CASFinder: Detecting Common Attack Surface

Mengyuan Zhang[1], Yue Xin[1], Lingyu Wang[1](✉), Sushil Jajodia[2], and Anoop Singhal[3]

[1] Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada
wang@ciise.concordia.ca

[2] Center for Secure Information Systems, George Mason University, Fairfax, USA
jajodia@gmu.edu

[3] Computer Security Division, National Institute of Standards and Technology, Gaithersburg, USA
anoop.singhal@nist.gov

Abstract. Code reusing is a common practice in software development due to its various benefits. Such a practice, however, may also cause large scale security issues since one vulnerability may appear in many different software due to cloned code fragments. The well known concept of relying on software diversity for security may also be compromised since seemingly different software may in fact share vulnerable code fragments. Although there exist efforts on detecting cloned code fragments, there lack solutions for formally characterizing their specific impact on security. In this paper, we revisit the concept of software diversity from a security viewpoint. Specifically, we define the novel concept of *common attack surface* to model the relative degree to which a pair of software may be sharing potentially vulnerable code fragments. To implement the concept, we develop an automated tool, *CASFinder*, in order to efficiently identify common attack surface between any given pair of software with minimum human intervention. Finally, we conduct experiments by applying our tool to real world open source software applications. Our results demonstrate many seemingly unrelated software applications indeed share significant common attack surface.

1 Introduction

Code reusing is a common practice in today's software industry due to the fact that it may significantly accelerate the development process [7,10]. However, such a practice also has the potential of leading to large scale security issues because a vulnerability may be shared by many different software applications due to the shared libraries or code fragments. A well known example is the *Heartbleed* vulnerability in *OpenSSL*, which caused widespread panic on the internet since the vulnerable library was shared by many popular Web servers, including Apache and Nginx [11]. In addition to shared libraries, the reusing of existing

© IFIP International Federation for Information Processing 2019
Published by Springer Nature Switzerland AG 2019
S. N. Foley (Ed.): DBSec 2019, LNCS 11559, pp. 338–358, 2019.
https://doi.org/10.1007/978-3-030-22479-0_18

code fragments may also lead to similar vulnerabilities shared by different software applications. Unlike libraries, such reused codes are typically not traced by any official documentation, which makes it more difficult to understand their security impact. Finally, this phenomenon may also compromise the well known concept of relying on software diversity for security, since seemingly unrelated software applications made by different vendors may in fact share common weaknesses.

The issue of identifying and characterizing the security impact of shared code fragments has received little attention (a more detailed review of the related work will be given in Sect. 6). Most existing vulnerability detection tools focus on identifying vulnerabilities for a specific software application based on static and/or dynamic analysis, with no indication whether different software may be sharing similar vulnerabilities due to common libraries or reused codes [12]. On the other hand, existing efforts on software clone detection mostly focus on identifying reused code fragments based on either the textual similarity or functional similarity, with no indication of the security impact [41]. Clearly, there exists a gap between the two, i.e., *how can we leverage existing efforts on software clone detection to characterize the likelihood that given software applications may share similar vulnerabilities?*

In this paper, we address the above issue through defining the novel concept of *common attack surface* and developing an automated tool, *CASFinder*, to calculate the common attack surface of given software applications. Specifically, we first extend the well-known attack surface concept to model the relative degree to which a pair of software may be sharing potentially vulnerable code fragments. Such a formal model enables the quantification of software diversity from the security point of view, and its results may be used as inputs to higher level diversity methods (e.g., network diversity [47] and moving target defense [20]). Second, we develop *CASFinder* which is an automated tool that takes the source code of two software applications as the input and outputs their common attack surface result in an XML file or to a database. Third, we conduct experiments by applying our tool to a large number of real-world open source software applications belonging to seven different categories from *Github*. More than 80,000 combinations of software applications are analyzed, and our results demonstrate many seemingly unrelated software applications indeed share a significant level of common attack surface. In summary, the contribution of this paper is threefold.

- First, to the best of our knowledge, this is the first effort on formally modeling the security impact of reused code fragments. The common attack surface model may serve as a foundation and provide quantitative inputs to higher level security-through-diversity methods.
- Second, the *CASFinder* tool makes it feasible to evaluate the common attack surface between open source software applications, which may have many practical use cases, e.g., providing useful references for security practitioners to choose the right combinations of software applications in order to maximize the overall software diversity in their networks, and reusing the knowledge

about existing vulnerabilities in one software to potentially identify similar ones in other software.
– Third, our experimental results prove the possibility of similar vulnerabilities shared by seemingly unrelated software applications made by different vendors. We believe such a finding may help attract more interest to re-examining the concept of software diversity and its security implication.

The remainder of this paper is organized as follows. Section 2 provides a motivating example and background information. Section 3 defines the common attack surface model. Section 4 designs and implements the *CASFinder* tool. Section 5 evaluates the tool through experiments using real open source software. Section 6 reviews related work and Sect. 7 concludes the paper and provides future directions.

2 Preliminaries

In this section, we first present a motivating example in Sect. 2.1 and then provide background knowledge and highlight the challenge in Sect. 2.2.

2.1 Motivating Example

As an example, consider an enterprise network with Web servers running either the Apache HTTP server (*Apache*) or the Nginx HTTP server (*Nginx*), as well as a Cyrus IMAP server (*Cyrus*). Assume all three software applications are of the vulnerable versions that are affected by the *Heartbleed* vulnerability. This vulnerability has reportedly affected an estimated 24–55% of popular websites and gave attackers accesses to sensitive memory blocks on the affected servers, which potentially contain encryption keys, usernames, passwords, etc. [11]. The vulnerability is discovered inside the popular *OpenSSL* library, which is an extension of many Web and email server software applications for supporting the *https* connections.

Specifically, Fig. 1 demonstrates how this vulnerability functions in relation to the three software applications in our example. Those software simply hand the encryption tasks to the *OpenSSL* extension, and the vulnerability appears when the software make external calls to the *OpenSSL* extension. In establishing the SSL connections, the API invocation *SSL_CTX_new(method)* is a function for establishing SSL content, *SSL_new()* is for creating SSL sessions, and *SSL_connect()* for launching SSL handshakes. To exploit the *Heartbleed* vulnerability, attackers would craft a heartbeat request with a special length and send it to the servers. This request would cause different software applications to invoke the same library function *memcpy()* without any boundary check enabling attackers to extract sensitive memory blocks from the servers.

