Certified Malware: Measuring Breaches of Trust in the Windows Code-Signing PKI

Doowon Kim
University of Maryland
College Park, MD
doowon@cs.umd.edu

Bum Jun Kwon
University of Maryland
College Park, MD
bkwon@umd.edu

Tudor Dumitraș
University of Maryland
College Park, MD
tdumitra@umiacs.umd.edu

ABSTRACT

Digitally signed malware can bypass system protection mechanisms that install or launch only programs with valid signatures. It can also evade anti-virus programs, which often forego scanning signed binaries. Known from advanced threats such as Stuxnet and Flame, this type of abuse has not been measured systematically in the broader malware landscape. In particular, the methods, effectiveness window, and security implications of code-signing PKI abuse are not well understood. We propose a threat model that highlights three types of weaknesses in the code-signing PKI. We overcome challenges specific to code-signing measurements by introducing techniques for prioritizing the collection of code-signing certificates that are likely abusive. We also introduce an algorithm for distinguishing among different types of threats. These techniques allow us to study threats that breach the trust encoded in the Windows code-signing PKI. The threats include stealing the private keys associated with benign certificates and using them to sign malware or by impersonating legitimate companies that do not develop software and, hence, do not own code-signing certificates. Finally, we discuss the actionable implications of our findings and propose concrete steps for improving the security of the code-signing ecosystem.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Operating systems security**;

KEYWORDS

Code signing; Windows Authenticode; Malware; PKI; Compromised certificates

1 INTRODUCTION

Each time we use our computers, we trust the programs executed, either deliberately or in the background, not to perform unwanted or harmful actions. Software that appears to come from reputable publishers, but that performs such actions, breaches this trust.

To establish trust in third-party software, we currently rely on the code-signing Public Key Infrastructure (PKI). This infrastructure includes Certification Authorities (CAs) that issue certificates to software publishers, vouching for their identity. Publishers use these certificates to sign the software they release, and users rely on these signatures to decide which software packages to trust (rather than maintaining a list of trusted packages). If adversaries can compromise code signing certificates, this has severe implications for end-host security. Signed malware can bypass system protection mechanisms that install or launch only programs with valid signatures, and it can evade anti-virus programs, which often neglect to scan signed binaries. More generally, the recent advances in trustworthy computing [30] rely on a functioning mechanism for bootstrapping trust in third-party programs.

In the past, compromised code-signing certificates have been associated with advanced threats, likely developed by nation-state adversaries. For example, the Stuxnet worm included device drivers that were digitally signed with keys stolen from two Taiwanese semiconductor companies, located in close proximity [10]. The Flame malware masqueraded as a file from Windows Update by conducting a previously unknown chosen-prefix collision attack against the MD5 cryptographic hash [33]. In both cases, the valid digital signatures allowed the malware to evade detection and to bypass anti-virus and Windows protections.

Anecdotal information suggests that a broader range of malicious programs may carry valid digital signatures, resulting from compromised certificates [10, 11, 33]. However, this threat has not been measured systematically. In particular, the methods, effectiveness window, and security implications of code-signing PKI abuse are not well understood, owing to the difficulty of distinguishing between malicious and potentially-unwanted behavior. The prior research on abuse in the code-signing ecosystem [18, 19, 21, 34] has focused on potentially unwanted programs (PUPs), such as adware, which are typically signed with certificates legitimately issued to the PUP publishers. While the signed PUPs substantiate the utility of valid signatures for abusive programs, the prior results do not distinguish between certificates issued to publishers of dubious software by following a legitimate process and abuse of the code-signing PKI.

In this paper, we conduct a systematic study of threats that breach the trust encoded in the Windows code-signing PKI. We focus on signed malware, which is more likely than PUPs to rely on abusive code signing certificates as malware creators typically try to hide their identities. Unlike the prior studies on other certificate ecosystems, such as measurements of the Web's PKI [3, 7, 14, 15, 24, 40], we cannot rely on a comprehensive corpus of code signing certificates. These certificates cannot be collected at scale by scanning

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3133958

the Internet; there is no official list of code signing certificates, or even of the organizations that can issue such certificates.

To overcome these challenges, we analyze a large data set of anti-virus reports, corresponding to 1,053,114 malicious files that carry digital signatures. This data set is made available by Symantec on the WINE platform [4]. By querying the VirusTotal service [35], this analysis allows us to prioritize the collection of code signing certificates that are likely abusive. We also utilize global corporate directories to identify publishers of benign software and we develop novel heuristics for distinguishing among different types of abuse.

We find that digitally signed malware was prevalent in the wild before Stuxnet; the earliest sample in our data set was signed in 2003. 88.8% of the malware families using abusive certificates rely on a single certificate, which suggests that, in most cases, these certificates are controlled by the malware authors rather than by third-party "signing" services. We also estimate that 80% of the abusive certificates remain a threat for over 5.6 years after they are first used to sign malware.

To characterize the real-world breaches of trust in the code signing PKI, we propose a threat model with three classes of weaknesses that an adversary can exploit: *inadequate client-side protections*, *publisher-side key mismanagement*, and *CA-side verification failures*. We infer the prevalence and evolution of each weakness, and we analyze how long users are exposed to these threats. Then, we conservatively select a subset of abusive code signing certificates for in-depth investigation. Below, we highlight some of our findings from this analysis:

- *Inadequate client-side protections.* We find that simply copying an Authenticode signature from a legitimate file to a known malware sample may cause anti-virus products to stop detecting it—even though the signature is invalid, as it does not match the file digest. 34 anti-virus products are affected, and this type of abuse accounts for 31.1% of malware signatures in the wild. We notified the anti-virus companies of the issue. Two companies confirmed that their products fail to check signature properly; one of them plans to fix the issue.
- *Publisher-side key mismanagement.* We identify 72 certificates that were likely compromised, and we were able to confirm this with eight publishers; five of them were not previously aware of the abuse. We analyze a malware family that infects developer machines and copies malicious code into files compiled and signed on those machines. We find that, starting from 180 developer machines, variants of this malware can reach 93,016 machines—an amplification factor of 517×.
- *CA-side verification failures.* We identify 27 certificates issued to malicious actors impersonating legitimate companies.

We utilize our findings to draw lessons about the trust that we can place in unknown software packages. We also discuss concrete proposals for improving the code signing ecosystem. We make the information of the abusive certificates publicly available at https://signedmalware.org. The information includes publisher names, issuer names, serial numbers, hash values of malware signed with the certificates, etc.

2 PROBLEM STATEMENT

Code signing is a mechanism for authenticating the software publisher that released a given executable program. The *publisher* generates a pair of cryptographic keys (the public key and the private key) computes a cryptographic hash of the executable code and signs the hash with the private key. To prove that it owns the signing key, the publisher requests from *Certificate Authority (CA)* a digital certificate. The certificate includes the name of the publisher, its public key, and a few other fields; and it is signed with the CA's key. A CA may itself have a certificate signed by another CA, resulting in a trust chain that must end in a root certificate that the end-user trusts. Like any authentication token, certificates must be revoked when they are compromised. Additionally, before signing any field of the binary may be forged, including the compilation timestamp. To prove that the certificate was valid at the time of signing, the publisher may obtain an additional signature from a *Time Stamping Authority (TSA)*. The code signature, the certificate and the timestamp are distributed with the binary. The user who installs the software and runs the installed executables can then authenticate the publisher and verify that the binary has not been tampered with after signing.

The code signing mechanism allows users to set policies on what executables to trust; for example all executables from a set of trusted publishers, all executables for which the publisher can be identified (i.e. they are signed with a valid certificate), or all executables signed with a certificate that was valid at the time of signing. Additionally, software updates may be validated and applied automatically if they are signed with the same certificate as the original program.

Code signing relies on a Public Key Infrastructure (PKI), composed of certificate authorities that can vouch for the identity of software publishers. Users place their trust in an ecosystem of software publishers, root CAs and root TSAs—a large trusted computing base (TCB) that provides many opportunities for miscreants to compromise security. Like in the TLS certificate ecosystem, every CA can sign certificates for any publisher, and there is no official list of code signing certificates or even of the organizations that can issue such certificates. Unlike for TLS, code signing is employed by many different platforms and operating systems, each with its own root certificate store: Windows and macOS executables and drivers, Firefox XPIs, Android/iOS apps, Java Jars, Visual Basic for Applications, Adobe Air apps, etc. This further expands the TCB for an end user. The TLS PKI has been the subject of several measurement studies [3, 7, 14, 15, 24, 40], which have illuminated vulnerabilities of the PKI and how it is abused in the wild. These findings have stimulated research on fixing these problems and have prompted several efforts for preventing abuse, such as certificate transparency [22], key pinning [20] and DANE [13]. In contrast, little is known about the various code signing ecosystems, including the opportunities for breaching the trust in various actors from these ecosystems, the prevalence of real-world abuse of the PKI and the extent to which code signing prevents security threats.

