log and have different ways of using the information. For example, the users can submit certain records or not, utilize the log in multiple ways for making download decisions, and have a different threshold for malware. The security analytics can have a different opinion regarding a sample, i.e., malware or not. Additionally, not all software publishers pre-announce their distribution to the log. We aim for a realistic design of the entities and their policies.
- **Q2. How effective will the Download Transparency be for blocking malware propagation with different policies applied?** One of the benefits, Download Transparency can introduce, is the blockage of malware propagation. However, the effectiveness may chance when different policies are applied.
- **Q3. How much performance can Download Transparency have in the presence of adversary influence?** As we discussed, the platform is exposed

to adversaries. We need to understand the effect of the adversary influence in Download Transparency.

- **Q4. Can we propose methods that can mitigate adversary influence and how will it disturb adversaries?** After we investigate the impact the adversaries can bring to the platform, we explore the attacks that the Download Transparency can mitigate and those that are hard to suppress.

5.2 Goals and Non-goals

Goals. The goal is to propose the platform where software distribution is reported in a transparent manner. We also design realistic policies that capture the potential usage of the Download Transparency. Based on those policies we aim to measure the benefit it can introduce to security, from the perspective of disturbing the malware distribution.

Non-goals. We do not introduce the details for implementing the platform, which includes the specifics of the data submission protocol, the management of the data in the back-end, and how we assign unique identifiers to the participants. We also do not aim to propose a perfect policy that can disturb the malware distribution. The goal is to measure the effect of transparency assuming we have such a platform.

We organize the following chapters as follows. We describe the Download Transparency, including the components and the participating entities and their policies in Chapter 5.3. In the consecutive Chapter 5.4, we provide an empirical analysis of the download events in our dataset and the existing poison instances.

The next chapters discuss the methods we take for measuring the benefits of transparency. We first describe the methods in Chapter 5.5, and present the results in Chapter 5.6. We then dedicate Chapter 5.7 to illustrate the possible adversarial influence. Finally, we have a chapter for discussion (Chapter 5.8).

5.3 Download Transparency

We start by illustrating the design of Download Transparency. The goal of Download Transparency is to provide a data sharing platform for download events in the wild, where individuals act as sensors around the world and report the download events they observed. The platform operates as a transparent and append-only log for the download events so that anyone can audit the behavior of the log and monitor download activities listed in the log. Note that, the log itself does not provide any security guarantees, i.e., identify malicious download events, but it allows agents who are interested in the download events (e.g., security analytics) to perform computer security experiments freely on this representative field data, which could benefit the community.

5.3.1 Components

Download Transparency consists of two logs each backed with a Merkle tree. As illustrated in Figure 5.1, the two logs are the download activity log and the analytic result log. It also depicts six possible usages of the platform: (i) the software publishers announce their distribution to the download activity log in advance of the

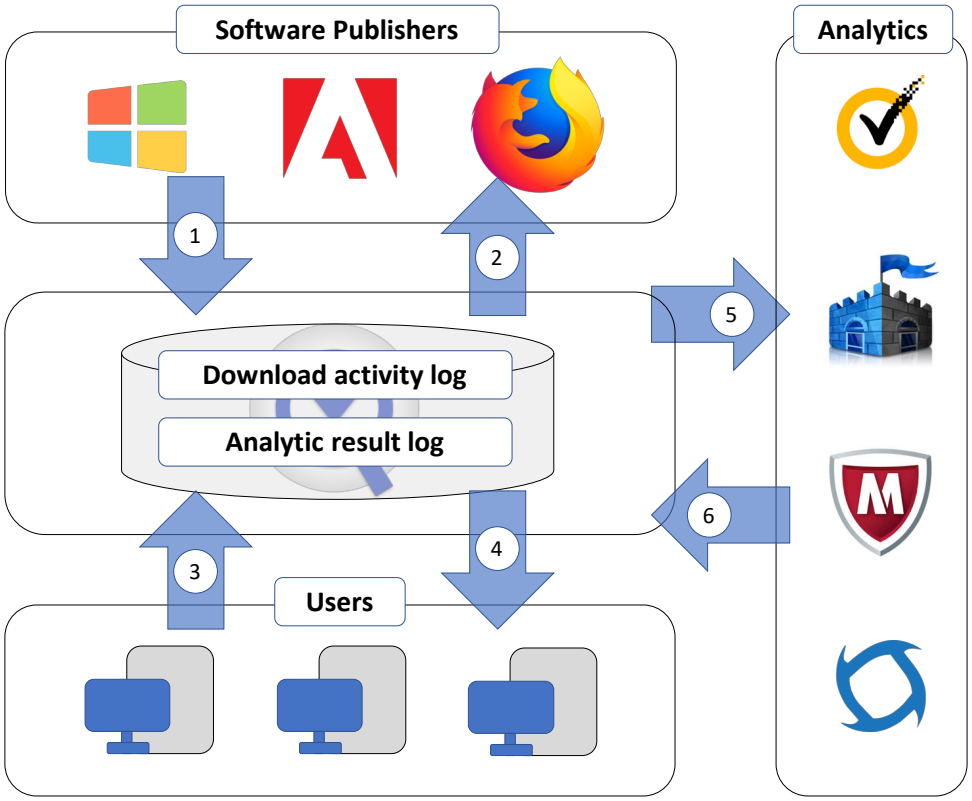

Figure 5.1: System diagram of Download Transparency.

dissemination, (ii) the software publishers monitor the logs to find any indicators of compromise, (iii) the users submit the download events they observed to the download activity log, (iv) the users refer to the analytic result log and the download activity log to make decisions of download, (v) the analytics use the download activity log to detect malicious download activities, and (vi) the analytics submit their findings to the analytic result DB.

Download activity log. The download activity log consists of a Merkle tree, which has the download events submitted from the users at the leaf. Each record implies the following information: *binary A was downloaded by binary B at time T through DNS domain C on machine D*. In some cases, it can be *binary B will download by binary A*, which is defined as "edge announcement". The details are

discussed at Chapter 5.3.2. Here, binary A indicates a payload and the binary B, which downloads A, is defined as a downloader. We represent both the payload and the downloader as the SHA256 hash value of the binary[1]. The DNS domain (C) is where the payload is hosted, for example, it can be the fully qualified domain name (FQDN). Every single record will also have the server-side submission timestamp (T), which is the time when the log server received the report.

Analytic result log. The analytic result log also consists of a Merkle tree. The record at the leaf has *binary A is detected as malicious by analytic B*.

5.3.2 Agents and Policies

The Download Transparency expects the participation from multiple parties. These parties (i.e., agents) include the users who submit the download activity reports to the log, the analytics who use the log to detect malicious software distribution and report the detections, the software publishers who announce their distribution (i.e., edge information consisting of a downloader and a payload) beforehand, and the monitors who check if the log is appropriately operating as in the certificate transparency.

5.3.2.1 Users

As mentioned previously, the log consists of the download events in the wild. We assume that these download events are submitted to the log by ordinary users

[1]In order to reduce the size of the log, we can use an identifier instead. In this case, it is necessary to have an ID–SHA256 mapping log that is also backed by a separate Merkle tree.

across the world. We also expect that these users can make their own decisions, e.g., which download records to submit, and how to use the information in the log for decision making (download or not). Based on these expectations, we design the policy model for these submission agents.

Submission policy. The users can freely submit the download events they observed in the field to the Download Transparency log. We assume there exists a submission policy these users stick to during the experiment. In other words, we do not consider the case of a user changing the policy in the middle of the observation period. We model the submission policy based on the edge (i.e., a downloader and a payload) recorded in the log.

- *Submit nothing (SN).* Users can decide not to submit their observation to the log. However, they can still check the log when deciding a download.
- *Submit only unseen edges (SUE).* A user will report only the new edges, which has not been reported to the log. We can assume the user, who wants to contribute to the log but wants to minimize the number of submissions, can set such a policy.
- *Submit only reported edges (SRE).* This is a policy very opposite to the one above. Users with this policy will only report the edges that are reported in the log. A possible justification for such behavior could be, they might feel much safer to report what others already have.
- *Submit everything (SE).* Users under this policy report every download events they observed.

Enforcement policy. Before we discuss how the users decide to accept a download,

we have to define how the user precept malware.

- *Threshold for malware (r_{mal}).* Users may consider a binary as malware if a certain portion of analytics flagged it as malware. We give a term r_{mal}, which is the rate of analytics flagged a binary as malware. For simplicity, a user can set r_{mal} from $\{0, 25, 50, 75\}$. $r_{mal} > 0$ implies he/she will consider a binary flagged as malware by at least one analytic as malware.

Download policy. Users can utilize, both the download activity log and the analytic result log from the Download Transparency, for deciding a download. For instance, by checking the existence of an edge in the download activity log, the user can block the unknown download. This policy encourages the software writers to report their software release to the log promptly. We discuss the pre-announcement of software distributions in Chapter 5.3.2.3. We design the download policy as follows:

- *Download everything (DE).* A user under this policy does not choose to utilize the transparency information at download.
- *Only accept reported edges (DR).* Some users might want to accept downloads from the Internet, only in the case that they are seen from somewhere else. Which could imply the software writer reported its release to the log or other users had accepted the corresponding download.
- *Only accept edges with no malicious reports (DN).* These users only take benefit of the detection results shared by the analytics. The download is blocked if the number of detection posted in the log exceeds the r_{mal}, for either the downloader

or the payload.

- *Only accept reported edges with no malicous records (DRN).* The most security-concerned user in our setting. Users under this policy check both logs and only perform a download, if and only if, others reported its existence and it has no known evidence of maliciousness.

5.3.2.2 Analytics

We expect the analytics to be interested in the download events in the wild. They can utilize the log to find out suspicious behaviors (e.g., malware distribution). The analytics can apply various methods for identifying the malicious software delivery. For example, an analytic can construct download graphs from the log, extract features, and train a classifier using their ground truth. Once the analytic detect malware, the hash of the binary will be reported to the log and be recorded with the report timestamp.