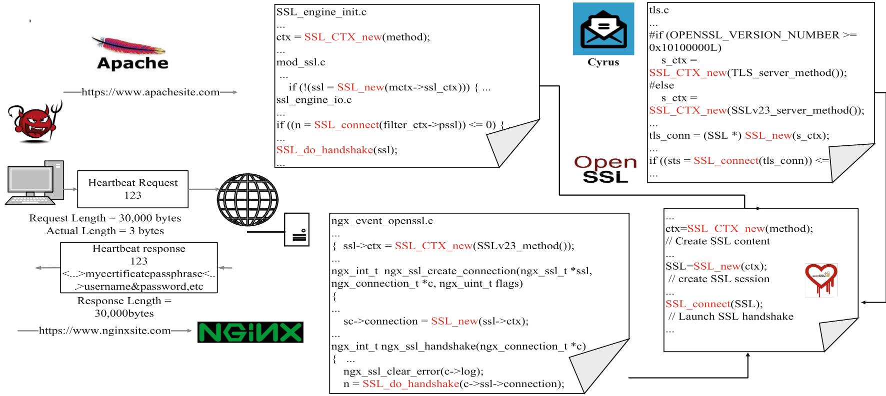

Fig. 1. An example of the *Heartbleed* vulnerability

The fact that this vulnerability exists inside the *OpenSSL* extension shared by all three software means an attacker can compromise those different software in a similar manner. This phenomenon is certainly not limited to this particular vulnerability. In this example, since both the *Apache* and *Nginx* projects are Web servers developed in C language, their similar functionality implies there is a high chance that the developers of both projects would not only import the same libraries, but also reuse the same or similar code fragments. In addition, as will be shown through our experimental results, code reusing also exists among software applications with very different functionalities. On the other hand, not all server software that use SSL connections are affected by this vulnerability, e.g., *Microsoft IIS* and *Jetty* are both immune to the vulnerability [11].

Clearly, there exists a need for identifying the software applications which may share such a common vulnerability, and for characterizing the level of such sharing since some software may share more than one such vulnerability. Such a desirable capability may have many practical use cases. For instance, it may allow similar software patches or fixes to be developed and applied to different software applications in order to mitigate a common vulnerability, which may significantly reduce the time and effort needed for developing such patches and fixes. This capability may also allow administrators to better judge the amount of software diversity in their networks, and to choose the right combinations of software applications (e.g., Apache and IIS w.r.t. this particular vulnerability) to increase the diversity. Finally, this capability would lead to a more refined approach to moving target defense (MTD) [13] since it could potentially allow us to quantify the amount of software diversity that is achieved by switching between different software resources under a MTD mechanism.

2.2 Background

We take two steps towards measuring the potential impact of cloned codes on security. The first step is to find similar code fragments in different software

applications. The second step is to characterize the security impact of such code fragments. We first review some of the background concepts related to each step.

First, to detect similar code fragments between software, most clone detection methods are based on either the textual similarity or the functional similarity, and existing tools are mostly based on text, token, tree, graph, or metrics [41,42]. Among the existing tools, we have chosen *CCFinder* [23], a language-based source code clone detection tool, to find cloned code fragments within given software applications. As one of the leading token-based detection tools, *CCFinder* has received the Clone Award in 2002, and it supports multiple languages, including C, C++, Java, and COBOL. *CCFinder* first divides the given source code into tokens using a lexical analyzer. It then normalizes some of those tokens by replacing identifiers, constants and other basic tokens with generic tokens representing their language role. Finally, it uses a suffix-tree based sub-string matching algorithm to find common subsequences corresponding to clone pairs and classes [23]. A key advantage of such a token-based tool is that it can tolerate minor code changes, such as formatting, spacing and renaming, in the reused code.

However, the result from clone detection tools, including *CCFinder*, only reveals similar code fragments between source codes, without indicating any security impact. The primary challenge is therefore to model and quantify the potential impact of clone detection on security in terms of leading to potential vulnerabilities. To this end, a promising solution is to apply the attack surface concept [36], which is a well known software security metric that measures the degree of software security exposure. The measurement is taken as counts along three dimensions, the entry and exit points (i.e., methods calling I/O functions), channels (e.g., TCP and UDP), and untrusted data items (e.g., registry entries or configuration files), and the counting results are then aggregated through weighted summation. Attack surface measures the intrinsic properties of a software application, e.g., how many times does each method invoke I/O functions (which provides an estimate of security risks such as buffer overflow), regardless of external factors such as the discovery of the vulnerability or the existence of exploit code. Therefore, attack surface can potentially cover both known and unknown vulnerabilities.

Therefore, we will combine clone detection (i.e., *CCFinder*) with attack surface to quantify the likelihood that cloned code fragments may lead to potentially similar vulnerabilities shared between different software applications. For simplicity, we will focus on entry and exit points in this paper, and will consider channels and untrusted data items in our future work. We also note that, since it is not guaranteed that every entry or exit point will map to a vulnerability, the attack surface concept is only intended as an estimation of the relative abundance of vulnerabilities in software [36]. Consequently, our model and tool also inherit this limitation, and the results will only indicate the potential, instead of the actual existence, of common vulnerabilities.

Combining the result of clone detection with the attack surface concept is not a straightforward task. We discuss a key challenge in the following. In Fig. 2,

function *handle_response()* and function *quicksand_mime()* are both entry points since they call I/O functions *fseek()* and *ftell* (from the standard C library). A naive application of the attack surface concept here would indicate each function count as one entry point and hence both have the same security implication. However, such a coarse-grained application ignores the exact number of I/O function calls (i.e., three calls in *handle_response()* and two in *quicksand_mime()*) whose difference may be significant in practice. In our model, we will take a more refined approach to address such issues.

```
1 fseek(fp, 0, SEEK_END);
2 size = ftell(fp);
3 fseek(fp, 0, SEEK_SET);
4 snprintf(fsize, 32, "Content-Length: %d\r\n\r\n", size);
```

```
1 long fsize = ftell(f);
2 fseek(f, 0, SEEK_SET);
3 free(decoded_mime);
```

Fig. 2. Examples of entry points: /Simple-Webserverche/server.c *handle_response()* (Top) and /quicksand_lite/libqs.c *quicksand_mime()* (Bottom)

3 The Model of Common Attack Surface

In this section, we model the security implication of cloned code fragments between software applications through two novel security metrics, namely, the *conditional common attack surface* (ccas) and the *probabilistic common attack surface* (pcas). Those two metrics are designed for different use cases as follows.

