Does OpenBSD and Firefox's Security Improve with Time?

Jian Shi, Deqing Zou, Shouhuai Xu, Xianjun Deng, and Hai Jin, *Fellow, IEEE*

Abstract—Ozment and Schechter (Usenix Security'2006) analyzed the evolution of OpenBSD vulnerabilities over the span of 7 years (1998-2005) and concluded that its security increases with age. In this paper, we extend their study by analyzing the evolution of OpenBSD vulnerabilities over the span of 22 years (1998-2020) and Firefox vulnerabilities over the span of 9 years (2011-2020). Our empirical study leads to a number of insights, including the following: both OpenBSD and Firefox get more secure (i.e., less vulnerable) with time, but today's developers do not necessarily produce more secure code; OpenBSD and Firefox developers tend to make similar security mistakes, but Firefox vulnerabilities are easier to exploit; finally, Firefox's vulnerability density is almost one order of magnitude higher than OpenBSD's, meaning Firefox is more vulnerable.

Index Terms—Security, Vulnerability, Vulnerability metric, Dependability.

1 Introduction

Software systems evolve over time for a number of reasons, such as when developers fix bugs, patch vulnerabilities, tune performance, or add new features. Unfortunately, these changes are often accompanied by new vulnerabilities in the software. It is important to understand the evolution of vulnerabilities in a software system because vulnerabilities are directly related to the security of a software system. This importance has motivated many studies from various perspectives of software security, including: analyzing operating system kernels via global static analysis [1], bug detection [2], [3], [4], and vulnerability detection [5], [6], [7], [8], [9], [10], [11], [12], [13], [14].

In particular, Ozment and Schechter investigated the evolution of OpenBSD vulnerabilities over the span of 7 years (1998-2005) [15]. They concluded that OpenBSD was getting increasingly secure with age. Given that 15 years have passed, we wonder whether their finding is still valid or not, and whether their finding is true for other large software systems. To compare these fairly, we must carefully consider the security of each software system with respect to its size. Specifically, we conduct an empirical study by *collecting* and *analyzing* vulnerabilities in OpenBSD and Firefox over a span of many years; we select these two software systems because they are representative of large, widely-employed systems.

Our contributions. Our first contribution is a methodology for collecting and pre-processing a software's vulnerabilities from multiple sources and for using the collected data to address a range of research questions we define in this paper. In the course of pre-processing, we observe that the vulnerability introduction time is rarely documented. We address this problem by tracing the vulnerable lines of code (LOC) back to the software release that introduced them. For addressing the research questions, we introduce a number of metrics, such as: *vulnerability lifetime*, *residual vulnerability*, and *software age* for comparing the security of software systems. The methodology (including the new metrics) can be adopted as is or adapted to study the evolution of any software system's vulnerabilities.

Our second contribution is to demonstrate the usefulness of the methodology by using it to conduct a case study on the evolution of OpenBSD and Firefox' vulnerabilities. For OpenBSD, we collect 263 vulnerabilities between release 2.3 (1998) and release 6.7 (2020); for Firefox, we collect 1,630 vulnerabilities between release 4.0 (2011) and release 75.0 (2020). We have made our vulnerability datasets anonymously available on GitHub (at `https://github.com/VulnSet/BetterOrWorse`). By analyzing the collected dataset, we draw a number of insights: (i) OpenBSD and Firefox developers tend to make similar security mistakes, especially with respect to memory management. (ii) Although OpenBSD and Firefox vulnerabilities cause a similar spectrum of damage, Firefox vulnerabilities are easier to exploit because they inherently allow remote exploitation without authentication. (iii) Firefox vulnerabilities have a shorter average lifetime than those found in OpenBSD, perhaps because Firefox has a much larger developer community, publishes multiple beta versions, and includes a mature bug tracking system via Bugzilla [16]. (iv) The OpenBSD module that has the most vulnerabilities

Corresponding author: Deqing Zou.

- *J. Shi, D. Zou is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {shijianwh, deqingzou} @hust.edu.cn*
- *S. Xu is with the Department of Computer Science, University of Colorado Colorado Springs, Colorado, USA 80918. This work was partly done when he was at University of Texas at San Antonio. E-mail: sxu@uccs.edu*
- *X. Deng is with School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: dengxj615@hust.edu.cn*
- *H. Jin is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: hjin@hust.edu.cn*

is `sys/kern`, and the Firefox module that has the most vulnerabilities is `js/src`. (v) Both OpenBSD and Firefox get more secure over time, but today's developers do not necessarily produce more secure code. For OpenBSD, our 22-year findings (1998-2020) reinforce those of Ozment and Schechter, which spanned only 7 years (1998-2005) [15]. (vi) Residual vulnerabilities for both OpenBSD and Firefox are decreasing, but Firefox has many more in general. (vii) Firefox's vulnerability density is almost one order of magnitude higher than OpenBSD's, despite Firefox having shorter vulnerability lifetimes on average.

Related work. We divide related prior studies into five perspectives: software quality; software evolution; bug and vulnerability characterization; vulnerability lifecycle; and vulnerability detection.

From the perspective of software quality, Behnamghader et al. [17] investigated software commits and found that different commits have different impacts on software quality; Clark et al. [18] showed that Firefox's rapid-release methodology does not necessarily cause more vulnerabilities; Vassallo et al. [19] investigated the practice of continuous code quality. In contrast to these studies, we focus on the *overall* evolution of a software as well as its vulnerabilities, rather than a particular aspect of software development (e.g., commit activities or release methodology).

From the perspective of software evolution, D'Ambros et al. [20] investigated software evolution in terms of developer effort, change coupling, and trend and hotspot; Ganpati et al. [21] investigated the maintainability of open-source software and found that Firefox has a high maintainability; Feng et al. [22] studied the evolution of open source projects and found that a project's active hotspots can reveal architecture problems. In contrast to these studies, we focus on analyzing the evolution of software vulnerabilities over time.

From the perspective of characterizing software bugs [23], [24] and vulnerabilities [15], [25], [26], [27], [28], Li et al. [29] and Tan et al. [24] showed that security-related bugs and vulnerabilities in open-source software are increasing; Jimenez et al. [26] characterized vulnerabilities in Linux kernel and OpenSSL; Wu et al. [30] investigated how to determine a bug's security impact; Mu et al. [31] investigated how to improve the reproducibility of crowdsourced vulnerabilities; Dong et al. [32] showed that vulnerability databases are often inconsistent with each other; Shen et al. [33] studied the evolution of exploits; Feng et al. [34] investigated vulnerabilities through the lens of bug reports. In contrast, we study the evolution of OpenBSD and Firefox vulnerabilities.

From the perspective of vulnerability lifecycle, Shahzad et al. [35], [36] investigated hackers' exploitation behaviors as well as vendors' patching behaviors and found that executable code, denial-of-service, and buffer overflow are the most exploited vulnerabilities; Bilge et al. [37] investigated zero-day attacks and zero-day vulnerability lifecycles; Li et al. [25] showed that security patches are more localized than non-security patches. In contrast, we analyze the evolution of OpenBSD and Firefox vulnerabilities.

From the perspective of vulnerability detection, there have been many studies on using fuzzing to detect vulnerabilities [38], [39], [40], using code-similarity to detect vulnerabilities [5], [6], [7], and using deep learning to detect vulnerabilities [8], [9], [10], [11]. These studies are complementary to the present study.

Paper outline. Section 2 presents the methodology. Section 3 applies the methodology to conduct a case study on OpenBSD and Firefox. Section 4 discusses the limitations of the present study. Section 5 concludes the paper.

2 METHODOLOGY

In order to analyze the evolution of vulnerabilities in a software system and compare the security of different software systems over time, we propose a methodology with three modules: (i) *defining vulnerability attributes*, (ii) *collecting vulnerability data*, (iii) defining research questions. Fig. 1 highlights these components, which are elaborated below.

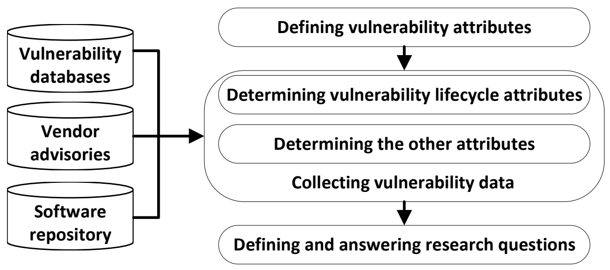

Vulnerability databases
Vendor advisories
Software repository
Defining vulnerability attributes
Determining vulnerability lifecycle attributes
Determining the other attributes
Collecting vulnerability data
Defining and answering research questions

Fig. 1: Methodology overview.

2.1 Defining Vulnerability Attributes

We propose defining 15 attributes to describe vulnerabilities. Among them, 6 attributes describe the vulnerability lifecycle and 9 attributes characterize other aspects.

2.1.1 *Defining Vulnerability Lifecycle Attributes*

A vulnerability's lifecycle includes several specific points in time, which are highlighted in Fig. 2 and elaborated below.