- *Malware definition* It is a well-known phenomenon that AV engines reach into different conclusions, i.e., malware or not, to reflect this in our analysis, we assume the malicious download event detection could vary among analytics. By allowing analytics to have different conclusions for a binary, we expect to observe how it affects the users.

5.3.2.3 Software Publishers

If the download transparency becomes prominent among their customer base, we believe the software publishers will have enough motivation to announce their distribution–the edges (i.e., a software updater will download a payload) to the log. Because not doing so might block the distribution at certain customers with download policy that blocks unseen downloads (e.g., DR and DRN). We model the aggressiveness of the announcement from only the edges of very benign publishers (i.e., a publisher in NSRL) to the edges of no detection history. We note that we only allow announcements for the downloaders that the publisher owns. The ownership can be proved by letting the publishers sign their announcement with their code-signing private key and append the signature to the edge. We discuss how this property can mitigate a part of the poisoning attacks in Chapter 5.7.2. In result, we introduce three different levels of an announcement: report edges where (1) the downloader is in NSRL (PN), (2) the downloader is signed (PS), and (3) the downloader is not malicious (PND). In case of generic downloaders, such as browsers or download managers, which cannot aggregate all the possible distributions in advance, we allow an edge submission stating the payload as '*' indicating the downloader can distribute any payload.

5.3.2.4 Monitors

Since we assume that the analytics' role is to identify malicious download activities, what remains for the monitors' role is to check if the log is operating

correctly. The monitor should do this by periodically downloading all the new download events that have been added to the log and checking the consistency.

5.4 Empirical Analysis of the Downloads

5.4.1 Data Summary

Download Activities. We utilize a large data set of download events from the downloader graph study in Chapter 4.1. We reconstructed these download events from observations on end hosts. In the previous study, we filtered out the events associated with browsers. We do not follow that filtering. Instead, we drop the hosts with less than 100 download records. From this data, we utilize the unique identifier of the machine, SHA2 hash of the downloader and the downloaded file (payload), the source domain of the download, and the timestamp of the event. Besides, we extract the information of the code signing certificates to simulate the edge announcements by the software publishers. Table 5.1 summarizes the data.

Ground Truth. We query all the hashes in the download activity dataset, which are above certain prevalence (i.e., number of machines a file appeared on), to VirusTotal. In result, we are able to retrieve reports on 909,281 binaries from VirusTotal.

Since we want to focus on malware and exclude gray-ware such as potentially unwanted programs (PUP), we use the AVClass [95] to filter out the PUPs. The AVClass tool, when provided with the VirusTotal reports, returns whether a binary is a PUP or not with the representative labels. If the AVClass says a binary is a PUP, then we filter it out from the malware ground truth. For the remaining malware,

Download Activites	
Download events	19.5 million
Total payloads	12.4 million
Domains accessed	64,545
Total hosts	48,709
NSRL downloaders	22,962
Signed downloaders	50,432
Not malicious downloaders	110,689
Ground Truth	
At least 1 detection	104,788
Above 25%	5,541
Above 50%	3,261
Above 75%	1,619
Benign	802,780

Table 5.1: Summary of the dataset.

we keep the first submission timestamp to VirusTotal (t_{vt}) and the number of total detection among the total AV engines utilized for the scanning (r_{mal}).

We use the results from VirusTotal to label benign files. We consider benign all the executables which VirusTotal report shows zero detection. We also list the resulting ground truth on Table 5.1 with the summary of the download activities.

5.4.2 Labeling the Edges with the Downloaders

We label the edges based on their downloaders to allow the edge announcement from the software publishers as described in Chapter 5.3.2.3 The first two categories e.g., *NSRL* and *Signed* require code-signing certificate information. We check the code-signing certificate information to label the samples to the software publisher groups. So, how do we collect the certificate information? For the binaries with VirusToal reports, we check the sigcheck field to see if the verified field indicates

valid or expired or revoked and to retrieve the publisher name. Those do not have VirusTotal report has to rely on the information from WINE: if it has both issuer and subject information, then it is considered to have a certificate. We also collect the name of the publisher. To identify the downloaders belonging to the *Not malicious* category, we utilize binary information from both VirusTotal and WINE. Once we obtain the necessary certificate information, we place the samples to the software publisher groups based on the following methods.

- *NSRL:* We extract the list of publishers from the NSRL RDS dataset [42]. The version of the RDS is at 2.52, released in April 2015. We utilize the python libraries `NLTK` [128] and `Cleanco` [129] to match the publisher names between the NSRL list and the code-signing certificates. The steps taken are: (1) clean the publisher name (i.e., remove legal terms and punctuation), (2) tokenize, and (3) compare the similarity.
- *Signed:* We assign this label to the binaries that have a certificate according to VirusTotal and WINE.
- *Not malicious:* We define a downloader to be not malicious if it has no information in VirusTotal or if it has less than 5% r_{mal}. We set the threshold for taking into account the false positives existing in the AV engines. We explore the downloaders that have at least one detection on VirusTotal and the number of benign payloads they distribute depicted in Figure 5.2. We observe that more than 80% of the benign payloads, which has been distributed by the flagged downloaders, are delivered by downloaders with $r_{mal} < 5\%$.

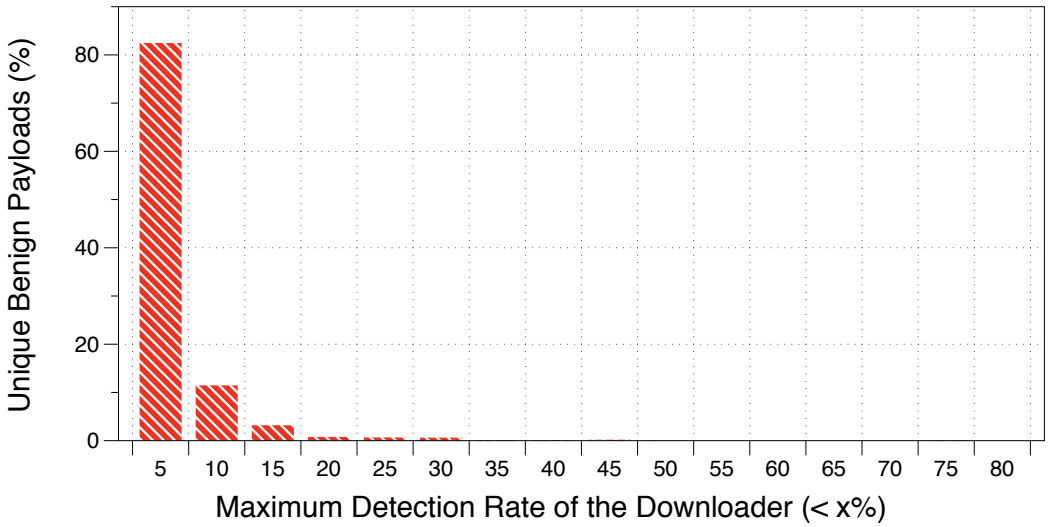

Figure 5.2: AV false positive decision: detection rate of the downloaders and the number of unique benign payloads they distribute.

Note that an edge can belong to multiple groups.

5.4.3 Characteristics of the Download Events (Edges)

We can illustrate a download as a directed edge of (downloader $\xrightarrow{attr}$ payload), which implies downloader delivered a payload. The edge can also have attributes, which describes the user who saw the edge, the source domain name where the payload originated from, and the time of the download. For this chapter, we ignore the attributes and take the unique combination of downloader and payload only, to investigate the properties of the edges. We illustrate the distribution of the number of edges per user in Figure 5.3. It shows a long-tail distribution, where a few numbers of users have more than 20,000 edges. The average number of edges is 349, and the median is 143.

Result of the edge labeling. Using the methods in Chapter 5.4.2, we attempt

to assign the publisher grouping to the edges and label them by their downloader and payload. We conduct the publisher grouping by using the information of the downloader. In result, the download events are assigned to the publisher groups and labeled by the downloader and payload as depicted in Table 5.2.

Existing poison instances. A majority of the download events in the dataset resulted from the benign downloaders and delivered non-malicious payloads (i.e., benign or has no VT record). However, we still observe a portion of download events that cannot be neglected, associated with malicious binaries. We revisit the findings regarding malware delivery shared in prior work [4, 23].

- *Abusive certificates:* Among 173,317 code-signed binaries labeled as NSRL, we observe 8,803 binaries at least one AV engine flagged it as malicious and not labeled as PUP. We investigate the ones with more than 25% detection rate. The most obvious and common type of abuse we see is the impersonation. These abusive certificates exploit the code-signing PKI in two ways: (1) request a certificate with a similar naming with a legitimate publisher, and (2) use a different CA from the one which the legitimate publisher uses. The publisher names are polymorphed by adding or removing the legal terms, for instance, `Adobe Systems Incorporated` $\xrightarrow{poly}$ `Adobe Systems` and `Microsoft` $\xrightarrow{ploy}$ `Microsoft Corporation`. Moreover, the abusive Microsoft certificate was issued by Ascertia Public CA whereas the other Microsoft certificates were from GlobalSign.
- *Benign downloaders as malware distribution channel:* We observe about 107,600 download events where benign downloaders deliver malicious binaries. For 50,773

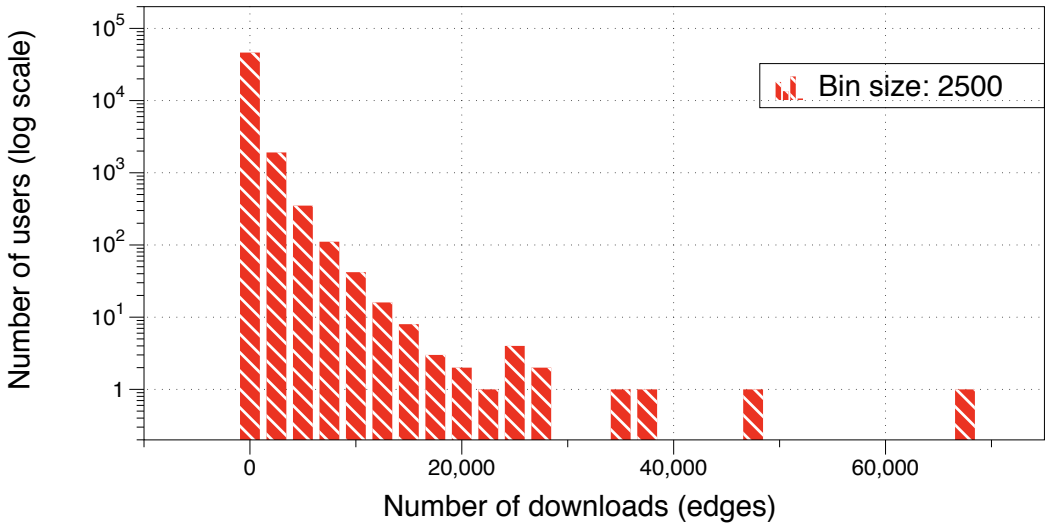

Figure 5.3: Number of edges per user.

