# The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching

Antonio Nappa*§, Richard Johnson†, Leyla Bilge‡, Juan Caballero*, Tudor Dumitraș†

*IMDEA Software Institute †University of Maryland, College Park
‡Symantec Research Labs §Universidad Politécnica de Madrid

antonio.nappa@imdea.org, rbjohns8@cs.umd.edu,
leylya_yumer@symantec.com, juan.caballero@imdea.org, tdumitra@umiacs.umd.edu

***Abstract*—Vulnerability exploits remain an important mechanism for malware delivery, despite efforts to speed up the creation of patches and improvements in software updating mechanisms. Vulnerabilities in client applications (e.g., browsers, multimedia players, document readers and editors) are often exploited in spear phishing attacks and are difficult to characterize using network vulnerability scanners. Analyzing their lifecycle requires observing the deployment of patches on hosts around the world. Using data collected over 5 years on 8.4 million hosts, available through Symantec's WINE platform, we present the first systematic study of patch deployment in client-side vulnerabilities.**

**We analyze the patch deployment process of 1,593 vulnerabilities from 10 popular client applications, and we identify several new threats presented by multiple installations of the same program and by shared libraries distributed with several applications. For the 80 vulnerabilities in our dataset that affect code shared by two applications, the time between patch releases in the different applications is up to 118 days (with a median of 11 days). Furthermore, as the patching rates differ considerably among applications, many hosts patch the vulnerability in one application but not in the other one. We demonstrate two novel attacks that enable exploitation by invoking old versions of applications that are used infrequently, but remain installed. We also find that the median fraction of vulnerable hosts patched when exploits are released is at most 14%. Finally, we show that the patching rate is affected by user-specific and application-specific factors; for example, hosts belonging to security analysts and applications with an automated updating mechanism have significantly lower median times to patch.**

## I. Introduction

In recent years, considerable efforts have been devoted to reducing the impact of software vulnerabilities, including efforts to speed up the creation of patches in response to vulnerability disclosures [16]. However, vulnerability exploits remain an important vector for malware delivery [2], [6], [20]. Prior measurements of patch deployment [14], [29], [36], [37], [46] have focused on server-side vulnerabilities. Thus, the lifecycle of vulnerabilities in client-side applications, such as browsers, document editors and readers, or media players, is not well understood. Such client-side vulnerabilities represent an important security threat, as they are widespread (e.g., typical Windows users are exposed to 297 vulnerabilities in a year [17]), and they are often exploited using spear-phishing as part of targeted attacks [7], [21], [26].

One dangerous peculiarity of client-side applications is that the same host may be affected by several instances of the same vulnerability. This can happen if the host has installed multiple instances of the same application, e.g., multiple software lines or the default installation and an older version bundled with a separate application. Multiple instances of the vulnerable code can also occur owing to libraries that are shared among multiple applications (e.g., the Adobe library for playing Flash content, which is included with Adobe Reader and Adobe Air installations). These situations break the *linear model for the vulnerability lifecycle* [3], [6], [16], [32], [39], which assumes that the vulnerability is disclosed publicly, then a patch released, and then vulnerable hosts get updated. In particular, vulnerable hosts may only patch one of the program installations and remain vulnerable, while patched hosts may later re-join the vulnerable population if an old version or a new application with the old code is installed. This extends the window of opportunity for attackers who seek to exploit vulnerable hosts. Moreover, the owners of those hosts typically believe they have already patched the vulnerability.

To the best of our knowledge, we present the first systematic study of patch deployment for client-side vulnerabilities. The empirical insights from this study allow us to identify several new threats presented by multiple installations and shared code for patch deployment and to quantify their magnitude. We analyze the patching by 8.4 million hosts of 1,593 vulnerabilities in 10 popular Windows client applications: 4 browsers (Chrome, Firefox, Opera, Safari), 2 multimedia players (Adobe Flash Player, Quicktime), an email client (Thunderbird), a document reader (Adobe Reader), a document editor (Word), and a network analysis tool (Wireshark).

Our analysis combines telemetry collected by Symantec's security products, running on end-hosts around the world, and data available in several public repositories. Specifically, we analyze data spanning a period of 5 years available through the Worldwide Intelligence Network Environment (WINE) [12]. This data includes information about binary executables downloaded by users who opt in for Symantec's data sharing program. Using this data we analyze the deployment of subsequent versions of the 10 applications on real hosts worldwide. This dataset provides a unique opportunity for studying the patching process in client applications, which are difficult to characterize using the network scanning techniques employed by prior research [14], [29], [36], [37], [46].

The analysis is challenging because each software vendor has its own software management policies, e.g., for assigning

program versions, using program lines, and issuing security advisories, and also by the imperfect information available in public vulnerability databases. To address these challenges we have developed a generic approach to map files in a host to vulnerable and not vulnerable program versions, using file meta-data from WINE and VirusTotal [44], and the NVD [31] and OSVDB [33] public vulnerability databases. Then, we aggregate vulnerabilities in those databases into clusters that are patched by the same program version. Finally, using a statistical technique called *survival analysis* [24], we track the global decay of the vulnerable host population for each vulnerability cluster, as software updates are deployed.

Using this approach we can estimate for each vulnerability cluster the delay to issue a patch, the rate of patching, and the vulnerable population in WINE. Using exploit meta-data in WINE and the Exploit Database [15], we estimate the dates when exploits become available and we determine the percentage of the host population that remains vulnerable upon exploit releases.

We quantify the race between exploit creators and the patch deployment, and we find that the median fraction of hosts patched when exploits are released is at most 14%. All but one of the exploits detected in the wild found more than 50% of the host population still vulnerable. The start of patching is strongly correlated with the disclosure date, and it occurs within 7 days before or after the disclosure for 77% of the vulnerabilities in our study—suggesting that vendors react promptly to the vulnerability disclosures. The rate of patching is generally high at first: the median time to patch half of the vulnerable hosts is 45 days. We also observe important differences in the patching rate of different applications: none of the applications except for Chrome (which employs automated updates for all the versions we consider) are able to patch 90% of the vulnerable population for more than 90% of vulnerability clusters.

Additionally, we find that 80 vulnerabilities in our dataset affect common code shared by two applications. In these cases, the time between patch releases in the different applications is up to 118 days (with a median of 11 days), facilitating the use of patch-based exploit generation techniques [8]. Furthermore, as the patching rates differ between applications, many hosts patch the vulnerability in one application but not in the other one. We demonstrate two novel attacks that enable exploitation by invoking old version of applications that are used infrequently, but that remain installed.

We also analyze the patching behavior of 3 user categories: professionals, software developers, and security analysts. For security analysts and software developers the median time to patch 50% of vulnerable hosts is 18 and 24 days, respectively, while for the general user it is 45 days—more than double.

In summary, we make the following contributions:

- We conduct a systematic analysis of the patching process of 1,593 vulnerabilities in 10 client-side applications, spanning versions released on a 5-year period.
- We demonstrate two novel attacks that enable exploitation by invoking old versions of applications that are used infrequently, but remain installed.
- We measure the patching delay and several patch deployment milestones for each vulnerability.

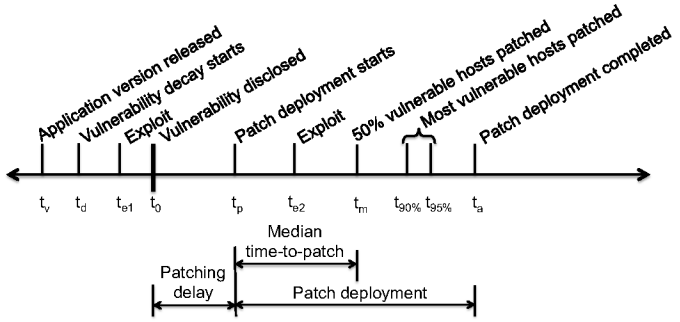



Fig. 1. Events in the vulnerability lifecycle. We focus on measuring the patching delay $[t_0, t_p]$ and on characterizing the patch deployment process, between $t_p$ and $t_a$.

- We propose a novel approach to map files on end-hosts to vulnerable and patched program versions.
- Using these techniques, we quantify the race between exploit creators and the patch deployment, and we analyze the threats presented by multiple installations and shared code for patch deployment.
- We release a dataset with curated vulnerability information and detailed program release histories for the 10 applications at http://clean-nvd.com/.

The rest of this paper is organized as follows. Section II describes the security model for patching vulnerabilities. Section III details our datasets and Section IV our approach. Our findings are presented in Section V. Section VI reviews prior work, Section VII discusses implications of our findings, and Section VIII concludes.

## II. Security Model for Patching Vulnerabilities.

Prior research [3], [6], [16], [32], [36], [37], [39], [46] generally assumes a *linear model* for the vulnerability lifecycle, illustrated in Figure 1. In this model, the introduction of a vulnerability in a popular software ($t_v$), is followed by the vulnerability's public disclosure ($t_0$), by a patch release ($t_p$) and by the gradual deployment of the patch on all vulnerable hosts, which continues until all vulnerable application instances have been updated ($t_a$). Important milestones in the patch deployment process include the *median time to patch*, i.e., the time needed to patch half of the vulnerable hosts ($t_m$), and the times needed to patch 90% and 95% of the vulnerable hosts ($t_{90\%}$ and $t_{95\%}$). Additional events may occur at various stages in this lifecycle; for example exploits for the vulnerability may be released before or after disclosure ($t_{e1}, t_{e2}$), and the vulnerable host population may start decaying earlier than $t_p$ if users replace the vulnerable version with a version that does not include the vulnerability ($t_d$). Notwithstanding these sources of variability, the linear model assumes that $t_v < t_0 \leq t_p < t_m < t_{90\%} < t_{95\%} < t_a$.

In practice, however, these events do not always occur sequentially, as a host may be affected by *several instances of the vulnerability*. Software vendors sometimes support multiple product lines; for example, Adobe Reader has several lines (e.g., 8.x, 9.x, 10.x) that are developed and released with some overlap in time and that can be installed in parallel on a host. Additionally, applications are sometimes bundled with other software products, which install a (potentially older)

version of the application in a custom directory. For example, device drivers such as printers sometimes install a version of Adobe Reader, to allow the user to read the manual. Furthermore, some applications rely on common libraries and install multiple copies of these libraries side-by-side. For example, Safari and Chrome utilize the WebKit rendering engine [45], Firefox and Thunderbird share several Mozilla libraries, and libraries for playing Flash files are included in Adobe Reader and Adobe Air. In consequence, releasing and deploying the vulnerability patch on a host does not always render the host immune to exploits, as the vulnerability may exist in other applications or library instances and may be re-introduced by the installation of an older version or a different application. The security implications of this *non-linear* vulnerability lifecycle are not well understood.

### A. Threats of Shared Code and Multiple Installations

The response to vulnerabilities is subject to two delays: the patching delay and the deployment delay. The *patching delay* is the interval between the vulnerability disclosure $t_0$ and the patch release $t_p$ in Figure 1. This delay allows attackers to create exploits based on the public details of the vulnerability and to use them to attack all the vulnerable instances of the application *before* they can be patched. After developing and testing a patched version of the application, the vendor must then deploy this version to all the vulnerable application instances. This *deployment delay* controls the window when vulnerabilities can be exploited *after* patches are available, but before their deployment is completed. In recent years, considerable efforts have been devoted to reducing the vulnerability patching delays, through efforts to speed up the creation of patches in response to vulnerability disclosures [16], and the patch deployment delays, through automated software updating mechanisms [11], [18]. While these techniques are aimed at patching vulnerabilities in the linear lifecycle model, attackers may leverage shared code instances and multiple application installations to bypass these defenses. Next, we review some of these security threats.