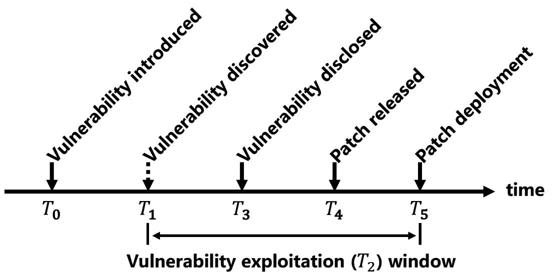


Fig. 2: Vulnerability lifecycle, where T_1 is indicated by a dashed arrow because it may not be known to the vendor/defender (e.g., when discovered by attackers), and T_2 is indicated by a time window because T_2 can happen at any point in the time window of $[T_1, T_5)$ or will never be exploited.

Vulnerability introduction time (T_0). This is the point in time at which the software containing a vulnerability is published for the first time. Suppose software release 1.0 is published on March 1, 2010 and release 2.0 is published on May 1, 2012, with no releases between the two. For a vulnerability that is introduced into the code base after release 1.0 but before release 2.0, T_0 is May 1, 2012 because it is the time at which both the attacker and defender may start to detect vulnerabilities in the software.

Vulnerability discovery time (T_1). This is the time at which a vulnerability is discovered by the vendor, defender, or attacker. In the case where the vulnerability is discovered by an attacker, the existence of the vulnerability may not be known to the vendor until after it has been exploited. This means that the precise T_1 may not be known to the vendor. Conversely, T_1 is known for the vulnerabilities that are discovered by the vendor or researchers. Note that $T_1 - T_0$ is the period of time that is spent discovering a vulnerability.

Vulnerability exploitation time (T_2). This is the time at which a vulnerability is exploited by an attacker. The exploitation of a vulnerability may not be known to the vendor until after the exploitation is detected by a defender. Given that attacks are often detected with significant delay, the precise T_2 may not be known to the vendor or defender.

Vulnerability disclosure time (T_3). This is the time at which a vulnerability is disclosed by the vendor or a public vulnerability database. T_3 is well documented in practice.

Vulnerability patch release time (T_4). This is the time at which a patch is published. T_4 is well documented in practice. Note that $T_4 - T_0$ is a vulnerability's *lifetime*, the period of time between its introduction and its patch is released.

Vulnerability patch deployment time (T_5). This is the time at which a specific end user deploys the patch to a particular software vulnerability. T_5 may not be well documented in practice. Note that $T_5 - T_0$ is a vulnerability's *exploitable lifetime*, namely the period of time between its introduction and it is patched in a specific system. This means that a single vulnerability may have different exploitable lifetimes in different environments. It is possible that $T_5 >> T_4$, meaning $T_5 - T_0 >> T_4 - T_0$, because some users may delay the patching of a vulnerability for a long period of time.

An ideal vulnerability repository maintained by a software vendor should document $T_0, \ldots, T_4$, while noting that T_1 and T_2 may not be known. An ideal end-user vulnerability management system should document $T_0, \ldots, T_5$, where $T_0, \ldots, T_4$ are inherited from the software vendor and T_5 is specific to the end-user and/or a specific instance of the vulnerable software.

2.1.2 Defining Other Attributes

We propose 9 attributes to describe the other aspects of vulnerabilities. We divided these 9 attributes into three categories.

Category 1: Attributes related to a vulnerability's Common Vulnerability Scoring System (CVSS) [41]. The CVSS base metrics describe a vulnerability's time- and environment-independent properties in two sub-categories. One sub-category describes the *exploitability* metrics reflecting the degree of difficulty in exploiting a vulnerability. This sub-category has three attributes: (i) `AccessVector` (what kind of access is required in order for an attacker to exploit a vulnerability?); (ii) `AccessComplexity` (what is the complexity that is required in order for an attacker to exploit a vulnerability?); and (iii) `Authentication` (how many times an attacker must authenticate to a target system in order to exploit a vulnerability?). In this sub-category, a higher value means it's easier to exploit.

The other sub-category describes the *impact* metrics reflecting the confidentiality, integrity, and availability consequences when a vulnerability is exploited. This sub-category also has three attributes: (iv) `ConfImpact` (what is the impact of a successful exploitation on confidentiality?); (v) `IntegImpact` (what is the impact of a successful exploitation on integrity?); and (vi) `AvailImpact` (what is the impact of a successful exploitation on availability?). In this sub-category, a higher value means a bigger damage.

Category 2: Attribute related to the Common Weakness Enumeration (CWE) [42]. This catgory has one attribute: (vii) `vulnerability type` according to the standard CWE classification. For example, CWE type 119 represents buffer overflows.

Category 3: Attributes describing the software releases that contain the vulnerability in question. This category has two attributes: (viii) `vulnerability introduction` release (what is the first or "birth" release of a software that contains the vulnerability in question, namely the time T_0 mentioned above?); and (ix) `vulnerability patch` release (what is the last release of a software that contains the vulnerability in question, namely the the time T_4 mentioned above?).

2.2 Collecting Vulnerability Data

It would be ideal to have a database that documents all of the vulnerabilities in a software over time, with all of the vulnerability details. However, vulnerability information is often scattered among multiple sources and lacks any standardized format. This is true despite the existence of several public vulnerability databases, such as NVD [43] and BID [44]. Unfortunately, these databases do not always provide all of the relevant information, perhaps because software vendors have been maintaining their own vulnerability repositories and security advisories (e.g., OpenBSD's errata bulletin [45], Mozilla Foundation's security advisories [46], and Ubuntu's security notices [47]). From these sources, one can extract a list of vulnerabilities corresponding to a given software system. In particular, every vulnerability listed in NVD has a Common Vulnerabilities and Exposures (CVE) identifier. Given this list of vulnerabilities, one can proceed to determine (i) vulnerability lifecycle attributes and (ii) other attributes.

2.2.1 Determining Vulnerability Lifecycle Attributes

It is often difficult to determine the vulnerability introduction time T_0, even for vendors or developers, because doing this requires one to understand the semantics of the source code. Nevertheless, the fact that a patch modifies certain portion(s) of source code can serve as a clue for identifying the vulnerable piece of code. These are the lines of code that are changed between (i) the committed revision that contains the patch and (ii) the immediate preceding revision. In order to determine the vulnerability introduction time T_0, one often needs to track the time at which the vulnerable lines of code are introduced. For this purpose, one first needs to identify the patch(es) corresponding to the vulnerability. Since software repositories often store the complete history of code changes, one can identify vulnerability patches as follows: (i) automatically extract the metadata of commits, including *commit message* and *commit time*; and (ii) traverse commit message to locate vulnerability patches according to vulnerability identifier pattern, such as "CVE-\d+-\d+" or "Bug \d+".

It is possible that the preceding process may identify multiple software revisions where the vulnerable lines of code are introduced. In this case, we consider two extreme choices. One choice is to use the *latest* (i.e., most recent) revision as T_0 for introducing the vulnerability in question. This is the *conservative* estimation because the earlier revisions that do not contain all of the changed lines of code may also be vulnerable (i.e., the vulnerability introduction time is actually earlier). The other choice is to use the other end of the same spectrum, namely the *earliest* revision as the time for T_0, denoted by T_0' because it is the *radical* estimation. Although there are other choices (e.g., considering revisions in between the earliest and the latest revisions), we only consider the two extremes because they would be sufficient for us to draw conclusion on the robustness of the results that are respectively derived from these two choices.

The preceding discussion leads to Algorithm 1, which takes as input the patches that are related to the vulnerability in question and outputs the two choices of vulnerability introduction time as mentioned above. In Algorithm 1, $parentRevision$ represents the immediate preceding revision of the committed revision that applies the patch; `diffHunk` is the lines of code corresponding to the patch; $changedFile$ and $lineNumbers$ are respectively the file and the line numbers of the lines of code that are changed according to the `diffHunk`; EXTRACT is a function that parses `diffHunk` and extracts $parentRevision$, $changedFile$, and $lineNumbers$ from the `diffHunk`; for the purpose of determining the revision that introduced a vulnerability, the ANNOTATE function annotates each line of code in a given program file by leveraging the revision in which the line of code is modified for the last time, while noting that ANNOTATE is readily available in a version control system. DETERMINE_RELEASE is a function that determines the software release corresponding to a given revision; RELEASE_TIME is a function that determines the release time of a given software release.

In order to determine the vulnerability discovery time T_1, one needs to know when the the vulnerability was detected. Unfortunately, it is often impossible to know when a vulnerability is detected by the attacker. In order to determine the vulnerability exploitation time T_2, one needs to obtain the publication time of the exploit in the wild or the time when the defender detects the attack exploiting the vulnerability for the first time. Generally speaking, this information is not available. Still, we should strive to extract such information whenever feasible because such information, even if only available for some vulnerabilities, would allow us to conduct further analysis. The vulnerability disclosure time T_3 and the vulnerability patch release time T_4 are often available. However, the patch deployment time T_5 often varies with the end users, depending on how quickly they patch known vulnerabilities. This information is only known to the end users and will not be analyzed in the present paper because of the lack of data. Nevertheless, some interesting research questions will be formulated in regards to T_5 so that a user with such information can analyze it to draw useful insights.