		Label by downloader/payload								
		m/m	m/b	m/n	b/m	b/b	b/n	n/m	n/b	n/n
Publisher Group	NSRL	6.5K	23K	47K	51.3K	467.6K	5.6M	563	10K	3.4K
	Signed	12.9K	57.9K	54.7K	60.6K	544.2K	5.6M	3.6K	24.7K	13.2K
	Not malicious	18.2K	112K	525.3K	108K	1.3M	11.1M	15.7K	136.8K	67.5K
	None	2.4K	8.1K	3.1K	0	0	0	839	86	18

Table 5.2: Download event labeling: m,b,n stands for malicious (has at least one detection), benign, and no ground truth.

downloads, the downloaders are code-signed by the NSRL publisher. There are 3,090 NSRL downloaders involved, and they are the generic downloaders (e.g., browsers) that are most frequently abused. We see instant messengers such as `Skype` and the remote control products like `TeamViewer` on the top of the list.

5.5 Experimental Methods

5.5.1 Evaluation Metrics

As we discussed in Chapter 5.3.2, we expect that the users will exploit the log under different policies in the submission of the download information, the decision

		Label by downloader/payload								
		m/m	m/b	m/n	b/m	b/b	b/n	n/m	n/b	n/n
Publisher Group	NSRL	256	21.9K	47.2K	44.4K	467.6K	5.6M	5	10K	3.4K
	Signed	588	47.4K	53.1K	47.3K	544.2K	5.6M	3.6K	24.7K	13.2K
	Not malicious	806	101.5K	523.7K	80.3K	1.3M	11.1M	989	136.8K	67.5K
	None	19.7K	18.7K	4.7K	27.8K	0	0	14.8K	0	0

Table 5.3: Download event labeling with defense assumptions: m,b,n stands for malicious (has at least one detection), benign, and no ground truth.

on which information to check at the time of download, and the tolerance in malicious reports. Here we present the evaluation metrics that will help us capture how much the users can benefit from the log under different policy settings.

Propagation block performance. The first metric aims to measure how effective the policies are regarding the blockage of incoming malware. We use the Matthews correlation coefficient (MCC)[2], which takes into account of both the true/false positives and negatives. The positive and negative corresponds to malware and benign, respectively. We chose this metric over the F1 score, which is another widely used metric that considers the precision and recall of the positive since the MCC is known to give a balanced measure in an imbalanced positive/negative dataset. The MCC ranges from -1 to +1, where +1 indicates perfect performance.

Early propagation block. Besides, we also evaluate how early we can block the malware propagation that is previously unknown. We define "early propagation block" as "it can block unknown malicious executables before their first submission to VirusTotal". Based on these definitions, we estimate the propagation block lead time introduced by exploiting the download transparency log.

Performance of the detector. When we utilize the downloader graph detector,

[2] $(TP * TN - FP * FN)/\sqrt{(P) * (TP + FN) * (TN + FP) * (N)}$

we measure how effective a malware is detected. The two above metrics are still employed, but instead of evaluating the "block", we focus on "detection". In other words, we evaluate the detection performance with MCC and the early detection (i.e., it can detect unknown malicious binaries before their first submission to VT).

5.5.2 Measuring the Benefit of Download Transparency

We measure the benefit of introducing download transparency in security under different policies. We evaluate the performance two ways, which includes, (1) a simulation where we assign a single policy to every user and (2) simulation with a single policy but with the downloader graph detector.

5.5.2.1 Simulating the Agents

We first describe how we populate information for the agents.

Users. We assign a set of policies (download policy, submission policy, enforcement policy) to each user. A user will decide weather download a binary or submit the download record to the log based on the given policy. For example, if a user with policy (DRN, SRE, 25) will check both the download record log and the detection log to see if the download has been reported before (DRN policy) and if not more than 25% of the analytics labeled the payload as malicious (25 policy). Once the user accepts the download based on those conditions, now the user reports the download edge to the download record log since the edge has been reported before (SRE policy).

Analytics. We use the information from VirusTotal to simulate the analytics. In case we have a scanning report of a binary A from VirusTotal, which is labeled as malware by 25 AV engines out of 62 and is submitted on 01 July 2013, then we populate this information to the detection log as 25 AV engines out of 62 reported the binary as malicious at 01 July 2013.

Software publishers. As mentioned in Chapter 5.3.2.3, the software publishers announce their edges to the download activity log before the distribution.

5.5.2.2 Simulations

Here, we describe the detail of the three simulations mentioned at the beginning of the chapter. We expect to observe the result of having different policies:

- *Effect of download policy.* We measure how much can the users benefit from utilizing the information shared through transparency. Specifically, we focus on how the download policy can help to prevent the propagation of malicious software.
- *Effect of submission policy.* The download transparency is built on top of the participation from the users, which implies the benefits the users can get also depends on the submissions. Throughout the experiment, we will observe the how the submission can affect the block performance.
- *Effect of tolerance in malware (enforcement policy).* In our model, we let not only the analytics but also the users to decide what they think is harmful to them. These difference in the tolerance can affect both the submission and the propagation block.

- *Effect of edge announcement.* The download policy can be though as a simple unsupervised decision with two features: is the edge is in the download record log and is the payload in the detection log, which implies the chance of having false positives (i.e., download of a benign file gets blocked). Therefore, we allow the software publishers to announce their distribution beforehand, which can act similar to a white-list. We will investigate the importance of the edge announcement for reducing the false positive, which will motivate the software publishers for releasing their distribution.

Full policy deployment. This simulation will have a set of policy assigned to all users. The control parameters will, therefore, be the policy set and the edge announcement (download policy, submission policy, enforcement policy, edge announcement). We exclude the case where submission policy is "download everything" since we cannot measure the propagation block. Therefore, we have 144 simulations with two output metrics, in total 288 data points.

Utilizing the downloader graph detector. We train a random-forest classifier using influence graphs introduced in prior work [23]. We utilize the same random-forest classifier with the features including graph structural features, source URL features, graph growth features, and globally aggregated features. We implement the detector in an online-fashion, which considers of the frequency of detection (i.e., run the classifier on the unknown influence graphs) and the periodicity of the model retraining (i.e., update the model with the new labeled influence graphs). The detail of how we manage the detector is discussed in Chapter 5.5.3.

With and without adversary influence. We experiment with two different conditions. At first, we reduce the poison instances discussed in Chapter 5.4.3 and run the simulations. The details on the reduction are stated in Chapter 5.5.4. We then utilize the real-world data that contains poison instances introduced in Chapter 5.4.3 and examine the effect.

5.5.3 Online Detection of Malicious Downloader Graphs

The prior work [23] introduced the downloader graph detection algorithm in offline mode; and conducted both the training and the testing on the graphs constructed with the entire five years of data. In this paper, we utilize the detection algorithm in online mode, where the graph keeps growing, and the unknown graphs are added as time goes. We allow this chapter to illustrate some parameter decisions in adapting to the online mode and other specifications of the online simulation including the graph labeling.

Classifier. We use the random forest classifier with $N_t = 200, N_f = sqrt$ (# of features), $N_d = 20, N_s = 10$, and $N_l = 4$. We also adjust the weight of the samples based on their class (e.g., malicious or benign) frequencies. Each of these parameters stands for the number of decision trees, the number of features per decision trees, the depth of the decision tree, the minimum samples for a split in the tree, and the minimum samples in the leaf of the tree, respectively. We decide the parameter by testing the performance of the classifier on the training set of the graphs built before 2013 and the graphs generated in the year 2013 as the testing set. We attempted

a 1000 random set of parameters and selected the set of parameters with the best performance.

Detection frequency. How frequently should the analytics perform the detection? The algorithm consists of features that depend on the time. For example, the globally aggregated features might require some time to populate in order to reach a sufficient amount for discrimination. However, if we perform the detection in a too long interval, it will result in late detection. Also, the graph growth features and graph structure features need the graphs to mature enough. Therefore, we try to find the best and shortest detection frequency. We try different frequency (3 days, 7 days, 14 days, 30 days, and 90 days) and evaluate the MCC and early detection. Considering the running time of each simulation, the detection performance, and the early detection rate, we decided to use the 30-day interval. Note that, we considered the running time since this is a simulation of 5 years of data. For practical use, we can use a shorter interval for better performance.

Retrain frequency. We retrain the classifier with the updated labels from the previous detection. For simplicity of the simulation, we set the retrain interval identical to that of the detection interval (30 days). Therefore, every time after we perform detection on the unknown graphs, the resulting labeled graphs are added to the ground truth and used as the training set. We utilize the retrained model for the next detection.