- The *conditional common attack surface* (ccas) is designed to be asymmetric for use cases in which one software is of particular interest and evaluated against all other software. For example, suppose a company has developed a new Web server application and wants to understand any similarity between their product and other existing Web servers such as *Apache* and *Nginx*. In such a case, the key is to rank those other software applications based on the relative percentage of shared attack surface, and the developer can apply the metric *ccas* for this purpose.
- Second, in a different scenario, suppose an administrator wants to understand the level of software diversity between all the software applications inside the same network. In such a case, both software in comparison are considered equally important, so the symmetric metric *pcas* would be more suitable, which will yield a unique measurement of shared attack surface between any pair of software. The following details the *ccas* and *pcas* metrics.

3.1 Conditional Common Attack Surface (CCAS) Metric

We first consider clone segments between two software applications identified using *CCFinder* [23] through an example.

Example 1. Figure 3 demonstrates clone segments between a Web server application *SimpleWebserver* and an ssh application *SSHBen*. In the figure, the *Clone id* is a unique number labelling a group of related clones inside both software applications. For instance, the code segments inside the solid line blocks indicate the clone segments with the same Clone id 28, and the dashed line blocks are for Clone id 78. Note that the same code may appear under different clone ids, e.g., line 146 and 147 in *Simple-Webserver* appear under both clone ids. Also note that, for Clone id 78, the matching between the two clone segments is *inexact* [23] since *strcat* does not exist in *SSHBen*.

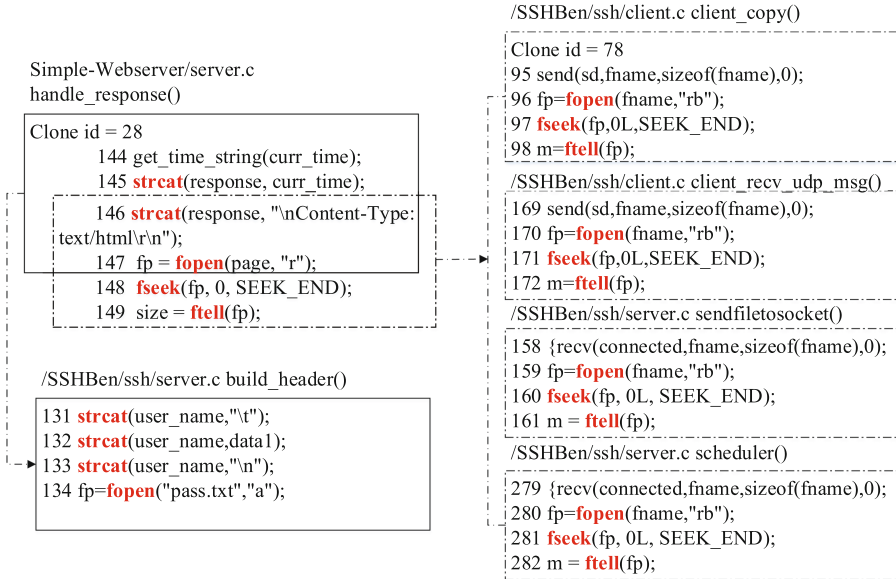

Fig. 3. An example of cloned segments

From the above example, it is clear that the clone segments belonging to the same Clone id are not identical between the two software applications. Therefore, the attack surface would be asymmetric as well. First, we define the *Common Attack Surface* as the collection of I/O function calls inside the clone segments as follows.

Definition 1 (Common Attack Surface). *Given two software applications A and B, the common attack surface of A w.r.t. B (or that of B w.r.t. A) under*

the Clone id i is defined as the multi-set (which preserves duplicates) of I/O function calls that exist inside the clone segments of A under the Clone id i, denoted as $cas_i(A|B)$ *(or* $cas_i(B|A)$*).*

Example 2. To follow our example, we have

- $\text{cas}_{28}(\text{SimpleWebserver}|\text{SSHBen}) = \langle strcat, strcat, fopen \rangle$,
- $\text{cas}_{28}(\text{SSHBen}|\text{SimpleWebserver}) = \langle strcat, strcat, strcat, fopen \rangle$,
- $\text{cas}_{78}(\text{SimpleWebserver}|\text{SSHBen}) = \langle fopen, fseek, ftell \rangle$, and
- $\text{cas}_{78}(\text{SSHBen}|\text{SimpleWebserver}) = \langle fopen, fseek, ftell, fopen, fseek, ftell, fopen, fseek, ftell, fopen, fseek, ftell \rangle$.

Since the attack surface concept is based on the number of entry and exit points (i.e., methods invoking I/O functions), we follow the similar approach to calculate the size of common attack surface by counting the number of I/O function calls across different Clone ids, with those appearing under different Clone ids counted only once. We demonstrate this through an example.

Example 3. For Clone id 78, this gives three for *Simple-Webserver* and 12 for *SSHBen*. As to Clone id 28, we have three for *Simple-Webserver* and four for *SSHBen*. Note that *fopen* is considered under both Clone ids for *Simple-Webserver*, and hence we should count it only once. Based on those discussions, we can calculate the total number of I/O function calls for both Clone ids as five for *Simple-Webserver* and 16 for *SSHBen*.

Finally, we define the *Conditional Common Attack Surface* as the ratio between the size of the common attack surface of a software application (w.r.t. to another software) and the size of its entire attack surface (i.e., the total number of I/O function calls inside that software). This ratio indicates the degree to which the software shares with others similar I/O function calls (entry/exit points).

Definition 2 (Conditional Common Attack Surface). *Given two software applications A and B with totally n clone segments, and* AS_A *and* AS_B *as the total number of I/O function calls inside A and B, respectively, the conditional common attack surface of A w.r.t B (or that of B w.r.t. A), denoted as* $ccas(A|B)$ *(or* $ccas(B|A)$*), is defined as:*

$$ccas(A \mid B) = \frac{\mid \bigcup_{i=1}^{n} cas(A \mid B) \mid}{AS_A}$$

$$ccas(B \mid A) = \frac{\mid \bigcup_{i=1}^{n} cas(B \mid A) \mid}{AS_B}$$

Example 4. The attack surface (i.e., the total number of I/O function calls) of *Simple-Webserver* and *SSHBen* are 16 and 182, respectively. We thus have $ccas(SSHBen \mid SimpleWebserver) = \frac{5}{16} = 0.3125$ and $ccas(Simple\text{-}Webserver \mid SSHBen) = \frac{16}{182} = 0.029$. The results show that *SSHBen* contains about 31% shared attack surface, whereas *SimpleWebserver* contains only

2.9%. By comparing a software application to many others, the developer of that application may gain useful insights from such results in terms of vulnerability discovery and security patch management.