As a first step in this direction, *our goal* in this paper is to measure breach-of-trust in the Windows code signing PKI. An adversary can breach trust relationships explicitly, e.g. by stealing the private keys associated with benign certificates and using them to sign malware, or implicitly, e.g. by impersonating legitimate companies

that do not develop software and, hence, do not own code-signing certificates. We aim to analyze the prevalence of this threat in the real world and to illuminate the mechanisms for beaching the trust. We also aim to understand the security implications of these types of abuse and to examine the effectiveness of proposed PKI improvements in defending against this threat. Our *non-goals* include fully characterizing the code signing ecosystems, analyzing certificates issued legitimately to real (but perhaps ill intentioned) publishers, or developing new techniques for authenticating executable programs.

2.1 Overview of code signing

On Windows, the prevalent code signing standard is Microsoft Authenticode [26]. The standard is based on Public-Key Cryptography Standard (PKCS) #7 [16] and is used to digitally sign Windows portable executables (PE). These include installers (.msi), cabinet (.cab) and catalog (.cat) files, Active X controls (.ctl, and .ocx), dynamically loaded libraries (.dll), and executables (.exe). PKCS #7-formatted content is called *signed data*. A signed data blob includes the signature, the hash value of a PE file, and the certificate chains. The chains should end in a trusted root certificate, but the signed data does not need to include the root certificate as long as the root certificate is present in the users' root stores. Authenticode supports MD5, SHA-1, and SHA-256 hashes.

Protections that rely on code signing. On Windows, User Account Control (UAC) verifies the signature and includes the publisher's name in the notification presented to the user when a program requests elevated privileges. Microsoft SmartScreen checks executable files downloaded from the Web and assesses the publisher's reputation. A publisher may also obtain an Extended Validation (EV) certificate, which must undergo stricter vetting procedures outlined in the Guidelines for Extended Validation produced by the CA/Browser Forum.[1] Because of the higher degree of trust in these certificates, they receive instant reputation in SmartScreen, while standard code signing certificates must build up their reputation to bypass the SmartScreen Filter. Google Safe Browsing is another protection system similar to the SmartScreen. The whitepaper mentions that "Chrome trusts potentially dangerous file types that match URLs in the whitelist, and it also trusts files signed by a trusted authority." [2] Starting with Vista, drivers should be signed by a trusted Certificate Authority (CA) to be installed on a Windows machine[3]. From Windows 10 (version 1607), a stricter requirement is set: EV certificates are necessary for any new kernel mode drivers [4].

Antivirus engines also utilize code signing information. To reduce false positives, some AV engines use whitelisting based on code signing certificates. For example, Symantec mention in their whitelisting page: "To prevent false positive detections we strongly recommend that you digitally sign your software with a class 3 digital certificate." [5]

[1] https://cabforum.org
[2] https://www.google.com/intl/en/chrome/browser/privacy/whitepaper.html
[3] https://www.digicert.com/code-signing/driver-signing-certificates.htm
[4] https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later-
[5] https://submit.symantec.com/whitelist/

Revocation. Beside issuing new certificates, CAs must sometimes revoke existing certificates, for a variety of reasons. One of the most common cases is when a private key associated with the certificate is compromised[6]. Certificates using weak cryptographic keys [39] must be revoked as well. In rare cases, CAs must also revoke erroneously issued certificates.[7]

CAs use two mechanisms for informing users of certificate revocations: Certificate Revocation Lists (CRLs) and the Online Certificate Status Protocol (OCSP). A CRL includes multiple certificates that have been revoked and is made available at a link specified in the certificate. These lists are not comprehensive, and users must periodically download multiple CRLs to receive all the information about revoked certificates. With OCSP, users query a server maintained by the CA to receive the revocation status of a certificate on demand. Both the CRLs and the OCSP responses are signed by the CA to ensure their integrity.

2.2 Differences between code signing and TLS

Code signing and the TLS protocols used to secure HTTP communications utilize X.509 v3 certificates for authentication and integrity. One type of certificate should not be used for another purpose (e.g., program code cannot be signed with a TLS certificate) because its purpose is indicated in the "Key Usage" and "Extended Key Usage" fields. The latter field must include "Code Signing" for code signing certificates and "Server Authentication" for TLS certificates. However, because executable programs may be invoked on a machine that is not connected to the Internet, code signing certificates may include trusted timestamps and have special requirements for revocation.

Trusted timestamping. A trusted timestamp certifies the signing time of the code and extends the trust in the program beyond the validity period of the certificate. In TLS, when a certificate expires, any service (e.g., web, email, etc.) associated with the certificate becomes invalid. However, if program code is properly signed and timestamped within its validity period, the code can remain valid after the certificate expires. To timestamp signed code, the hash value of an original program code is sent to a Time-Stamping Authority (TSA). The TSA appends the current timestamp to the received hash value, calculates a new hash, and digitally signs it with its private key. The TSA then returns the signature and the current timestamp in plain text. The publisher appends the received signature, timestamp, and TSA certificate chains to the signed program code.

Revocation. For TLS, CAs do not provide the revocation status of expired certificates. In contrast, because of timestamping, code-signing CAs must keep updating the certificate revocation information even after the certificate expires.

As trusted timestamps cause an indefinite extension of the certificate validity, revocation mechanisms play a more important role in the code signing ecosystem. Suppose a code signing certificate is valid between t_i (issue date) and t_e (expiration date). The certificate is revoked at some point (no matter before or after the expiration

[6] https://www.globalsign.com/en/blog/casc-code-signing-certificate-requirements-for-developers/
[7] For example, in 2001 VeriSign issued two code signing certificates with the common name of "Microsoft Corporation" to an adversary who claimed to be a Microsoft employee [25].

date) and its revocation date is set to t_r. Any program that is signed and timestamped between t_i and t_r is still valid even though the certificate is revoked. This is based on the assumption that we know when the certificate has been compromised, and all the code signed and timestamped before t_r is valid. However, the CA may not know the exact date when the certificate was compromised. Consequently, CAs may set the revocation date equal to the issue date. In this case, all the (even benign) programs signed with that certificate become invalid. The policy on setting revocation dates varies from one CA to another.

2.3 Threat model

We consider an adversary with two goals: (1) to distribute and install malware on end-user machines; and (2) to conceal its identity. In consequence, the adversary aims to sign malware samples in order to evade AV detections and platform security policies such as User Account Control and SmartScreen. At the same time, the adversary does not want to obtain a legitimate code-signing certificate, which would reveal its identity. This second goal distinguishes our adversary from the PUP publishers considered in the prior work on code-signing abuse [18, 19, 21, 34]. To achieve these goals, the adversary exploits weaknesses in the code-signing PKI to obtain signed malware samples, which bypass various defenses. These weaknesses fall into three categories: inadequate client-side protections, publisher-side key mismanagement, and CA-side verification failures.

Inadequate client-side protections. While Windows operating systems have the ability to verify code-signing signatures, they often allow unsigned or improperly signed software to be installed and executed. If a program requires elevated privileges, UAC notifies the user and includes a message about the publisher's identity (or lack thereof, if the signature is invalid). However, if the user choses to grant the privilege, Windows does not take further enforcement actions. To fill this gap, anti-virus tools may block malicious programs with invalid signatures. However, each tool includes an independent implementation of the signature verification, which may result in different interpretations of certificate validity.

Publisher-side key mismanagement. Publishers are expected to restrict access to their code signing keys and to keep them secure. If the adversary gains access to the development machines involved in the signing process, it can steal the private key that corresponds to the publisher's certificate—thus compromising the certificate—or it can use those machines to sign malware.

A certificate is compromised when the private key is no longer in the sole possession of its owners. Signing keys may be stolen in two ways:

- An adversary may breach the publisher's network and gain access to a machine storing the private key. The adversary may then use the key for signing malware. This method was likely employed for Stuxnet and Duqu 2.0 [10, 31], advanced pieces of malware that carried valid signatures and certificates belonging to legitimate companies located in Taiwan.
- When a private key is compromised, the certificate associated with the key should be revoked and then re-issued with a new key pair. However, developers may reuse the compromised key in the new certificate. Key reuse was previously documented in 14% of TLS certificates revoked following Heartbleed [6].