Online detection. We simulate the online detection with the above settings. Since we need a trained model to initiate the simulation, we construct the influence graphs

from the download events before the year 2013 and label them with the analytic reports also until the end of the year 2012. Note that, we do not report these edges to the log. We only use them to generate the influence graphs. From the year 2013, users report and block the downloads based on the assigned policies. For the simplicity of the experiment, we assume all the analytics use the same model with the same ground truth. Therefore, the result from the downloader graph detector is interpreted as 100% r_{mal}. We illustrate the specifics of the method like the following:

- *Labeling the influence graphs.* We first have to define the node as malicious or benign or unknown. As we discovered that binaries with less than 5% detection rate might be false positives in Chapter 5.4.2, we say a node is malicious if it has $r_{mal} \geq 5\%$. The node is benign if it has zero detection from VirusTotal. Note that when we consider no adversary influence, we add the downloaders of the announced edges to the benign ground truth. The rest are considered unknown. After we label the nodes, we label the influence graphs. We label the influence graphs as malicious, benign, mixed, and unknown. An influence graph that has a malicious node and the root is not benign is labeled as malicious. If the root is benign and none of the nodes are malicious, then it is labeled as benign. In case the root is benign but has a malicious node in the graph, we label it as mixed and ignore at training. The rest of the influence graphs, which has no information about the nodes in the graph, are labeled as unknown and used at detection.
- *Dealing with the detections.* We apply the trained classifier to the unknown influ-

ence graphs and label it as malicious or benign. We report all the nodes in the malicious influence graphs to the detection log with the timestamp. We add these nodes to the malware ground truth for the next training. In case of the benign detection, we only add the root node to the ground truth. Once we complete the report and the ground truth update, we retrain the classifier with the new labels.

For the online detector, we only apply variation to the download policy and submission policy. Since we only rely on the single detector, we cannot manipulate the enforcement policy. In case of the edge announcement, we expect the implication is similar (i.e., the change in the number of ground truth) with the previous simulation in Chapter 5.6.1.1.

5.5.4 Reducing the Poison Instances

Since the original dataset contains poison instances as described in Chapter 5.4.3, we have to exclude these edges from being announced when we simulate the case without adversary influence. We assume the situation where the adversary decided not to submit information regarding their distribution to the log, considering the risk of being revealed to the public early. Therefore, we will not see the distributions utilizing abusive certificates in the list of announcements. For the malware delivered through benign downloaders, we eliminate ones from benign software updater, due to the same reason. However, we leave the downloads via generic downloaders. We apply a heuristic to determine generic downloaders by looking at their number of payloads they deliver and the domains they access. We say a downloader

is generic if it utilizes more than or equal to 5 distinct effective domain names to deliver payloads and the number of unique payloads distributed by a single effective domain name should be between 0.5 and 2. We made this decision based on the intuition that it will have a more significant number of domains and less number of dedicated payload hosting compared to a software update. As a result, we acquire a refined edge labeling for the software publisher edge announcement, depicted on Table 5.3.

5.6 Experimental Results

By performing a series of simulations, we want to answer to the main research question of the paper "the benefit of transparency in security". Based on the methods we present in Chapter 5.5.2, we perform a quantitative evaluation of the benefit of the Download Transparency from the perspective of obstructing the spread of malware. We begin with the result of the download events with reduced adversarial instances. The result consists of simulations with two different scenarios assuming (1) full deployment of the policies and (2) online downloader graph detector with full policy deployment. We then explore the effect of the poison instances.

5.6.1 Measuring the Benefit

We present the benefit of the Download Transparency in terms of blocking the malware propagation associated with the realistic policies introduced in Chapter 5.3.2. For the experiments in this chapter, we use the download events *with the*

	MCC		Early block rate	
Policy	t	P >$\|t\|$	t	P >$\|t\|$
Download	-12.14	0.000	17.98	0.000
Submission	0.03	0.976	-0.29	0.771
Enforcement	0.61	0.541	-6.89	0.000
Announcement	4.35	0.000	-4.25	0.000

Table 5.4: Regression result of full policy deployment.

poison instances reduced as illustrated in Chapter 5.5.4. We initiate the simulation with the full policy deployment, to take a look at the pure effect of each policy on disturbing the malware delivery. We then present the results with the online downloader graph detector. We note that the target of evaluation differs from the downloader graph analytic in Chapter 4.1. In the previous work, we evaluated the detection performance based on the graph or the root downloader. In other words, if the root of the influence graph is malicious, it is a true positive detection. On the other hand, for the current evaluation, we target the detection of nodes, i.e., files in the influence graph. Therefore, the performance might differ from the previous work. The detail of the methods can be referred from Chapter 5.5.2.

5.6.1.1 Full Policy Deployment

We present the result of the initial experiment, which we conduct assuming the same policy applies to all the users. As stated in Chapter 5.5.2.2, we explore the correlation of each policy, when solely applied, for disturbing the malware distribution. We interpret the result as follows. First, we perform a regression between the controlled parameters (policies) and the output (the propagation block performance

as MCC and the early block rate). Then, we present the result of the regression with the t value and P value. We first define the null hypothesis as the parameter does not affect the output. We reject the null hypothesis when we have a low P value (less than 0.05). In other words, the parameter has a significant effect on the output. The t value is presented to show if the estimated trend is positive or negative. We depict the result of the regression at Table 5.4.

Propagation block performance. The performance of the propagation block is heavily dependent on the download policy. We present the result in Figure 5.4(a). When we utilize the download activity log by making download decisions, the performance decreases due to (1) the decrease in true positive and (2) the increase in false positive. The decrease in true positive is due to the malware delivered by benign generic downloaders, which cannot be blocked by the 'DR' policy. The next issue results from the malicious binaries that drop benign files. The false positive is similar for all download policies when the threshold for enforcement is low ($r_{mal} > 0$). However, the false positive gets significantly lower when we rely on the analytic result log when r_{mal} is set higher than 0. The result implies a large portion of downloaders, which are detected as malicious and distributing benign files, may be false positives of the AV engines. Which also suggests that in the ideal scenario, where all the benign publishers announce their distribution to the log, the actual blocking performance will be higher for the downloader policies that check the activity log. The result of the regression supports the positive effect of the announcement. The announcement has a low p-value, which means the policy has strong statistical

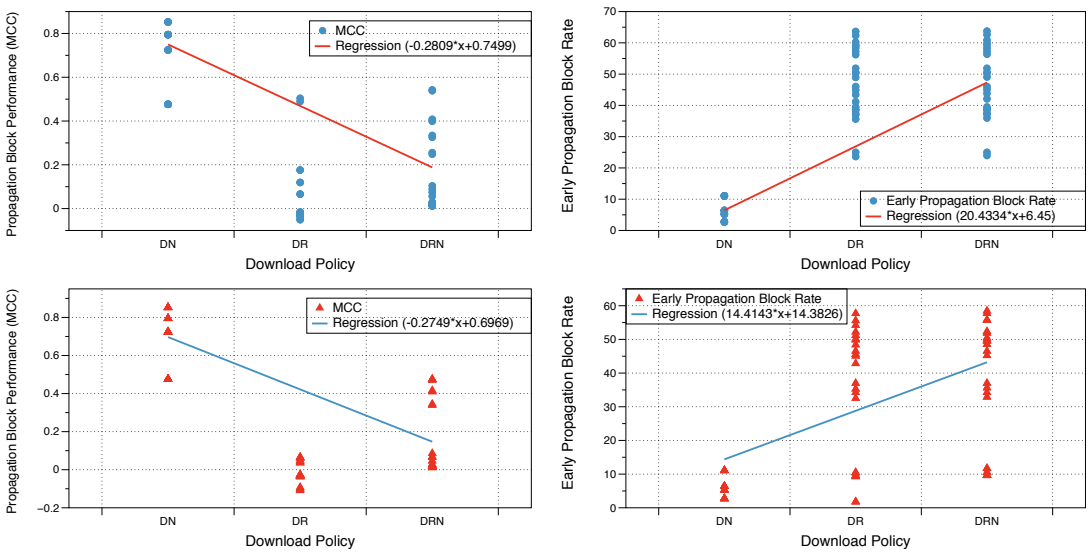

Figure 5.4: Performance and regression of full-policy deployment: without adversary (a) propagation block performance and (b) early propagation block rate, and with adversary (c) propagation block performance and (d) early propagation block rate.

meaning on the performance; it also shows a positive trend on the performance.

Early propagation block. The download policy strongly contributes to the early propagation block as well in a positive trend. The result is depicted in Figure 5.4(b). Employing the download activity log at making download decisions gives about 63.75% early detection rate at best, whereas only 10% or less malware is blocked earlier when only checking the analytic result log. The enforcement and announcement explain the result well in a negative trend, which may imply the effort of the security community for detecting highly malicious samples. The early detection rate decreases for the announcement due to the increased number of malware from the generic downloaders.