3.2 Probabilistic Common Attack Surface Metric

The conditional common attack surface metric *ccas* is designed for evaluating one software application against others. We now take a different approach of defining a symmetric probabilistic common attack surface metric for two software applications. Such a metric can be used to estimate the amount of effort that a potential attacker may reuse while attempting to compromise both software applications. The nature of such a use case implies the metric should be symmetric.

We apply Jaccard index for this purpose, which is commonly defined as $J(A,B) = \frac{A \cap B}{A \cup B}$ and used for analyzing the similarity and diversity between the two sets. To apply this metric in our case, we need to define both the intersection and union of the attack surface of two software applications. The common attack surface defined in previous section (Definition 1) can be considered as the intersection, but such a definition is not sufficient here since it is asymmetric in nature. Instead, we will define the intersection between the attack surface of two software applications using the standard multi-set intersection operation [43], which is described below.

Definition 3 (Intersection of Multi-Sets [43]**).** *Given two multi-sets* $A = \langle A, f \rangle$ *(where* f *is the multiplicity function such that for any* $a \in A$*,* $f(a)$ *gives the number of occurrences of* a *in the multiset) and* $B = \langle A, g \rangle$*, then their intersection, denoted as* $A \cap B$*, is the multi-set* $\langle A, s \rangle$*, where for all* $a \in A$*:*

$$s(a) = min(f(a), g(a)).$$

Example 5. Assume U ={a,a,a,b} and V = {a,a,b,b}, if we apply the multi-set operation as defined above, we have $U \cap V$ = {a,a,b}.

The union of the attack surface between two software applications can be defined as $AS_A \cup AS_B = AS_A + AS_B - cas(B \mid A) \cap cas(A \mid B)$. With both the union and intersection operations defined, we can now define the probabilistic common attack surface metric as follows.

Definition 4 (Probabilistic Common Attack Surface Metric). *Given two software applications* A *and* B*, with their attack surface* AS_A *and* AS_B *and the common attack surface* $cas(B|A)$ *and* $cas(A|B)$*, respectively, the probabilistic common attack surface of* A *and* B *is defined as:*

$$pcas(A.B) = \frac{\mid cas(B \mid A) \cap cas(A \mid B) \mid}{\mid AS_A \cup AS_B \mid}$$

Example 6. The size of attack surface in *Simple-Webserver* and *SSHBen* is 16 and 182, respectively. From our previous discussions, we have $cas(SSHBen \mid SimpleWebserver) \cap cas(SimpleWebserver \mid SSHBen) = \langle strcat, strcat, fopen, fseek, ftell \rangle$ whose size is 5, and hence $pcas(SSHBen.SimpleWebserver) = \frac{5}{16+182-5} = 2.6\%$. Intuitively, this result indicates that, among all the I/O function calls, about 2.6% are shared between the two software applications. Such a result, when applied to all pairs of software applications inside a network, may allow administrators to estimate the degree of software diversity in the network from a security point of view.

4 Design and Implementation

To automate the evaluation of common attack surface between software applications, we design and implement a tool, *CASFinder*. Figure 4 depicts the architecture of *CASFinder*, which consists of three main components, the clone detection module, the source code labeling module, and the visualization module. The following describes those modules in more details.

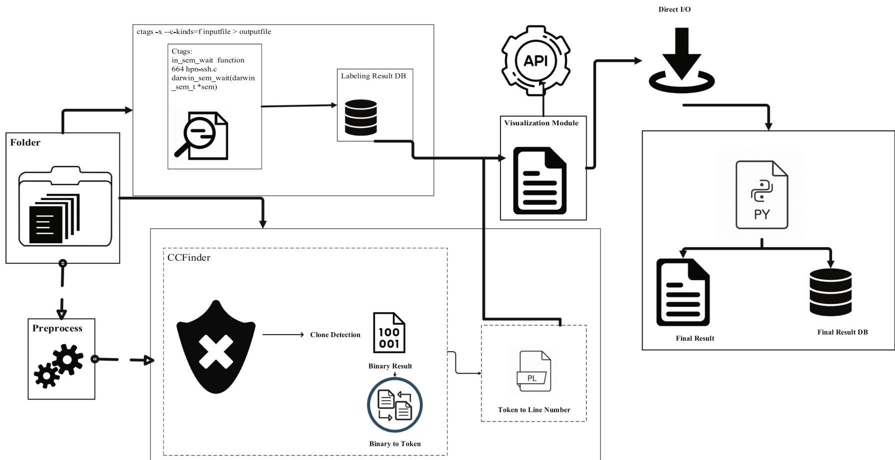

Fig. 4. The architecture

– *The Clone Detection Module.* As mentioned earlier, we choose *CCFinder* [23] as the basis of our clone detection module. The following details challenges and solutions for applying *CCFinder*. First, since our tool is developed and operated under Linux, we apply only the back end of *CCFinder*. One challenge is that, since the default Linux version of *CCFinder* is designed to work on *Ubuntu 9*, the newer versions of many libraries are no longer valid for *CCFinder*. Therefore, several libraries need to be installed separately,

e.g., *libboost-dev* and *libicu-dev*, which will depend on the specific version of the Linux system and can be determined based on the warnings and errors produced by *CCFinder*. Second, various parameters can be fine tuned in *CCFinder* to customize its execution mode [22]. In particular, the most important parameters include b, the minimum length of the detected code clones, and t, the minimum number of types of tokens involved. We have chosen $b = 20$ and $t = 8$ based on experiences obtained through extensive experiments. In addition, parameter w is used to determine whether *CCFinder* will perform inner-file clone detection whose results contain clones between different parts of the same software application, which is not our focus, and therefore w is set to be *f-w-g+* to focus on inter-file clones. Finally, the default output of the *CCFinder* is stored in a binary file with *.ccfd* extension. Since we do not install any front end of *CCFinder*, we apply the command *./$PATH/ccfx -p name.ccfd* to translate the *.ccfd* file into a human-readable version. The resultant file contains only the token information, which cannot be directly mapped back to the source code files. Therefore, we have developed a script, *post-prettyprint.pl* [38], to convert the token information into corresponding line numbers in the source code.