**Overhead of maintaining multiple product lines.** When a vendor supports multiple product lines in parallel and a vulnerability is discovered in code shared among them, the vendor must test the patch in each program line. Prior work on optimal patch-management strategies has highlighted the trade-off between the patching delay and the amount of testing needed before releasing patches [32]. The overhead of maintaining multiple product lines may further delay the release of patches for some of these lines.

**Threat of different patch release schedules.** When a vulnerability affects more than one application, patch releases for all these applications seldom occur at the same time. Coordinated patch releases are especially difficult to achieve when applications from different vendors share vulnerabilities, e.g., when the vulnerabilities affect code in third-party libraries. When the patch for the first application is released, this gives attackers the opportunity to employ patch-based exploit generation techniques to attack the other applications, which remain unpatched.

**Threat of multiple patching mechanisms.** When shared code is patched using multiple software update mechanisms, some instances of the vulnerability may be left unpatched. The attacker can use existing exploits to target all the hosts where the default application used for opening a certain type of file (e.g., PDF documents) or Web content (e.g., Flash) remains vulnerable. This situation may happen even after the vendor has adopted automated software updates—for example, when the user installs (perhaps unknowingly, as part of a software bundle) an older version of the application, which does not use automated updates, and makes it the default application, or when the user disables automatic updates on one of the versions installed and forgets about it. The use of multiple software updating mechanisms places a significant burden on users, as a typical Windows user must manage 14 update mechanisms (one for the operating system and 13 for the other software installed) to keep the host fully patched [17].

This problem can also occur with shared libraries because they typically rely on the updating mechanisms of the programs they ship with, but one of those programs may not have automatic updates or they may have been disabled. This scenario is more common with third-party libraries deployed with programs from different vendors using different update policies. However, we have also observed it in a shared vulnerability between Adobe Reader and Adobe Flash (CVE-2011-0609), which happened at a time when Adobe Reader had silent automatic updates enabled by default, but the Adobe Flash plugin did not yet.

**Attacks against inactive program versions through multiple content delivery vectors.** Even if all the applications that are actively used on a host are all up to date, an attacker may deliver exploits by using a vector that will open a different runtime or application, which remains vulnerable. Here, we discuss an attack that allows exploiting a vulnerable version even if the patched version is the default one. The user runs both an up-to-date Flash plugin and an old Adobe Air (a cross-platform runtime environment for web applications), which includes a vulnerable Flash library (npswf32.dll). Adobe Air is used by web applications that want to run as desktop applications across different platforms. In this case the user runs FLVPlayer over Adobe Air. The attacker can deliver the exploit to the user in two ways: as a .flv file to be played locally or as a URL to the .flv file. If the user clicks on the file or URL, the file will be opened with FLVPlayer (associated to run .flv files) and the embedded Flash exploit will compromise the vulnerable Flash library used by Adobe Air. Similarly, Adobe Reader includes a Flash library. The attacker can target hosts that have up-to-date Flash plugins, but old Adobe Reader installations, by delivering a PDF file that embeds an exploit against the old version of the Flash library.

**Attacks against inactive program versions through user environment manipulation.** The previous attack relies on the presence of applications to which the attacker can deliver exploit content (e.g., Flash content). Here, we discuss another attack, which allows the attacker to replace the default, up-to-date, version of an application with a vulnerable one. The user is running two versions of Adobe Reader, a default up-to-date version and a vulnerable version. The attacker convinces the user to install a Firefox add-on that looks benign. The malicious add-on has filesystem access through the XPCOM API [5]. It locates the vulnerable and patched versions of the Adobe Reader library (nppdf32.dll) and overwrites the patched version with the vulnerable one. When the user visits

a webpage, the malicious add-on modifies the DOM tree of the webpage to insert a script that downloads a remote PDF exploit. The exploit is processed by the vulnerable Adobe Reader version and exploitation succeeds. We have successfully exploited a buffer overflow vulnerability (CVE-2009-1861) on Firefox 33.1, Adobe Reader 11.0.9.29 as patched version and Acrobat Reader 7.0.6 as vulnerable version.

These two attacks demonstrate the dangers of inactive application versions that are forgotten, but remain installed. Note that our goal is not to find exploits for all the programs we analyze, but rather to provide evidence that a sufficiently motivated attacker can find avenues to exploit multiple installations and shared code.

### B. Goals and Non-Goals

**Goals.** Our goal in this paper is to determine how effective are update mechanisms in practice and to quantify the threats discussed in Section II-A. We develop techniques for characterizing the patching delay and the patch deployment process for vulnerabilities in client-side applications, and use these techniques to assess the impact of current software updating mechanisms on the vulnerability levels of real hosts. The *patching delay* depends on the vendor's disclosure and patching policies and requires measuring $t_0$ and $t_p$ for each vulnerability. $t_0$ is recorded in several vulnerability databases [31], [33], which often include incomplete (and sometimes incorrect) information about the exact versions affected by the vulnerability. Information about $t_p$ is scattered in many vendor advisories and is not centralized in any database. To overcome these challenges, we focus on analyzing the presence of vulnerable and patched program versions on real end-hosts, in order to estimate the start of patching and to perform sanity checks on the data recorded in public databases.

The *patch deployment process* depends on the vendor (e.g., its patching policy), the application (e.g., if it uses an automated updating mechanism), and the user behavior (e.g., some users patch faster than others). The daily changes in the population of vulnerable hosts, reflected in our end-host observations, give insight into the progress of patch deployment. We focus on modeling this process using statistical techniques that allow us to measure the *rate* at which patching occurs and several patch deployment *milestones*. To characterize the *initial deployment phase* following patch deployment ($t_p$), we estimate the median time to patch ($t_m$). The point of *patch completion* $t_a$ is difficult to define, because we are unable to observe the entire vulnerable host population on the Internet. To characterize the *tail of the deployment process*, we estimate two additional deployment milestones, ($t_{90\%}$ and $t_{95\%}$), as well as the fraction of vulnerabilities that reach these patching milestones. Our focus on application vulnerabilities (rather than vulnerabilities in the underlying OS) allows us to compare the effectiveness of different software updating mechanisms. These mechanisms are used for deploying various kinds of updates (e.g., for improving performance or for adding functionality); we focus on updates that patch security vulnerabilities. We also aim to investigate the impact of application-specific and user-specific factors on the patching process.

To interpret the results of our analysis, it is helpful to compare our goals with those of the prior studies on vulnerability patching and software update deployment. Several studies [29], [36], [37], [46] conducted remote vulnerability scans, which allowed them to measure vulnerabilities in server-side applications, but not in client-side applications that do not listen on the network. Another approach for measuring the patch deployment speed is to analyze the logs of an update management system [18], which only covers applications utilizing that updating system and excludes hosts where the user or the system administrator has disabled automated updating, a common practice in enterprise networks. Similarly, examining the User-Agent string of visitors to a popular website [11] only applies to web browsers, is confounded by the challenge of enumerating hosts behind NATs, and excludes users not visiting the site. In contrast, we aim to compare multiple client-side applications from different vendors, employing multiple patch deployment mechanisms. Because we analyze data collected on end hosts, we do not need to overcome the problem of identifying unique hosts over the network and our results are not limited to active instances. Instead, we can analyze the patch deployment for applications that are seldom used and that may remain vulnerable under the radar (e.g., when a user installs multiple browsers or media players).

**Applications selected.** We select 10 desktop applications running on Windows operating systems: 4 browsers (Chrome, Firefox, Opera, Safari), 2 multimedia players (Adobe Flash Player, Quicktime), an email client (Thunderbird), a document reader (Adobe Reader), a document editor (Word), and a networking tool (Wireshark). We choose these applications because: (1) they are popular, (2) they are among the top desktop applications with most vulnerabilities in NVD [31], and (3) they cover both proprietary and open source applications. Across all these applications, we analyze the patching process of 1,593 vulnerabilities, disclosed between 2008–2012. All programs except Word can be updated free of charge. All programs replace the old version with the new one after an upgrade except Adobe Reader, for which new product lines are installed in a new directory and the old line is kept in its current directory.

**Non-goals.** We do not aim to analyze the entire vulnerability lifecycle. For example, determining the precise dates when vulnerability exploits are published is outside the scope of this paper. Similarly, we do not aim to determine when a patch takes effect, e.g., after the user has restarted the application. Instead, we focus on patch deployment, i.e., patch download and installation. Finally, we do not aim to determine precisely when the patch deployment is completed, as old versions of applications are often installed along with driver packages or preserved in virtual machine images, and can remain unpatched for very long periods.

## III. Datasets

We analyze six datasets: WINE's binary reputation [12] to identify files installed by real users, the NVD [31] and OSVDB [33] vulnerability databases to determine vulnerable program versions and disclosure dates, the EDB [15] and WINE-AV for exploit release dates, and VirusTotal [44] for additional file meta-data (e.g., AV detections, file certificates). These datasets are summarized in Table I and detailed next.

**WINE–binary reputation.** The Worldwide Intelligence Network Environment (WINE) [13] provides access to data collected by Symantec's anti-virus and intrusion-detection prod-

TABLE I. SUMMARY OF DATASETS USED.

| Dataset | Analysis Period | Hosts | Vul. | Exp. | Files |
|---|---|---|---|---|---|
| WINE-BR | 01/2008–12/2012 | 8.4 M | – | – | 7.1 M |
| VirusTotal | 10/2013-04/2014 | – | – | – | 5.1 M |
| NVD | 10/1988-12/2013 | – | 59 K | – | – |
| OSVDB | 01/1972-01/2012 | – | 77 K | – | – |
| EDB | 01/2014 | – | – | 25 K | – |
| WINE-AV | 12/2009–09/2011 | – | – | 244 | – |

ucts on millions of end-hosts around the world. Symantec's users have a choice of opting-in to report telemetry about security events (e.g., executable file downloads, virus detections) on their hosts. WINE does not include user-identifiable information. These hosts are real computers, in active use around the world.

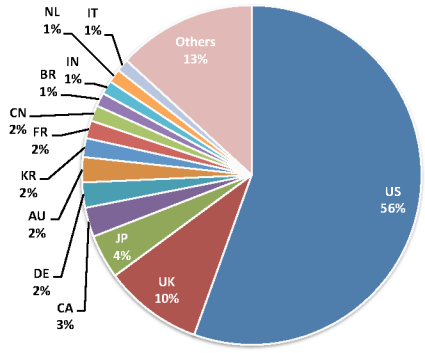



Fig. 2. The geographical distribution of machines that contribute to the binary reputation data in WINE.

We use the binary reputation dataset in WINE (WINE-BR), which is collected from 8.4 million Windows hosts that have installed Symantec's consumer AV products. WINE-BR records meta-data on all—benign or malicious—executable files (e.g., EXE, DLL, MSI). As shown in Figure 2, the WINE hosts are concentrated in North America and Europe with the top-20 countries accounting for 89.83% of all hosts. Since these are real hosts, their number varies over time as users install and uninstall Symantec's products. Figure 3 shows the number of simultaneous reporting hosts over time. It reaches a plateau at 2.5 M hosts during 2011.

The hosts periodically report new executables found. Each report includes a timestamp and for each executable, the hash (MD5 and SHA2), the file path and name, the file version, and, if the file is signed, the certificate's subject and issuer. The binaries themselves are not included in the dataset. Since we analyze the vulnerability lifecycle of popular programs our analysis focuses on the subset of 7.1 million files in WINE-BR reported by more than 50 users between 2008 and 2012.