Infected developer machines may also be utilized to silently sign malicious code with the publisher's valid certificate. The *W32/Induc.A* [27] malware was first found in 2009, and infected *Delphi* files. When executed, the malware searched for files required by the Delphi compilers, and injected malicious code into those files. In consequence, additional malware could be signed with a valid certificate, during the compilation process, and then distributed in a legitimate software package.

CA-side verification failures. CAs are responsible for verifying clients' identity before issuing a code signing certificate. However, this verification process may fail, resulting in certificates issued to publishers who hide their real identities.

Identity theft occurs when the adversary successfully masquerades as a reputable company and convinces a CA to issue a code-signing certificate bearing that company's name. For example, in January 2001 Verisign mistakenly issued two code signing certificates to an adversary who claimed to be an employee of Microsoft, owing to a human error in the verification process [25]. However, identity theft does not necessarily need to target large software corporations. Companies in other industries, which do not release software and do not need code signing certificates, may be easier targets as they do not expect to be affected by this form of identity theft.

Shell companies may also help the adversary acquire code signing certificates legally from a CA. Because the shell company appears legitimate, CAs have no reasons to decline the application for the certificate. A disadvantage for this form of abuse is that the valid, but unfamiliar, publisher name has not accumulated reputation in defenses like SmartScreen and may not be trusted by users. However, valid signatures may nevertheless prevent anti-virus scanning, and users may feel encouraged to install the software if the Windows dialog indicates that the publisher is verified.

2.4 Challenges for measuring code signing

The challenges in this study for measuring code signing are collecting of binaries and distinguishing between the abuse cases. For TLS it is possible to get a comprehensive list of certificates by scanning the IP spaces. However, for code signing there exists no easy way to collect all the certificates used in the field. Even for the certificates we are able to collect, it is hard to capture all the binaries signed by these certificates. A further challenge is to identify the abuse case, e.g., compromised certificate, identity theft, shell company. While some well-studied malware samples, such as Stuxnet and Duqu, are known to use compromised certificates, in most cases it is hard to find a ground truth about the type of abuse. The only precise information available is whether a certificate is revoked, as CAs distribute lists of revoked certificates. For most of the certificates on these lists, the revocation reason is left unspecified [19].

3 MEASUREMENT METHODS

To characterize breaches of trust in the Windows code signing PKI, we collect information on Authenticode certificates used to sign malware samples in the wild. We then classify these cases according to the threat model from Section 2.3 and we investigate the PKI

weaknesses that facilitated the abuse. To overcome the challenges that have prevented this analysis until now, we propose methods for prioritizing the collection of certificates that are likely to be abusive and an algorithm for distinguishing among the three types of threats we consider.

3.1 Data sources

We identify 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). WINE [4] provides telemetry collected by Symantec's products on millions of end-hosts around the world (10.9 million). Users of these products can opt-in to report telemetry about security events (e.g., executable file downloads, virus detections) on their hosts. These hosts are real computers in active use, rather than honeypots. From the WINE data set, we query the (a) *anti-virus (AV) telemetry* and (b) *binary reputation*.

The AV telemetry data contains information on the anti-virus signatures triggered on user machines. From this data, we collect the SHA256 hash of the binary that triggered the report and the name of the AV detection signature assigned to the binary. We extract about 70,293,533 unique hashes.

The binary reputation data reports download events on the user machines. From this data, we extract the SHA256 hash of the binary, the server-side timestamp of the event, and the names of the publisher and the CA from the code signing certificate. There are 587,992,001 unique binaries here. This data set does not provide more detailed information about the certificate such as its serial number. In consequence, WINE does not allow us to distinguish between files signed with different certificates belonging to the same publisher.

VirusTotal. To collect information about code signing certificates, we query VirusTotal [35]. This service provides an API for scanning files using up to 63 different anti-virus (AV) products, and for querying the previous scanning reports. We query VirusTotal using a private API, provided by VirusTotal, and retrieve the following information: the first-submission timestamp to VirusTotal, the number of AV engines that detected the file as malicious, the assigned detection name, and the file code signing information.

OpenCorporates. OpenCorporates[8] maintains the largest open database of businesses around the world, providing information on over 100 million companies. We use this database to determine whether publishers that own certificates used to sign malware correspond to legitimate companies.

HerdProtect. We further investigate the reputation of the company, as software publisher, using *HerdProtect*[9]. For each publisher in our dataset, we collect the following information: whether the publisher is known to release PUPs, a summary of the company (location, business type, etc.), and a sample of the certificates issued to the publisher.

[8]https://opencorporates.com
[9]http://www.herdprotect.com/

3.2 System overview

Pipeline overview. As illustrated in Figure 1, our data collection and analysis consists of four steps: seed collection, data filtration, input data preparation and identifying potentially abusive certificates.

- *Seed collection.* We start by collecting a set of unique SHA256 file hashes from the AV telemetry data set in WINE. We exclude the hashes detected by signatures that do not necessarily indicate a malicious activity, such as hacktools (e.g. password crackers, vulnerability scanners) or adware. To this set, we add a list of hashes of known malicious binaries, provided separately by Symantec. We then join this list of hashes to the binary reputation schema, to determine which files have a digital signature. This results in a tentative list of binaries that are both digitally signed and malicious.
- *Filtering data.* The tentative list generated in the previous step may contain PUPs and benign programs. We filter out PUPs in three ways. First, we exclude the PUP publishers identified in prior work [18, 19, 21, 34]. Second, we query *HerdProtect* for all company names (the common names in certificates). If they are labeled as PUP publishers, we remove their hashes from the list. Third, we pick 10 samples for each common name, and filter out them if at least one of them is determined to be a PUP as discussed in Section 3.3. For the files whose subject is specified as Microsoft or Anti-virus companies, we sample the files using the Symantec ground truth. In order words, we only take the ones that have bad reputation in the Symantec ground truth for those files.
- *Input data preparation.* We use the filtered hashes to query VirusTotal. The VirusTotal reports provide detailed information on the code signing certificate and AV detection results for each binary. We consider a binary to be signed malware if 1) it is properly signed, and 2) the detection names suggest it is malware, as detailed in Section 3.3. Note that at this stage we expect to see a number of malware samples with malformed signatures. We analyze these samples in Section 4, but we do not pass them further down the pipeline.
  After we obtain the list of signed malware, we use the <publisher, CA> pairs from our list to query binary reputation schema in WINE. The result of the query is a list of potentially benign binaries that are signed with the same certificates used to signed the malware. We then query VirusTotal again with this list, and we use the reports to identify benign files, as described in Section 3.3.
- *Identify potentially abusive certificates.* For each code signing certificate used to sign malware, we infer the type of abuse using the algorithm described in Section 3.4.

3.3 Binary labeling

Malware. We distinguish malware from benign and potentially unwanted programs using the approach proposed in [21]. Specifically, for each binary we define c_{mal} as the number of anti-virus products invoked by VirusTotal that flagged the binary as malicious. We set $c_{mal} \geq 20$ as the threshold for suspicious binaries. We then inspect the labels given by these AV products and compute r_{pup} as the fraction of labels indicative of PUPs (we utilize the same keywords

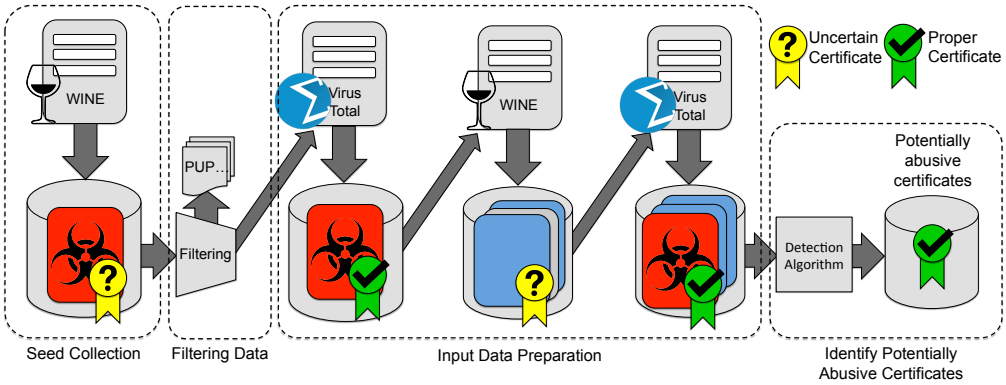

Figure 1: Data analysis pipeline.