Algorithm 1: Determining vulnerability introduction time T_0

```
Input: patches related to a given vulnerability
Output: vulnerability introduction time $T_0$ (the conservative
        estimation) and $T_0'$ (the radical estimation)
1  $latestRevision \leftarrow 0$;
2  $earliestRevision \leftarrow INF$;
3  for each diffHunk in patches do
4      $(parentRevision, changedFile, lineNumbers) \leftarrow$
         EXTRACT(diffHunk);
5      $result \leftarrow$ ANNOTATE($parentRevision, changedFile$);
6      for each $l$ in $lineNumbers$ do
7          $latestRevision \leftarrow \max(latestRevision, result[l])$;
8          $earliestRevision \leftarrow \min(latestRevision, result[l])$;
9      end
10 end
11 $release \leftarrow$ DETERMINE_RELEASE($latestRevision$);
12 $release' \leftarrow$ DETERMINE_RELEASE($earlistRevision$);
13 $T_0 \leftarrow$ RELEASE_TIME($release$);
14 $T_0' \leftarrow$ RELEASE_TIME($release'$);
15 $returnT_0, T_0'$
```

2.2.2 Determining the Other Attributes

In order to determine the other vulnerability attributes, one may extract them from various sources. In this process, one may encounter a consistency problem, namely that different sources may provide descriptions that do not include the attributes defined above. For example, one may encounter inconsistencies between different sources' vulnerability identifiers, vulnerability disclosing times (T_3), and affected versions. In order to tackle this problem, special care must be taken. Since the problem is specific to a given source, we consider it on a case-by-case basis. Details can be found in the case study.

2.3 Defining and Answering Questions

One application of the methodology is to characterize and compare the evolution of vulnerabilities in software. This prompts us to define the following RQ1-RQ6 for individual software and RQ7 for comparing two software.

RQ1: What are the dominating types of vulnerabilities? Answering this question will identify common programming mistakes, which in turn can help developers enhance their skills in secure programming. Vulnerability types may be based on those defined in the CWE, as discussed above.

RQ2: How severe are the vulnerabilities? The severity of a vulnerability can be measured by the impact and difficulty of its exploitation. Answering this question will deepen our understanding of the severity of vulnerabilities in a software.

RQ3: What are the vulnerability lifetimes? Analyzing vulnerability lifetime, $T_4 - T_0$, allows us to draw insights into how long a vulnerability exists in a software, which may impact the ability of attackers to discover and exploit it.

RQ4: Where do the vulnerabilities reside? Answering this question requires analysis of program source code and version control, which in turn may help attribute vulnerabilities to developers for accountability management.

RQ5: Does a software system get more secure with time? This question is important because it highlights the intuition that security is an ongoing contest between attackers and defenders. In order to answer this question, one may use the *vulnerability density* metric, which is the number of vulnerabilities per a certain number of lines of code (e.g., 1,000 LOC). In order to answer this RQ, we propose using the following *basic* and *derived* metrics. The basic metrics include: (i) $\texttt{LOC}_{\text{total},r}$, which is the total number of lines of non-commented source code in a software release r; (ii) $\texttt{vuls}_{\text{total},r}$, which is the number of vulnerabilities in a software release; (iii) $\texttt{LOC}_{\text{introduced},r}$, which is the number of lines of newly introduced code in a software release r; (iv) $\texttt{vuls}_{\text{introduced},r}$, which is the number of vulnerabilities that are newly introduced in a software release r; (v) $\texttt{LOC}_{\text{inherited},r}$, which is the number of lines of code in release r that are inherited from the immediate preceding software release; (vi) $\texttt{vuls}_{\text{inherited},r}$, which is the number of vulnerabilities in release r that are inherited from the immediate preceding software release. These basic metrics can be used to derive more metrics:

- **Vulnerability density (D_1):** This metric can be defined as $D_1 = \texttt{vuls}_{\text{total},r}/\texttt{LOC}_{\text{total},r}$, namely the ratio of the number of vulnerabilities contained in a software release to the total number of lines of code in the release. D_1 reflects the overall security of a new software release.
- **Vulnerability density in new code (D_2):** This metric can be defined as $D_2 = \texttt{vuls}_{\text{introduced},r}/\texttt{LOC}_{\text{introduced},r}$, the ratio of the number of vulnerabilities that are newly introduced in a software release to the number of lines of code that are newly introduced into the release. D_2 reflects the security of the code that is newly introduced into a software release.
- **Vulnerability density of inherited code (D_3):** This metric, $D_3 = \texttt{vuls}_{\text{inherited},r}/\texttt{LOC}_{\text{inherited},r}$, is the ratio of the number of the vulnerabilities inherited from the immediate preceding release to the LOCs inherited from the immediate preceding release. D_3 reflects the security of the code inherited from the preceding release.

RQ6: How prevalent are residual vulnerabilities in a software system?

We use the notion of *residual vulnerability density* ($D_{residual}$) to describe the ratio of the number of vulnerabilities that are currently unknown but discovered later (by the attacker, the defender, or the vendor) to the total number of lines of code in the release. One application of this notion is to describe a software's unpatched vulnerability density, including zero-day vulnerabilities. Given a time horizon $[0, T]$, we use a set α_t to denote a software's set of *residual vulnerabilities* with respect to a *reference time* $t \in [0, T]$ (i.e., the point in time at which residual vulnerabilities are measured). Let $\texttt{vul}.T_i$ denote the time T_i, where $1 \leq i \leq 5$, as described in a vulnerability $\texttt{vul}$'s lifecycle (cf. Fig. 2). In principle, we have $\alpha_t = \{\texttt{vul} : \texttt{vul}.T_0 < t < \texttt{vul}.T_5\}$. Since T_5 is not known to us in our case study, we propose defining $\alpha_t = \{\texttt{vul} : \texttt{vul}.T_0 < t < \texttt{vul}.T_4\}$; this corresponds to the ideal case that every patch is immediately applied whenever becoming available (i.e., the best-case scenario with $T_4 = T_5$). Let $\texttt{LOC}_{\text{total},t}$ denote the total number of lines of code in a software release at the reference time t. Then, we define residual vulnerability density as $D_{\text{residual}} = |\alpha_t| / \texttt{LOC}_{\text{total},t}$.

RQ7: How should we compare security of two software systems? When comparing the security of two systems, it is important to ensure *fairness*, which is an elusive notion whose rigorous definition is beyond the scope of the present paper. Nevertheless, it is clearly not fair to compare the number of vulnerabilities in two systems unless they have the same size of code base. It may be tempting to compare their vulnerability densities. While intuitive, this comparison needs to be refined because two software systems may be developed at different times (e.g., one is first released in 1980's and the other is first released in 1990's). This prompts us to consider the notion of a software system's *age*, which is similar to a human's age, such that we can compare security of two software systems at the same age. For this purpose, we need to define the *birth* time of a software system. In practice, this is not straightforward because the release of a software, especially an open source one, does not have any widely accepted standard. One possibility, which we will adopt, is to use the release of the first stable code base as the birth of a software.

We propose comparing security of two software systems at the same *age* through the following metrics: (i) the afore-defined vulnerability density D_1, which also reflects the secure programming competence of developers in the respective time; (ii) the vulnerability lifetime $T_4 - T_0$, which also reflects the time window during which a software can be exploited (i.e., overall vulnerability); and (iii) the patch release time $T_4 - T_3$, which also reflects a vendor's responsiveness;

Remark. Two potentially important metrics are *vulnerability exposure time windows*, which can be measured by $T_5 - T_3$ and $T_5 - T_4$. The former reflects the time window during which an attacker may exploit the vulnerability after a vulnerability is disclosed, whereas the latter reflects the responsiveness of the end user in applying patches that are already available. These metrics can only be measured by practitioners who patch vulnerabilities and know the patch deployment time T_5.

3 Case Study on OpenBSD and Firefox

Now we apply the methodology to conduct a case study on OpenBSD and Firefox. We choose OpenBSD and Firefox for the case study because: (i) they are widely employed and major targets of attackers; (ii) they use software version control systems and their source code is publicly available, which enables us to look back into their source code history; (iii) the 1998-2005 code base of OpenBSD was analyzed in [15], making us curious about whether the 2005-2020 code bases would continue the trend; and (iv) we are curious which is more secure. Since OpenBSD is completely implemented in C/C++ and Firefox is mainly implemented in C/C++ but partly implemented in Java, JavaScript or other languages, we focus on C and C/C++ vulnerabilities for fair comparison purposes.

3.1 Collecting Vulnerability Data

For OpenBSD, we set its birth time as May 19, 1998, when release 2.3 was made public. This release is deemed as the first stable code base (as in [15]). For collecting vulnerabilities, we extract its vulnerabilities from OpenBSD's errata bulletin [45] and use crawlers to retrieve the information about their vulnerabilities from NVD and BID. In total, we collected 263 vulnerabilities from 44 releases, including 114 vulnerabilities that were detected after the previous study on the 1998-2005 code base [15]. Among the 263 vulnerabilities, 136 have no CVE identifiers, 104 have one CVE identifier, 10 have two CVE identifiers, and 13 have three or more CVE identifiers; this is caused by the fact that OpenBSD developers sometimes fix multiple vulnerabilities via a single patch. We treat these sets of multiple vulnerabilities as one each because we do not have enough information to distinguish them from each other.