**NVD.** The National Vulnerability Database [31] focuses on vulnerabilities in commercial software and large open-source projects. Vulnerability disclosures regarding less-well-known software, e.g., small open source projects, are typically redirected to other vulnerability databases (e.g., OSVDB). NVD uses CVE identifiers to uniquely name each vulnerability and publishes XML dumps of the full vulnerability data. We use

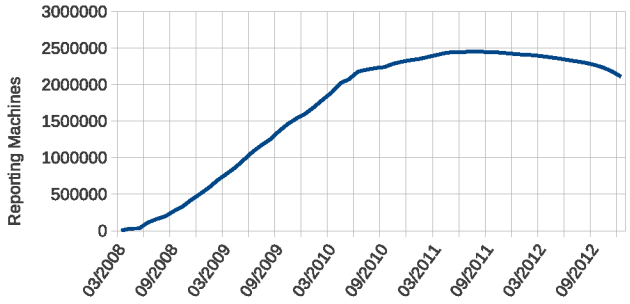



Fig. 3. Reporting hosts in WINE's binary reputation data over time.

the NVD dumps until the end of 2013, which comprise 59,875 vulnerabilities since October 1988[1].

**OSVDB.** The Open Sourced Vulnerability Database[2] [33] has the goal of providing technical information on every public vulnerability, so it contains vulnerabilities in more programs than NVD. OSVDB uses its own vulnerability identifier but also references the CVE identifier for vulnerabilities with one. Up to early 2012, OSVDB made publicly available full dumps of their database. However, OSVDB has since moved away of their original open source model and public dumps are no longer available. Our OSVDB data comes from one of the latest public dumps on January 12, 2012. It contains 77,101 vulnerabilities since January 2004.

**EDB.** The Exploit Database [15] is an online repository of vulnerability exploits. We crawl their web-pages to obtain meta-data on 25,331 verified exploits, e.g., publication date and vulnerability identifier (CVE/OSVDB).

**WINE–AV.** The AV telemetry in WINE contains detections of known cyber threats on end hosts. We can link some of these threats to 244 exploits of known vulnerabilities, covering 1.5 years of our observation period. We use this dataset to determine when the exploits start being detected in the wild.

**VirusTotal.** VirusTotal [44] is an online service that analyzes files and URLs submitted by users with multiple security / anti-virus products. VirusTotal offers a web API to query meta-data on the collected files including the AV detection rate and information extracted statically from the files. We use VirusTotal to obtain additional meta-data on the WINE files, e.g., detailed certificate information and the values of fields in the PE header. This information is not available otherwise as we do not have access to the WINE files, but we can query VirusTotal using the file hash. Overall, VirusTotal contains an impressive 5.1 million (72%) of the popular files in WINE's binary reputation dataset.

**Release histories.** In addition to the 6 datasets in Table I, we also collect from vendor websites the release history for the programs analyzed, e.g., Chrome [19], Safari [38]. We use release histories to differentiate beta and release program versions and to check the completeness of the list of release versions observed in WINE.

[1]CVE identifiers start at CVE-1999-0001, but CVE-1999-* identifiers may correspond to vulnerabilities discovered in earlier years.

[2]Previously called Open Source Vulnerability Database

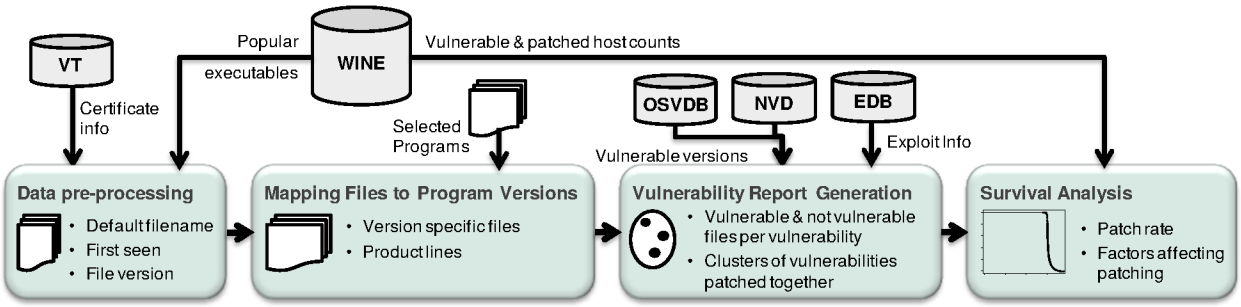



Fig. 4. Approach overview.

## IV. Vulnerability Analysis

Our study of the vulnerability lifecycle comprises two main steps. First, we develop an approach to map files on end-hosts to vulnerable and patched program versions (Sections IV-A through IV-E). Then, we perform a survival analysis to study how users deploy patches for each vulnerability (Section IV-D). Our approach to map files to vulnerable and patched program versions is novel, generic, and reusable. As far as we know there is no such approach proposed in the literature. While it comprises some program-specific data collection, the approach itself is generic, as it has been applied to 10 programs from 7 software vendors. Making it generic was challenging as each vendor has their own policies. Another challenge was dealing with the scattered, often incomplete, and sometimes incorrect data available in public repositories. We are publicly releasing the results of our data processing for the benefit of the community. Our approach has been designed to minimize manual work so that it can be applied to hundreds of vulnerabilities for each program and that adding new programs is also easy. The approach is not specific to WINE; it can be used by any end host application that periodically scans the host's hard-drive producing tuples of the form <machine, timestamp, file, filepath> for the executable files it finds. Most antivirus software and many other security tools fit this description.

Figure 4 summarizes our approach. First, we pre-process our data (Section IV-A). Then, we identify version-specific files that indicate a host has installed a specific program version (Section IV-B). Next, our vulnerability report generation module automatically identifies which files indicate vulnerable and not-vulnerable versions for each vulnerability (Section IV-C). We detail our cleaning of NVD data in Appendix A. Finally, we perform a survival analysis for each vulnerability (Section IV-D).

### A. Data Pre-Processing

The first step in our approach is to pre-process the 7.1 million files in our dataset to have the information needed for the subsequent steps. Note that while we focus on 10 selected applications, we do not know a priori which files belong to those. We first assign to each file (identified by MD5 hash) a default filename, first seen timestamp, and file version. The default filename is selected by majority voting across all WINE hosts reporting files. Since most users do not modify default filenames and the files are reported by at least 50 hosts, the filename used by the largest number of hosts is highly likely the vendor's default. The first seen timestamp is the earliest time that the file was reported by any host. For the file version, each Windows executable contains version information in its PE header, which WINE normalizes as four decimal numbers separated by dots, e.g., 9.0.280.0.

In addition, we query VirusTotal for the files' metadata including the number of AV products flagging the file as malicious and the file certificate. We consider malware the 1.3% of the 7.1 M files flagged by 3 or more AV products as malicious, removing them from further analysis. If the file has a valid certificate, we extract the publisher and product information from its metadata.

### B. Mapping Files to Program Versions

To determine if a WINE host installed a specific program version we check if the host has installed some files specific to that program version. Such *version-specific files* should not be part of other program versions (or other programs) and should not correspond to installer files (e.g., firefox_setup_3.0.7.exe) but rather to the final executables or libraries (e.g., firefox.exe) that are only present on the host's file system if the user not only downloaded the program version, but also installed it. The goal is to generate a list of triples $< h, p, v_p >$ capturing that the presence of a file with hash $h$ in a host indicates the installation of program $p$ and version $v_p$.

An intuitive method to identify version-specific files would be to install a program version, monitoring the files it installs. Files installed by one program version but not by any other version would be version-specific. However, such approach is difficult to generalize because for some programs it is not easy to obtain all the program versions released between 2008–2012, and because automating the installation process of arbitrary programs is challenging given the diversity in installation setups and required user interaction.

We have developed an alternative analyst-guided approach to identify version-specific files. Our approach leverages the fact that large software vendors, e.g., those of our selected programs, sign their executables to provide confidence in their authenticity, and that, to keep software development manageable, they keep filenames constant across program versions (e.g., Firefox's main executable is named firefox.exe in all program versions) and update the file version of files modified in a new program version.

For each program analyzed, we first query our database for how many different file versions each default filename associated with the program has (e.g., files with vendor

TABLE II. SUMMARY OF THE SELECTED PROGRAMS.

| | Configuration | | | WINE | | | NVD vulnerabilities | | | Auto Updates Introd. | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Program** | **Vendor** | **Leading Filename** | **Lines** | **Files** | **Ver.** | **Rel.** | **Total** | **Selected** | **Clust.** | **Ver.** | **Date** |
| Chrome | Google | chrome.dll | 1 | 544 | 531 | 531 | 886 | 545 (61%) | 87 | 1.0 | 2008-12-12 |
| Firefox | Mozilla | firefox.exe | 10 | 457 | 440 | 140 | 1,013 | 226 (22%) | 88 | 15 | 2012-08-26 |
| Flash | Adobe | npswf32%.dll | 12 | 187 | 121 | 121 | 316 | 123 (38%) | 22 | 11.2 | 2012-03-27 |
| Opera | Opera | opera.exe | 1 | 95 | 90 | 90 | 25 | 10 (40%) | 8 | 10 | 2009-08-30 |
| Quicktime | Apple | quicktimeplayer.exe | 1 | 68 | 43 | 41 | 206 | 67 (33%) | 15 | - | 2008-01-01 |
| Reader | Adobe | acrord32.dll | 9 | 102 | 73 | 66 | 330 | 159 (48%) | 30 | 10.1 | 2011-06-14 |
| Safari | Apple | safari.exe | 1 | 35 | 29 | 26 | 460 | 227 (49%) | 39 | - | 2008-01-01 |
| Thunderbird | Mozilla | thunderbird.exe | 17 | 91 | 91 | 83 | 594 | 31 (15%) | 60 | 17 | 2012-11-20 |
| Wireshark | Wireshark | wireshark.exe | 1 | 56 | 38 | 38 | 261 | 110 (42%) | 66 | 1.10.0 | 2013-06-05 |
| Word | Microsoft | winword.exe | 7 | 247 | 104 | 7 | 92 | 36 (39%) | 12 | 2003 | 2005-06-01 |
| | | | TOTAL | 2,006 | 1,610 | 1,242 | 4,347 | 1,593 (39%) | 408 | | |

"Adobe%" and product "%Reader%"[3]). We select the non-installer filename with most file versions as the *leading filename*. In 7 of our 10 programs the leading filename corresponds to the main executable (e.g., firefox.exe, wireshark.exe) and in other 2 to the main library (chrome.dll for Chrome, acrord32.dll for Reader). For Flash Player, Adobe includes the version in the default filename since March 2012, e.g., npswf32_11_2_202_18.dll, so we use a wildcard to define the leading filename, i.e., npswf32%.dll.

**Mapping file version to program version.** The file version of files with the leading filename may correspond directly to the program version (e.g., chrome.dll 5.0.375.70 indicates Chrome 5.0.375.70), to a longer version of the program version (e.g., acrord32.dll 9.1.0.163 indicates Reader 9.1), or the relationship may not be evident, (e.g., firefox.exe 1.9.0.3334 indicates Firefox 3.0.7). For each program we build a *version mapping*, i.e., a list of triples $< v_f, v_p, type >$ indicating that file version $v_f$ corresponds to program version $v_p$. We use the type to indicate the maturity level of the program version, e.g., alpha, beta, release candidate, release. We limit the patch deployment analysis to release versions of a program. The mapping of file to program versions is evident for 7 programs, the exception being Safari and old versions of Firefox and Thunderbird[4]. For Safari we use its public release history [38] and for Firefox/Thunderbird we leverage the fact that the developers store the program version in the product name field in the executable's header, available in the VirusTotal metadata. For the program version type, we leverage the program release history from the vendor's site.