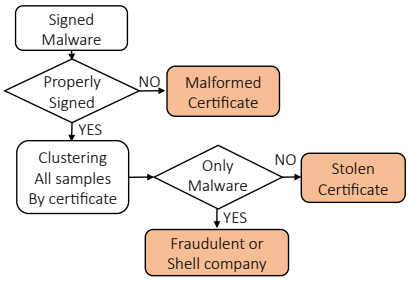

Figure 2: Flowchart of the abuse detection algorithm.

as in the prior work). We consider that a binary is malware if it has $c_{mal} \geq 20$ and $r_{pup} \leq 10\%$.

Benign programs. We also utilize the VirusTotal reports to determine if a program is benign. If a file has $c_{mal} = 0$ and a valid signature, we treat it as benign.

3.4 Abuse detection algorithm

As explained in Section 3.2, in the third step of our pipeline we identify binaries that carry malformed digital signatures. These signatures do not match the hash of the binary and were likely copied literally from other signed binaries. In this case, the adversary does not control the code signing certificate, but nevertheless tries to produce a binary that may appear to be signed.

The rest of binaries identified by our pipeline are properly signed with certificates that may be valid, expired or revoked. The binaries include malware and benign samples and exclude PUPs. We group binaries according to their code signing certificates. A publisher's binaries may be split among multiple groups if we identify distinct certificates owned by the publisher. Each group may include (i) only benign samples; (ii) only malware or (iii) both malware and benign binaries. As it is generally difficult to identify benign samples signed with a specific certificate, we consult additional data sources to increase our coverage. Specifically, we further query *HerdProtect* to determine if more samples signed with the same certificate are known to exist, and we manually investigate the publishers by visiting their websites. For each certificate used to sign malware, we then infer the corresponding type of abuse using the method illustrated in Figure 2.

Compromised certificates. As described in Section 2.3, a compromised certificate is initially issued to a legitimate publisher and is used to sign benign programs. After the private key is compromised, the certificate is shared between the legitimate owner and the malware authors. In consequence, we expect to see both benign and malicious programs in a compromised certificate's group. For each group of binaries in this category, we further analyze the trusted timestamps to reconstruct the timeline of the abuse.

Identity theft & shell companies. When the malware authors are able to convince a CA to issue them a code signing certificate, they have no motivation to use the certificate to sign benign code as well. We therefore expect to see only malicious programs in such a certificate's group. We distinguish between cases of identity theft and certificates belonging to shell companies by querying *OpenCorporates* and *HerdProtect*. Specifically, if we find a publisher in either of these directories, and the company address in the X.509 certificate matches the information in the directory, we suspect that the company is a victim of identity theft.

Verification. The VirusTotal reports allow us the reliably identify all the binaries with malformed digital signatures. Because this case does not involve any failures on the part of the publisher or the CA, we run experiments to determine if these signatures bypass client-side protections. Among the other types of abuse, we have a higher degree of confidence in our determination that a certificate is compromised, as we can observe that it is likely utilized by multiple actors. Some of the other certificates may also be compromised, but we cannot determine this because we cannot collect all the benign binaries in the wild. Similarly, we have a higher degree of confidence in the identity fraud determination than in recognizing shell companies, because some companies may be missing from the directories we utilize. To verify our results, we manually analyze their timelines, and we contact their owners and CAs to confirm our findings.

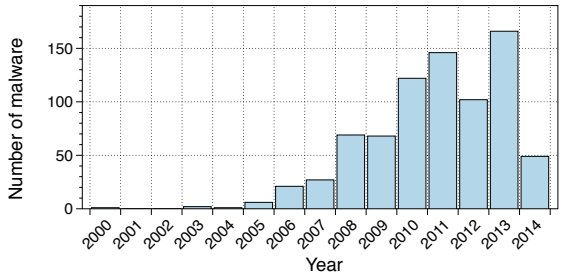

Figure 3: Number of properly signed malware per year ($c_{mal} \geq 5$).

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

4 MEASUREMENT RESULTS

4.1 Summary of the input data

We start by summarizing the effect of each step from our data analysis pipeline, described in Section 3.2. Among the 70,293,533 samples from the AV telemetry reports, 1,053,114 include the file signer information in the binary reputation data set. This suggests that 1 out of 67 samples detected by a leading anti-virus carries a digital signature. We note that this is an approximation, as we do not have visibility into all the malware present on end-hosts around the world, and other AV products may handle digital signatures differently. However, WINE provides a representative sample for the data collected by Symantec [28], which is the largest security vendor.

We filter out potentially unwanted programs using the method from Section 3.3. This step returns 526,487 hashes. We further reduce the number of samples by removing 268,404 executables signed with 2,648 PUP certificates. This yields 258,083 executables. We query VirusTotal with these samples. Out of these, we could not find VirusTotal reports for 88,154 . After further filtering out 104,230 samples without a full chain of certificates, we have 153,853 signed samples in the seed set. To apply the abuse detection algorithm from Section 3.4, we search for other potentially benign samples signed with the certificates found in the seed set. We identify a total of 415,377 such samples in VT.

We set a very conservative number $c_{mal} \geq 20$ for the malware detection threshold to identify suspicious binaries. This conservative threshold number identifies only obvious malware, and enables us to examine every single malware manually. (587 and 1136 malware samples are observed when we set the threshold as 10 and 5 respectively.) We present the number of properly signed malware per year for $c_{mal} \geq 5$ in Figure 3.

We further examine the validity of the digital signatures from these samples using the *verified* message in the VT reports. VT checks Authenticode signatures using the `sigcheck` tool provided by Microsoft.[10] For example, the message "a certificate was explicitly revoked by its issuer" indicates that the affixed certificate is no longer valid because it was revoked by its CA; this corresponds to error code "0x800B010C" in Microsoft Authenticode. Table 1 shows a breakdown of the validity status. The 325 samples are detected as malware in a total of 153,853 signed samples. Of these 325 signed malware, 58.2% samples are properly signed while 41.8% are signed with malformed certificates. Most (74.3%) improperly signed malware results from bad digests. We categorize any malware that has parsing errors into "Parsing Error." Of a total of 189 properly signed malware, 22.8% samples were signed with revoked certificates, and 19.6% samples have expired certificates and no valid timestamp. More than a half of all properly signed samples (57.7%) are still valid as of this writing.

[10] https://technet.microsoft.com/en-us/sysinternals/bb897441.aspx

CA	Count (%)
Symantec/Verisign	23,325,279 (60.47%)
Symantec/Thawte	7,054,263 (18.29%)
Comodo	2,059,601 (5.34%)
GlobalSign	708,618 (1.84%)
Go Daddy	704,036 (1.83%)
DigiCert	429,159 (1.11%)
Certum	48,677 (0.13%)
WoSign/StartCom	43,578 (0.11%)
WoSign	38,758 (0.10%)
Go Daddy/StarField	21,410 (0.06%)
Total	38,572,995 (100%)

Table 2: The number of executables signed with certificates issued by each CA in WINE.

4.2 The code signing ecosystem

In this section, we analyze the code signing ecosystem using the binary reputation data set in WINE. This analysis encompasses both benign and malicious files that carry digital signatures. The numbers in WINE are likely biased because all the hosts run at least one Symantec product. Consequently, we remove the binaries where *Symantec* is in the subject field. We then extract the name of the Certification Authority from 38.6 million executables, which allows us to identify 210 unique CAs.

CA market share. We first estimate the market share of code signing certificates by investigating the file signer information from the binary reputation data set. Table 2 shows the market share for the top-10 most popular CAs and the number of unique binaries signed with the certificates issued by those CAs. 4.1 million binaries (10.7%) in our data set are either signed by minor CAs, not included

in the table, or are self-signed. These numbers suggest that Symantec (including the Versign and Thawte brands) dominates the code signing ecosystem, with a combined market share above 78%. This is not necessarily bad for security, as placing trust in a single CA reduces the TCB for code signing. However, this raises the expectations for the leading CA to ensure the security of the ecosystem. The latest version of Windows, Windows 10, by default contains 28 root certificates for code signing that belong to 14 CAs[11]. In topmost popular CAs, WoSign, Certum, and Startcom root certificates are not pre-installed in Windows. When executables signed with the three certificate are installed, the root certificates are installed without a prompt message for users.

Misuse of code signing certificates for TLS. Code signing and TLS certificates cannot be used interchangeably. However, the two types of certificates follow the same format and can be generated with the same tools. We searched censys.io [5], a site that periodically scans the Internet and collects certificates from TLS sessions, for the keywords [12] that explicitly indicate code signing usage. We identified 122 code signing certificates used for TLS, including for the website of "*marketedge.com*" in the Alexa Top 1 Million list. Two different certificates are used for the domain; a code signing certificate is used for the domain without "www" while a proper TLS certificate is used for the domain with "www". This is surprising because these certificates are considered invalid by the browsers initiating the TLS connections. It suggests that people tend not to differentiate between code signing certificates and TLS certificates.