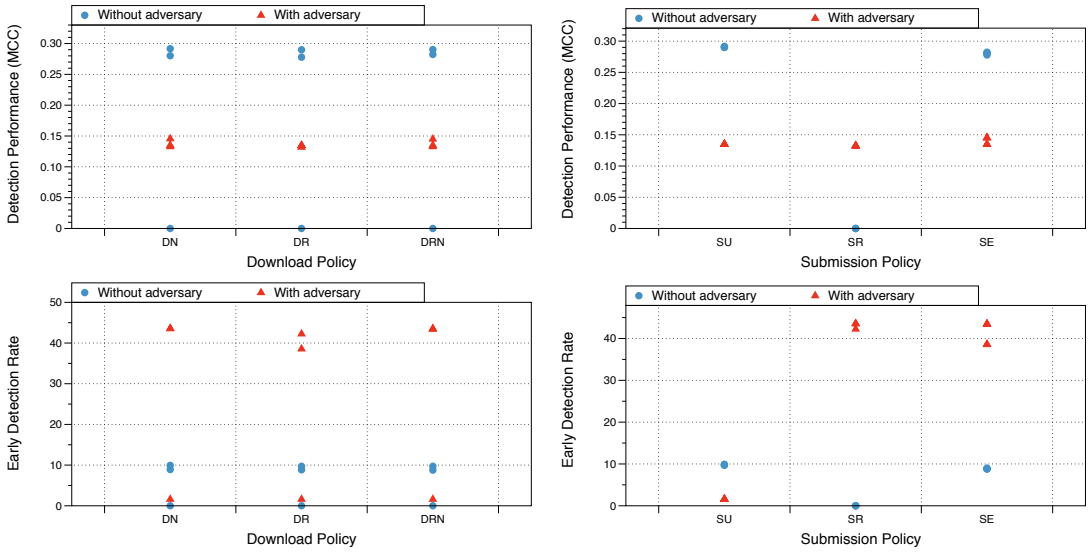

Figure 5.5: Performance of the online downloader graph detector: (a) detection performance v.s. download policy, (b) detection performance v.s. submission policy, (c) early detection rate v.s. download policy, and (d) early detection rate v.s. submission policy (blue circle for without adversary and red triangle for with adversary).

5.6.1.2 Online Downloader Graph Detector

We depict the performance of the online detector in Fig. 5.5. Note that we present results of both with and without adversarial influence in the same figure. We mark the current result (without adversarial influence) with blue circles. Both the detection performance and early detection show minor change due to the download policy. We get zero positive detection when we apply submission policy 'SR' (i.e., submit only reported edges). Since most of the malicious downloads get blocked and only reported edges are submitted, not enough malicious graphs remain for detection. Besides this single case, submission policy also shows the weak effect to the detection performance.

5.6.1.3 Summary of the Result

We can state the takeaway of the result can as: (1) Download Transparency can help to block a significant part of malware distribution before it is flagged as malicious. However, it introduces false positives and (2) the aggressive edge announcement can help to reduce the false positive. The submission policy shows no significant effect compared to the other policies. One reason could be, the high ratio of singletons. In the real-world download record dataset we use for this experiment, only 9% (1.15 million out of 12.4 million) of the binaries have an appearance more than once. In other words, 91% of the binaries had no chance of being utilized as regardless of the information has been shared or not. Which also could be one of the factors that decreased the performance by the download policies relying on the download activity log.

We can interpret the result from the online detector as the policies seem to have no significant effect on the detection performance when we have a trustable ground truth.

5.6.2 Impact of the Adversarial Influence

We now perform the same simulation in Chapter 5.6.1, this time with adversarial influence. We present the results and try to analyze the impact of the poison instances on the performance.

Impact to the policies. We begin with the simulation with the full policy deployment. We present the regression result on Table 5.5. The trend is similar to

	MCC		Early block rate	
Policy	t	P $>\|t\|$	t	P $>\|t\|$
Download	-10.91	0.000	11.41	0.000
Submission	0.04	0.972	-0.19	0.851
Enforcement	1.97	0.050	-3.31	0.001
Announcement	3.33	0.001	-10.60	0.000

Table 5.5: Regression result of full policy deployment with adversary influence.

the result of without adversarial influence; however, we can see the difference in the t value. Since the download policy fits the regression best among the other policies and we expect it will be most affected by the adversarial influence, we try to examine the impact by comparing the results of the download policy. For the comparison, we depict the result of the adversarial case in Figure 5.4(c),(d). We can see a slight decrease in the overall performance in both MCC and early block rate. Also, we observe several points with a huge decrease in the performance of download policies that consider the download activity logs. Which can be explained by the existence of the poisonous instances in the announcement of NSRL or Signed edges, which caused high false negatives (i.e., malware not blocked). However, not all of the data points are affected, since the detection rate these poisonous samples reside is mostly below 75% and the 'Not malicious' announcement is less affected by the types of poisoning we have.

Impact to the online downloader graph detector. As we mentioned in Chapter 5.6.1.2, we depict the performance of the online detector in Fig. 5.5 and we use the red triangles to mark the result with adversarial influence. Similar with the case without adversarial influence, the detection performance seems not to be affected

significantly by the download policy and even by the submission policy. However, we see a considerable drop in the early detection rate when the submission policy is 'SU' (i.e., submit only unknown edges). The reason is the malware included in the announcement cannot be detected since they are not reported.

The overall detection performance decrease by about 0.15. It is because when we have better ground truth for the case of without poison instances. However, the early detection rate is much higher (about 30%). We can provide two explanation: (1) we allow more malware to get downloaded in the current simulation, which also provides a better opportunity for the detector to detect more malware and (2) malware involved in these poisonous edges remain undetected for a long time in the wild.

5.7 Adversary Influences and Defenses

Due to the nature of a public platform, the Download Transparency is also inevitable from adversary influence. We illustrate the possible attacks against the Download Transparency and discuss possible countermeasures with assumptions. We set the potential targets of these adversaries as (1) the propagation block spring from the download policy and (2) the machine learning models based on prevalence features like the downloader graph classifier. For the prevalence feature, we assume the model tends to detect as malicious if it has low prevalence as reported by prior work [23]. Then we discuss what benefits can be introduced by Download Transparency with these assumptions.

5.7.1 Evasion Attack

We place a situation where the adversaries already know how the propagation is blocked by the download policies and about the prevalence features. Under the evasion attack scenario, the goal of the adversary is to make its malware successfully land on the victim without being blocked. A large part of the evasion attacks against the target defense mechanisms must accompany with the poisoning attempts. For example, to bypass the download policy, the adversary should announce their distribution to the log or to fool the prevalence features fake records to increase the prevalence of the malware should be submitted to the log, which all can be considered as a type of poisoning attacks although the purpose is on evading the propagation block. Therefore, we discuss in details about the attacks above later where we talk about the poisoning attacks. A pure evasion strategy the attacker can come up with is to distribute the malware through generic downloaders exploiting the fact that it does not need to specify the payload. However, it still exposes the downloader, which is also a valuable resource for the adversaries, to the security community and provides the opportunity to identify the threat of these downloaders and come up with the countermeasure earlier. They can use the benign generic downloaders to distribute malware. The log itself cannot provide a defense against such evasion attempt. However, we claim that benign generic downloaders such as browsers have already been widely used as an attack vector and introducing the Download Transparency does not deteriorate the current state. Instead, the Download Transparency provides an opportunity for identifying the distribution earlier

with the help of users who submit the download records to the log.

5.7.2 Poisoning Attack

As we already discussed while exploring the evasion attacks in Chapter 5.7.1, the poisoning attack in the Download Transparency indicates attempts to make malware bypass the block by submitting reports that poisons the platform. The poisoning attack can be achieved in high-level as follows: poisonous edge submission (1) to mimic the benign distribution or (2) to pollute the benign distributions.

Mimic the benign distribution. The attacker who knows the features that block the malware propagation (i.e., block unseen edge or block binaries within a low prevalence download graph), can take two strategies.

First, as the benign software publishers announce their edges to the transparency log, the adversary can also announce the edges in advance. It may help the initial propagation however it implies exposing the malware and providing the opportunity to the security community to identify the threat and come up with the countermeasure earlier. In case the malware is signed, which is a well-known technique the malware writers use to bypass the detection, it even exposes the certificate to the risk of revocation. We have observed in the study of code-signing abuse [4] that obtaining the code-signing certificates are challenging. However, by introducing the Download Transparency with the "download based on the transparency log" policy, the malicious distributor has to append the information to the log, revealing its information. Such behavior is beneficial to the security community

for detecting abusive/compromised certificates and initiates a prompt and efficient revocation, which implies the Download Transparency combined with the existing security mechanisms such as code-signing could let malware writers face a dilemma of: *the increase in the cost of mimicking the benign standards.*

Now, we can imagine about an attacker trying to exploit the known features such as the prevalence by submitting a large number of download records corresponding to its malware distribution. We may adopt a reputation system for the users and the downloaders and combine with the ideas from the collaborative platforms such as Wikipedia [130]. For instance, we can provide an alert for the submission volume increase for downloaders with low reputation. The security analytics may investigate these surges and report the poisoning attempts. We may also limit the attacker's ability by placing a delay on the reports for unknown downloaders and a delay for users with low reputation, which will require the attackers spend resources to build up the reputation.

Pollute the benign distribution. Attackers may also try to damage the reputation of the benign distribution by submitting false information to the log. For example, an attacker with malicious intent could submit information such as downloader A downloads C (which is not true) or a malicious downloader B is signed by a particular key (which is not true either).We can mitigate such attacks by introducing a cryptographic guarantee at the announcement. By allowing the publishers to sign their software distribution record with their code-signing private key and append the signature to the announcement (e.g., dlr, pld, Sig(dlr,pld)), the ownership can

be proved with the code-signing PKI. Either the log can only accept the record with a valid signature, or we can detect the submitted fake information by verifying the signature.

5.8 Discussion

5.8.1 Complement of the Code Signing PKI

We observed the breaches of trust in code-signing PKI and the failure of the revocation in Chapter 3. We also identified that the primary cause of these weaknesses in the code-signing PKI is due to the opaque ecosystem, where it is hard to identify what files are signed by the code-signing key. We believe the Download Transparency can complement the code-signing PKI by introducing transparency to the files carrying the valid digital signatures. As we discussed in Chapter 5.7.2, we can allow the software distribution announcements from the publishers 5.3.2.3 to append a signature of the *downloader–payload* pair generated with the code-signing key of the publisher. Note that the code-signing key should be the same as the one used to sign the downloader, which implies the owner of the downloader submits the record. To allow the announcement of the signed payloads, which does not have a download functionality, the publisher may submit a digitally signed record of *payload–none*.