- *The Source Code Labeling Module.* As mentioned above, the converted output of *CCFinder* provides only the file name and line number of the clone segments, without information needed for mapping them back to the original source code. For the purpose of generating traceable output with source code fragments, a mapping between the line number of the clone segments and the source code needs to be established. This second module is designed for this purpose by automatically retrieving a clone code segment from the source code according to the result of *CCFinder*.
- *The Visualization and CAS Calculation Module.* The visualization module generates the results of clone segments. The results include clone ID, file path, function name, clone segment, start line number, and end line number. The visualized output is organized as an *XML* tree with labels. The label *contents* contains the source clone segments from *CCFinder* outputs. Label *funcname* reveals the function names corresponding to the clone segments, and label *io* contains the common I/O functions. To calculate the common attack surface, we first need to identify the I/O functions. In our experiments, we have obtained the list of I/O functions from the GNU C library [40] (glibc), which is the GNU project's implementation of C standard library, as the database for examining the entry/exit points. In total, 256 I/O functions are stored in our database, e.g., function *memcpy* or *strcpy*, which could take user inputs as the source, and copy them directly to the memory block pointed to by the destination. Such functions have caused many serious security flaws including CVE-2014-0160 (i.e., the Heartbleed bug [8]). The final result of common attack surface is calculated based on the I/O functions shared among all software applications, and can be stored either in a file or into the database.

5 Experiments

This section presents experimental results on applying our tool *CASFinder* to real world open source software.

5.1 Dataset

To study the common attack surface among real world software applications, we need a large amount of open-source software to apply our tool. For this purpose, we have developed a script to automatically parse the download links at the open-source software hosts. Our research shows that *GitHub* [15] provides the customized API for users to search open-source software applications with customized requirements and to download them automatically. The results are presented in json code, which contains the download link of each application together with other information. In our experiments, we have set the parameter *language* to C programs, and use parameters *q*, *sort*, and *order* to specify the query conditions and to customize the sequence of results. We have developed the script to parse the json format output from the *GitHub* automatically and to store the information of the software download link, authors, publish time, size, and other descriptions into our local database. All the download links for each software application are stored separately. Since *Github* has a limitation with respect to the maximum requests in a certain amount of time, we design the process to sleep for certain time after each query. Our experimental environment is a virtual machine running Ubuntu 14.04, with the Intel core i3-4150 CPU and 8.0 GB of RAM. We have applied our tool to totally 293 different software applications belonging to seven categories. The software applications belong to several categories as follows: 32 in Databases, 62 in Web servers, 25 in ssh servers, 79 in FTP servers, 41 in TFTP servers, 6 in IMAP servers, and 48 in firewalls. Those amount to totally $\binom{293}{2} = 42,778$ pairs of software applications tested using our tool in the experiments.

5.2 Cross-Category Common Attack Surface

In this section, we apply the two proposed common attack surface metrics to totally 42,778 pairs of real world software. The first set of experiments reveal the existence of common attack surface between different categories of software applications. To convert the results to a comparable scale, we have normalized the absolute value of common attack surface reported by *CASFinder* by the size of the software. Figure 5 shows the existence of common attack surface across seven categories. The percentages on top of the bars inside each figure indicate the level of common attack surface between the category mentioned in the title of the figure and all the seven categories. We can observe that common attack surface exists in all of the category combinations. For example, the *DB* category has the highest level of common attack surface inside its own category (between different software inside that category), 27.9%, and it also shares more than 9% common attack surface with any other category.

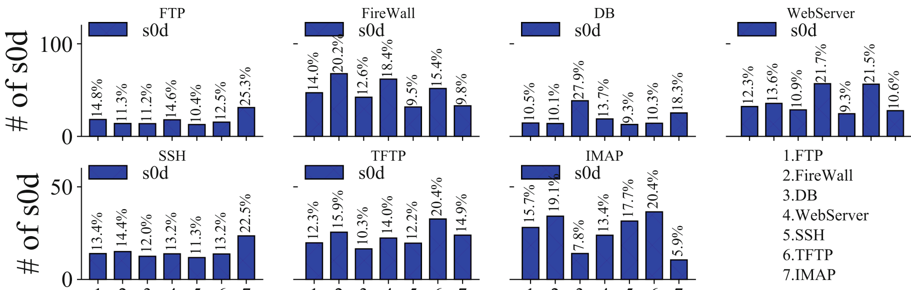

Fig. 5. Common attack surface across categories

In summary, the results across all categories are shown in the heat map in Table 1 where a darker color indicates a larger CAS value between the pair of categories. A visible diagonal with the darkest color in the heat map indicates the expected trend that different software in the same category yield the highest level of common attack surface, most likely due to their similar functionality, except for *SSH*. In fact, the category *SSH* has the lowest level of common attack surface within its category. The reason is that the *SSH* category only contains 25 software applications, which is not sufficiently large to produce any reliable trend. Due to similar reasons, we have omitted the results from the *IMAP* category in the heat-map.

Table 1. HeatMap for common attack surface in different categories

	FTP	FireWall	DB	WebServer	SSH	TFTP
FTP	18.2	13.8	13.7	17.9	12.8	15.3
FireWall	47.1	67.6	42.2	61.8	31.7	51.7
DB	14.4	13.9	38.4	18.8	12.8	14.1
WebServer	32.2	35.6	28.6	56.9	24.4	56.3
SSH	13.9	15.0	12.5	13.8	11.8	13.7
TFTP	19.8	25.5	16.5	22.4	19.6	32.6

After understanding the general existence of common attack surface among the seven categories of software applications, we aim to study more specific trends in our second sets of experiments. The left chart in Fig. 6 shows the accumulated number of pairs of software applications in the absolute value of common attack surface. The figure depicts only the results with a nonzero value, which include totally 9,852 pairs (which amounts to about 1/8 of the total number of pairs). We can observe that the accumulated number of pairs of software applications increases quickly before the value of common attack surface reaches about 12 and afterwards the accumulation flattens out. About 20% of software share common clone segments, and 56% of the clone segments contain at least one common

attack surface. The right chart in Fig. 6 depicts the relationship between common attack surface and sizes of the software. We use the absolute values of common attack surface in this experiment. For the sizes, we use the normalized combined sizes $\log_{1000}(A^B)/1000$ when software A is compared with software B. We can observe that, with increasing sizes of the software, the value of common attack surface generally increases. This is as expected since the number of I/O functions would be roughly proportional to the size of the software.