For Firefox, we set its birth time as March 22, 2011, when release 4.0 is made public. This release is deemed by many as the first stable code base because Firefox had a huge iteration in this release. For collecting vulnerabilities, we extract its vulnerabilities from MFSA [46] and use crawlers to retrieve vulnerability information from NVD and BID. In total, we collected 1,630 vulnerabilities from 72 releases. Among these vulnerabilities, 634 CVE identifiers correspond to a single vulnerability, 34 CVE identifiers correspond to two vulnerabilities, 141 CVE identifiers correspond to three or more vulnerabilities. The latter two scenarios can happen because a CVE identifier may accommodate multiple unspecified vulnerabilities (e.g. CVE-2016-1952).

3.1.1 Determining Vulnerability Lifecycle Attributes

Since a vulnerability's patch deployment time T_5 is only known to end users, we will only investigate $T_0, \ldots, T_4$.

Determining a vulnerability's introduction time T_0. This is where we encounter the challenge mentioned above, determining T_0 when it is not available from the data sources. At a high level, we address this challenge as follows: (i) we leverage the modified lines of code in a patch to help identify the vulnerable piece of code; (ii) we track the modified lines of code back to the software release that introduced them; (iii) we treat this release as the vulnerability's introduction time T_0. Given that the release identified as such may not be unique because some modified lines of code may be introduced in one release and the others may be introduced in another release, we propose considering *conservative* vs. *radical* estimation of T_0, which are elaborated below.

For OpenBSD, developers use a sequence number to identify a vulnerability in each release, such as "012_openssl" (meaning the 12th patch to openssl). Since OpenBSD vulnerability identifiers do not provide enough information and some commit descriptions do not include vulnerability or CVE identifiers, we identify OpenBSD patches in its errata bulletin, rather than searching for vulnerability patches through their commit messages. After identifying OpenBSD patches, we use Algorithm 1 to determine T_0. The input to Algorithm 1 is the list of relevant patches corresponding to the vulnerability in question. We extract the lines of code that are changed between two consecutive revisions and use these as a vulnerability candidate (Line 4 in Algorithm 1). In OpenBSD's version control system, the `annotate` command extracts each line of code in a given program file, along with the information regarding when the line of code is introduced (Line 5 in Algorithm 1). This allows us to locate vulnerable revisions, namely the latest and earliest modifications to these lines of code (Line 6-9 in Algorithm 1). We use these two ends of the spectrum as the estimated vulnerability introduction time T_0 (i.e., *conservative* vs. *radical* estimation mentioned above), which will be compared with each other to demonstrate the robustness of our methodology.

For Firefox, we locate a patch by traversing the commits corresponding to a given vulnerability according to its identifier pattern "Bug \d+". This method is reliable because Firefox mandates developers to include a bug number and a clear explanation for each fix. In the case a commit is later reverted (e.g., because they introduced new bugs or were unreliable), we ignore the problematic versions in order to avoid any unintended side-effects. Given a patch, we can use Algorithm 1 to determine the corresponding vulnerability's introduction time T_0. In order to further clarify the idea and demonstrate the execution of Algorithm 1, consider the example of Firefox vulnerability #682335. In order to patch this vulnerability, Firefox revision 73339, dated August 26, 2011, evolved to revision 73348, dated September 9, 2011, whereby three lines of code (#759, #770 and #779) in revision 73339 are deleted. These three lines of code are the vulnerability candidate (Line 4 in Algorithm 1). Then, we use command "`hg annotate -r 73339 content/canvas/src/WebGLContext.h`" to extract the last revisions, 63070 (dated February 24, 2011), 63070 (dated February 24, 2011), and 43009 (dated June 1, 2010), which introduced these three lines of code (Line 5-9 in Algorithm 1). Since the release corresponding to revision 63070 is Firefox 4.0, Firefox's birth time for the sake of this study (as explained above), we treat the release corresponding to revision 43009 as Firefox 4.0 even though it may correspond to an earlier release. Therefore, we set the release time of Firefox 4.0 (March 22, 2011) as the introduction time T_0 of vulnerability #682335. As discussed above, we do not use June 1, 2010 or February 24, 2011 as T_0 because these revisions are internal to the developers but are not released to the public.

Determining $T_1, \ldots, T_4$. For OpenBSD, the vulnerability discovery time T_1 is not available. The vulnerability exploitation time T_2 can be set as the earliest publication date of the corresponding exploit. By looking into the exploits in NVD, BID and Exploit-DB [48], we find 90 OpenBSD exploits in total. The vulnerability disclosure time T_3 can be set as the earliest time when a vulnerability is published in NVD and BID. The vulnerability patch release time T_4 can be set as a patch's release time in OpenBSD's errata bulletin.

For Firefox, a vulnerability discovery time T_1 can be approximated as the vulnerability's *opened date* in Bugzilla, which is the date indicating when the developer discovered the vulnerability. The vulnerability exploitation time T_2 can be set as the earliest publication date of exploits, and we find 128 exploits in total. The vulnerability disclosure time T_3 can be set as the earliest time when a vulnerability is published in NVD and BID. The vulnerability patch release time T_4 can be set as a patch's release time in MFSA.

3.1.2 Determining the Other Attributes

In order to determine the other attributes, we aggregate the vulnerabilities extracted from NVD, BID, and vendor advisories into a single dataset by reconciling the discrepancy between the data sources. The discrepancy is caused by differences in the indexing methods of different sources (e.g., CVE-2011-2895 and BID #49124 correspond to the same vulnerability). Whenever possible, we can automatically leverage a common identifier, such as the CVE identifier that may be used in the different sources and/or the links that are used by one source for cross-referencing to another source to reconcile the discrepancy. When the preceding automated process does not work because there are no obvious references to leverage, we manually check and map the vulnerabilities. To do this, we follow these three steps: (i) extract some reference that may help identify an record in another data source; (ii) leverage such references to automatically match the data records extracted from different data sources; and (iii) manually check and map vulnerabilities that are not resolved by the preceding automated method.

For OpenBSD, we obtain 263 vulnerabilities from OpenBSD's errata bulletin and 186 CVE identifiers from NVD and BID, where 136 (out of the 263) vulnerabilities have no CVE identifiers. Therefore, we use the vulnerability identifiers in OpenBSD's errata bulletin rather than the CVE identifiers.

For Firefox, we use Bugzilla identifiers (rather than CVE identifiers) because some CVE identifiers correspond to multiple vulnerabilities (rather than uniquely pointing to vulnerabilities). For example, CVE-2016-2804 points to multiple unspecified vulnerabilities. Bugzilla identifiers are more appropriate here because they uniquely point to vulnerabilities.

TABLE 1
CWE Types of OpenBSD Vulnerabilities Ranked by The Number of Associated CVEs.

	CWE-ID	Weakness Summary	Number
1	189	Numeric Error	18
2	119	Buffer Overflow	13
3	399	Resource Management Error	11
4	264	Access Control Error	5
5	20	Improper Input Validation	5
6	310	Cryptographic Issues	4
7	200	Information Disclosure	4
8	362	Race Condition	3
9	17	Code	2
10	284	Improper Access Control	2
11	611	Improper Restriction of XML External Entity Reference	1
12	287	Improper Authentication	1
13	190	Integer Overflow or Wraparound	1
14	476	NULL Pointer Dereference	1

TABLE 2
Top 20 CWE Types of Firefox Vulnerabilities Ranked by The Number of Associated CVEs.

	CWE ID	Weakness Summary	Number
1	119	Buffer Overflow	221
2	416	Use After Free	73
3	399	Resource Management Error	62
4	264	Access Control Error	56
5	200	Information Disclosure	49
6	20	Improper Input Validation	38
7	79	Cross-site Scripting	28
8	189	Numeric Error	18
9	17	Code	13
10	787	Out-of-bounds Write	12
11	254	Security Features	12
12	190	Integer Overflow or Wraparound	10
13	362	Race Condition	10
14	94	Code Injection	9
15	125	Out-of-bounds Read	8
16	284	Improper Access Control	7
17	310	Cryptographic Issues	6
18	704	Incorrect Type Conversion or Cast	5
19	120	BClassic Buffer Overflow	5
20	352	Cross-Site Request Forgery	4

3.2 Answering the RQs

3.2.1 RQ1: What are the dominating types of vulnerabilities?

Table 1 lists the 14 CWE types of OpenBSD vulnerabilities. These types only cover 71 (out of the 263) OpenBSD vulnerabilities, because many vulnerabilities' CVE entries are missing or the CWE types in many CVE entries are missing. The major types of OpenBSD vulnerabilities are *Numeric Error*, *Buffer Overflow* and *Resource Management Error*.

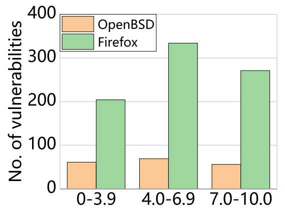
Firefox vulnerabilities are covered by 41 CWE types. These 41 types only cover 1259 (out of the 1,630) Firefox vulnerabilities, as the CVE entries in NVD for the other 371 vulnerabilities do not include their types. Table 2 lists the top 20 of the 41 types. The major types of Firefox vulnerabilities are *Buffer Overflow*, *Use After Free*, and *Resource Management Error*, while noting that buffer overflow dominates among them.

