

COMMON ATTACK SURFACE DETECTION

YUE XIN

A THESIS

IN

THE DEPARTMENT

OF

CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE

IN INFORMATION SYSTEMS SECURITY AT

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

MARCH 2018

© YUE XIN, 2018

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Yue Xin**

Entitled: **Common Attack Surface Detection**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Y.Zeng _____ Chair

Dr. W.Lucia _____ Examiner

Dr. J.X.Zhang _____ External Examiner

Dr. Lingyu Wang _____ Supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 2018 _____

Dr. Christopher Trueman, Interim Dean
Faculty of Engineering and Computer Science

Abstract

Common Attack Surface Detection

Yue Xin

In the current software development market, many software is being developed using a copy-paste mechanism with little to no change made to the reused code. Such a practice has the potential of causing severe security issues since one fragment of code containing a vulnerability may cause the same vulnerability to appear in many other software with the same cloned fragment. The concept of relying on software diversity for security may also be compromised by such a trend, 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 the specific impact on security.

In this thesis, 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, *Dupsec*, in order to efficiently identify common attack surface between any given pair of software applications with minimum human intervention. Finally, we conduct experiments by applying our tool to a large number of open source software. Our results demonstrate many seemingly unrelated real-world software indeed share significant common attack surface.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Lingyu Wang for his continuous support in my study and research. His guidance has been invaluable in the research and writing of this thesis.

I would also like to thank my friend and lab-mate Mengyuan Zhang, who is a source of inspiration and who helped finish this thesis.

Thank you to all my lab-mates at Concordia University, I am grateful to have the chance to be a member of this research group.

Contents

List of Figures	vii
List of Tables	ix
List of Algorithms	x
1 Introduction	1
2 Preliminaries	5
2.1 Motivating Examples	5
2.2 BackGround	10
3 The Model of Common Attack Surface	13
3.1 Conditional Common Attack Surface	14
3.2 Probabilistic Common Attack Surface	17
4 Design and Implementation	20
4.1 The Main Components of DupSec	20
4.1.1 The Clone Detection Module	22
4.1.2 The Source Code Labeling and Prediction Module	25
4.1.3 The Visualization and CAS Calculation Module	31
4.2 Automated Analysis Using <i>DupSec</i>	33
5 Experimental Results	35

5.1	Dataset	35
5.2	Common Attack Surface and Vulnerabilities	37
5.3	Cross-Category Common Attack Surface	38
5.4	Common Attack Surface in the Same Category of Software Applications	52
5.5	The Distribution of Function Names	56
6	Related Work and Background Knowledge	59
6.1	Clone detection	59
6.2	Attack Surface	67
6.3	Software Diversity	68
7	Conclusions and Future Work	70
	Bibliography	76

List of Figures

1	<i>Heartbleed</i> in Apache and Nginx	7
2	/Simple-Webserverche/server.c <i>handle_response()</i>	12
3	/quicksand_lite/libqs.c <i>quicksand_mime()</i>	12
4	An Example of Clone Segments	15
5	Architecture of <i>DupSec</i>	21
6	The False Positive and False Negative Rates in Different Parameter Ranges	24
7	.ccfd file (a),the first translated file(b), and the second translated file(c))	26
8	Keywords Distribution (a) and The <i>Read</i> keyword Vulnerability Distribution (b)	28
9	An Example of DupSec Output	32
10	The Common Attack Surface between FTP and All the Categories . .	40
11	The Common Attack Surface between FireWall and All the Categories	40
12	The Common Attack Surface between DB and All the Categories . .	41
13	The Common Attack Surface between WebServer and All the Categories	41
14	The Common Attack Surface between SSH and All the Categories . .	42
15	The Common Attack Surface between TFTP and All the Categories .	42
16	The Common Attack Surface between IMAP and All the Categories .	43
17	The Common Attack Surface in Accumulated Pairs of Software Applications	44
18	Common Attack Surface in Accumulated Software Application Pairs vs Size	45

19	Common Attack Surface Trend in Years	46
20	The Probabilistic cas measure	47
21	cas vs Functionalities in FTP with All the Categories	48
22	cas vs Functionalities for FireWall with All the Categories	49
23	cas vs Functionalities for DB with All the Categories	49
24	cas vs Functionalities for WebServer with All the Categories	50
25	cas vs Functionalities for SSH with All the Categories	50
26	cas vs Functionalities for TFTP with All the Categories	51
27	cas vs Functionalities for IMAP with All the Categories	51
28	Top 10 I/O Functions (a) and CVE (b) for Different Functionalities .	53
29	Common Attack Surface(s0d) vs Size for WebServers	54
31	Common Attack Surface(s0d) vs Size over Time	55
30	Common Attack Surface(s0d) vs Size for FTP	55
32	Common Attack Surface vs Size over Time in the Same Category . .	56
33	Common Attack Surface vs Funtionalities in Different Categories . . .	57
34	The Distribution of Keywords in Function Names	58

List of Tables

1	The Parameters of CCFinder	22
2	The Effect of Different Parameters (FP: false positive; FN: false negative)	24
3	The Keywords and Variations	29
4	The Comparison of Different Tools	29
5	Logging Parameters of <i>DupSec</i>	34
6	Percentage of Detected Vulnerabilities Which Correlate to Reported Common Attack Surface	38
7	HeatMap for Common Attack Surface in Different Categories	43
8	Percentage Value of Common Attack Surface	57

List of Algorithms

1 Automated Analysis 33

Chapter 1

Introduction

The concept of security through software diversity is less dependable today due to the fact that many libraries are shared between different software applications. If a library contains vulnerabilities, many seemingly unrelated software may be at risk of exploits if they all import this same library. As a well-known example, on April 7th, 2014, the *Heartbleed* vulnerability in *OpenSSL* caused widespread panic on the Internet, affecting many popular Web browsers, including Apache and Nginx [13]. Researchers have since studied the popular libraries in order to better predict potential vulnerabilities [39]. Similar to libraries, the reusing of other existing code may also lead to similar vulnerabilities shared by different software applications. In fact, code reusing is a common phenomenon in today's software industry due to the fact that copying and pasting accelerates the code development process [7, 12]. However, unlike organized libraries, reused codes are not traced by any official documentation, which makes it more difficult to understand their security impact.