**Checking for missing versions.** In most cases, the file with the leading filename is updated in every single program version. However, for Adobe Reader we found a few program versions that do not modify the file with the leading filename but only other files (e.g., Adobe Reader 9.4.4 did not modify acrord32.dll or acrord32.exe). To identify if we are missing some program version we compare the version mapping with the vendor's release history. For any missing version, we query our database for all files with the missed file version and signed by the program's vendor (regardless of the filename). This enables identifying other files updated in the missing version, e.g., nppdf32.dll in Reader 9.4.4. We add one of these file hashes to our list of version-specific files. Of course, if the vendor did not update the version of any executable file in a new program version, we cannot identify that program version.

[3]We use % as a wildcard as in SQL queries

[4]Firefox and Thunderbird use file versions similar to program versions since Firefox 5 and Thunderbird 5.

**Product lines.** Some programs have product lines that are developed and released with some overlap in time. For our analysis it is important to consider product lines because a vulnerability may affect multiple lines and each of them needs to be patched separately. For example, vulnerability CVE-2009-3953 is patched in versions 8.2 and 9.3 of Adobe Reader, which belong to lines 8 and 9 respectively. To map product versions to product lines we use regular expressions. Next section describes our handling of product lines.

As the output of this process the analyst produces a program configuration file that captures the list of version-specific files, the version mapping, and the product line regular expressions. This configuration is produced once per program, independently of the number of program vulnerabilities that will be analyzed.

The left-side section of Table II summarizes the configuration information for the 10 selected programs: the program name, the vendor, the leading filename, and the number of product lines. The right-side section shows on what date, and in which version, the application started using an automated updating mechanism. The WINE section captures the number of version-specific files (Files), the number of file versions for those files (Ver.), and the number of program versions they correspond to (Rel.) The 3 numbers monotonically decrease since two files may occasionally have the same file version, e.g., 32-bit and 64-bit releases or different language packs for the same version. Two file versions may map to the same program version, e.g., beta and release versions.

Note that we start with 7.1 M files and end up with only 2,006 version-specific files for the 10 programs being analyzed. This process would be largely simplified if program vendors made publicly available the list of file hashes corresponding to each program version.

### C. Generating Vulnerability Reports

Next, our vulnerability report generation module takes as input the configuration file for a program and automatically determines for each vulnerability for the program in NVD, which versions are vulnerable and not vulnerable.

For this, it first queries our database for the list of NVD vulnerabilities affecting the program. If a vulnerability affects multiple programs (e.g., Firefox and Thunderbird), the vulnerability will be processed for each program. For each vulnerability, it queries for the list of vulnerable program versions in NVD and splits them by product line using the regular expressions in the configuration. For each vulnerable product line (i.e., with at least one vulnerable version), it

determines the range of vulnerable program versions $[x_l, y_l]$ in the line. Note that we have not seen lines with multiple disjoint vulnerable ranges. Finally, it annotates each version-specific file as not vulnerable (NV), vulnerable (V), or patched (P). For a file specific to program version $v_p$, it compares $v_p$ with the range of vulnerable versions for its line. If the line is not vulnerable, the file is marked as not vulnerable. If the line is vulnerable but $v_p < x_l$, it is marked as not vulnerable; if $x_l \leq v_p \leq y_l$ as vulnerable; and if $v_p > y_l$ as patched.

We discard vulnerabilities with no vulnerable versions, no patched versions, or with errors in the NVD vulnerable version list. We may not find any vulnerable versions for vulnerabilities in old program versions (that no WINE host installs between 2008–2012). Similarly, we may find no patched versions for vulnerabilities disclosed late in 2012 that are patched in 2013. We detail how we identify NVD errors in Appendix A.

Multiple vulnerabilities may affect exactly the same program versions. Such vulnerabilities have the same vulnerable population and identical patching processes. We therefore group them together into *vulnerability clusters* to simplify the analysis. All vulnerabilities in a cluster start patching on the same date, but each vulnerability may have a different disclosure date.

The "NVD vulnerabilities" section of Table II shows the total number of vulnerabilities analyzed for each program (Total), the selected vulnerabilities after discarding those with no vulnerable or patched versions, or with errors (Selected), and the number of vulnerability clusters (Clust.). Overall, we could analyze 39% of all vulnerabilities in the 10 programs.

**Patching delay.** For each cluster, we compute the disclosure date $t_0$ as the minimum of the disclosure dates in NVD and OSVDB. The start of patching $t_p$ is the first date when we observe a patched version in the field data from WINE. The start of patching date often differs from the disclosure date and is not recorded in NVD or OSVDB. The patching delay for a cluster is simply $pd = t_p - t_0$.

### D. Survival Analysis

To analyze the patch deployment speed we employ *survival analysis techniques*, which are widely used in medicine and biology to understand the mortality rates associated with diseases and epidemics [24]. In our case, survival analysis measures the probability that a vulnerable host will "survive" (i.e., remain vulnerable) beyond a specified time. Intuitively, the population of vulnerable hosts decreases (i.e., the vulnerability "dies" on a host) when one of two death events happen: (1) a user installs a *patched* program version or (2) a user installs a *non-vulnerable* program version. While both events decrease the vulnerable population, only the first one is directly related to patch deployment. When we analyze patch deployment we consider only the first death event; when analyzing vulnerability decay we consider both.

**Survival function.** To understand how long a vulnerability remains exploitable in the wild, we consider that the vulnerability lifetime is a random variable $T$. The *survival function* $S(t)$ captures the likelihood that the vulnerability has remained unpatched until time $t$:

$$S(t) = \Pr[T > t] = 1 - F(t)$$

where $F(t)$ is the cumulative distribution function.

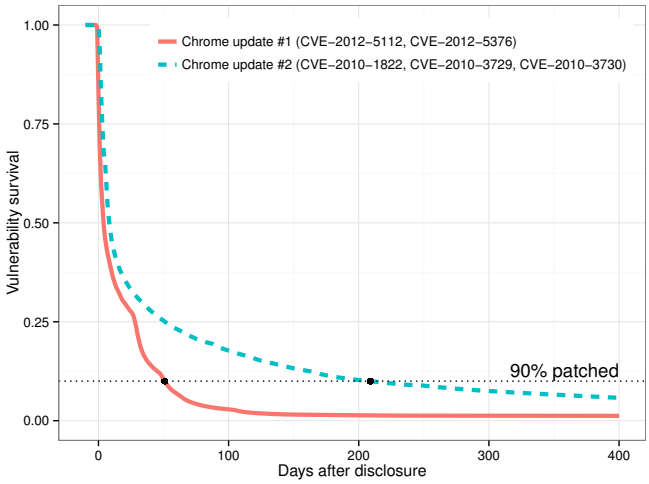



Fig. 5. Examples of vulnerability survival, illustrating the deployment of two updates for Google Chrome.

We estimate $S(t)$ for each vulnerability cluster. Figure 5 illustrates the output of this analysis through an example. Using our $S(t)$ estimations, we can compare the deployment of two software updates in Chrome. Both updates reached 50% of the vulnerable hosts in a few days, but update #1 reached 90% of vulnerable hosts in 52 days, while update #2 needed $4\times$ as much time to reach the same milestone.

The survival function describes how the probability of finding vulnerable hosts on the Internet decreases over time. $S(t)$ is 1 in the beginning, when no vulnerable hosts have yet been patched, and 0 when there are no vulnerable hosts left. By definition, $S(t)$ is monotonically decreasing; it decreases when one of the two death events mentioned earlier occurs on a vulnerable host. In our analysis of the patch deployment process, the installation of a patched version closes the vulnerability on the host. In this case, the point in time when $S(t)$ starts decreasing corresponds to the *start of patching* $t_p$, which we use to estimate the patching delay as described in Section IV-C. In our analysis of the vulnerability decay process, the installation of either a patched version or a non-vulnerable version closes the vulnerability window. The point where $S(t)$ starts decreasing corresponds to the *start of the vulnerability decay* ($t_d$ in Figure 1). In both cases, the host's death event for a given vulnerability corresponds to the first installation of a patched or non-vulnerable version *after* the installation of a vulnerable version on the host.

The time needed to patch a fraction $\alpha$ of vulnerable hosts corresponds to the inverse of the survival function: $t_\alpha = S^{-1}(1-\alpha)$. The survival function allows us to quantify the milestones of patch deployment such as the median time to patch $t_m = S^{-1}(0.5)$ and the time to patch most vulnerable hosts ($t_{90\%} = S^{-1}(0.1)$, $t_{95\%} = S^{-1}(0.05)$).

**Right censoring and left truncation.** When estimating $S(t)$ we must account for the fact that, for some vulnerable hosts, we are unable to observe the patching event before the end of our observation period. For example, some hosts may leave our study before patching, e.g., by uninstalling the Symantec product, by opting out of data collection, or by upgrading the OS. In statistical terms, these hosts are independently *right-*

*censored*. We determine that a host becomes right censored by observing the last download report (not necessarily for our selected applications) in WINE. For these hosts, we do not know the exact time (or whether) they will be patched; we can only ascertain that they had not been patched after a certain time. In other words, when a host becomes censored at time $t$, this does not change the value of $S(t)$; however, the host will no longer be included in the vulnerable host population after time $t$. In addition, we cannot determine whether vulnerable versions have been installed on a host before the start of our observation period, so some vulnerable hosts may not be included in the vulnerable population. In statistical terms, our host sample is *left-truncated*. Right-censoring and left-truncation are well studied in statistics, and in this paper we compute the values of $S(t)$ using the Kaplan-Meier estimator, which accounts for truncated and censored data. We compute this estimator using the `survival` package for R [43]. We consider that we have reached the end of the observation period for a vulnerability when the vulnerable population falls below 3 hosts or when we observe that 95% or more of the vulnerable hosts become censored within 5% of the total elapsed time. This prevents artificial spikes in the hazard function at the end of our observation period, produced by a large decrease in the vulnerable host population due to right-censoring.

### E. Threats to Validity

**Selection bias.** The WINE user population is skewed towards certain geographical locations; for example, 56% of the hosts where the data is collected are located in the United States (Figure 2). Additionally, WINE's binary reputation only covers users who install anti-virus software. While we cannot exclude the possibility of selection bias, the prevalence of anti-virus products across different classes of users and the large population analyzed in our study (1,593 vulnerabilities on 8.4 million hosts) suggests that our results have a broad applicability. Similarly, our study considers only Windows applications, but the popular client applications we analyze are cross-platform, and often use the same patching mechanism on different platforms. Moreover, cyber attacks have predominantly targeted the Windows platform.

**Sampling bias.** The WINE designers have taken steps to ensure WINE data is a representative sample of data collected by Symantec [34]. For our study, this means that the vulnerability survival percentages we compute are likely accurate for Symantec's user base, but the absolute sizes of the vulnerable host population are underestimated by at least one order of magnitude.

**Data bias.** The WINE binary reputation does not allow us to identify program uninstalls. We can only identify when a user installs new software and whether the newly installed version overwrote the previous one (i.e., installed on the same path). The presence of uninstalls would cause us to underestimate the rate of vulnerability decay, as the hosts that have completely removed the vulnerable application would be counted as being still vulnerable at the end of our observation period. We do not believe this is a significant factor, as we observe several vulnerabilities that appear to have been patched completely during our observation (for example, update #1 from Figure 5).