We observe that 11 types of vulnerabilities are common to OpenBSD and Firefox, including two top vulnerabilities in *Buffer Overflow* and *Resource Management Error*. Moreover, the common vulnerabilities *Buffer Overflow*, *Use After Free* and *Improper Input Validation* are mainly memory corruption vulnerabilities. This suggests that OpenBSD and Firefox developers tend to make the same kinds of security-critical errors in programming, especially memory corruption errors. This phenomenon can be attributed to that OpenBSD and Firefox are written in memory-unsafe languages (i.e., C and C++). On the other hand, Firefox suffers from web-specific vulnerabilities, such as Cross-site Scripting, which are not applicable to OpenBSD. Note that many vulnerabilities can be prevented by practicing secure coding. Proper input validation can eliminate many vulnerabilities, such as *Buffer Overflow*, *Cross-Site Scripting* and *Code Injection*; practicing standard memory management can prevent most memory management errors, such as *Use After Free*; and performing range and overflow checking can prevent *Numeric Error* vulnerabilities.

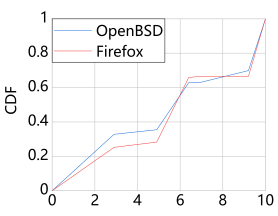
Insight 1. OpenBSD and Firefox developers make similar security mistakes, especially memory management ones.

3.2.2 RQ2: How severe are the vulnerabilities?

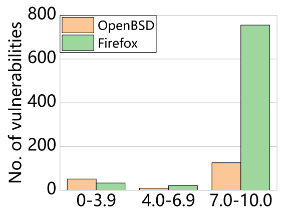
We study the severity of a vulnerability based on its *impact* and *exploitability*, which can be quantified by the CVSS [41]. Intuitively, impact score reflects a vulnerability's impact on confidentiality, integrity, and availability; a higher impact score means a greater potential damage. The exploitability score reflects the degree of difficulty in exploiting a vulnerability; a higher exploitability score means a vulnerability is easier to exploit. Vulnerabilities reported in NVD after 2016 were scored using CVSS 3.x and 2.0, while vulnerabilities reported in NVD prior to 2016 were scored using CVSS 2.0. Since our dataset consists of vulnerabilities that were reported prior to 2016, we use CVSS 2.0 for the sake of compatibility.



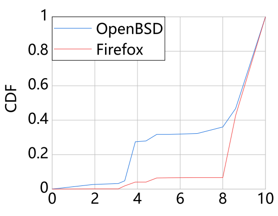
(a) Impact score



(b) Impact score



(c) Exploitability score



(d) Exploitability score

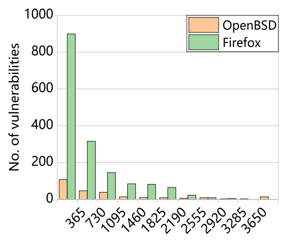
Fig. 3: Vulnerability impact and exploitability.

Fig. 3(a) plots the density function of the impact score of OpenBSD and Firefox vulnerabilities. We observe that Firefox has many more vulnerabilities than OpenBSD. Moreover, OpenBSD vulnerabilities have somewhat evenly distributed impact scores, while Firefox has many more vulnerabilities with medium or higher impact than with low impact. Fig. 3(b) plots the cumulative distribution function (CDF) of their impact scores. We observe that the two CDFs are similar, which can be explained by the fact that their density functions are somewhat close to the uniform distribution. These phenomena suggest that the proportions of high- and medium-impact vulnerabilities within OpenBSD and Firefox are similar, and that OpenBSD developers and Firefox developers make programming mistakes with similar consequences. Fig. 3(c) plots the density function of the exploitability score for OpenBSD and Firefox vulnerabilities. We observe that most OpenBSD and Firefox vulnerabilities have high exploitability scores, meaning that they are easy to exploit. Fig. 3(d) plots the CDF of the exploitability score. Let $F(x)$ denote the CDF of OpenBSD vulnerability exploitability score and $G(x)$ denote the CDF of Firefox vulnerability exploitability score. We observe that $F(x) \geq G(x)$ for any x, meaning that Firefox vulnerabilities are easier to exploit than OpenBSD vulnerabilities. One factor contributing to this is that 1628 (out of the 1,630) Firefox vulnerabilities can be exploited remotely without requiring authentication (e.g., CVE-2016-2808 can be exploited when a user visiting a malicious website). This may explain why there are more Firefox exploits than OpenBSD's (128 vs. 90), as shown by the exploits we collected.

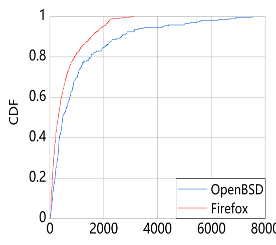
Insight 2. OpenBSD and Firefox vulnerabilities cause similar damages, but Firefox vulnerabilities are easier to exploit.

3.2.3 RQ3: What are the vulnerability lifetimes (i.e., $T_4 - T_0$)?

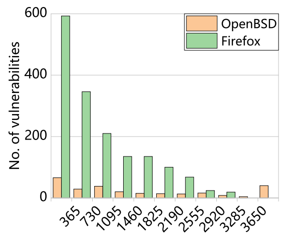
As mentioned above, we will focus on two choices when determining T_0, namely the *conservative* one that treats the most recent revision time as T_0 vs. the *radical* one that treats the earliest possible time as T_0 (denoted by T_0' for ease of reference). We observe that $T_0 = T_0'$ for 38% of the OpenBSD vulnerabilities and 44% of the Firefox vulnerabilities, which correspond to the cases where T_0 can be uniquely determined. In order to clarify the difference between T_0 and T_0', let us look at OpenBSD vulnerability "62-002_fktrace" as an example.In order to patch this vulnerability, the OpenBSD file `kern_ktrace.c` in revision 1.92, dated August 12, 2017, evolved to revision 1.93, dated November 28, 2017, whereby the function `sys_fktrace` in `kern_ktrace.c` is deleted. This function contains 64 lines of code, among which 63 were introduced in revision 1.92 and 1 was introduced in revision 1.45. Therefore, T_0 is set to be the time when revision 1.92 (or OpenBSD release 6.2) was published and T_0' is set to the time when revision 1.45 (or OpenBSD release 4.5) was published.



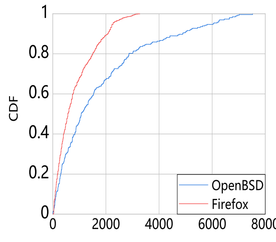
(a) $T_4 - T_0$ density function



(b) $T_4 - T_0$ CDF



(c) $T_4 - T_0'$ density function



(d) $T_4 - T_0'$ CDF

Fig. 4: Vulnerability lifetime (days).

Fig. 4(a) plots the density function of vulnerability lifetime $T_4 - T_0$ (unit: day). We observe that for any lifetime, there are many more Firefox vulnerabilities than OpenBSD vulnerabilities. Nevertheless, most OpenBSD and Firefox vulnerabilities have a lifetime under 730 days (or 2 years). Fig. 4(b) plots the CDF of vulnerability lifetime $T_4 - T_0$. For OpenBSD, we further observe that 50% of the vulnerabilities have lifetimes exceeding 477 days (15.9 months); 75% have lifetimes under 1,156 days (3.2 years); 90% have lifetimes under 2,655 days (7.3 years). For Firefox, we further observe that 50% vulnerabilities have lifetimes exceeding 308 days (10.3 months); 75% have lifetimes under 737 days (2.0 years); 90% have lifetimes under 1,547 days (4.2 years). Let $F(x)$ denote CDF of OpenBSD vulnerability lifetime and $G(x)$ denote CDF of Firefox vulnerability lifetime. We observe that $F(x) \leq G(x)$ for any x, meaning that Firefox vulnerabilities tend to have a shorter lifetime than OpenBSD vulnerabilities. In particular, OpenBSD has a significant fraction of vulnerabilities that have a long lifetime (> 3136 days), but Firefox has no vulnerabilities with this long lifetime. The relatively shorter lifetime of Firefox vulnerabilities can be attributed to Firefox's practice of multiple bate releases before a stable release.

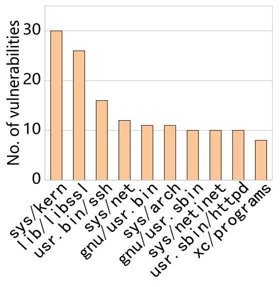
Fig. 4(c) plots the density functions of OpenBSD and Firefox's vulnerability lifetimes $T_4 - T_0'$ (unit: day). We observe that both density functions are somewhat different from the density functions of $T_4 - T_0$, which are depicted in Fig. 4(a). This is because replacing T_0 with T_0' makes some vulnerabilities' lifetimes appear longer because $T_0' \leq T_0$. This explains, for example, why there are 306 Firefox vulnerabilities whose lifetimes with respect to T_0 are within 365 days, but are longer than 365 days with respect to T_0'. Fig. 4(d) plots the CDFs of OpenBSD and Firefox's vulnerability lifetimes $T_4 - T_0'$. We observe that both CDFs are somewhat "smoother" than their counterparts with respect to $T_4 - T_0'$ because of the reason discussed above.