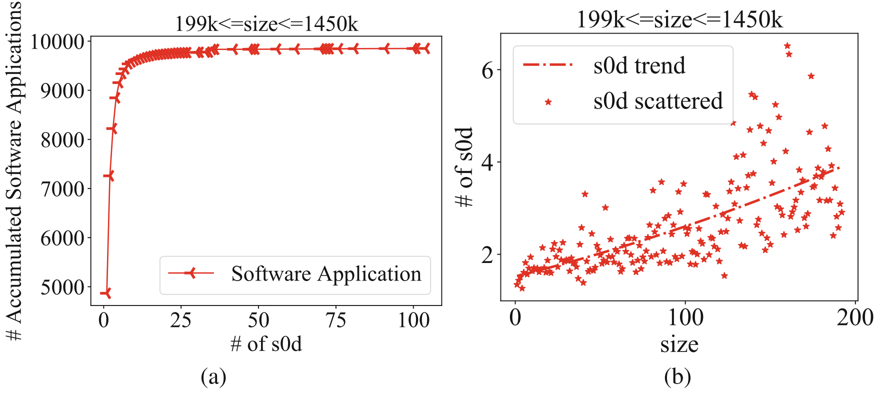

Fig. 6. CAS in accumulated software application pairs (a), CAS trend vs size (b) (Color figure online)

The left chart in Fig. 7 compares the average number of I/O functions and the average common attack surface over several years. The blue bars indicate the average number of I/O functions used in the software applications tested in our experiments based on the publishing year. The average number of I/O functions per software application does not have a simple trend and is used as a baseline for comparison. We can observe a clear downward trend in the average value of common attack surface over time, with software published around 2010 having a much higher value of common attack surface compared with more recent years, regardless of the number of average I/O functions. We believe this trend shows that code reusing plays a major role in common attack surface, since the trend can be easily explained by the backward nature of code reusing (i.e., programmers can only reuse older code). The right chart in Fig. 7 explores the trend of the probabilistic common attack surface metric versus the size. The value of the probabilistic common attack surface metric decreases since the increase of the number of I/O functions in software applications is faster than the increase of common attack surface.

In fact, those results match the results of existing vulnerability discovery models, which generally show that larger software applications typically have more vulnerabilities but a lower probability for having vulnerabilities per unit of software size. For example, Google Chrome (with the number of lines at 14,137,145 [2]) has 1,453 vulnerabilities over nine years [9], while Apache (with

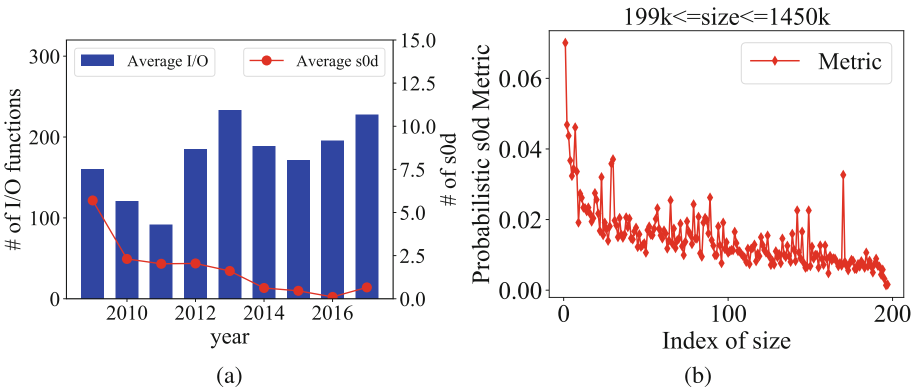

Fig. 7. CAS trend in years (a) and the probabilistic CAS metric (b) (Color figure online)

the number of lines at 1,800,402) has 815 over 19 years. However, the probability of having one vulnerability per unit of software size per year is $1.15 * 10^{-3}\%$ for Chrome and $2.4 * 10^{-3}\%$ for Apache (i.e., the larger Chrome has less vulnerabilities per unit of software size).

5.3 Common Attack Surface in the Same Category

We study the trend of common attack surface between software within the same category in this section. Figure 8 depicts the common attack surface for different sizes of software in the category *WebServer* and *FTP*, respectively, represented in both scattered and trending results. The orange scattered points and the dotted line indicate the result and the red dotted line is the same trend borrowed from Fig. 6 for comparison. We can observe that the trend of common attack surface in both categories increase with the size, which follows a similar trend as the cross category result. However, the trend of *WebServer* increases faster than the cross-category trend, which matches the results shown in Table 1. On the other hand, the trend in the *FTP* category grows slightly slower than the cross category trend, which can be explained by the fact that *FTP* shares a large amount of common attack surface with *WebServer* and *TFTP*.

The left chart in Fig. 9 depicts the trend of common attack surface over time in the same category. Each blue bar represents the average number of I/O functions in the years in the same category of the experiments. The red line shows the average number of common attack surface in those years. Compared to Fig. 7, the common attack surface in the same category has higher values, which also match the previous observations. The right chart in Fig. 9 reveals the trend of the probabilistic common attack surface metric versus the size in the same category, which shows a similar trend as the cross category result, although the trend within the same category starts from a higher value around 0.20 (in contrast, the cross-category metric starts from 0.06).

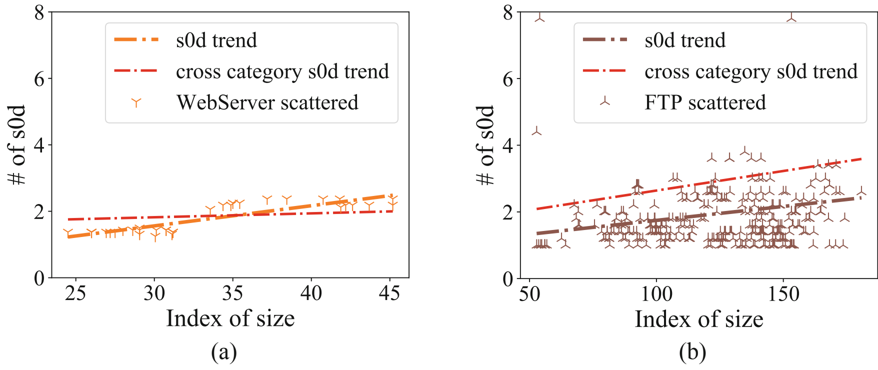

Fig. 8. Size trend in same category, WebServer (a) and FTP (b) (Color figure online)

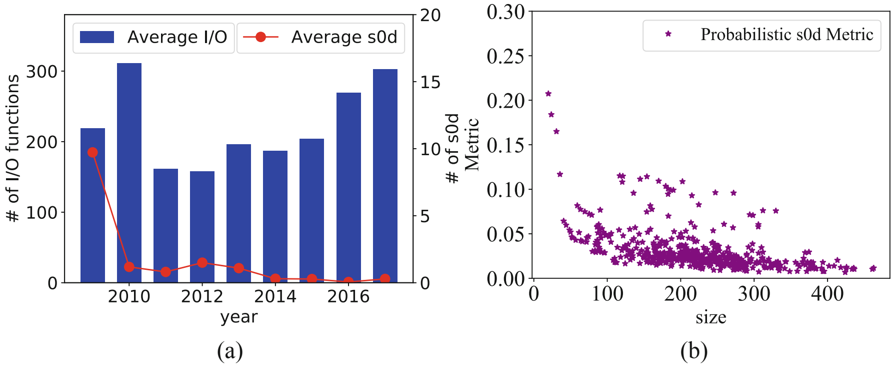

Fig. 9. Common attack surface over time and vs size (Color figure online)