TABLE III. MILESTONES FOR PATCH DEPLOYMENT FOR EACH PROGRAM (MEDIANS REPORTED).

| Program | %Vers. Auto | Vul. Pop. | Patch Delay | Days to patch (%clust.) $t_m$ | $t_{90\%}$ |
|---|---|---|---|---|---|
| Chrome | 100.0% | 521 K | -1 | 15 (100%) | 246 (93%) |
| Firefox | 2.7% | 199 K | -5.5 | 36 (91%) | 179 (39%) |
| Flash | 14.9% | 1.0 M | 0 | 247 (59%) | 689 (5%) |
| Opera | 33.3% | 2 K | 0.5 | 228 (100%) | N/A (0%) |
| Quicktime | 0.0% | 315 K | 1 | 268 (93%) | 997 (7%) |
| Reader | 12.3% | 1.1 M | 0 | 188 (90%) | 219 (13%) |
| Safari | 0.0% | 146 K | 1 | 123 (100%) | 651 (23%) |
| Thunderbird | 3.2% | 11 K | 2 | 27 (94%) | 129 (35%) |
| Wireshark | 0.0% | 1 K | 4 | N/A (0%) | N/A (0%) |
| Word | 37.4% | 1.0 M | 0 | 79 (100%) | 799 (50%) |

## V. EVALUATION

We analyze the patching of 1,593 vulnerabilities in 10 applications, which are installed and are actively used on 8.4 M hosts worldwide. We group these vulnerabilities into 408 clusters of vulnerabilities patched together, for an average of 3.9 vulnerabilities per cluster. We conduct survival analysis for all these clusters, over observation periods up to 5 years.

In this section, we first summarize our findings about the update deployment process in each application (Section V-A). Then, we analyze the impact of maintaining parallel product lines on the patching delay (Section V-B), the race between exploit creators and patch deployment (Section V-C), the opportunities for patch-based exploit generation (Section V-D), and the impact of parallel installations of an application on patch deployment (Section V-E). Next, we analyze the time needed to reach several patch deployment milestones (Section V-F) and, finally, the impact of user profiles and of automated update mechanisms on the deployment process (Section V-G).

### A. Patching in Different Applications

The prior theoretical work on optimal patch-management strategies makes a number of assumptions, e.g., that there is an important trade-off between the patching delay and the amount of testing needed before releasing patches, or that patch deployment is instantaneous [32]. In this section, we put such assumptions to the test. We focus on the patching delay and on two patch deployment milestones: reaching 50% and 90% of the vulnerable hosts. For this analysis, we consider only the deployment of a patched version as a vulnerability death event, and we aggregate the results of the survival analysis for each of the 10 selected applications. Because it is difficult to compare $t_{90\%}$ for two applications with different vulnerable populations, we report both the time to reach this milestone and the percentage of updates that reach it. Applications with effective updating mechanisms will be able to reach 90% deployment for a large percentage of updates. The time needed to reach this milestone illustrates how challenging it is to patch the last remaining hosts within a large vulnerable population.

Table III summarizes the patch deployment milestones for each application. The second column shows the percentage of versions that were updated automatically. Chrome had an automated updating mechanism since its first version; Word used Microsoft Update throughout the study; Wireshark had completely manual updates throughout our study; Safari and

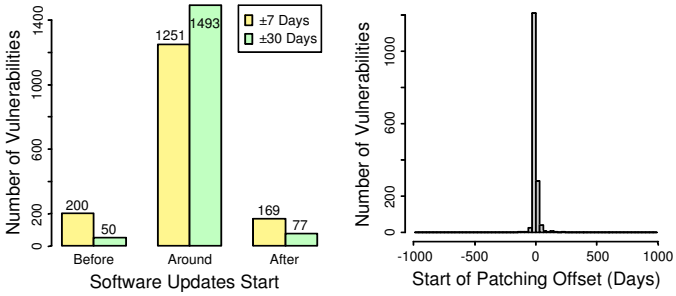
Quicktime use the Apple Software Updater that periodically checks and prompts the user with new versions to install; the remaining programs introduced silent updates during our study.

The next column shows the vulnerable host population (we report the median across all vulnerability clusters for the program). For 7 out of the 10 applications the median vulnerable population exceeds 100,000 hosts and for 3 (Flash, Reader, Word) it exceeds one million hosts. In comparison, the 2014 Heartbleed vulnerability in OpenSSL affected 1.4 million servers [22], and the Internet worms from 2001–2004 infected 12K–359K hosts [28], [29], [40]. As explained in Section IV-E the host population in our study only reflects the hosts in WINE, after sampling. The vulnerable host population in the unsampled Symantec data is likely to be at least one order of magnitude higher [34], and the real vulnerable population when including hosts without Symantec software much larger. These numbers highlight that vulnerabilities in client-side applications can have significant larger vulnerable populations than server-side vulnerabilities.

The fourth column reports the median patch delay (in days) for the vulnerability clusters. Chrome and Firefox have negative values indicating that patching starts before disclosure for most of their clusters. Flash, Reader, and Word have zero values indicating that Adobe and Microsoft are coordinating the release of patches and advisories, e.g., at Patch Tuesday for Microsoft. The remaining programs have positive patch delay; the larger the patch delay the longer users are exposed to published vulnerabilities with no patch available.

The last two columns capture the times needed to reach 50% and 90% patch deployment (we report medians across clusters), as well as the percentage of clusters that reached these milestones before the end of our observation period. Chrome has the shortest $t_m$, followed by Thunderbird and Firefox. At the other extreme, Wireshark exhibits the slowest patching: no cluster reaches 50% patch deployment by the end of 2012—not even for vulnerabilities released in 2008. Excluding Wireshark and Flash, all applications reach 50% patching for over 90% of vulnerability clusters, with Chrome, Opera, Safari, and Word reaching this milestone for 100% of clusters. At $t_{90}$ only Chrome patches more than 90% of the vulnerability clusters. We also observe that Firefox and Thunderbird appear to reach this milestone faster than Chrome, but this comparison is biased by the fact that for these two applications only 35–40% of clusters reach this milestone, in contrast with Chrome's 93%.

Chrome's automated update mechanism results in the fastest patching among the 10 applications. However, it still takes for Chrome 246 days (approximately 8.2 months) to achieve 90% patching. This finding contradicts prior research that concluded that 97% of active Chrome instances are updated to the new version within 21 days after the version's release of a new version [11]. The main difference in our approach is that, because we analyze end-host data, our measurements include inactive applications, which are installed on a host but rarely used. Moreover, our measurements do not suffer from confounding factors such as applications on hosts behind NATs, which complicate the interpretation of version numbers extracted from network traces.




(a) Correlation with disclosure. (b) Histogram.

Fig. 6. Distribution of the patching delay. For each vulnerability, we compare the start of patching with the disclosure date. The left plot shows how many vulnerabilities start patching within a week (±7 days) or a month (±30 days) of disclosure and how many start patching outside these intervals. The right plot shows a histogram of the patch delay (each bar corresponds to a 30-day interval).

### B. Patching Delay

In this section we ask the questions: *How quickly are patches released, following vulnerability disclosures?* and *What is the impact of maintaining multiple product lines on these releases?* The patch can be released *on* the disclosure date (e.g., coordinated disclosure, where the vendor is notified and given some time to create a patch before publicizing the vulnerability), *after* the disclosure date, (e.g., discovery of a zero-day attack in the wild or full disclosure, where the vulnerability is publicized despite the lack of a patch in order to incentivize the vendor to create the patch), or *before* disclosure (e.g., the vulnerability does not affect the latest program version). Thus, the patching delay may be positive, zero, or negative. Figure 6 illustrates the distribution of the patching delay for all the 1,593 vulnerabilities in our study. We find that the start of patching is strongly correlated with the disclosure date (correlation coefficient $r = 0.994$). Moreover, Figure 6 shows that 77% of the vulnerabilities in our study start patching within 7 days before or after the disclosure dates. If we extend this window to 30 days, 92% of vulnerabilities start patching around disclosure. This suggests that software vendors generally respond promptly to vulnerability disclosures and release patches that users can deploy to avoid falling victim to cyber attacks.

It is interesting to observe that maintaining multiple parallel lines does not seem to increase the patching delay. Firefox, Flash, Reader, Thunderbird and Word all have multiple product lines; in fact, each vulnerability cluster in these products affects all the available product lines. When a vulnerability affects multiple product lines, patches must be tested in each program line, which may delay the release of patches. However, Table III suggests that the median patching delay is not significantly higher for these applications than for the applications with a single product line, like Google Chrome. We note that this doesn't suggest that multiple product lines do not impact software security, as there may be additional software engineering issues associated with product lines; however, we do not find empirical evidence for these effects.

When we take into consideration both types of vulnerability death events in the survival analysis, (i.e., the installation of non-vulnerable versions contributes to the vulnerability

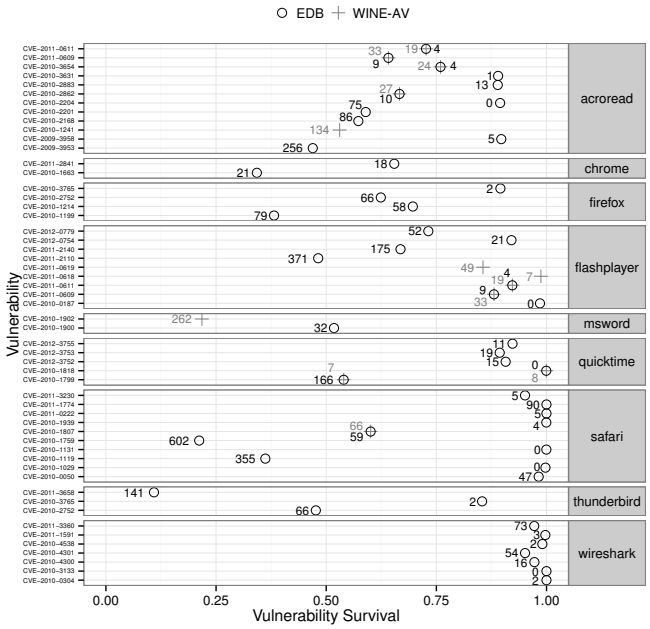



Fig. 7. Vulnerability survival (1 = the vulnerability decay has not yet started) when exploits are released. The data points are annotated with estimations of the exploitation lag: the number of days after disclosure when the the exploits were publicized (for EDB) and when the first detections occurred (for WINE-AV).

decay), we observe that the vulnerable host population starts decaying almost as soon as a vulnerable version is released, as some users revert to older versions. In this case, 94% of the clusters start decaying earlier than 30 days before the disclosure date. While this is a mechanism that could prevent zero-day exploits in the absence of a patch, users are not aware of the vulnerability before its public disclosure, and the vulnerability decay due to non-vulnerable versions is small.

### C. Patches and Exploits

We now ask the question: *When attackers create exploits for these vulnerabilities, what percentage of the host population can they expect to find still vulnerable?* This has important security implications because exploit creators are in a race with the patch deployment: once the vulnerability is disclosed publicly, users and administrators will start taking steps to protect their systems.

We use the WINE-AV dataset to identify attacks for specific vulnerabilities. This dataset only covers 1.5 years of our observation period (see Table I). After discarding vulnerabilities disclosed before the start of the WINE-AV dataset and intersecting the 244 exploits in WINE-AV with our vulnerability list, we are left with 13 exploits, each for a different vulnerability. We also add 50 vulnerabilities that have an exploit release date in EDB. For each of the 54 vulnerabilities in the union of these two sets, we extract the earliest record of the exploit in each of the two databases, and we determine the value of the survival function $S(t)$ on that date. In this analysis, we consider both types of death events, because installing non-vulnerable program versions also removes hosts from the population susceptible to these exploits.