Signed installer or application. While UAC checks file signatures when programs are installed, code signing best practices [23] recommend signing the installed files as well. This protects the installed files from tampering, for instance from file infectors that copy malicious code into an otherwise benign executable. We investigated if this advice is well kept in practice. The binary reputation data set allows us to determine when an executable file is created on disk by another executable, typically a downloader or installer. We identified 25,051,415 unique installation/download events. Within those events, 2,946,950 events (11.8%) have both the installer/downloader and the payload digitally signed. This represents an upper bound, as some of the digital signatures may no longer be valid. Among the events where both are signed, about 666,350 events (2.66%) have installer/downloader and the payload signed by the same publisher. Meanwhile, 19,752,592 unique unsigned files were installed.

4.3 Malformed digital signatures

For 101 samples in our data set, the signature and the authentihash do not match. In this case, the error message in VT reports is "the digital signature of the object did not verify," which corresponds to Authenticode error code "0x80096010". This error typically results from copying a digital signature and a certificate from one file to another. This does not correspond to a breach of trust in the publisher or the CA, since an adversary does not need access to a private code signing key to produce such malformed signatures. However, these signatures account for 31.1% of the signed malware in our data set. We therefore conduct experiments to determine if such signatures can help malware bypass client-side protections provided by browsers, operating systems and anti-virus products.

Browser protections. Google Chrome and Microsoft IE9 include components called Safe Browsing and SmartScreen, respectively, which protect against malicious downloads from the Web. To test these protections, we copied a legitimate certificate and signature to a benign and simple calculator, which does not require elevated privileges (i.e. Administrator mode in Windows). This resulted in a sample with a malformed digital signature. We then tried to download this sample from the Web to see how the browsers react. Both Safe Browsing and SmartScreen blocked our sample. However, we found that it is possible to bypass the protection by removing the extension. If we remove the file extension (`.exe`), the browsers do not block the download.

Operating system protections. In Windows 7, 8.1, and 10, the OS alerts the user with a message saying the file is unsigned when a file downloaded from the Web is executed. However, if the file does not originate from the Web—e.g. it was copied from a USB drive—then the execution does not trigger any warnings. We also tested the behavior of executables that require elevated privileges by adding a malformed signature to a Windows installer. This program triggers UAC when it asks for Administrator rights, regardless of where the file originated. UAC displays a message saying that the file is from a unknown source. We note that these are the same warnings displayed for unsigned binaries. While Windows appears to detect malformed signatures, it does not block the executables that carry such signatures. If the user chooses to ignore the warnings, no further checks are performed. The OS protections exhibited the same behavior in our experiments with improperly signed malware, which are described next.

In summary, Windows provides minimal protections against executables using forged signatures, while browser defenses apply only to files downloaded from the Web and can be bypassed. The last line of defense, therefore, is anti-virus products.

Anti-virus protections. We conducted an experiment to determine if malformed signatures can affect the AV detections. We first downloaded five random unsigned ransomware samples recently reported to VT. These binaries are known to be malicious—they are detected by 56–58 AV products invoked by VirusTotal. We extracted two expired certificates and the corresponding signatures. These were issued to different publishers and had already been used in malformed signatures in the wild. From each ransomware samples we created two new samples, each including one certificate and its signature, for a total of ten new samples.

Surprisingly, we found that this simple attack prevents many anti-virus products from detecting the malware. Table 3 lists the 34 AVs that detect the unsigned ransomware but fail to detect the same sample after we include the incorrect signatures. We did not observe a significant difference between the impact of the two certificates. However, the impact of this attack varies with the AV products. The top three AVs affected are *nProtect*, *Tencent*, and *Paloalto*. They detected unsigned ransomware samples as malware, but considered eight of out our ten crafted samples as benign. Even well-known AV engines, e.g. Kaspersky, Microsoft, Symantec, and Commodo, allow some of these samples to bypass detection. On average, the

[11] USERTrust, DigiCert, Verisign, Comodo, Entrust, GeoTrust, GlobalSign, Go Daddy, Microsoft, Trustwave, Starfield, StarCom, Symantec, and Thawte

[12] 443.https.tls.certificate.parsed.extensions.extended_key_usage.code_signing:true

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: Bogus Digest Detection (AV and the number of detection fail).

malformed signatures reduced the VirusTotal detection rate r_{mal} by 20.7%. We believe that this is due to the fact that AVs take digital signatures into account when filter and prioritize the list of files to scan, in order to reduce the overhead imposed on the user's host. However, the incorrect implementation of Authenticode signature checks in many AVs gives malware authors the opportunity to evade detection with a simple and inexpensive method.

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.

4.4 Properly signed malware

189 malware samples in our data set carry correct digital signatures, generated using 111 unique certificates. To generate these signatures, adversaries must have controlled the private keys of these certificates. We will analyze the weaknesses in the code-signing PKI that contributed to this abuse in Section 4.5. But first we investigate how these certificates are used in the wild and for how long users are exposed to these threats.

At the time of writing, 27 of these certificates had been revoked. While all the abusive certificates in our data set had expired, executable files signed with one of the 84 certificates that were not revoked may still be valid, as long as they carry a trusted timestamp obtained during the validity of the certificate. For example, the digital signatures from 109 malware samples in our data set remain valid. We notified the CAs of the compromised certificates and asked them for revocation of the certificates except for two CAs (GlobalSign and GoDadday) due to their abuse report system errors.

Malware families. We determined the malware family from the AV labels, using *AVClass* [32]. We identify a total of 116 unique families in the 189 properly signed malware samples. The most prevalent family is *delf* (7 samples), followed by *fareit* (4 samples).

Figure 4 illustrates the number of unique certificates used per malware family. 103 families utilize a single certificate, and 13 use more than two certificates. Among the families with multiple certificates we observe droppers (autoit, banload, agentb, dynamer, delf), bots (Zeus), and fake AVs (smartfortress and onescan). Similar types of malware appear again in the list of families with a single certificate. However, here we also find malware used in targeted attacks. For example, Krbanker was reported to be involved in targeted attacks against customers of South Korean banks. Shylock is also known as a banking trojan that targeted customers of UK banks. The large faction (88.8%) of malware families relying on a

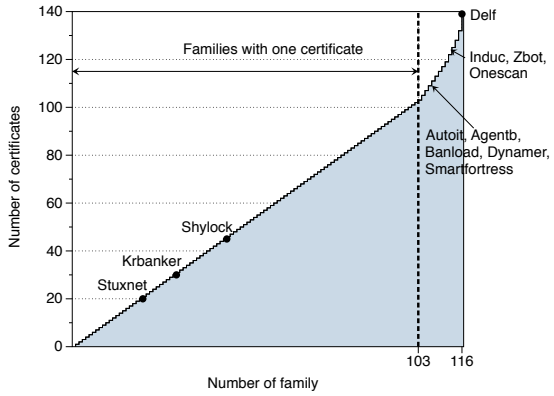

Figure 4: Number of unique certificates per family.

single certificate suggests that, in most cases, abusive certificates are controlled by the malware authors rather than by third parties.

Certificates. On average, an abusive certificate signs samples from 1.5 malware families. Most certificates (79.3%) were issued to publishers in five countries (China, Korea, USA, Brazil, and UK). We believe that this observation reflects the reputation of these publishers, which makes them attractive targets for abuse. This is particularly important for targeted attacks against users or organizations located in one of these countries.

In a total of the 189 properly signed malware, most (111, 66.8%) were signature-timestamped through TSA; *Verisign* was the preferred TSA for most samples (38, 34.2%). This suggests that malware authors value the extended validity provided by trusted timestamps and that they are not concerned about submitting hashes of their malware samples to the timestamping authorities.

Certificate lifecycle. To determine how long users are exposed to these threats, we examine the lifecycle of these abusive certificates. For each certificate, we investigate the expiration date specified in the certificate, the revocation date specified in the CRL (if available), and the dates when benign and malicious binaries are signed with these certificates. If a binary has a trusted timestamp, we use that timestamp as the signing date. This corresponds to 66.8% of the binaries in our data set. For the other samples, we inspect their first appearance in WINE and the first submission to VT; we use the earliest timestamp as the signing date.

Figure 5 illustrates the timelines reconstructed in this manner. For example, the compromised certificate used by Stuxnet had previously been utilized to sign several legitimate binaries. These binaries have been signed both before and after the malware's signing date. After the abuse was discovered, the certificate was revoked as of the date when the malware was signed, which also invalidated all the benign binaries signed after that date. We note that the revocation date indicates when the certificate should cease to be valid and not when the certificate was added to a CRL. In other words, revocation dates do not allow us to determine for how long users were exposed to the abuse.