6 Related Work

There exist extensive research on clone code detection although many of these tools are mainly for research purposes [42]. One of the popular tools in text-based clone detection is the *Dup* [3]; if two lines of code are identical after removing all whitespaces and comments, they are assigned as clone codes; the longest line matches are the output, but the minimum length of the reported code can be customized according to different needs. Another well-known approach [21] is applying the fingerprint in order to identify the redundancy on a substring of the source code. The fingerprinting calculation uses KARP-Rabins string matching approach [25,26] to calculate the length of all n substrings. Ducasse developed [10] *duploc* which was designed to be a parsing free, language-independent tool which first reads the source file and sequences of the lines, then removes all comments and whitespace to create a set of condensed lines; afterward, a comparison is made based on the hash result, where scatter-plots indicate the visualization of a cloned result. Token-based clone detection is also one of the

widely applied methods. One of the representative tools in token-based detection is *CCFinder* [23], which is applied in our work. Bakers Dup [3,4] implements a similar approach as *CCFinder*. The detection process begins by tokenizing the source code, then using a suffix-tree algorithm to compare tokens. Unlike *CCFinder*, *Dup* does not apply transformation, but rather consistently renames the identifier. Raimar Falke [30] develops a tool called *iclones* [16], which uses suffix-trees to find clones in abstract syntax trees, which can operate in linear time and space. CP-Miner [32] as a well-designed token-based clone detector, uses frequent subsequence mining algorithms to detect tokenized segments. RTF [6] is a token-based clone detector that uses string algorithms for efficient detection; rather than using the more common suffix-tree, it utilizes more memory-efficient suffix array.

One of the leading tools using AST-based algorithm is the *CloneDR* developed by Baxter [7] which can detect exact and near-miss clone through applying hashing and dynamic algorithm. The *ccdiml* [39] developed by Bauhaus is similar to the *CloneDR* in the way of dealing with hash and code sequences, but instead of using AST, it applies IML algorithm in the comparing process. David and Nicholas [14] develop a tool named *Sim* which uses a standard lexical analyzer to generate a parsing-tree of two given software applications. The code similarity is determined by applying the maximum common subsequence and dynamic programming. One of the leading PDG-based tools is PDG-DUP presented by Komondoor and Horwit [27] and Komondoor and Horwitz's PDG-DUP [27] is another leading PDG-based detection tool, which identifies clones together and keeping the semantics of the source code to reflect software. As to metric-based clone detection, Mayrand et al. [37] uses the tool *Darix* to generate the metric and the clone identification is based on four values, which are name, layout, expression and control flow. Kontogiannis [28] uses Markov models to compute the dissimilarity of the code by applying the abstract pattern matching. Five widely used metrics are applied in a direct comparison in [29]. There are also some other approaches that using hybrid clone detections. In [30], the authors apply the suffix trees to find clones in AST; this approach can find clones in linear time and space.

The concept of attack surface is originally proposed for specific software, e.g., Windows, and requires domain-specific expertise to formulate and implement [17]. Later on, the concept is generalized using formal models and becomes applicable to all software [35]. Furthermore, it is refined and applied to large scale software, and its calculation can be assisted by automatically generated call graphs [33,34]. Attack surface has attracted significant attentions over the years. It is used as a metric to evaluate Android's message-passing system [24], in kernel tailing [31], and also serves as a foundation in Moving Target Defense, which basically aims to change the attack surface over time so to make attackers' job harder [18,19]. The study on automating the calculation of attack surface is another interesting domain, e.g., COPES uses static analysis from bytecode to calculate attack surface and to secure permission-based software [5]. Stack traces from user crash reports is used to approximate attack surface automatically [44].

The correlation between attack surface and vulnerabilities has also been investigated, such as using attack surface entry points and reachability to assess the risk of vulnerability [46]. A study about the relationship between attack surface and the vulnerability density is given in [45], although the result is only based on two releases of Apache HTTP Server. Despite such interest in attack surface, to the best of our knowledge, the common attack surface between different software has attracted little attention.

7 Conclusion

In this paper, we have defined the concept of common attack surface and implemented an automated tool for evaluating the common attack surface between given software applications. We have conducted experiments on real open source software and examined the common attack surface both within and between software categories. Our results have shown common attack surface to be pervasive among software. Our work still has some limitations which will lead to our future work. First, since we rely on *CCFinder* our tool also inherits its limitations, and one future direction is to explore other clone detection tools. Second, we have focused on entry/exit points of attack surface, and one future direction is to also consider channels and untrusted data items. Third, we have focused on the C language in this work, and extending it to other languages with different entry and exit libraries is an interesting future direction. Forth, we plan to extend the effort on correlating between common attack surface and known vulnerabilities. We have focused on reused codes only, and a future direction is to also consider their indirect impact on other parts of the software. Finally, one interesting future direction is to evaluate common attack surface between two binary files. Existing disassembling and de-compiling tools, such as IDA Pro [1], could reverse the binary code to source code for further common attack surface study.

Acknowledgment. Authors with Concordia University are partially supported by the Natural Sciences and Engineering Research Council of Canada under Discovery Grant N01035. Sushil Jajodia was supported in part by the National Institute of Standards and Technology grants 60NANB16D287 and 60NANB18D168, National Science Foundation under grant IIP-1266147, Army Research Office under grant W911NF-13-1-0421, and Office of Naval Research under grant N00014-15-1-2007.