Figure 7 illustrates the survival levels for these vulnerabilities; on the right-side 100% (1.00) of hosts remain vulnerable, and on the left-side there are no vulnerable hosts left. Each vulnerability is annotated with the number of days after disclosure when we observe the first record of the exploit. This exploitation lag is overestimated (it is $\geq 0$ even for zero-day exploits) because publicizing an exploit against an unknown vulnerability amounts to a disclosure (for EDB) and because AV signatures that can detect the exploit are often deployed after disclosure (for WINE-AV). For example, CVE-2011-0611, CVE-2011-0609, CVE-2010-3654, CVE-2010-2862, CVE-2010-1241, and CVE-2011-0618 are known to have been exploited in zero-day attacks [6], [41], [42]. In consequence, the vulnerability survival levels in Figure 7 must be interpreted as *lower bounds*, as the exploits were likely released prior to the first exploit records in WINE-AV and EDB.

Additionally, these results illustrate the opportunity for exploitation, rather than a measurement of the successful attacks. Even if vulnerabilities remain unpatched, end-hosts may employ other defenses against exploitation, such as anti-virus and intrusion-detection products or mechanisms such as data execution prevention (DEP), address space layout randomization (ASLR), or sandboxing. Our goal is therefore to assess the effectiveness of vulnerability patching, by itself, in defending hosts from exploits.

All but one of the real-world exploits in WINE found more than 50% of hosts still vulnerable. Considering both databases, at the time of the earliest exploit record between 11% and 100% of the vulnerable population remained exploitable, and the median survival rate was 86%. As this survival rate represents a lower bound, *the median fraction of hosts patched when exploits are released is at most 14%*.

### D. Opportunities for Patch-Based Exploit Generation

Attackers have an additional trump card in the race with the patch deployment: once a patch is released, it may be used to derive working exploits automatically, by identifying the sanitization checks added in the patched version and generating inputs that fail the check [8]. While prior work has emphasized the window of opportunity for patch-based exploit generation provided by slow patch deployment, it is interesting to observe that in some cases a vulnerability may affect more than one application. For example, Adobe Flash vulnerability CVE-2011-0611 affected both the Flash Player and Acrobat Reader (which includes a library allowing it to play Flash objects embedded in PDF documents). For Reader, the patching started 6 days later than for Flash, giving attackers nearly one week to create an exploit based on the Flash patch.

80 vulnerabilities in our dataset affect common code shared by two applications. The time between patch releases ranges from 0 (when both patches are released on the same date, which occurs for 7 vulnerabilities) to 118 days, with a median of 11 days. 3 of Flash vulnerabilities also affect Adobe Reader (as in the case of CVE-2011-0611 described above), and the patches for Flash were released before or on the same day as the Reader patches. 7 vulnerabilities affect the Chrome and Safari browsers, which are based on the WebKit rendering engine; in one case, the Safari patch was released first, and in the other cases the Chrome patch was released first. 1

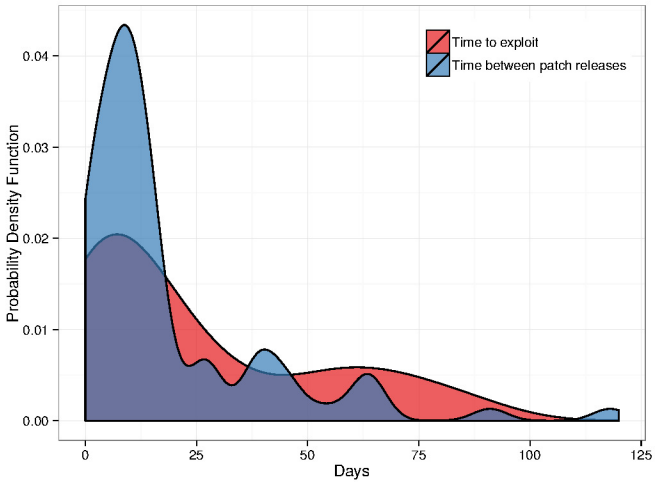



Fig. 8. Distributions of the time between patch releases for vulnerabilities affecting multiple applications and the time needed to exploit vulnerabilities.

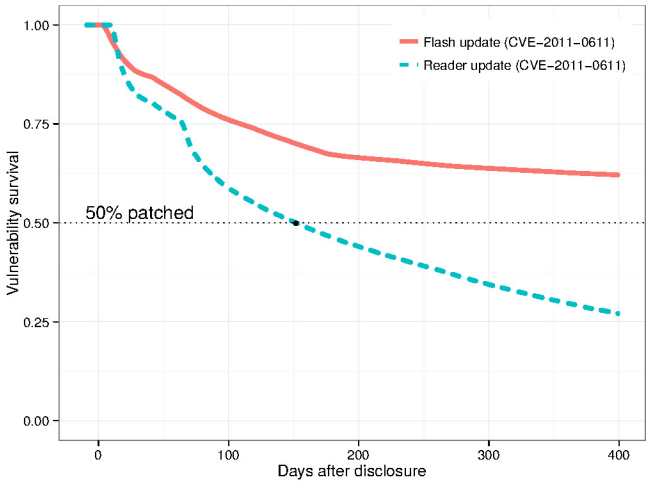



Fig. 9. Patching of one vulnerability from the Flash library, in the stand-alone installation and in Adobe Reader.

vulnerability affects Chrome and Firefox, which use the Angle[5] library for hardware-accelerated graphics, and in this case the Chrome patch was released first. Finally, 69 vulnerabilities affect Firefox and Thunderbird, which share multiple Mozilla libraries, and in all these cases the Firefox patches were released before or on the same day as the Thunderbird patches.

These delays in releasing patches provide ample opportunity for attackers employing patch-based exploit generation techniques, as the run time of these techniques is typically less than an hour [4], [8]. In practice, however, the attacker may experience additional delays in acquiring the patch, testing the exploit and delivering it to the intended target. We therefore compare the time between patch releases with our empirical observations of the exploitation lag, illustrated in Figure 7. While the median exploitation lag is slightly longer (19 days) the two distributions largely overlap, as shown in Figure 8. In consequence, the time between patch releases for applications sharing code is comparable to the typical time that attackers need to create exploits in practice. This is a serious threat because an exploit derived from the first patch to be released is essentially a *zero-day exploit for the other applications*, as long as patches for these applications remain unavailable.

### E. Impact of Multiple Installations on Patch Deployment

While in Section V-C we investigated the threat presented by the time elapsed between patch releases for applications with common vulnerabilities, we now ask the question *how do multiple versions of an application, installed in parallel, impact the patch deployment process?* Failing to patch all the installed versions leaves the host exposed to the attack described in Section II-A. This is a common situation: for example, our analysis suggests that 50% of WINE users who have installed the Flash plugin have Adobe Air installed as well.

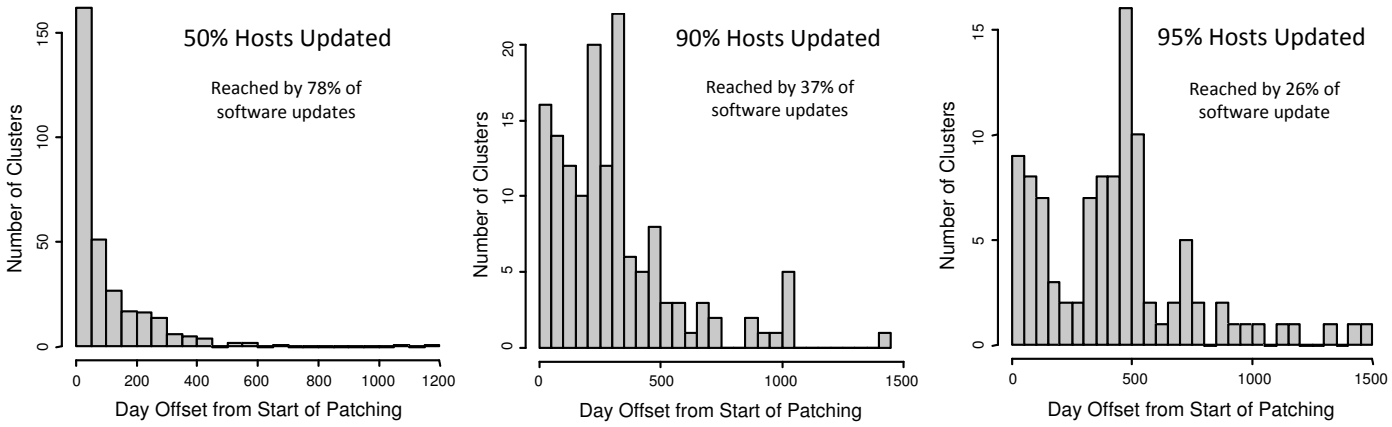
Figure 9 compares the patching of CVE-2011-0611 in Flash and Adobe Reader. This vulnerability is a known zero-day attack, discovered on 12 April 2011. For Flash, the patching started 3 days after disclosure. The patching rate was high, initially, followed by a drop and then by a second wave of patching activity (suggested by the inflection in the curve at $t = 43$ days). The second wave started on 25 May 2011, when the vulnerability survival was at 86%. According to analyst reports, a surge of attacks exploiting the vulnerability started on 15 May 2011 [27]. The second wave of patching eventually subsided, and this vulnerability did not reach 50% completion before the end of our observation period. Perhaps because of this reason, CVE-2011-0611 was used in 30% of spear phishing attacks in 2011 [1]. In contrast, for Reader the patching started later, 9 days after disclosure, after CVE-2011-0611 was bundled with another vulnerability in a patch). Nevertheless, the patch deployment occurred faster and reached 50% completion after 152 days. This highlights the fact that, in general, Adobe Reader patched faster than Flash Player, as Table III indicates.

This highlights the magnitude of the security threat presented by keeping multiple versions installed on a host, without patching all of them. Even if some of these installations are used infrequently, the attacker may still be able to invoke them, as demonstrated in Section II-A. Moreover, a user patching the frequently used installation in response to news of an exploit active in the wild may unknowingly remain vulnerable to attacks against the other versions that remain installed.

### F. Patching Milestones

In this section we ask the question: *How quickly can we deploy patches on all the vulnerable hosts?* Figure 10 illustrates the distribution for the median time-to-patch ($t_m$), and the time needed to patch 90% and 95% of the vulnerable hosts ($t_{90\%}$ and $t_{95\%}$, respectively), across all vulnerability clusters for the 10 applications. We find that the rate of updating is high in the beginning: 5% of clusters reach $t_m$ within 7 days, 29% of clusters reach it within 30 days and 54% of clusters reach it within 100 days. Figure 10a suggests that the median time-to-patch distribution is decreasing, with a long tail, as most vulnerability clusters reach this milestone within a few weeks, but a few clusters need as much as three years to reach it.

[5]https://code.google.com/p/angleproject/




(a) Time to patch 50% of hosts. (b) Time to patch 90% of hosts. (c) Time to patch 95% of hosts.

Fig. 10. Distribution of the time needed to reach three patch deployment milestones. Each bar corresponds to a 50-day window.