Such software distribution announcements provide an additional property to the code-signing PKI, which is *the publisher is aware of the signed binary*. Moreover, since the information of the digitally signed binaries is recorded in the public log, we

can identify the abuse and revoke the certificate effectively when a breach happens (e.g., malware signed with the compromised code-signing key).

5.8.2 Privacy Concerns

Although we can introduce anonymization schemes to users, such as hashing the user's identification, an adversary may still be able to utilize the information in the log to determine the user's identity. For example, (1) the user's location can be identified by the submission timestamp or due to the software localization or (2) the user's occupation can be referred by the software he/she has. Moreover, the adversary might utilize the information to find the vulnerable population based on the software they have or the domains they visit and plan the attack. A possible solution is to adopt differential privacy [131] to the data submission from the users, so that the log can still provide the aggregated information such as the prevalence of a binary while preserving the privacy.

Blocking user submission. Another option might be not allowing submission from users. The evaluation of the policies in Chapter 5.6 suggest the contribution from the submission policies are not critical compared to the other policies such as the download policy or the announcement. Therefore, a valid option could be only utilizing the log where the software publishers make their distribution transparent; still, the users can benefit from the log by blocking the propagation of malware in its early stage. However, the detectors that utilize user telemetry data such as the downloader graph will be hard to be deployed.

Chapter 6: Conclusion

In this dissertation, we conducted studies on malware distribution. We take the approach in three directions. We begin with several measurement studies to understand the malware distribution, explicitly focusing on the digitally signed malware. We then develop a couple of algorithms for detecting malware distribution and share the insights of malware delivery. Lastly, we propose a transparency platform for software distribution that can prevent malware propagation.

Understanding malware distribution. The measurements provided unique insights to understand the malware delivery. It showed that various types of malware often infiltrate benign software ecosystems to be stealthy. The finding motivated us to build the detection algorithm based on sub-graphs rooted in each downloader, rather than the full downloader graph, which we discussed in Chapter 4.1. We also examined the abuse in code signing and the effectiveness of the revocation process. We identified various abuse that exploited the weakness in the code-signing PKI and revealed that the revocation is not done correctly in many cases. These issues are due to the challenges in monitoring the code-signing PKI ecosystem. The opaque ecosystem is making the responsible actors underestimate the security implications that result from the mismanagement. This lesson extends to the study

on transparency, which is discussed in Chapter 5.

Detecting malware distribution. We developed two detection algorithms for malware distribution. In the first study, we proposed abstracts defined as the downloader graphs and the influence graphs, which captures the relationships between downloaders and the payloads. By analyzing the downloader graphs in the wild, we designed features that can differentiate between the malicious and benign download activity. The classifier builds upon these features show their effectiveness in terms of precisely discriminating malicious from benign, and introducing a way to detect malware distribution earlier than the existing anti-virus systems. Then we introduce Beewolf, a system for systematically detecting silent delivery campaigns. With Beewolf, we identify the silent delivery campaigns conducted in the year of 2013. These campaigns provide novel findings regarding the malware distribution, such as the overlap between the PUP and malware delivery ecosystems and the business relationships among PPI providers, which may remain hidden. The studies on download graph analytics and lockstep detection highlight the importance of global analysis in detecting malicious software distribution. A transparency platform for download events could be an excellent environment to conduct such global analysis.

Blocking malware distribution. In the last part of the dissertation, we proposed Download Transparency, a platform where we can publicize software distribution with guarantees of integrity. We studied the benefits by introducing such platform to the software distribution ecosystem. The first possible benefit could be disturbing malware distribution. To measure the performance, we designed the participants and

the policies they might take when utilizing the platform. We then simulate different policies with five years of download events and measure the block performance. The results suggest that the Download Transparency can help to block unknown malware distribution in advance. However, it comes with false positive due to the corruptive nature of the software distribution (e.g., benign software distributing malware and distributed by malware). We can reduce the false positives by having benign software publishers announce their distribution in advance to the log. We also investigated the effect of a machine learning system such as the malicious downloader graph detector. The detection performance did not show a significant change depending on the policies. We also investigated the effect of the adversarial influences and proposed possible mitigation for different types of attacks. For example, we can enforce the digital signatures to be submitted along with the software distribution announcement. These mitigations suggest the potential of Download Transparency as the complement of code signing PKI. Such enforcement disturbs the malware writers mimicking the benign software distribution such as utilizing code signing certificates. They now have to expect their digitally signed malware have a shorter validity, which implies the cost of the attack will increase to achieve the similar impact. It adds another benefit of introducing transparency.

6.1 Future Directions

In this section, we discuss some potential topics that could extend our study.

6.1.1 Improvements in the Code Signing PKI

The findings in Chapter 3 suggest the current revocation systems based on CRLs and OCSP are facing several problems including (1) revocation delay, (2) inaccurate revocation dates, (3) availability of revocation information, and (4) invalidated benign files by the revocation. We describe several necessary properties for an effective code signing PKI and how a new design could address the current problems in revocation.

Design goals. We desire the following properties:

- *Publicize certificate issuing.* The code signing certificate issuing should be open to the public to prevent certificate issuance for a publisher without the publishers knowing it.
- *Publicize signed binaries.* The new system should help the owner of the certificates to monitor which binaries are signed by their certificates.
- *Pin-pointed revocation.* The proposed system should support the targeted invalidation for a signing binary or targeted whitelisting to revoke all except a specific subset so that we can set a revocation date with a finely controlled impact on benign files.
- *Better availability.* The system should have better availability and better failure modes on the client that do not negatively impact users but allow the system to work as intended.

One possible direction could be introducing transparency in the code signing certificates as in the TLS certificate transparency or as we discussed in Chap-

ter 5.8.1, we can use the software distribution transparency such as the Download Transparency as a way to monitor the digitally signed binaries.

6.1.2 Improvements in the Downloader Graph Detector

Refine the labeling. We train the downloader graph classifier with the labeling method based on the root downloader of the influence graph. The two different evaluations in Chapter 4.1 and Chapter 5, which used the downloader and the payload as the target, respectively, suggests that the performance significantly decreases in the latter case. The difference in the performance might be due to the difference in the conditions such as 1) including the downloads from the browsers, 2) distinguishing PUP from malware, and 3) online-learning. However, the most significant factor is the chaotic nature of download graphs, i.e., malware download benign software and benign downloaders deliver malware. Therefore, it is critical to come up with a better method for labeling the graphs.

Cross-machine influence graph features. We explored the influence graphs across different hosts. There we employed globally aggregated features and observed the high discrimination power of them. However, these cross-machine features focused only on the nodes, and we did not pay enough attention to influence graphs. The influence graphs that share the same root downloader or that has a similar pattern might be able to provide us with essential insights of malware distribution. For example, the different shape among the influence graphs with the same root may indicate a failure/success of some delivery mechanism or a targeted distribu-

tion strategy. A feature that captures such information might be useful in detecting graphs of malicious distribution. Moreover, studying the graph patterns may reveal a way to come up with an unsupervised algorithm for identifying specific distribution strategies, such as pay-per-install or botnets.

Differential privacy. We relied on a centralized setting, where all the download event information is available either transparently or not. However, what if there is no such environment and all the graphs are kept on each users machine? To make users willing to share such information, we need a method that can guarantee privacy. As we discussed in Chapter 5.8.2, applying differential privacy to the downloader graph might be a possible direction for this problem.

6.1.3 A Better Data Sharing Platform for Security Research

We take this section as an opportunity to overview the current data collection sources found in the cybersecurity research, and we introduce the common challenges in conducting research when utilizing these data sources. Those challenges then lead to several properties a new platform for data exchange should meet.

6.1.3.1 Data Sources in Cybetsecurity Research

Private data. VirusTotal [41] is one of the most widely used sources of private data both in academy and industry. They provide various information about the malware including the detection rate by different anti-virus engines and the behavioral report from the sandbox. In some cases, research is conducted on the data from industry

partners, who shares it with the academics under a non-disclosure agreement. For example, the WINE [40] dataset provided by Symantec corresponds to this case.

Public data. Several repositories share the data for free without limiting the scope of access. A public data repository such as VirusShare [132] hosts the malware samples. Sometimes the research institutes share the malware analysis report to the public [133]. However, often the analysis report has to be acquired by utilizing the private data sources or by setting up an own sandbox environment.

6.1.3.2 Challenges in Data Collection

The challenges in the data collection fundamentally result from the surge in the number of malware. For the public data service, it becomes much difficult to have good coverage of the scope of data. Even for the largest provider VirusShare, the total number of samples they cover is about 31 million, which is less than 5% of the reported number of the year 2018 [134]. The private data providers, which likely have a better coverage compared to the public data (e.g., VirusTotal sees about 0.4 million new binaries that are detected by at least one antivirus engine [135], the value of the data and the cost of maintenance naturally lead to a business model of providing data in exchange of price. The cost of the data in case of VirusTotal is reported to be around $100,000 per year [136, 137].

6.1.3.3 Properties Required for the New Platform

The challenges in data collection suggest several properties desirable for a new platform.

A public dataset with strategies to increase motivation. The aim here is to have a public dataset with better coverage and to achieve this we allow data submission from users around the world. However, it is challenging to motivate the voluntary submission. Therefore, considerations should be made in the platform design to motivate the participation of the users.

Reduce the burden of data maintenance. We discuss the difficulties of maintaining and hosting the data due to a large number of resources to take care of. We can distribute the burden of the maintenance of the dataset (e.g., hosting the binaries) among the users utilizing the dataset.

Bibliography

[1] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All Your iFRAMEs Point to Us. In *USENIX Security Symposium*, 2008.

[2] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security Symposium*, 2011.