Recall that we collected OpenBSD vulnerabilities in versions 2.3-6.7 over the span of 22 years (1998-2020); by contrast, we collected Firefox vulnerabilities in releases 4.0-75.0 over the span of 9 years (2011-2020). Since vulnerability lifetime is affected by the availability of vulnerability detection techniques, we compare the lifetime of the same type of vulnerabilities in Firefox and different periods of OpenBSD. The median lifetime of *Buffer Overflow* vulnerabilities introduced into OpenBSD prior to 2011 is 1,073 days; the median lifetime of *Buffer Overflow* vulnerabilities introduced into OpenBSD after 2011 is 603 days; and the median lifetime of *Buffer Overflow* vulnerabilities in Firefox is 364 days.

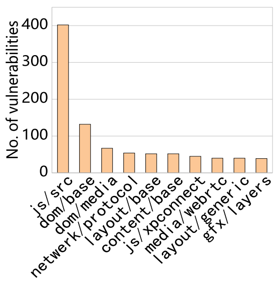
Insight 3. Firefox vulnerabilities have a shorter lifetime than OpenBSD vulnerabilities, hinting that operating systems vulnerabilities take more time and effort to detect.

3.2.4 RQ4: Where do the vulnerabilities reside?

We divide OpenBSD and Firefox into modules according to their second-level source code directories. Fig. 5 depicts the 10 OpenBSD and Firefox modules that have most vulnerabilities. The module having the most vulnerabilities is `sys/kern` for OpenBSD and `js/src` for Firefox, which are their main function modules, respectively. The vulnerability density of a module in a release is the ratio of the number of vulnerabilities in that module to the number of lines of code (LoC) in that module. We calculate the average vulnerability density of each module over its releases considered in this paper. Since there are many modules (at the order of magnitude of hundreds), we plot in Fig. 6 the 10 modules of OpenBSD and Firefox that have the highest average vulnerability density (unit: number of vulnerabilities per 10^3 LoC). The module with the highest vulnerability density is `usr.bin/su` for OpenBSD and `dom/media` for Firefox. We observe that modules with a large number of vulnerabilities do not necessarily have a high vulnerability density; instead, small modules containing few vulnerabilities can exhibit a high vulnerability density. In Firefox, `dom/media` and `dom/base` have both a large number of vulnerabilities and a high vulnerability density.

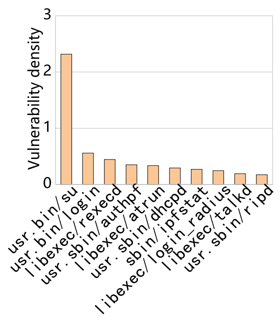


(a) Top 10 OpenBSD modules

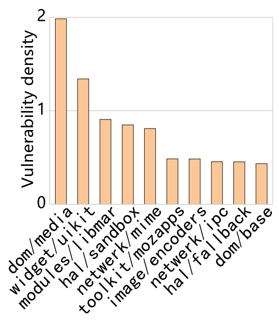


(b) Top 10 Firefox modules

Fig. 5: Vulnerable OpenBSD and Firefox modules.



(a) Top 10 OpenBSD modules



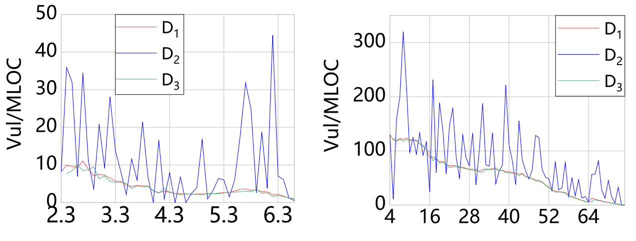
(b) Top 10 Firefox modules

Fig. 6: Ten highest vulnerability densities of modules (unit: # of vulnerabilities per 10^3 LoCs).

Insight 4. Most OpenBSD vulnerabilities belong to the `sys/kern` module and most Firefox vulnerabilities belong to the `js/src` module. The module with the highest vulnerability density is `usr.bin/su` for OpenBSD and `dom/media` for Firefox.

3.2.5 RQ5: Does a software system get more secure with time?

Guided by the vulnerability density metric defined in Section 2, Fig. 7(a) plots OpenBSD's vulnerability density (D_1), vulnerability density in new code (D_2), and vulnerability density of inherited code (D_3) with respect to its releases. We make the following observations. First, D_1 almost always decreases with time because vulnerabilities get patched when they become known and when patches become available, except for release 5.6 where a slight increase occurs; this increase is caused by the fact that there are 9 vulnerabilities in the newly introduced code and there are 6 vulnerabilities that are newly detected from the inherited code. This suggests that that OpenBSD gets more secure over the time span of 1998-2020, which reinforces the finding by Ozment and Schechter based on OpenBSD's time span of 1998-2005 [15]. Second, D_2 exhibits substantial fluctuations, this suggests that both the size of newly introduced code and the number of vulnerabilities in newly introduced code are in a sense random, hinting that newly introduced code can, but does not always, include a substantial number of vulnerabilities and that today's OpenBSD developers do *not* necessarily produce more secure code. More specifically, vulnerability density D_1 reflects the overall security of a software release, whereas vulnerability density in new code D_2 reflects the security of the code that is newly introduced into a software release (which reflects more on whether or not developers are producing more secure code with time). Third, D_3 exhibits a similar pattern as D_1. This can be explained as follows: most of the code and most of the vulnerabilities in a software release are largely inherited from the immediately preceding release, explaining why their ratios, namely D_1 and D_3, are also similar.



(a) OpenBSD (b) Firefox

Fig. 7: Vulnerability density-related metrics with time (unit: # of vulnerabilities per 10^6 LoCs).

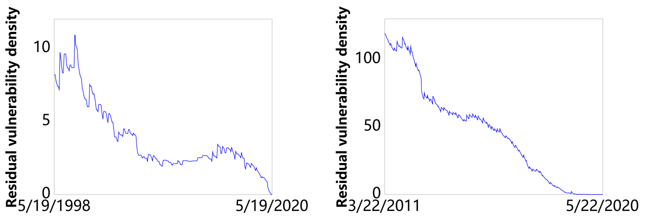
Fig. 7(b) plots Firefox's vulnerability density (D_1), vulnerability density in new code (D_2), and vulnerability density of inherited code (D_3) with respect to its releases. We observe that Firefox vulnerabilities exhibit similar phenomena to OpenBSD's. It is worth mentioning that the *actual* vulnerability density of the last few releases may be higher because they may contain vulnerabilities that are yet to be discovered. This is possible because vulnerability lifetime, even under the conservative estimation, $T_4 - T_0$, can be multiple years.

Insight 5. Both OpenBSD and Firefox get more secure with time, but today's developers do not necessarily produce more secure code. This highlights the importance of continually checking for and repairing vulnerabilities.

3.2.6 RQ6: How prevalent are residual vulnerabilities in a software system?

Guided by the residual vulnerability metrics described in Section 2, we define the reference time points, namely the t's. Since a new version of OpenBSD is released every six months and the span of time we consider is 1998-2020, we make every *month* a reference time t for measuring its residual vulnerabilities. Since a new release of Firefox is made every six weeks and the span of time we consider is 2011-2020, we make every *week* a reference time t for measuring its residual vulnerabilities.

Fig. 8(a) plots OpenBSD's residual vulnerability density where the x-axis time unit is month. We observe that by and large, OpenBSD's residual vulnerability density decreases with time. However, few vulnerabilities are discovered between November 1, 2011 (release 4.8) and May 1, 2013



(a) OpenBSD (b) Firefox

Fig. 8: Residual vulnerability density (unit: # of vulnerabilities per 10^6 LoCs).

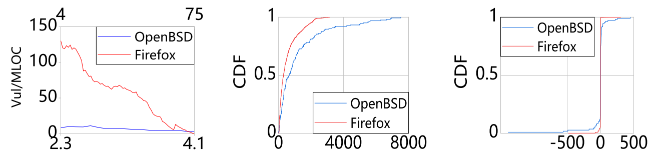
(release 5.3), which explains why the residual vulnerability density does *not* monotonically decrease with time. Fig. 8(b) plots Firefox's residual vulnerability density where the x-axis time unit is week. We observe that the residual vulnerability density generally decreases over time, with some exceptions. Roughly speaking, the chances for OpenBSD and Firefox to be attacked by 0-day attacks are generally decreasing, but the chance for Firefox to suffer from such attacks is one order of magnitude higher that of OpenBSD's because the residual vulnerability density is 3.86 (vulnerabilities per 10^6 LoCs) and Firefox's is 45.94 (vulnerabilities per 10^6 LoCs) on average. It is worth mentioning that the *actual* residual vulnerabilities corresponding to the recent releases may be higher than what is plotted because their vulnerabilities may continue to be detected as time goes by.

Insight 6. OpenBSD and Firefox's residual vulnerabilities are overall decreasing, but Firefox has one order of magnitude more residual vulnerabilities.