In contrast, the distribution of the time needed to reach high levels of update completion seems to have two modes, which are not visible in the distribution of $t_m$. The bimodal pattern starts taking shape for $t_{90\%}$ (Figure 10b) and is clear for $t_{95\%}$ (Figure 10c). We observe that the second mode starts after 200 days at 90% update completion and after 300 days at 95% update completion. These 100 additional days needed to update 5% more hosts suggest that the update deployment slows down over time for a large class of vulnerabilities. Moreover, the first mode stretches 150 days after the start of patching, suggesting that, even for the vulnerabilities that exhibit the highest updating rate, the median time to patch may be up to several months.

### G. Human Factors Affecting the Update Deployment

Finally, we analyze whether specific user profiles and automated patching mechanisms have a positive effect on how fast the vulnerable applications are patched. To this end, we first define three user categories that are presumably more security-aware than common users: professionals, software developers, and security analysts. We classify WINE hosts into these 3 categories by first assigning a set of applications to each category and then checking which hosts have installed some of these applications. A host can belong to multiple categories. For professionals we check for the existence of applications signed, among others, by SAP, EMC, Sage Software, and Citrix. For software developers we check for software development applications (Visual Studio, Eclipse, NetBeans, JDK, Python) and version control systems (SVN, Mercurial, Git). For security analysts, we check for reverse engineering (IdaPro), network analysis (Wireshark), and forensics tools (Encase, Volatility, NetworkMiner). Using these simple heuristics, we identify 112,641 professionals, 32,978 software developers, and 369 security analysts from our dataset.

We perform survival analysis for each user category separately to obtain the median time-to-patch. Table IV presents the results for Adobe Reader, Flash, Firefox, and the mean for the 10 applications. We focus on these three applications because they are popular and because the update mechanisms used in most of the versions in our study were manual and, therefore, required user interaction. In addition, we are interested in checking if automated update mechanisms improve the success of the patching process, and these three applications started using automated updates in 2012. As shown in Table IV, all 3 user categories reach 50% patching faster than the common category encompassing all users. This indicates that these categories react to patch releases more responsibly than the average user. Among our three categories, the security analysts patch fastest, with a patching rate almost three times higher than the general-user category.

TABLE IV. NUMBER OF DAYS NEEDED TO PATCH 50% OF VULNERABLE HOSTS, FOR DIFFERENT USER PROFILES AND UPDATE MECHANISMS.

| Categories | Median time-to-patch (% reached) | | | |
|---|---|---|---|---|
| | All | Reader | Flash | Firefox |
| Professionals | 30 (79%) | 103 (90%) | 201 (73%) | 25 (92%) |
| Software Developers | 24 (80%) | 68 (90%) | 114 (86%) | 23 (90%) |
| Security Analyst | 18 (93%) | 27 (87%) | 51 (91%) | 13 (89%) |
| All users | 45 (78%) | 188 (90%) | 247 (60%) | 36 (91%) |
| Silent Updates | 27 (78%) | 62 (90%) | 107 (86%) | 20 (89%) |
| Manual Updates | 41 (78%) | 97 (90%) | 158 (81%) | 26 (88%) |

The bottom of Table IV we compare manual and automated updating shows the survival analysis results when splitting the program versions into those with manual and automated patching, based on when silent updates were introduced by the program. As expected, automated update mechanisms significantly increase patching deployment, improving security.

## VI. RELATED WORK

Several researchers [3], [16], [39] have proposed vulnerability lifecycle models, without exploring the patch deployment phase in as much detail as we do. Prior work on manual patch deployment has showed that user-initiated patches [29], [36], [37], [46] occur in bursts, leaving many hosts vulnerable after the fixing activity subsides. After the outbreak of the Code Red worm, Moore et. at [29] probed random daily samples of the host population originally infected and found a slow patching rate for the IIS vulnerability that allowed the worm to propagate, with a wave of intense patching activity two weeks later when Code Red began to spread again. Rescorla [37] studied a 2002 OpenSSL vulnerability and observed two waves

of patching: one in response to the vulnerability disclosure and one after the release of the Slapper worm that exploited the vulnerability. Each fixing wave was relatively fast, with most patching activity occurring within two weeks and almost none after one month.

Rescorla [37] modeled vulnerability patching as an exponential decay process with decay rate 0.11, which corresponds to a half-life of 6.3 days. Ramos [36] analyzed data collected by Qualys through 30 million IP scans and also reported a general pattern of exponential fixing for remotely-exploitable vulnerabilities, with a half-life of 20-30 days. However, patches released on an irregular schedule had a slower patching rate, and some do not show a decline at all. While $t_m$ for the applications employing silent update mechanisms and for two other applications (Firefox and Thunderbird) is approximately in the same range with these results, for the rest of the applications in our study $t_m$ exceeds 3 months.

Yilek et al. [46] collected daily scans of over 50,000 SSL/TLS Web servers, in order to analyze the reaction to a 2008 key generation vulnerability in the Debian Linux version of OpenSSL. The fixing pattern for this vulnerability had a long and flat curve, driven by the baseline rate of certificate expiration, with an accelerated patch rate in the first 30 days and with significant levels of fixing (linked to activity by certification authorities, IPSes and large Web sites) as far out as six months. 30% of hosts remained vulnerable six months after the disclosure of the vulnerability. Durumeric et al. [14] compared these results with measurement of the recent Heartbleed vulnerability in OpenSSL and showed that in this case the patching occurred faster, but that, nevertheless, more than 50% of the affected servers remained vulnerable after three months.

While the references discussed above considered manual patching mechanisms, the rate of updating is considerably higher for systems that employ automated updates. Gkantsidis et al. [18] analyzed the queries received from 300 million users of Windows Update and concluded that 90% of users are fully updated with all the previous patches (in contrast to fewer than 5%, before automated updates were turned on by default), and that, after a patch is released, 80% of users receive it within 24 hours. Dübendorfer et al. [11] analyzed the User-Agent strings recorded in HTTP requests made to Google's distributed Web servers, and reported that, within 21 days after the release of a new version of the Chrome Web browser, 97% of active browser instances are updated to the new version (in contrast to 85% for Firefox, 53% for Safari and 24% for Opera). This can be explained by the fact that Chrome employs a *silent update* mechanism, which patches vulnerabilities automatically, without user interaction, and which cannot be disabled by the user. These results cover only instances of the application that were active at the time of the analysis. In contrast, we study multiple applications, including 500 different versions of Chrome, and we analyze data collected over a period of 5 years from 8.4 million hosts, covering applications that are installed but seldom used. Our findings are significantly different; for example, 447 days are needed to patch 95% of Chrome's vulnerable host population.

Despite these improvements in software updating, many vulnerabilities remain unpatched for long periods of time. Frei et al. [17] showed that 50% of Windows users were exposed to 297 vulnerabilities in a year and that a typical Windows user must manage 14 update mechanisms (one for the operating system and 13 for the other software installed) to keep the host fully patched. Bilge et al. [6] analyzed the data in WINE to identify zero-day attacks that exploited vulnerabilities disclosed between 2008–2011, and observed that 58% of the anti-virus signatures detecting these exploits were still active in 2012.

## VII. Discussion

Recent empirical measurements suggest that only 15% of the known vulnerabilities are exploited in real-world attacks and that this ratio is decreasing [30]. Our findings in this paper provide insight into the continued effectiveness of vulnerability exploits reported in prior work [20]. For example, the median fraction of vulnerable hosts patched when exploits are released is at most 14%, which suggests that vulnerability exploits work quite well when they are released—even if they are not zero-day exploits. Additionally, because vulnerabilities have a non-linear lifecycle, where the vulnerable code may exist in multiple instances on a host or may be re-introduced by the installation of a different application, releasing and deploying the vulnerability patch does not always provide immunity to exploits. In the remainder of this section, we suggest several improvements to software updating mechanisms, to risk assessment frameworks, and to vulnerability databases, in order to address these problems.

### A. Improving Software Updating Mechanisms

**Handling inactive applications.** Inactive applications represent a significant threat if an attacker is able to invoke (or convince the user to invoke) installed, but forgotten, programs that remain vulnerable. In Section II-A we present two attacks against such inactive versions, and we find that it is common for users to have multiple installations of an application (for example, 50% of the hosts in our study have Adobe Flash installed both as a browser plugin and as part of the Adobe Air runtime). We therefore recommend the developers of software updating systems to check the hard drive for all the installed versions and to implement the updater as a background service, which runs independently of the application and which automatically downloads and updates all the vulnerable software detected.

**Consolidating patch mechanisms.** We find that, when vendors use multiple patch dissemination mechanisms for common vulnerabilities (e.g., for Acrobat Reader and Flash Player), some applications may remain vulnerable after the user believes she has installed the patch. This highlights a bigger problem with code shared among multiple applications. Many applications include third-party software components—both commercial and open source—and they disseminate security patches for these components independently of each other. In this model, even if the developer notifies the application vendor of the vulnerability and provides the patch, the vendors must integrate these patches in their development and testing cycle, often resulting in delays. For example, vulnerabilities in the Android kernel and device drivers have a median time-to-patch of 30–40 weeks, as the mobile device manufacturers are responsible for delivering the patches to the end-users [25]. Unpatched code clones are also common

in OS code deployed in the field [23]. Our measurements suggest that, when a vulnerability affects several applications, the patches are usually released at different times—even when the two applications are developed by the same organization—and that the time between patch releases gives enough window for attackers to take advantage of patch-based exploit generation techniques. In consequence, we recommend that patch dissemination mechanisms be consolidated, with all software vendors using one or a few shared channels. However, in ecosystems without a centralized software delivery mechanism, e.g., workstations and embedded devices, consolidation may be more difficult to achieve. Moreover, coordinating patch releases among multiple vendors raises an interesting ethical question: when one vendor is not ready to release the patch because of insufficient test coverage, is it better to delay the release (in order to prevent patch-based exploit generation) or to release the patch independently (in order to stop other exploits)?

**Updates for libraries.** A more radical approach would be to make library maintainers responsible for disseminating security patches for their own code, independently of the applications that import these libraries. This would prevent patching delays, but it would introduce the risk of breaking the application's dependencies if the library is not adequately tested with all the libraries that use it. This risk could be minimized by recording the behavior of the patch in different user environments and making deployment decisions accordingly [9]. These challenges emphasize the need for further research on software updating mechanisms.

**Program versioning.** We observe that identifying all vulnerabilities affecting a host is challenging due to the various vendor approaches for maintaining program and file versions. We conclude that software vendors should have at least one filename in a product whose file version is updated for each program version. Otherwise, it becomes very complicated for a user or analyst to identify the installed program version. This situation happens with popular programs such as Internet Explorer and in some versions of other programs like Adobe Reader. This situation also creates issues for the vendor, e.g., when clicking the "About" menu entry in Reader 9.4.4 it would still claim it was an earlier version. So, even if it is possible to patch a vulnerability by modifying only a secondary file, we believe vendors should still update the main executable or library to reflect a program version upgrade. This would also establish a total ordering of how incremental patches should be applied, simplifying the tracking of dependencies.