[3] Cliff Changchun Zou and Ryan Cunningham. Honeypot-aware advanced botnet construction and maintenance. In *DSN*, 2006.

[4] Doowon Kim, Bum Jun Kwon, and Tudor Dumitraș. Certified malware: Measuring breaches of trust in the windows code-signing pki. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, 2017.

[5] Nicholas Falliere, Liam O'Murchu, and Eric Chien. W32.Stuxnet dossier. Symantec Whitepaper, February 2011.

[6] DAN GOODIN. Stuxnet spawn infected kaspersky using stolen foxconn digital certificates, Jun 2015.

[7] Swiat. Flame malware collision attack explained, Jun 2012.

[8] Kaspersky Lab's Global Research and Analysis Team. The duqu 2.0 persistence module, Jun 2015.

[9] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 debian openssl vulnerability. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 15–27, New York, NY, USA, 2009. ACM.

[10] Liang Zhang, David Choffnes, Dave Levin, Tudor Dumitras, Alan Mislove, Aaron Schulman, and Christo Wilson. Analysis of ssl certificate reissues and revocations in the wake of heartbleed. In *Proceedings of the 2014 Conference*

on Internet Measurement Conference, IMC '14, pages 489–502, New York, NY, USA, 2014. ACM.

[11] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[12] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. An End-to-End Measurement of Certificate Revocation in the Web's PKI. pages 183–196. ACM Press, 2015.

[13] Lin Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. Analyzing forged SSL certificates in the wild. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 83–97. IEEE, 2014.

[14] Frank Cangialosi, Taejoong Chung, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. Measurement and analysis of private key sharing in the https ecosystem. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 628–640, New York, NY, USA, 2016. ACM.

[15] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. The SSL landscape: a thorough analysis of the x. 509 PKI using active and passive measurements. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 427–444. ACM, 2011.

[16] Zhou Li, Sumayah A. Alrwais, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Finding the linchpins of the dark web: a study on topologically dedicated hosts on malicious web infrastructures. In *S&P*, 2013.

[17] Phani Vadrevu, Babak Rahbarinia, Roberto Perdisci, Kang Li, and Manos Antonakakis. Measuring and Detecting Malware Downloads in Live Network Traffic. In *ESORICS*, 2013.

[18] Luca Invernizzi, Sung-Ju Lee, Stanislav Miskovic, Marco Mellia, Ruben Torres, Christopher Kruegel, Sabyasachi Saha, and Giovanni Vigna. Nazca: Detecting Malware Distribution in Large-Scale Networks. In *NDSS*, 2014.

[19] Zhou Li, Sumayah Alrwais, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Finding the linchpins of the dark web: a study on topologically dedicated hosts on malicious web infrastructures. In *IEEE S&P*, 2013.

[20] Zhaoyan Xu, Antonio Nappa, Robert Baykov, Guangliang Yang, Juan Caballero, and Guofei Gu. AUTOPROBE: towards automatic active malicious server probing using dynamic binary analysis. In *CCS*, 2014.

[21] Kurt Thomas, Danny Huang, David Wang, Elie Bursztein, Chris Grier, Thomas J Holt, Christopher Kruegel, Damon McCoy, Stefan Savage, and Giovanni Vigna. Framing dependencies introduced by underground commoditization. In *WEIS*, 2015.

[22] Improved digital certificate security. https://security.googleblog.com/2015/09/improved-digital-certificate-security.html, Sep 2015.

[23] Bum Jun Kwon, Jayanta Mondal, Jiyong Jang, Leyla Bilge, and Tudor Dumitraș. The dropper effect: Insights into malware distribution with downloader graph analytics. In *CCS*, 2015.

[24] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 public key infrastructure certificate and certificate revocation list (crl) profile. RFC 5280, RFC Editor, May 2008. http://www.rfc-editor.org/rfc/rfc5280.txt.

[25] Microsoft. Windows authenticode portable executable signature format, Mar 2008.

[26] Burt Kaliski. Pkcs #7: Cryptographic message syntax version 1.5. RFC 2315, RFC Editor, March 1998. http://www.rfc-editor.org/rfc/rfc2315.txt.

[27] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 internet public key infrastructure online certificate status protocol - ocsp. RFC 6960, RFC Editor, June 2013. http://www.rfc-editor.org/rfc/rfc6960.txt.

[28] CodeSigningWorkingGroup. Minimum requirements for the issuance and management of publicly-trusted code signing certificates. Technical report, 2016.

[29] Microsoft. Code-Signing Best Practices. Technical report, 2007. https://msdn.microsoft.com/en-us/library/windows/hardware/dn653556.

[30] Symantec. W32.Sobig.F. http://www.symantec.com/security_response/writeup.jsp?docid=2003-081909-2118-99, 2003.

[31] P. Porras, H. Saidi, and V. Yegneswaran. An anlysis of conficker's logic and rendezvous points. http://mtc.sri.com/Conficker/, 2009.

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

[33] Gilou Tenebro. The Bredolab Files. Symantec Whitepaper. http://securityresponse.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_bredolab_files.pdf, 2009.

[34] Antonio Nappa, Zhaoyan Xu, M. Zubair Rafique, Juan Caballero, and Guofei Gu. Cyberprobe: Towards internet-scale active detection of malicious servers. In *NDSS*. The Internet Society, 2014.

[35] Christian Rossow, Christian Dietrich, and Herbert Bos. Large-scale analysis of malware downloaders. In *DIVMA*. 2013.

[36] Nektarios Leontiadis, Tyler Moore, and Nicolas Christin. A Nearly Four-Year Longitudinal Study of Search-Engine Poisoning. In *CCS*, 2014.

[37] Thomas Dübendorfer and Stefan Frei. Web browser security update effectiveness. In *CRITIS Workshop*, September 2009.

[38] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitraș. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. In *IEEE S&P*, San Jose, CA, 2015.

[39] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitraș. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *S&P*, 2015.

[40] Tudor Dumitraș and Darren Shou. Toward a standard benchmark for computer security research: The Worldwide Intelligence Network Environment (WINE). In *EuroSys BADGERS Workshop*, Salzburg, Austria, Apr 2011.

[41] VirusTotal. `www.virustotal.com`, 2017.

[42] National software reference library (nsrl). `http://www.nsrl.nist.gov/`.

[43] Bum Jun Kwon, Virinchi Srinivas, Amol Deshpande, and Tudor Dumitraş. Catching worms, trojan horses and pups: Unsupervised detection of silent delivery campaigns. In *Proc. NDSS*, 2017.

[44] Doowon Kim, Bum Jun Kwon, Kristián Kozák, Christopher Gates, and Tudor Dumitras. The broken shield: Measuring revocation effectiveness in the windows code-signing PKI. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 851–868, Baltimore, MD, 2018. USENIX Association.

[45] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 605–620, Berkeley, CA, USA, 2013. USENIX Association.

[46] D. Kumar, Z. Wang, M. Hyder, J. Dickinson, G. Beck, D. Adrian, J. Mason, Z. Durumeric, J. A. Halderman, and M. Bailey. Tracking certificate misissuance in the wild. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 288–301.

[47] Mike Wood. Want My Autograph? The Use and Abuse of Digital Signatures by Malware. *Virus Bulletin Conference September 2010*, (September):1–8, 2010.

[48] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. Certified pup: Abuse in authenticode code signing. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 465–478, New York, NY, USA, 2015. ACM.

[49] Omar Alrawi and Aziz Mohaisen. Chains of distrust: Towards understanding certificates used for signing malicious applications. In *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, pages 451–456, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[50] Sascha Fahl, Sergej Dechand, Henning Perl, Felix Fischer, Jaromir Smrcek, and Matthew Smith. Hey, nsa: Stay away from my market! future proofing app markets against powerful attackers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1143–1155, New York, NY, USA, 2014. ACM.

[51] Antonio Nappa, M. Zubair Rafique, and Juan Caballero. The MALICIA dataset: identification and analysis of drive-by download operations. *IJIS*, 2015.

[52] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of HTTP-based malware and signature generation using malicious network traces. In *NSDI*, 2010.

[53] Marco Cova, Corrado Leita, Olivier Thonnard, Angelos D. Keromytis, and Marc Dacier. An analysis of rogue AV campaigns. In *RAID*, 2010.

[54] Phani Vadrevu, Babak Rahbarinia, Roberto Perdisci, Kang Li, and Manos Antonakakis. Measuring and detecting malware downloads in live network traffic. In *ESORICS*, 2013.

[55] Luca Invernizzi, Sung-Ju Lee, Stanislav Miskovic, Marco Mellia, Ruben Torres, Christopher Kruegel, Sabyasachi Saha, and Giovanni Vigna. Nazca: Detecting malware distribution in large-scale networks. In *NDSS*, 2014.

[56] Jialong Zhang, Sabyasachi Saha, Guofei Gu, Sung-Ju Lee, and Marco Mellia. Systematic mining of associated server herds for malware campaign discovery. In *ICDCS*, 2015.

[57] Duen Horng Chau, Carey Nachenberg, Jeffrey Wilhelm, Adam Wright, and Christos Faloutsos. Polonium: Tera-scale graph mining and inference for malware detection. In *SDM*, 2011.

[58] Acar Tamersoy, Kevin Roundy, and Duen Horng Chau. Guilt by association: large scale malware detection by mining file-relation graphs. In *KDD*, 2014.

[59] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *CCS*, 2009.

[60] Deguang Kong and Guanhua Yan. Discriminant malware distance learning on structural information for automated malware classification. In *KDD*, 2013.

[61] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

[62] Yao Zhao, Yinglian Xie, Fang Yu, Qifa Ke, Yuan Yu, Yan Chen, and Eliot Gillum. BotGraph: Large Scale Spamming Botnet Detection. In *NSDI*, 2009.