3.2.7 RQ7: How should we compare security of two software systems?

Recall that for OpenBSD, we collected vulnerabilities in versions 2.3-6.7 over the span of 22 years (1998-2020); for Firefox, we collected vulnerabilities in releases 4.0-75.0 over the span of 9 years (2011-2020). In order to make a fair comparison between their security, we need to take into consideration the notion of their *age* as described in the methodology. Since Firefox is much "younger" than OpenBSD, we contrast them for their first 9 years, namely OpenBSD releases 2.3-4.1 (1998-2007) vs. Firefox releases 4.0-75.0 (2011-2020). For OpenBSD, this means that we only focus on the vulnerabilities whose introduction times are earlier that OpenBSD 4.1 (dated May 1, 2007).

Fig. 9(a) contrasts OpenBSD and Firefox's vulnerability density D_1 (unit: number of vulnerabilities per 10^6 LOC). We observe that Firefox has more vulnerabilities than OpenBSD does, and that Firefox's vulnerability density is, for most of the time, almost one order of magnitude higher than OpenBSD's. These suggest that the community is committed to making OpenBSD a secure operating system. This is confirmed by the fact that (i) many security features have been adopted into OpenBSD, such as Kernel randomization, Privilege separation, Pledge, and Unveil [49], [50], [51], [52], [53], [54], and (ii) the entire OpenBSD code base has undergone a large-scale manual source code audit [45], [50], [55]. On the other hand, Firefox's rapid release cycle (i.e, 6 weeks) may have contributed to the fact that more than 91% of its vulnerabilities are detected after a number of subsequent releases. Nevertheless, Firefox uses the Security Bug Bounty Program to detect vulnerabilities. This explains why many vulnerabilities in Firefox are discovered and fixed after its releases, namely that Firefox initially exhibits a higher vulnerability density which however drops more rapidly than its OpenBSD counterpart. It is worth mentioning that this finding does not contradict [18], which says that Firefox's rapid release cycle does not cause a higher vulnerability rate. This is because their study was based on comparing Firefox *with* vs. *without* employing the rapid release cycle; by contrast, we compare Firefox against OpenBSD.



(a) Density D_1 (b) Lifetime $T_4 - T_0$ (c) Patch delay $T_4 - T_3$

Fig. 9: OpenBSD vs. Firefox vulnerabilities in their "childhood" (0-9 years old).

Fig. 9(b) plots the CDF of vulnerability lifetime $T_4 - T_0$ during OpenBSD and Firefox's "childhood" (i.e., 0-9 years old). These CDFs are similar to their counterparts with respect to OpenBSD and Firefox's lifetime, which are depicted in Fig. 4(b). Nevertheless, we observe that Firefox vulnerabilities have a shorter lifetime (549 days on average) than OpenBSD's (1227 days on average). This is a fair comparison in the sense that we only compare their "childhood", while noting that OpenBSD is 11 years older than Firefox.

Fig. 9(c) plots the patch delay $T_4 - T_3$, where $T_4 - T_3 < 0$ means that a patch is released before the vulnerability is disclosed. A greater $T_4 - T_3$ value means that the vendor has a longer response time and that the software suffers a longer period of time during which it can be attacked because of the known but yet-to-be-patched vulnerabilities. We observe that for most vulnerabilities, we have $T_3 = T_4$, indicating the practice of responsible disclosure. A few OpenBSD vulnerabilities have a large $T_4 - T_3$ because they belong to some OpenBSD modules (e.g., httpd) that have to be fixed by third-party vendors. For Firefox vulnerabilities, we observe very few instances with $T_4 - T_3 > 0$, meaning that patches are almost always available when vulnerabilities are disclosed.

Insight 7. Firefox's vulnerability density is almost one order of magnitude higher than OpenBSD's.

4 Limitations

First, the *completeness* of the vulnerability datasets we collected could be improved because we extracted vulnerabilities from NVD, BID, and vendor advisories. This is relevant because not all OpenBSD vulnerabilities are widely reported (especially so for the ones that are not confirmed to be exploitable [45]).

Second, the *quality* of the vulnerability datasets may need to be improved because NVD and BID may contain some inaccurate information. This can be seen from the following examples. (i) OpenBSD patch "40-015_file" fixes a heap overflow vulnerability, which can be mapped to CVE-2007-1536, BID #23021 and BID #24146, but BID #24146 also contains CVE-2007-2799. This causes the aggregation of "(40-015_file, CVE-2007-1536, CVE-2007-2799, 23021, 24146)" as a vulnerability, even though OpenBSD may *not* suffer from CVE-2007-2799. (ii) We encountered 136 OpenBSD vulnerabilities that do *not* correspond to any CVE identifier; for Firefox, we encountered 175 CVE identifiers which correspond to multiple vulnerabilities each.

Third, the *accuracy* of the estimated vulnerability introduction time T_0 may not be perfect because of the following. (i) The modified portion of code in a patch may contain some code that is not related to the vulnerability. This can be improved by precisely locating the vulnerable lines of code. (ii) Using the `annotate` command to determine the vulnerability introduction time T_0 may be overly conservative because changes in syntax does not necessarily change the semantics (e.g., dividing a line of code into two). This explains why we considered two extreme choices, *conservative* vs. *radical* estimation, to alleviate this problem. (iii) Neither the conservative estimation nor the radical estimation of T_0 is perfect. For example, OpenBSD patch "42-012_xorg2" fixed five vulnerabilities (i.e., CVE-2008-2360, CVE-2008-2361, CVE-2008-2362, CVE-2008-1379, and CVE-2008-1377). As a consequence, this patch leads to the same T_0 assigned to these five vulnerabilities. Unfortunately, there is not enough information available to distinguish these vulnerabilities from each other, explaining why we cannot offer more accurate estimation of their respective T_0's.

Fourth, the *fairness* in comparing OpenBSD and Firefox may not be perfect. This is because Firefox is more widely used than OpenBSD, meaning that attackers and researchers may be more interested in investigating and searching for vulnerabilities in Firefox. This may have introduced a bias in the number of vulnerabilities reported, and consequently, exploited and patched in these two software systems.

Fifth, the *timespan* of software systems also makes it hard to fairly compare two systems. This is because both the knowledge of software developers and the vulnerability discovery techniques evolve with time and are difficult to quantify. For example, if we compare OpenBSD and Firefox with respect to the same period of time (say 2011-2020), the comparison may not be fair because OpenBSD has evolved for 13 years (or 13 years old since its introduction in 1998) but Firefox is first released in 2011 (as the first stable code base). If we compare Firefox and OpenBSD during the same age (say, 1998-2007 for OpenBSD and 2011-2020 for Firefox), the comparison may not be fair because programmers and programming techniques are quite different during these two periods of time. Eliminating such biases is an outstanding open problem for future studies because the result would be widely applicable.

Sixth, the *subjectivity* of our case study. Our case study focuses on OpenBSD and Firefox, which is somewhat subjective. Nevertheless, this subjectivity on our case study may not cause substantial damage, because of the following: (i) the methodology we proposed is equally applicable to other software systems than OpenBSD and Firefox; (ii) the choice of OpenBSD is to achieve backward compatibility in the sense that Ozment and Schecter [15] analyzed the vulnerabilities of OpenBSD about 15 years ago; (iii) we additionally analyze Firefox because it is widely used, it is a large and complex software, and its source code is available. This means that analyzing OpenBSD and Firefox would suffice to show the usefulness of the methodology.

5 Conclusion

We have presented an empirical study on the evolution of OpenBSD and Firefox vulnerabilities, respectively over the span of 2009-2020 and 2011-2020. To the best of our knowledge, this is the most systematic study about the vulnerabilities in these two important software systems. We have collected and processed the most comprehensive dataset on OpenBSD and Firefox vulnerabilities, and have made the dataset publicly available. Our empirical analysis has led to a number of insights, which shed light on the security of OpenBSD and Firefox. The aforementioned limitations of the present study offer interesting future research directions.

Acknowledgment. We thank the reviewers for their insightful comments and suggestions, which have guided us in improving the paper. The authors affiliated with HUST were supported by the Key Program of the National Science Foundation of China under Grant No. U1936211. Shouhuai Xu was supported in part by the National Science Foundation under Grants #2122631 and #2115134, Army Research Office Grant #W911NF-17-1-0566, and Colorado State Bill 18-086.

References

[1] D. Gens, S. Schmitt, L. Davi, and A. Sadeghi, "K-miner: Uncovering memory corruption in linux," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[2] M. Pradel, V. Murali, R. Qian, M. Machalica, E. Meijer, and S. Chandra, "Scaffle: Bug localization on millions of files," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2020.

[3] A. Habib and M. Pradel, "How many of all bugs do we find? a study of static bug detectors," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 317–328.

[4] M. Pradel and K. Sen, "Deepbugs: a learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, 2018.

[5] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Proc. IEEE Symp. Secur. Priv.* IEEE, 2017, pp. 595–614.

[6] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *Proc. IEEE Symp. Secur. Priv.* IEEE, 2012, pp. 48–62.

[7] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, 2016, pp. 201–213.

[8] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[9] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 5, pp. 2224–2236, 2021.