### B. Improving Security Risk Assessment

This work contributes new vulnerability metrics that can complement the Common Vulnerability Scoring System (CVSS), currently the main risk assessment metric for software vulnerabilities [35]. The CVSS score captures exploitability and impact of a vulnerability, but it does not address the vulnerable population size, the patching delay for the vulnerability, the patching rate, and the updating mechanisms of the vulnerable program. Enhancing CVSS with the above information would provide stronger risk assessment, enabling system administrators to implement policies such as subjecting hosts patched infrequently to higher scrutiny, prioritizing patching of vulnerabilities with larger vulnerable populations, or scanning more frequently for vulnerabilities as the updating rate decreases. These metrics have several potential applications:

- *Customizing security:* Security vendors could customize the configuration of their security tools according to the risk profile of a user. For example, they could enable more expensive, but also more accurate, components of their product for users at higher risk.
- *Improving software whitelists:* One issue when using software whitelists for malware protection [10] is that newly developed benign software would be considered malicious since it is not yet known to the static whitelist. Our software developer profiles can reduce false positives by not flagging as malicious new executables from developers that have not been externally obtained (e.g., from the Internet, USB, or optical disk).
- *Educating the users:* Many users may be interested in receiving feedback about their current security risk profile and suggestions to improve it.
- *Cyber-insurance:* The new vulnerability metrics and risk profiles could significantly improve the risk assessment methods adopted by insurance companies that currently rely on questionnaires to understand how specific security measures are taken by a given organization or individual to establish their policy cost.

## VIII. Conclusion

We investigate the patch deployment process for 1,593 vulnerabilities from 10 client-side applications. We analyze field data collected on 8.4 million hosts over an observation period of 5 years, made available through the WINE platform from Symantec. We show two attacks made possible by the fact that multiple versions of the same program may be installed on the system or that the same library may be distributed with different software. We find that the median fraction of vulnerable hosts patched when exploits are released is at most 14%. For most vulnerabilities, patching starts around the disclosure date, with a high initial rate of patching. The patching rate decreases over time, but for some vulnerabilities we observe multiple waves of intense patching activity. The patch mechanism has an important impact on the rate of patch deployment: applications updated automatically have a median time-to-patch 1.5 times lower than applications that require the user to apply patches manually. However, only 28% of the patches in our study reach 95% of the vulnerable hosts during our observation period. This suggests that there are additional factors that influence the patch deployment process. For example, users have an important impact on the patch deployment process, security analysts and software developers deploy patches faster than the general user population. Most of the vulnerable hosts remain exposed when exploits are released in the wild. Our findings will enable system administrators and security analysts to assess the the risks associated with vulnerabilities by taking into account the milestones in the vulnerability lifetime, such as the patching delay and the median time-to-patch.

## IX. Acknowledgements

We thank Symantec for access to WINE. We also thank VirusTotal and the people behind NVD, EDB, and OSVDB for making their information publicly available. This work utilized dataset WINE-2015-001, archived in the WINE infrastructure.

This research was partially supported by the Regional Government of Madrid through the N-GREENS Software-CM project S2013/ICE-2731, and by the Spanish Government through Grant TIN2012-39391-C04-01, a Juan de la Cierva Fellowship for Juan Caballero. Partial support was also provided by the Maryland Procurement Office, under contract H98230-14-C-0127 for Tudor Dumitraș. All opinions, findings and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.

## References

[1] Private communication from leading security vendor.

[2] L. Allodi and F. Massacci. A preliminary analysis of vulnerability scores for attacks in wild. In *CCS BADGERS Workshop*, Raleigh, NC, Oct 2012.

[3] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of vulnerability: A case study analysis. *IEEE Computer*, 33(12), December 2000.

[4] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: automatic exploit generation. In *Network and Distributed System Security Symposium, San Diego, California, USA*. The Internet Society, 2011.

[5] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*. USENIX Association, 2010.

[6] L. Bilge and T. Dumitraş. Before we knew it: an empirical study of zero-day attacks in the real world. In *ACM Conference on Computer and Communications Security*, pages 833–844. ACM, 2012.

[7] S. L. Blond, A. Uritesc, C. Gilbert, Z. L. Chua, P. Saxena, and E. Kirda. A Look at Targeted Attacks through the Lense of an NGO. In *USENIX Security Symposium*, August 2014.

[8] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy*, pages 143–157, Oakland, CA, May 2008.

[9] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel. Staged deployment in Mirage, an integrated software upgrade testing and distribution system. In *ACM Symposium on Operating Systems Principles, Stevenson, Washington, USA*, pages 221–236. ACM, 2007.

[10] S. Dery. Using whitelisting to combat malware attacks at fannie mae. *IEEE Security & Privacy*, 11(4), August 2013.

[11] T. Dübendorfer and S. Frei. Web browser security update effectiveness. In *CRITIS Workshop*, September 2009.

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

[13] T. Dumitras and P. Efstathopoulos. The provenance of wine. In *EDCC 2012*, May 2012.

[14] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. In *Internet Measurement Conference*, Vancouver, Canada, Nov 2014.

[15] Exploit database. http://exploit-db.com/.

[16] S. Frei. *Security Econometrics: The Dynamics of (In)Security*. PhD thesis, ETH Zürich, 2009.

[17] S. Frei and T. Kristensen. The security exposure of software portfolios. In *RSA Conference*, March 2010.

[18] C. Gkantsidis, T. Karagiannis, P. Rodriguez, and M. Vojnovic. Planet scale software updates. In *SIGCOMM*, pages 423–434. ACM, 2006.

[19] Google. Chrome releases. http://googlechromereleases.blogspot.com.es/.

[20] C. Grier et al. Manufacturing compromise: the emergence of exploit-as-a-service. In *ACM Conference on Computer and Communications Security*, Raleigh, NC, Oct 2012.

[21] S. Hardy, M. Crete-Nishihata, K. Kleemola, A. Senft, B. Sonne, G. Wiseman, P. Gill, and R. J. Deibert. Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In *USENIX Security Symposium*, 2014.

[22] Heartbleed bug healt report. http://zmap.io/heartbleed/.

[23] J. Jang, A. Agrawal, and D. Brumley. Redebug: Finding unpatched code clones in entire OS distributions. In *IEEE Symposium on Security and Privacy,May 2012, San Francisco, California, USA*, pages 48–62. IEEE Computer Society, 2012.

[24] D. G. Kleinbaum and M. Klein. *Survival Analysis: A Self-Learning Text*. Springer, third edition, 2011.

[25] A. Lineberry, T. Strazzere, and T. Wyatt. Don't hate the player, hate the game: Inside the Android security patch lifecycle. In *Blackhat*, 2011.

[26] W. R. Marczak, J. Scott-Railton, M. Marquis-Boire, and V. Paxson. When governments hack opponents: A look at actors and technology. In *USENIX Security Symposium*, 2014.

[27] Microsoft. Zeroing in on malware propagation methods. Microsoft Security Intelligence Report, 2013.

[28] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford-Chen, and N. Weaver. Inside the slammer worm. *IEEE Security & Privacy*, 1(4):33–39, 2003.

[29] D. Moore, C. Shannon, and K. C. Claffy. Code-red: a case study on the spread and victims of an internet worm. In *Internet Measurement Workshop*, pages 273–284. ACM, 2002.

[30] K. Nayak, D. Marino, P. Efstathopoulos, and T. Dumitraş. Some vulnerabilities are different than others: Studying vulnerabilities and attack surfaces in the wild. In *International Symposium on Research in Attacks, Intrusions and Defenses*, Gothenburg, Sweeden, Sep 2014.

[31] U.s. national vulnerability database. http://nvd.nist.gov/.

[32] H. Okhravi and D. Nicol. Evaluation of patch management strategies. *International Journal of Computational Intelligence: Theory and Practice*, 3(2):109–117, 2008.

[33] Osvdb: Open sourced vulnerability database. http://osvdb.org/.

[34] E. E. Papalexakis, T. Dumitraş, D. H. P. Chau, B. A. Prakash, and C. Faloutsos. Spatio-temporal mining of software adoption & penetration. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, Niagara Falls, CA, Aug 2103.

[35] S. Quinn, K. Scarfone, M. Barrett, and C. Johnson. Guide to adopting and using the security content automation protocol (SCAP) version 1.0. NIST Special Publication 800-117, Jul 2010.

[36] T. Ramos. The laws of vulnerabilities. In *RSA Conference*, 2006.

[37] E. Rescorla. Security holes... who cares. In *Proceedings of the 12th USENIX Security Symposium*, pages 75–90, 2003.

[38] Safari version history. http://en.wikipedia.org/wiki/Safari_version_history.

[39] M. Shahzad, M. Z. Shafiq, and A. X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *International Conference on Software Engineering*, 2012.

[40] C. Shannon and D. Moore. The spread of the witty worm. *IEEE Security & Privacy*, 2(4):46–50, 2004.

[41] Symantec Corporation. Symantec Internet security threat report, volume 16, April 2011.

[42] Symantec Corporation. Symantec Internet security threat report, volume 17. http://www.symantec.com/threatreport/, April 2012.

[43] T. M. Therneau. *A Package for Survival Analysis in S*, 2014. R package version 2.37-7.

[44] Virustotal. http://www.virustotal.com/.

[45] The webkit open source project. http://webkit.org.

[46] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage. When private keys are public: results from the 2008 debian openssl vulnerability. In *Internet Measurement Conference*, pages 15–27. ACM, 2009.

## APPENDIX A
## CLEAN NVD

An important challenge to the automatic generation of vulnerability reports are NVD inaccuracies. We have spent significant effort on a `Clean NVD` subproject, whose goal is identifying and reporting discrepancies in NVD vulnerability entries. This section briefly introduces our efforts. We have contacted the NVD managers about these issues and have set up a website to detail our findings and help track fixes to NVD at http://clean-nvd.com/.

The 3 main NVD inaccuracies we found are programs with vulnerable product lines rather than program versions, and *missing* and *extraneous* vulnerable versions. Surprisingly, we found that vulnerabilities in popular programs such as Microsoft Word and Internet Explorer only contain vulnerable program lines. For example, NVD states that CVE-2009-3135 affects Word 2002 and 2003, but those are product lines rather than program versions. Note that these programs do not have program versions proper, i.e., there is no such thing as Word 2003.1 and Word 2003.2. Instead, Microsoft issues patches to those programs that only update file versions. Currently, an analyst/user cannot use the NVD data to determine if the Word version installed on a host is vulnerable. We believe that NVD should add the specific file versions (for a version-specific filename such as msword.exe) that patched the vulnerability. To validate that this is possible, we crawled the Word security advisories in MSDN for 2008-2012, which contain the CVEs fixed and link to pages describing the specific msword.exe file versions released with the advisory. We have used that information to build the Word version mapping and cluster files needed for the analysis.

A vulnerability in NVD may miss some vulnerable program versions. These are likely due to errors when manually entering the data into the database as the missing versions typically appear in the textual vulnerability descriptions and the vendor's advisories. A vulnerability may also contain extraneous vulnerable versions. We find two reasons for these: errors when inserting the data and vendors conservatively deciding which program versions are vulnerable. Specifically, vendors seem to often determine the last vulnerable version in a product line and then simply consider all prior versions in the line vulnerable, without actually testing if they are indeed vulnerable. This cuts vulnerability testing expenses and helps pushing users to the latest version.

**Finding errors.** To automatically identify missing and extraneous vulnerable versions in NVD we use two approaches. For Firefox and Thunderbird, we have built scripts that collect Mozilla security advisories, parse them to extract the patched versions in each line, and compare them with the last vulnerable versions in NVD. In addition, we have developed a generic differential testing approach to compare the vulnerable ranges in the textual vulnerability description with the vulnerable versions in the NVD XML dumps. To automate this process we use natural language processing (NLP) techniques to extract the vulnerable ranges from the textual description. We have applied the differential testing approach to the 10 programs finding 608 vulnerabilities with discrepancies, which we exclude from the analysis. While these approaches are automated and identify a large number of NVD errors, they may not find all errors. Thus, we also manually check one vulnerability in each remaining cluster comparing the version list against the vendor's advisories and the disclosure dates against the release dates of the patched versions.