While the Stuxnet incident raised awareness about digitally signed malware, we observe that this problem was prevalent in the wild before Stuxnet. We found a certificate from Skyline Sotftware

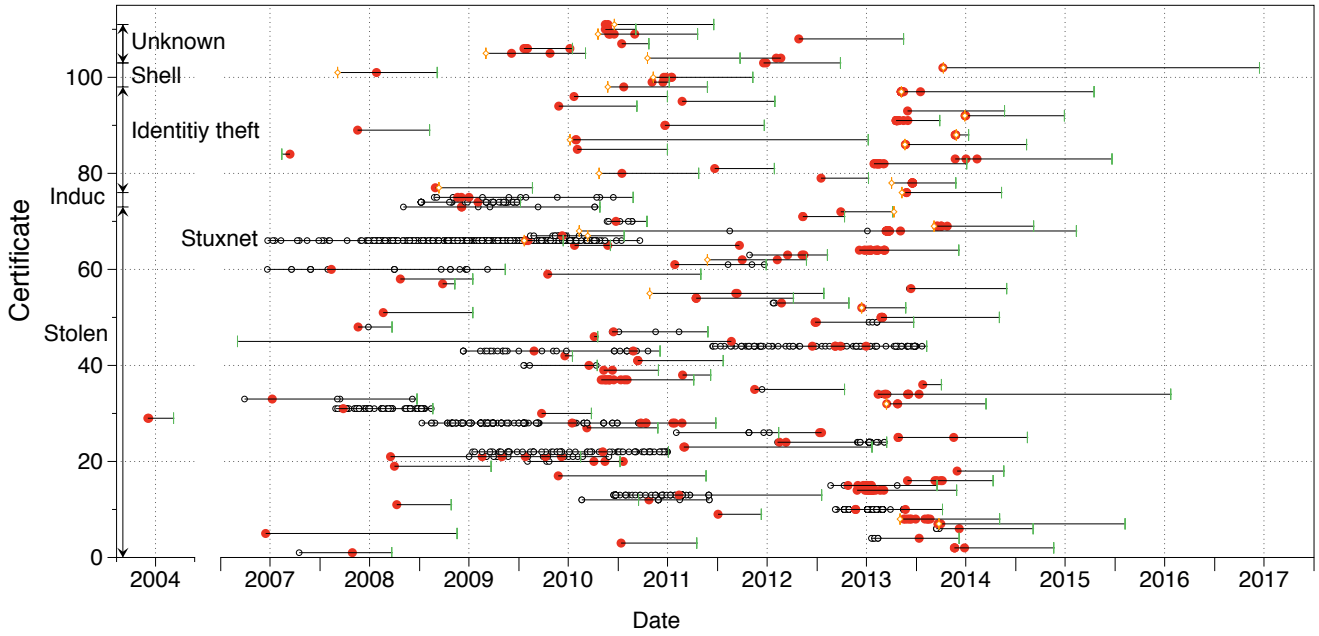

Figure 5: Lifecycle of the abusive certificates. (The red-filled circle, empty circle, green bar, and orange diamond indicate malware, benign sample, expiration date, and revocation date, respectively.)

Systems compromised to sign a malware at the year 2003, which is before Stuxnet appeared. Moreover, in Figure 3, we have 195 signed malware which appeared before the year 2010, when Stuxnet was discovered. Additionally, we observed two interesting behaviors. 5 certificates were used to sign malware that was not timestamped and was seen in the wild after the certificate expired. Other than the opportunity to evade some AV products as discussed in Section 4.3 (which does not require a certificate), there is no motivation for the malware writers to release their code after the expiration date. We therefore believe that these samples correspond to stealthy attacks and were present in the wild for a long time but managed to evade detection. We also find 7 certificates with ineffective revocations. In these cases, the revocation dates were set after the timestamping dates of the malware samples, which allowed the malware to remain valid after revocation. This illustrates the challenge of estimating when certificates are compromised: setting the revocation date too early may invalidate many benign executables (as in Stuxnet's case), but setting it too late may prevent the invalidation of certain malware samples in the wild.

To determine for how long the compromised certificates remain a threat, we perform *survival analysis* [17]. This statistical technique allows us to estimate the probability that an abused certificate will "survive" (i.e. it will not be revoked) after a given number of days. We consider the signing date (estimated as described above) of the oldest malware sample signed with the same certificate as the "birth" of the abuse. We estimate "death events"—the dates when certificates are added to CRLs—as follows. For a revoked certificate, we collect the scan date and the state of the certificate in VirusTotal for all the binaries signed with the certificate. We sort the scan dates, and take the very last date when state was "valid" right before the

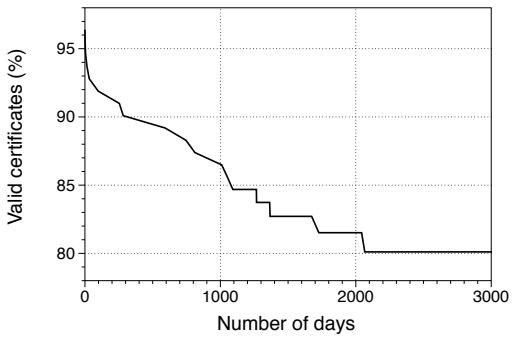

Figure 6: Estimation of the threat effectiveness.

first scan date where the state is "revoked". We then calculate the time difference in days between birth and death events for the abused certificate. This represents a conservative estimation of the threat exposure, as the birth is an upper bound for the compromise date and the death is a lower bound for the revocation date. We also account for the fact that we cannot observe the death of some of the abusive certificates—the ones that are not yet revoked. In these cases, we do not know how big the revocation delay is, but we know that the certificates were not yet revoked on May 18, 2017; in survival analysis terminology, these data points are *censored*. We compute the Kaplan-Meier estimator [17] of the survival function, as it can account for censored data in the estimation.

We present the estimation in Figure 6. The probability that a certificate is not yet revoked decreases to 96% after the first day,

Compromised		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 4: Type of abuse and the top 5 frequent CAs.

owing to some certificates for which all the VT reports indicated a "revoked" status. The probability continues to decrease slowly for 5.6 years, then it stops decreasing after reaching 80%. This suggests that the threat of abused certificates is very resilient: only 20% of the certificates used to sign malware are likely to be revoked during our observation period, which spans 15 years. If the malware samples signed with the remaining certificates also carry a trusted timestamp, they remain valid today.

4.5 Measuring the abuse factors

To gain insight into the attackers' methods, we utilize the algorithm from Section 3.4 to identify the PKI weakness exploited in abusing the 111 certificates.

Publisher-side key mismanagement. We considered a certificate as falling into this category if it was used for signing both benign and malicious programs. Of the 111 clusters (i.e., certificates), at least 75 certificates were used for signing both.

We examined the validity of the samples in this case. Surprisingly, as of this writing, most (50, 66.7%) are still valid while only some certificates (10, 13.3%) were explicitly revoked. Although all certificates were already expired, the executable files signed with the certificates are still valid beyond the expiration date due to trust time stamping. Therefore, users will be displayed a message saying the publisher is verified and legitimate when they run the malware.

To categorize the certificates, we manually and deeply investigated malware samples signed with each certificate.

- ***Compromised certificate.*** Out of 75 certificates, we believe that most (72) were compromised and used for signing malware. Using this method, we found the *Stuxnet* malware, which is known to have been signed with a compromised certificate [10]. In our dataset, it was signed with the *Realtek Semiconductor Corp.* certificate issued by Verisign. Our systems also detected that an Australian department's private key was also stolen and used to sign malware, labeled as *autoit*.
- ***Infected developer machines.*** We also identified developer machines that were infected and used to sign malicious code with a legitimate certificate. This resulted in signed malicious code shipped with a legitimate package. We found three certificates used to sign *W32/Induc.A* that infects only Delphi developer machines. We investigated the prevalence of *Induc* in the wild using the WINE dataset. About 1,554 binaries were detected as a variant of Induc and 93,016 machines were infected. Among these machines, 180 were Delphi compiler machines. This suggests that infecting developer machines is an effective method for amplifying the impact of signed malware and ultimately infecting 517× more machines.

As depicted in Table 4, 70% of them are issued by Symantec group (Thawte and Verisign).

CA-side verification failure. This weakness is caused by CAs' failure in verifying the publisher's identity. CAs may issue certificates to an adversary who impersonates someone else or uses shell company information.