[10] S. Liu, G. Lin, L. Qu, J. Zhang, O. De Vel, P. Montague, and Y. Xiang, "Cd-vuld: Cross-domain vulnerability discovery based on deep domain adaptation," *IEEE Trans. Dependable Secure Comput.*, pp. 1–1, 2020.

[11] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proc. IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.

[12] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Dependable Secur. Comput.*, 2021.

[13] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: A deep learning-based fine-grained vulnerability detector," *IEEE Trans. Dependable Secur. Comput.*, 2021.

[14] D. Zou, Y. Zhu, S. Xu, Z. Li, H. Jin, and H. Ye, "Interpreting deep learning-based vulnerability detector predictions based on heuristic searching," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 23:1–23:31, 2021.

[15] A. Ozment and S. E. Schechter, "Milk or wine: Does software security improve with age?" in *Proc. 15th USENIX Conf. Secur. Symp.*, 2006.

[16] Mozilla, "Bugzilla main page," 2020. [Online]. Available: https://bugzilla.mozilla.org

[17] P. Behnamghader, R. Alfayez, K. Srisopha, and B. W. Boehm, "Towards better understanding of software quality evolution through commit-impact analysis," in *Proc. IEEE Int. Conf. Softw. Quality, Rel. Secur.*, 2017, pp. 251–262.

[18] S. Clark, M. Collis, M. Blaze, and J. M. Smith, "Moving targets: Security and rapid-release in firefox," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1256–1266.

[19] C. Vassallo, F. Palomba, A. Bacchelli, and H. C. Gall, "Continuous code quality: are we (really) doing that?" in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 790–795.

[20] M. D'Ambros, H. C. Gall, M. Lanza, and M. Pinzger, "Analysing software repositories to understand software evolution," in *Software Evolution*, 2008, pp. 37–67.

[21] A. Ganpati, A. Kalia, and H. Singh, "A comparative study of maintainability index of open source software," *Int. J. Emerg. Technol. Adv. Eng*, vol. 2, no. 10, pp. 228–230, 2012.

[22] Q. Feng, Y. Cai, R. Kazman, D. Cui, T. Liu, and H. Fang, "Active hotspot: An issue-oriented model to monitor software evolution and degradation," in *Proc. 34th ACM/IEEE Int. Conf. Autom. Softw. Eng.* IEEE, 2019, pp. 986–997.

[23] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 298–308.

[24] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empir. Softw. Eng.*, vol. 19, no. 6, pp. 1665–1705, 2014.

[25] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2201–2215.

[26] M. Jimenez, M. Papadakis, and Y. Le Traon, "An empirical analysis of vulnerabilities in openssl and the linux kernel," in *Proc. 23rd Asia-Pacific Softw. Eng. Conf.*, 2016, pp. 105–112.

[27] Y. Wu, H. Siy, and R. Gandhi, "Empirical results on the study of software vulnerabilities," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 964–967.

[28] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Software Eng.*, vol. 37, no. 6, pp. 772–787, 2010.

[29] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006, pp. 25–33.

[30] Q. Wu, Y. He, S. McCamant, and K. Lu, "Precisely characterizing security impact in a flood of patches via symbolic rule comparison," in *Proc. 27th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2020.

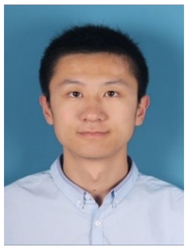
[31] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *Proc. 27th USENIX Conf. Secur. Symp.*, 2018, pp. 919–936.

[32] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *Proc. 28th USENIX Conf. Secur. Symp.*, 2019, pp. 869–885.

[33] Y. Shen and G. Stringhini, "Attack2vec: Leveraging temporal word embeddings to understand the evolution of cyberattacks," in *Proc. 28th USENIX Conf. Secur. Symp.*, 2019, pp. 905–921.

[34] X. Feng, X. Liao, X. Wang, H. Wang, Q. Li, K. Yang, H. Zhu, and L. Sun, "Understanding and securing device vulnerabilities through automated bug report analysis," in *Proc. 28th USENIX Conf. Secur. Symp.*, 2019, pp. 887–903.

[35] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 771–781.
[36] ——, "Large scale characterization of software vulnerability life cycles," *IEEE Trans. Dependable Secure Comput.*, vol. 17, no. 4, pp. 730–744, 2020.
[37] L. Bilge and T. Dumitras, "Before we knew it: an empirical study of zero-day attacks in the real world," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2012, pp. 833–844.
[38] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proc. IEEE Symp. Secur. Priv.* IEEE, 2019, pp. 754–768.
[39] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *Proc. IEEE Symp. Secur. Priv.* IEEE, 2019, pp. 769–786.
[40] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *Proc. IEEE Symp. Secur. Priv.* IEEE, 2018, pp. 679–696.
[41] "Common vulnerability scoring system," 2019. [Online]. Available: https://www.first.org/cvss/v2/guide
[42] "Common weakness enumeration," 2019. [Online]. Available: https://cwe.mitre.org/about/
[43] "National vulnerability database," 2020. [Online]. Available: https://nvd.nist.gov/
[44] "Securityfocus," 2020. [Online]. Available: https://www.securityfocus.com
[45] "Openbsd security," 2019. [Online]. Available: http://www.openbsd.org/security.html
[46] "Mozilla foundation security advisories," 2019. [Online]. Available: https://www.mozilla.org/en-US/security/advisories/
[47] "Ubuntu security notices," 2020. [Online]. Available: https://usn.ubuntu.com/
[48] "Exploit database - exploits for penetration testers, researchers, and ethical hackers," 2020. [Online]. Available: https://www.exploit-db.com
[49] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation." in *Proc. 12th USENIX Conf. Secur. Symp.*, 2003.
[50] L. Teo, "Writing exploit-resistant code with openbsd," https://lteo.net/assets/pdf/lteo-openbsd-carolinacon15-20190427.pdf, CarolinaCon, 2019.
[51] T. Mortimer, "Removing rop gadgets from openbsd," *Proc. AsiaBSDCon*, pp. 13–21, 2019.
[52] T. de Raadt, "Deterministic behaviours are your attacker's friend," http://www.openbsd.org/papers/cuug2019-predictable.pdf, Calgary Unix Users Group, 2019.
[53] ——, "Advances in openbsd," http://www.openbsd.org/papers/csw03/index.html, CanSecWest, 2003.
[54] ——, "Mitigations and other real security features," https://www.openbsd.org/papers/bsdtw.pdf, BSD Taiwan, 2017.
[55] A. Lindqvist, "Fuzzing the openbsd kernel," http://www.openbsd.org/papers/fuzz-slides.pdf, BSD Users Stockholm Meetup, 2018.



Jian Shi received the B.E. degree in computer science and technology from China University of Petroleum (UPC), Qingdao, China, in 2015, and is currently pursuing the Ph.D. degree in School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, China. His research interests mainly include vulnerability detection and software security.

Deqing Zou received the Ph.D. degree at Huazhong University of Science and Technology (HUST), in 2004. He is currently a professor of School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, China. His main research interests include system security, trusted computing, virtualization and cloud security. He has always served as a reviewer for several prestigious journals, such as IEEE TDSC, IEEE TOC, IEEE TPDS, and IEEE TCC. He is on the editorial boards of four international journals, and has served as PC chair/PC member of more than 40 international conferences.

Shouhuai Xu (M'14–SM'20) received the Ph.D. degree in computer science from Fudan University in 2000. He is the Gallogly Chair Professor in the Department of Computer Science, University of Colorado Colorado Springs (UCCS). Prior to joining UCCS, he has been with University of Texas at San Antonio. He pioneered the Cybersecurity Dynamics approach as foundation for the emerging science of cybersecurity, with three pillars: first-principle cybersecurity modeling and analysis (the x-axis); cybersecurity data analytics (the y-axis, to which the present paper belongs); and cybersecurity metrics (the z-axis). He co-initiated the International Conference on Science of Cyber Security and is serving as its Steering Committee Chair. He is/was an Associate Editor of IEEE Transactions on Dependable and Secure Computing (IEEE TDSC), IEEE Transactions on Information Forensics and Security (IEEE T-IFS), and IEEE Transactions on Network Science and Engineering (IEEE TNSE).

Xianjun Deng received the B.S. and M.S. degrees in communications engineering from Chongqing University of Posts and Telecommunications (CQUPT), China, in 2005 and 2008, respectively, and the Ph.D. degree in the School of Electronic Information and Communications, Huazhong University of Science and Technology (HUST), Wuhan, China, in 2014. He is currently a professor with School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, China. He obtained the IEEE TCSC Award for Excellence for Early Career Researcher on December, 2019. He is Hibiscus Scholar of Hunan Province, China, and the outstanding Young Instructor of Hunan Province, China. His research interests focus on coverage optimization, network reliability, security and privacy algorithms in wireless sensor networks and Internet of Things. He has published more than 50 technical papers in international journals and conferences.

Hai Jin received the Ph.D. degree in computer engineering from Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994. He is a Cheung Kung Scholars Chair Professor of computer science and engineering at HUST in China. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of the IEEE, a fellow of the CCF, and a member of the ACM. He has co-authored 22 books and published over 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.