References

1. Interactive disassembler. https://www.hex-rays.com/products/ida/
2. Open hub (2017). https://www.openhub.net/
3. Baker, B.S.: A program for identifying duplicated code. Comput. Sci. Stat. **24**, 49 (1993)
4. Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Proceedings of 2nd Working Conference on Reverse Engineering, pp. 86–95. IEEE (1995)

5. Bartel, A., Klein, J., Le Traon, Y., Monperrus, M.: Automatically securing permission-based software by reducing the attack surface: an application to Android. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 274–277. ACM (2012)
6. Basit, H.A., Jarzabek, S.: Efficient token based clone detection with flexible tokenization. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 513–516. ACM (2007)
7. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: 1998 Proceedings of International Conference on Software Maintenance, pp. 368–377. IEEE (1998)
8. Carvalho, M., DeMott, J., Ford, R., Wheeler, D.A.: Heartbleed 101. IEEE Secur. Privacy **12**(4), 63–67 (2014)
9. CVE Community. Common vulnerabilities and exposures (1999). https://cve.mitre.org/
10. Ducasse, S., Rieger, M., Demeyer, S.: A language independent approach for detecting duplicated code. In: Proceedings of IEEE International Conference on Software Maintenance, ICSM 1999, pp. 109–118. IEEE (1999)
11. Durumeric, Z., et al.: The matter of heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference, pp. 475–488. ACM (2014)
12. Ghaffarian, S.M., Shahriari, H.R.: Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey. ACM Comput. Surv. (CSUR) **50**(4), 56 (2017)
13. Ghosh, A.K., Pendarakis, D., Sanders, W.H.: Moving target defense co-chair's report-national cyber leap year summit 2009. Technical report, Federal Networking and Information Technology Research and Development (NITRD) Program (2009)
14. Gitchell, D., Tran, N.: Sim: a utility for detecting similarity in computer programs. In: ACM SIGCSE Bulletin, vol. 31, pp. 266–270. ACM (1999)
15. GitHub. Inc. A web-based hosting service for version control using Git. https://github.com
16. Göde, N., Koschke, R.: Incremental clone detection. In: 13th European Conference on Software Maintenance and Reengineering, CSMR 2009, pp. 219–228. IEEE (2009)
17. Howard, M., Pincus, J., Wing, J.: Measuring relative attack surfaces. In: Workshop on Advanced Developments in Software and Systems Security (2003)
18. Jajodia, S., Ghosh, A.K., Subrahmanian, V.S., Swarup, V., Wang, C., Wang, X.S.: Moving Target Defense II: Application of Game Theory and Adversarial Modeling. Springer, Heidelberg (2012). https://doi.org/10.1007/978-1-4614-5416-8
19. Jajodia, S., Ghosh, A.K., Swarup, V., Wang, C., Wang, X.S.: Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats, 1st edn. Springer, Heidelberg (2011). https://doi.org/10.1007/978-1-4614-0977-9
20. Jajodia, S., Ghosh, A.K., Swarup, V., Wang, C., Wang, X.S.: Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats, vol. 54. Springer, Heidelberg (2011). https://doi.org/10.1007/978-1-4614-0977-9
21. Johnson, J.H.: Substring matching for clone detection and change tracking. In: ICSM, vol. 94, pp. 120–126 (1994)
22. Kamiya, T.: Tutorial of CLI tool ccfx (2008). http://www.ccfinder.net/doc/10.2/en/tutorial-ccfx.html
23. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. **28**(7), 654–670 (2002)

24. Kantola, D., Chin, E., He, W., Wagner, D.: Reducing attack surfaces for intra-application communication in Android. In: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 69–80. ACM (2012)
25. Karp, R.M.: Combinatorics, complexity, and randomness. Commun. ACM **29**(2), 97–109 (1986)
26. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987)
27. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 40–56. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-47764-0_3
28. Kontogiannis, K., Galler, M., DeMori, R.: Detecting code similarity using patterns. In: Working Notes of 3rd Workshop on AI and Software Engineering, vol. 6 (1995)
29. Kontogiannis, K.A., DeMori, R., Merlo, E., Galler, M., Bernstein, M.: Pattern matching for clone and concept detection. Autom. Softw. Eng. **3**(1–2), 77–108 (1996)
30. Koschke, R., Falke, R., Frenzel, P.: Clone detection using abstract syntax suffix trees. In: 13th Working Conference on Reverse Engineering, WCRE 2006, pp. 253–262. IEEE (2006)
31. Kurmus, A., et al.: Attack surface metrics and automated compile-time OS kernel tailoring. In: NDSS (2013)
32. Li, Z., Shan, L., Myagmar, S., Zhou, Y.: CP-miner: finding copy-paste and related bugs in large-scale software code. IEEE Trans. Softw. Eng. **32**(3), 176–192 (2006)
33. Manadhata, P., Wing, J.: An attack surface metric. Technical report CMU-CS-05-155 (2005)
34. Manadhata, P., Wing, J.: An attack surface metric. IEEE Trans. Softw. Eng. **37**(3), 371–386 (2011)
35. Manadhata, P., Wing, J.: Measuring a system's attack surface. Technical report CMU-CS-04-102 (2004)
36. Manadhata, P.K., Wing, J.M.: An attack surface metric. IEEE Trans. Softw. Eng. **37**(3), 371–386 (2011)
37. Mayrand, J., Leblanc, C., Merlo, E.: Experiment on the automatic detection of function clones in a software system using metrics. In: ICSM, vol. 96, p. 244 (1996)
38. Petersenna. Ccfinder core. https://github.com/petersenna/ccfinderx-core
39. Raza, A., Vogel, G., Plödereder, E.: Bauhaus – a tool suite for program analysis and reverse engineering. In: Pinho, L.M., González Harbour, M. (eds.) Ada-Europe 2006. LNCS, vol. 4006, pp. 71–82. Springer, Heidelberg (2006). https://doi.org/10.1007/11767077_6
40. Rothwell, T.: The GNU C reference manual (2006). https://www.gnu.org/software/gnu-c-manual/
41. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Sci. Comput. Program. **74**(7), 470–495 (2009)
42. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Queen's Sch. Comput. TR **541**(115), 64–68 (2007)
43. Syropoulos, A.: Mathematics of multisets. In: Calude, C.S., PĂun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2000. LNCS, vol. 2235, pp. 347–358. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45523-X_17
44. Theisen, C., Herzig, K., Morrison, P., Murphy, B., Williams, L.: Approximating attack surfaces with stack traces. In: Proceedings of the 37th International Conference on Software Engineering, vol. 2, pp. 199–208. IEEE Press (2015)

45. Younis, A.A., Malaiya, Y.K.: Relationship between attack surface and vulnerability density: a case study on apache HTTP server. In: Proceedings on the International Conference on Internet Computing (ICOMP), p. 1. The Steering Committee of the World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) (2012)
46. Younis, A.A., Malaiya, Y.K., Ray, I.: Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In: 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering (HASE), pp. 1–8. IEEE (2014)
47. Zhang, M., Wang, L., Jajodia, S., Singhal, A., Albanese, M.: Network diversity: a security metric for evaluating the resilience of networks against zero-day attacks. IEEE Trans. Inf. Forensics Secur. **11**(5), 1071–1086 (2016)