We believe that 27 certificates were issued to malicious publishers due to verification failures. To distinguish between identity theft and shell companies, we also manually investigated each certificate by searching for the publisher names in the Internet or in *openCorporates* to see if the publishers are legitimate. 22 certificates out of 27 certificates issued through identify theft and 5 certificates were done through shell company information. For example, a certificate issued to a delivery service company in Korea was used to sign malware. Another certificate was issued to an Oregon resident. We believe that the company is not related to software developments, and has never released any software. Moreover, we doubt that for a malware writer it is worth to reveal his/her identity. Therefore, we consider that these are cases of identity theft.

We investigate the process for issuing code signing certificates to understand the weakness that allowed these certificates to be issued. The policy might have changed from that of the time when these certificates were issued; however, we set an assumption that the policy will not downgrade. Around the end of 2016, Certificate Authority Security Council (CASC) announced a minimum requirements for code signing certificates.[13] The new requirements include:

- *Stronger protection of private keys.* Now the private keys should only be stored on secure cryptographic hardware, e.g., a USB token or Hardware security module (HSM).
- *Careful identity verification.* The new requirement asks CAs to strictly verify the identity of the publisher, which includes checking the legal identity of the publisher and the cross-checking with the known bad publisher list.
- *Better response to the abuse.* The CAs now have to quickly respond to the revocation request. They have to revoke the certificate within two days, or notify the reporter that the investigation has started.
- *TSA is now a requirement.* Now every code signing provider must operate a RFC-3161 compliant TSA.

These guidelines suggest CASC is aware of the abuse happening in the wild. Increased protection of the private keys would help prevent certificates from being compromised; strict identification check will make it hard to acquire a certificate by impersonating. Moreover, Microsoft announced the CAs must follow these guidelines starting February 1, 2017.

We investigated the policies of the top ten code signing CAs listed in Table 2. We found that only Certum follows the guidelines. The result of the survey suggest that the code signing is still vulnerable to the certification thefts and fraudulent applications.

[13] https://casecurity.org/2016/12/08/leading-certificate-authorities-and-microsoft-introduce-new-standards-to-protect-consumers-online/

Revocation. We investigate the revocation practice in the field. By category, 15.3%, 40.9%, and 80.0% of the certificates were revoked for compromised, identity theft, and shell company, repectively. Interestingly, the revocation rate was significantly less for the compromised certificates compared to the other abuse types.

Verification and further investigation. We decided to contact the owners of the compromised certificates we found to inform them that their certificates were used for signing malware, and to better understand the code signing ecosystem. We manually searched for their websites, and sent 23 publishers email to ask them to check if the certificate was owned by them. We were unable to send more publishers email due to unrecognizable publisher names, closures, etc.

As of this writing, we received eight replies from those who we emailed. All of them said that they issued and used the certificates to sign their benign programs. Three of them were already aware that their certificates were abused by adversaries and revoked by their CAs because the CAs notified them that the certificates were compromised. One publisher told us that their private key may have been stolen because the machines storing the code-signing keys were accessible to all developers. Most publishers asked us to send the malware samples signed with their certificate for further investigation. Two publishers claimed that it was false positive results of AVs. We also manually investigated to see if those files signed with each certificates had false positives. Microsoft, IE, Chrome, GMail, and Dropbox detected them as malware and did not allow us to share or download them from the Internet. Moreover, one of them was obviously labeled as trojan, called *Vilsel*. Therefore, we believe that the signed samples that the two publishers claimed as false positives are obvious malware.

5 DISCUSSION

Improvements for code signing PKI. The major difference between TLS and code signing certificates is that it is hard for the owners of code signing certificates to know where and how they are abused, while the owners of TLS certificates are readily aware of the certificate abuse because the certificates are tightly bound to a certain domain name. If the owners are informed and aware of what program code are signed with their certificate, the owners can easily see if their certificate is abused. To achieve this goal, we suggest to make signing tools (e.g., *signtool.exe* in Windows) log all history and inform the original owners of (1) what and when program code is signed and (2) who tries to sign (e.g., IP address). As long as the owners have all history of singing, they can readily see if it is abused by periodically checking the log. This model is effective for the compromised certificates, but may not work for other cases like identify theft and shell company. In these cases, we can introduce transparency in code signing. In this idea, the hash value of program code and the certificates are logged when signed. Other third parties can periodically audit the log and identify code signing abuse.

Other threats. In addition to the three threats described in Section 2.3, a Certificate Authority may be compromised or may act maliciously. This is a severe threat, as it would allow the adversary to *issue* fake code signing certificates for signing malware. For example, they could issue certificates setting the publisher name (common name) to a reputable software company like Microsoft, Google, etc. In the past, hackers have compromised two CAs (DigiNotar and Comodo) and have issued fake TLS certificates. However, we do not observe this threat in our data set.

Recently, Google and CWI Amsterdam developed code that can generate the two different files with the same SHA-1 hash value [12]. We have not observed any malware sample that exploits this SHA-1 collision attack. However, this represents an important threat, as demonstrated in the past by the Flame malware, which conducted an attack against MD5 shortly after it became practical to find MD5 collisions.

6 RELATED WORK

We discuss related work in three key areas: measuring the ecosystems of HTTPS and TLS certificates; measuring code signing abuse, specifically Authenticode and Android; and attempts at the improvements for PKIs.

Measurements of the TLS certificate ecosystem. The TLS certificates and HTTPS ecosystems have been thoroughly studied due to the introduction of many network scanners such as ZMap. ZMap can scan the entire IPv4 address space less than one hour [8]; researchers can readily obtain the large, but limited number of TLS certificate datasets while code signing certificates in the wild are hard to collect. Durumeric et al. have uncovered the bad practices in the HTTPS ecosystem using the datasets that ZMap produced [7]. Two measurement studies with regard to the impacts of *Heartbleed*, has been conducted [6, 40]. *Heartbleed* is a serious security bug in the *OpenSSL* cryptography library.

Code signing abuse. For Authenticode code signing abuse, similar to our work, the computer security company, *Sophos* examined the signed Windows PE files collected by the company from 2008 to 2010 [37]. They observed that the number of signed malicious PEs including trojan, adware, spyware, etc., increased over time in the measurement period. Kotzias et al. and Alrawi et al. evaluated 356 thousands and three million Windows PE samples respectively [2, 19]. In their findings, signed malware was not prevalent since only a small number of signed malware were found, while most signed samples were PUPs. Unlike the Authenticode code signing that obtains a certificate from a CA, Android applications are signed with self-signed certificates; thus, there is no effective revocation system. Many Android developers use the same key for their many applications, which can lead to unexpected security threats such as signature-based permissions [9].

PKI Improvements. The efforts to prevent certificate abuse have focused primarily on the Web PKI. The proposals can be classified into three groups; client-based, CA-based, and domain-based. In the client-based approaches, *Perspective* [36] and *Convergence* [1] require clients to ask a third party (*notary authority*, independent from CAs) to check if the certificate of a service (e.g., web service) that they access is legitimate. *Certificate Transparency* (CT) [22] is the representative proposals in the domain-based class. In CT, all TLS certificates are logged in a signed chronologically ordered Merkle Hash Tree when they are issued, and the logs are publicly opened. That enables anyone (e.g., CAs, owners of certificates, etc.) to monitor and audit the logs. For Android application, Fahl et al. have proposed *Application Transparency* (AT) [9]. It is based on CT,

and aims to prevent the threat model of "targeted-and-stealthy." However, the all proposed improvements cannot be applied to code signing certificates because they were designed particularly for TLS certificates or Android applications.

Comparatively, there have been fewer effort to improve the code-signing PKI. Papp at al. have presented a repository system, called ROSCO where software code are uploaded by publishers, and looked up by end-users to check if the certificates are compromised [29]. However, the repository is maintained by a single group so that the no one can audit the repository system. Moreover, publishers are required to voluntarily upload their program code to the repository. In other words, it is not able to cover all program code in the wild. Another proposal was *CT for Binary Codes* [38]. This systems is based on CT to support logging program code so that anyone can audit the system. However, the proposal does not have an explicit protocol for publishers how to log program code.

7 CONCLUSIONS

We introduce a threat model that highlights three types of weaknesses in the code signing PKI: *inadequate client-side protections*, *publisher-side key mismanagement*, and *CA-side verification failures*. We propose an algorithm for analyzing the abuse recorded in the code signing ecosystem and for identifying the weaknesses exploited by the adversary. Using the algorithm, we conducted a systematic measurement of the three weakness types.