[63] Chao Yang, Robert Chandler Harkreader, and Guofei Gu. Die Free or Live Hard? Empirical Evaluation and New Design for Fighting Evolving Twitter Spammers. In *RAID*, 2011.

[64] Anirudh Ramachandran, Anirban Dasgupta, Nick Feamster, and Kilian Weinberger. Spam or ham?: characterizing and detecting fraudulent not spam reports in web mail systems. In *CEAS*, 2011.

[65] Leman Akoglu, Mary McGlohon, and Christos Faloutsos. Oddball: Spotting anomalies in weighted graphs. In *KDD*. 2010.

[66] Chris Kanich, Christian Kreibich, Kirill Levchenko, Brandon Enright, Geoffrey M. Voelker, Vern Paxson, and Stefan Savage. Spamalytics: an empirical analysis of spam marketing conversion. In *CCS*, 2008.

[67] Kirill Levchenko, Andreas Pitsillidis, Neha Chachra, Brandon Enright, Márk Félegyházi, Chris Grier, Tristan Halvorson, Chris Kanich, Christian Kreibich, He Liu, Damon McCoy, Nicholas Weaver, Vern Paxson, Geoffrey M. Voelker, and Stefan Savage. Click trajectories: End-to-end analysis of the spam value chain. In *S&P*, 2011.

[68] Hongyu Gao, Jun Hu, Christo Wilson, Zhichun Li, Yan Chen, and Ben Y. Zhao. Detecting and characterizing social spam campaigns. In *SIGCOMM*, 2010.

[69] Chris Grier, Kurt Thomas, Vern Paxson, and Chao Michael Zhang. @spam: the underground on 140 characters or less. In *CCS*, 2010.

[70] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *WWW*, 2013.

[71] Meng Jiang, Peng Cui, Alex Beutel, Christos Faloutsos, and Shiqiang Yang. Catching synchronized behaviors in large networks: A graph mining approach. *TKDD*, 2015.

[72] Deepayan Chakrabarti. Autopart: Parameter-free graph partitioning and outlier detection. In *Knowledge Discovery in Databases: PKDD*. 2004.

[73] Qiang Cao, Xiaowei Yang, Jieqi Yu, and Christopher Palow. Uncovering large groups of active malicious accounts in online social networks. In *CCS*, 2014.

[74] Peter Eckersley. Sovereign keys: A proposal to make https and email more secure. *Electronic Frontier Foundation*, 18, 2011.

[75] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. Technical report, 2013.

[76] Ben Laurie and Emilia Kasper. Revocation transparency. *Google Research, September*, 2012.

[77] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable key infrastructure (aki): A proposal for a public-key validation infrastructure. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 679–690, New York, NY, USA, 2013. ACM.

[78] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. Arpki: Attack resilient public-key infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 382–393, New York, NY, USA, 2014. ACM.

[79] Mark D Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *In Network and Distributed System Security Symposium (NDSS). Internet Society*. Citeseer, 2014.

[80] Jiangshan Yu, Vincent Cheval, and Mark Ryan. Dtki: a new formalized pki with verifiable trusted parties. *The Computer Journal*, 59(11):1695–1713, 2016.

[81] Liang Xia, Dacheng Zhang, Daniel Kahn Gillmor, and Behcet Sarikaya. CT for Binary Codes. Internet-Draft draft-zhang-trans-ct-binary-codes-04, Internet Engineering Task Force, March 2017. Work in Progress.

[82] Dorottya Papp, Balázs Kócsó, Tamás Holczer, Levente Buttyán, and Boldizsár Bencsáth. Rosco: Repository of signed code. In *Virus Bulletin Conference, Prague, Czech Republic*, 2015.

[83] Mozilla binary transparency. `https://wiki.mozilla.org/Security/Binary_Transparency`.

[84] Amonetize. http://www.amonetize.com/about-us/.

[85] Platon Kotzias, Leyla Bilge, and Juan Caballero. Measuring pup prevalence and pup distribution through pay-per-install services. In *Proceedings of the USENIX Security Symposium*, 2016.

[86] Kurt Thomas, Juan A. Elices Crespo, Ryan Rasti, Jean Michel Picod, Cait Phillips, Marc-André Decoste, Chris Sharp, Fabio Tirelo, Ali Tofigh, Marc-Antoine Courteau, Lucas Ballard, Robert Shield, Nav Jagpal, Moheeb Abu Rajab, Panayiotis Mavrommatis, Niels Provos, Elie Bursztein, and Damon McCoy. Investigating commercial pay-per-install and the distribution of unwanted software. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 721–739, 2016.

[87] Microsoft. Virus: Win32/induc.a, April 2011.

[88] Microsoft. Erroneous verisign-issued digital certificates pose spoofing hazard, 2001.

[89] Opencorporates. https://opencorporates.com.

[90] Herdprotect. http://www.herdprotect.com.

[91] Microsoft sigcheck. https://technet.microsoft.com/en-us/sysinternals/bb897441.aspx.

[92] Virustotal hunting. https://www.virustotal.com/#/hunting-overview.

[93] Yara rules. http://virustotal.github.io/yara/.

[94] Charles J Krebs et al. Ecological methodology. Technical report, Harper & Row New York, 1989.

[95] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[96] Alexa top 1 million. http://www.alexa.com/.

[97] Gain ratio. http://www.csee.wvu.edu/~timm/cs591o/old/Lecture6.html.

[98] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1986.

[99] Receiver operating characteristic. http://en.wikipedia.org/wiki/Receiver_operating_characteristic.

[100] René Peeters. The maximum edge biclique problem is np-complete. *Discrete Applied Mathematics*, 131(3):651–654, 2003.

[101] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. Certified PUP: Abuse in Authenticode code signing. In *CCS*, 2015.

[102] List of pay per install companies. http://www.blackhatworld.com/seo/list-of-pay-per-install-ppi-networks.646987/.

[103] List of pay per install networks. http://www.blackhatworld.com/seo/list-of-pay-per-install-ppi-networks.646987/.

[104] Reason labs knowledge base. https://www.reasoncoresecurity.com/knowledgebase.aspx.

[105] Networkx. https://networkx.github.io/.

[106] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM Sigmod Record*, volume 29, pages 1–12. ACM, 2000.

[107] Jayanta Mondal and Amol Deshpande. Eagr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. In *SIGMOD*, 2014.

[108] Multiprocessing. https://docs.python.org/2/library/multiprocessing.html.

[109] Marc Kührer, Christian Rossow, and Thorsten Holz. Paint it black: Evaluating the effectiveness of malware blacklists. In *RAID*, 2014.

[110] Affiliate summit. http://affiliatesummit.com/.

[111] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008.

[112] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 1977.

[113] Shetef solutions and consulting. https://www.reasoncoresecurity.com/signer-shetef-solutions-consulting-1998-ltd-40812da0f7cb2ecd4955fd76e0a6c493.aspx.

[114] Kurt Thomas, Juan A. Elices Crespo, Ryan Rasti, Jean-Michel Picod, Cait Phillips, Marc-André Decoste, Chris Sharp, Fabio Tirelo, Ali Tofigh, Marc-Antoine Courteau, Lucas Ballard, Robert Shield, Nav Jagpal, Moheeb Abu Rajab, Panayiotis Mavrommatis, Niels Provos, Elie Bursztein, and Damon McCoy. Investigating commercial pay-per-install and the distribution of unwanted software. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 721–739, Austin, TX, August 2016. USENIX Association.

[115] Platon Kotzias, Leyla Bilge, and Juan Caballero. Measuring PUP prevalence and PUP distribution through Pay-Per-Install services. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 739–756, Austin, TX, August 2016. USENIX Association.

[116] Kmp media, stack overflow vulnerability. https://www.krcert.or.kr/data/secNoticeView.do?bulletin_writing_sequence=2147&queryString=cGFnZT0xJnNvcnRfY29kZT0mc2VhcmNoX3NvcnQ9a2V5d29yZCZzZWFyY2hfd29yZD1pb3M=.

[117] Virustotal report of the kmp sample. https://www.virustotal.com/en/file/ff49e145515bdecbca61b7d97439959be5b04b1c29d77a0e8c42a1c1bed42aa8.

[118] Brian Krebs. Signed malware = expensive "oops" for hp. http://krebsonsecurity.com/2014/10/signed-malware-is-expensive-oops-for-hp/, Oct 2014.

[119] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a dynamic reputation system for dns. In *Proceedings of the 19th USENIX Conference on Security*, 2010.

[120] Leyla Bilge, Sevil Sen, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. Exposure: A passive dns analysis service to detect and report malicious domains. *ACM Trans. Inf. Syst. Secur.*, 2014.

[121] Malware domain list. https://www.malwaredomainlist.com.

[122] Malware domains. http://www.malwaredomains.com.

[123] Phish tank. https://www.phishtank.com.

[124] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[125] igraph. http://igraph.org/.

[126] Amazon aws. https://aws.amazon.com/ec2/.

[127] Andy Greenberg. Software has a serious supply-chain security problem. https://www.wired.com/story/ccleaner-malware-supply-chain-software-security/, Sep 2017.

[128] Nltk. https://www.nltk.org.

[129] Cleanco. https://pypi.org/project/cleanco/.

[130] Wikipedia: Vandalism. https://en.wikipedia.org/wiki/Wikipedia:Vandalism.

[131] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3–4):211–407, August 2014.

[132] Virusshare. https://virusshare.com.

[133] Us cert analysis reports. https://www.us-cert.gov/ncas/analysis-reports.

[134] Avtest, number of malware statistics. https://www.av-test.org/en/statistics/malware/.

[135] Virustotal statistics. https://www.virustotal.com/en/statistics/.

[136] Wikileaks, virustotal price. https://wikileaks.org/hackingteam/emails/emailid/177430.

[137] Virustotal access to be limited. https://antivirus.comodo.com/blog/computer-safety/virustotal-access-to-be-limited-google/.