We identify 325 signed malware samples in our data set. Of these, 189 (58.2%) samples are properly signed while 136 carry malformed digital signatures, which do not match the binary's digest. Such malformed signatures are useful for an adversary: we find that simply copying an Authenticode signature from a legitimate sample to an unsigned malware sample may help the malware bypass AV detection. The 189 samples signed correctly rely on 111 unique certificates. We find that only 27 certificates were revoked; malware signed with one of the remaining 84 certificates would still be trusted today as long as is carries a trusted timestamp. A large fraction (88.8%) of malware families rely on a single certificate, which suggests that the abusive certificates are mostly controlled by the malware authors rather than by third parties.

Of the 111 certificates used to sign malware, 75 were abused due to publisher-side mismanagement (72 were compromised and 3 were used on infected developer machines). The compromised certificates identified with our algorithm include the one used by Stuxnet. Additionally, 27 certificates were issued to malware authors due to the CAs' verification failure (22 through identify theft and 5 using shell company information). For further investigation, we sent the publishers of the certificates emails to inform them that their certificates were abused. We received replies from eight publishers. They confirmed that the certificates were issued to them and five of them are unaware of the abuse.

ACKNOWLEDGMENTS

We thank Dave Levin, Michelle Mazurek, Dylan O'Reagan, and the anonymous reviewers for their feedback. We also thank VirusTotal for access to their service and Symantec for making data available through the WINE platform. This research was partially supported by the National Science Foundation (award CNS-1564143) and the Department of Defense.

REFERENCES

[1] 2017. Convergence. https://github.com/moxie0/Convergence. (2017).
[2] Omar Alrawi and Aziz Mohaisen. 2016. 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)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 451–456. https://doi.org/10.1145/2872518.2888610
[3] Frank Cangialosi, Taejoong Chung, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. 2016. 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)*. ACM, New York, NY, USA, 628–640. https://doi.org/10.1145/2976749.2978301
[4] Tudor Dumitraș and Darren Shou. 2011. Toward a Standard Benchmark for Computer Security Research: The Worldwide Intelligence Network Environment (WINE). In *EuroSys BADGERS Workshop*. Salzburg, Austria.
[5] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. 2015. A Search Engine Backed by Internet-Wide Scanning. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 542–553. https://doi.org/10.1145/2810103.2813703
[6] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC '14)*. ACM, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755
[7] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. 2013. Analysis of the HTTPS Certificate Ecosystem. In *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC '13)*. ACM, New York, NY, USA, 291–304. https://doi.org/10.1145/2504730.2504755
[8] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2013. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13)*. USENIX Association, Berkeley, CA, USA, 605–620. http://dl.acm.org/citation.cfm?id=2534766.2534818
[9] Sascha Fahl, Sergej Dechand, Henning Perl, Felix Fischer, Jaromir Smrcek, and Matthew Smith. 2014. 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)*. ACM, New York, NY, USA, 1143–1155. https://doi.org/10.1145/2660267.2660311
[10] Nicholas Falliere, Liam O'Murchu, and Eric Chien. 2011. W32.Stuxnet Dossier. Symantec Whitepaper. (February 2011). http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf
[11] DAN GOODIN. 2015. Stuxnet spawn infected Kaspersky using stolen Foxconn digital certificates. (Jun 2015). https://arstechnica.com/security/2015/06/stuxnet-spawn-infected-kaspersky-using-stolen-foxconn-digital-certificates/
[12] Google. 2017. Announcing the first SHA1 collision. (February 2017). https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html
[13] P. Hoffman and J. Schlyter. 2012. *The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA*. RFC 6698. RFC Editor. http://www.rfc-editor.org/rfc/rfc6698.txt http://www.rfc-editor.org/rfc/rfc6698.txt.
[14] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. 2011. 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*. ACM, 427–444. http://dl.acm.org/citation.cfm?id=2068856
[15] Lin Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. 2014. Analyzing forged SSL certificates in the wild. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 83–97. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6956558
[16] Burt Kaliski. 1998. *PKCS #7: Cryptographic Message Syntax Version 1.5*. RFC 2315. RFC Editor. http://www.rfc-editor.org/rfc/rfc2315.txt http://www.rfc-editor.org/rfc/rfc2315.txt.
[17] David. G. Kleinbaum and Mitchell Klein. 2011. *Survival Analysis: A Self-Learning Text* (3 ed.). Springer.
[18] Platon Kotzias, Leyla Bilge, and Juan Caballero. 2016. Measuring PUP prevalence and PUP distribution through Pay-Per-Install services. In *Proceedings of the USENIX Security Symposium*.
[19] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. 2015. Certified PUP: Abuse in Authenticode Code Signing. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 465–478. https://doi.org/10.1145/2810103.2813665
[20] Michael Kranch and Joseph Bonneau. 2015. Upgrading HTTPS in mid-air: An Empirical Study of Strict Transport Security and Key Pinning. In *Network and Distributed System Security (NDSS) Symposium*. Internet Society. https://doi.org/10.14722/ndss.2015.23162

[21] Bum Jun Kwon, Virinchi Srinivas, Amol Deshpande, and Tudor Dumitraş. 2017. Catching Worms, Trojan Horses and PUPs: Unsupervised Detection of Silent Delivery Campaigns. In *Proc. NDSS*.

[22] B. Laurie, A. Langley, and E. Kasper. 2013. *Certificate Transparency*. RFC 6962. RFC Editor.

[23] Eric Lawrence. 2011. Everything you need to know about Authenticode Code Signing. (Mar 2011). https://blogs.msdn.microsoft.com/ieinternals/2011/03/22/everything-you-need-to-know-about-authenticode-code-signing/

[24] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. 2015. An End-to-End Measurement of Certificate Revocation in the Web's PKI. ACM Press, 183–196. https://doi.org/10.1145/2815675.2815685

[25] Microsoft. 2001. Erroneous VeriSign-Issued Digital Certificates Pose Spoofing Hazard. (2001). https://technet.microsoft.com/en-us/library/security/ms01-017.aspx

[26] Microsoft. 2008. Windows Authenticode Portable Executable Signature Format. (Mar 2008). http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx

[27] Microsoft. 2011. Virus: Win32/Induc.A. (April 2011). https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?name=Virus%3AWin32%2FInduc.A

[28] Evangelos E. Papalexakis, Tudor Dumitras, Duen Horng (Polo) Chau, B. Aditya Prakash, and Christos Faloutsos. 2103. Spatio-temporal Mining of Software Adoption & Penetration. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. Niagara Falls, CA.

[29] Dorottya Papp, Balázs Kócsó, Tamás Holczer, Levente Buttyán, and Boldizsár Bencsáth. 2015. ROSCO: Repository Of Signed Code. In *Virus Bulletin Conference, Prague, Czech Republic*.

[30] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. 2010. Bootstrapping Trust in Commodity Computers. In *IEEE Symposium on Security and Privacy*. 414–429.

[31] Kaspersky Lab's Global Research and Analysis Team. 2015. The Duqu 2.0 persistence module. (Jun 2015). https://securelist.com/blog/research/70641/the-duqu-2-0-persistence-module/

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

[33] Swiat. 2012. Flame malware collision attack explained. (Jun 2012). https://blogs.technet.microsoft.com/srd/2012/06/06/flame-malware-collision-attack-explained/

[34] 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. 2016. 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*. 721–739. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/thomas

[35] VirusTotal. 2017. www.virustotal.com. (2017).

[36] Dan Wendlandt, David G. Andersen, and Adrian Perrig. 2008. Perspectives: Improving SSH-style Host Authentication with Multi-path Probing. In *USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, Berkeley, CA, USA, 321–334. http://dl.acm.org/citation.cfm?id=1404014.1404041

[37] Mike Wood. 2010. Want My Autograph? The Use and Abuse of Digital Signatures by Malware. *Virus Bulletin Conference September 2010* September (2010), 1–8. http://www.sophos.com/medialibrary/PDFs/technicalpapers/digital

[38] Liang Xia, Dacheng Zhang, Daniel Gillmor, and Behcet Sarikaya. 2017. *CT for Binary Codes*. Internet-Draft draft-zhang-trans-ct-binary-codes-04. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-zhang-trans-ct-binary-codes-04.txt http://www.ietf.org/internet-drafts/draft-zhang-trans-ct-binary-codes-04.txt.

[39] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. 2009. 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)*. ACM, New York, NY, USA, 15–27. https://doi.org/10.1145/1644893.1644896

[40] Liang Zhang, David Choffnes, Dave Levin, Tudor Dumitras, Alan Mislove, Aaron Schulman, and Christo Wilson. 2014. Analysis of SSL Certificate Reissues and Revocations in the Wake of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC '14)*. ACM, New York, NY, USA, 489–502. https://doi.org/10.1145/2663716.2663758