To the best of our knowledge, detecting similar vulnerabilities shared among different software applications due to code reusing has received little attention in the literature (a more detailed review of the related work will be discussed in Section 6). Most existing vulnerability detection tools, either based on static or dynamic analysis, focus on generating a list of potential vulnerabilities for a specific software application, with no indication whether different software may be sharing similar vulnerabilities

due to common libraries or reused codes. On the other hand, existing efforts on source code clone detection has developed two main types of detection methodologies, either based on textual similarity or function similarity, to identify cloned codes in the source code. While clone detection tools like *CCFinder* [25] could provide detailed reports on code similarities between two software, they do not indicate any security impact of such similarities. In other words, they do not answer the question: *Knowing that two software applications share some similar code fragments, how can we characterize the likelihood that those software may also share some common vulnerabilities?*

The above question is especially relevant at a higher abstraction level, the network, where software diversity has been widely recognized as a viable means to defend against various security threats, such as worm propagation [41], and to boost the overall resilience of the network [38]. More recently, diversity has also been applied to emerging applications, e.g., cloud computing security [45, 48], Moving Target Defense(MTD) [22], and network routing [8]. Most of those works rely on an intuitive or over simplified notion of software diversity, e.g., different software will not share common vulnerabilities. Such a notion of software diversity has become insufficient due to the aforementioned trend of code sharing among seemingly unrelated software, which can potentially diminish the value of such diversity-based security mechanisms. For example, using different Web servers in a network provides no security advantages if they can all be exploited through the same Heartbleed vulnerability.

In this thesis, we aim to address the above issue through defining the novel concept of *common attack surface* and developing an automated tool, *DupSec*, to calculate the common attack surface of given software. Specifically,

- First, 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. This formal model provides numeric outputs to measure software diversity from security point of view, and such results may be used as inputs to higher level diversity measures.

- Second, to automate the calculation of common attack surface, we have developed an automated tool, *Dupsec*, in order to efficiently identify common attack surface between any given pair of software with minimum human intervention. This tool takes the source code of two software as input, and the result can be either reported in an XML file or stored in the database.
- Third, we conduct experiments by applying our tool to a large number of open source software. To make our experiment results more convincing, we have collected software applications belonging to seven different categories from *Github* in order to evaluate our tool. More than 80,000 combinations of software applications have been downloaded and analyzed in our experiments. The results are analyzed based on various features of the software applications. Our results demonstrate many seemingly unrelated real-world software indeed share significant common attack surface.
- Finally, we also study the correlation between our experimental results and real-world vulnerabilities related to the *C* language. We have manually examined the shared vulnerable code fragments discovered by our tool, and verified that similar vulnerabilities which correspond to the common attack surface do exist in the software.

This thesis makes three main contributions as follows.

- First, to the best of our knowledge, this is the first effort on formally modeling the impact of code reusing on security. Our common attack surface model provides a foundation and inputs to other higher level security-through-diversity models.
- Second, the tool we have developed to calculate common attack surface between software applications may provide useful references for end users to choose software applications in order to achieve sufficient software diversity in a network,

and to reuse knowledge about existing vulnerabilities in a software to potentially identify similar ones in other software.

- Third, our experimental results prove the existence of similar vulnerabilities shared by seemingly unrelated software due to code sharing. We believe this finding would attract interest from both practitioners and researchers to re-examine the concept of software diversity and its security implication.

The remainder of this thesis is organized as follows. In Section 2, we present the concept of common attack surface following a motivating example and discussions on the challenges of designing such a formal model. In Section 3, we define two measures to calculate the common attack surface among software applications. In Section 4, we design and implement a tool called *DupSec* to calculate common attack surface among software applications. In Section 5, we evaluate our model and tool through experiments with real world software. In Section 6 we review related work and in Section 7 we discuss about the limitations of the work and future directions.

Chapter 2

Preliminaries

In this section, we first present motivating examples through discussing several use cases in Section 2.1. We then describe the background knowledge in Section 2.2. Finally, we highlight the challenges before defining the formal model of common attack surface in Section 3.

2.1 Motivating Examples

We will use two fictitious websites to describe our use cases. The first website is referred to as *https://apachesite.com* (apache site), which is equipped with an Apache HTTP server running on host 1 (h1) and Cyrus IMAP server on host 2 (h2). The second website, called *https://nginxsite.com* (nginx site), is equipped with only Nginx HTTP server running on host 3 (h3).

User Case 1 *Vulnerability Discovery and Patch Reuse:* Clearly, all the hosts are equipped with different software applications in each website. Assume an attacker on the Internet wants to compromise the websites, without considering any code clones among those software applications, the system administrators of both websites as well as most existing security metrics would consider that the attacker would need to employ three different exploits (for the three software applications) in order to

achieve his/her goals. However, as we will show, similar vulnerabilities due to cloned codes may exist among those different software applications, which may render the attacker's job much easier by using the same exploit repeatedly.

For example, consider again the well known *Heartbleed* vulnerability which drew world-wide attention in April 2014 [13]. This vulnerability affects an estimated 24-55% of popular HTTPS sites, and it gives attackers accesses to sensitive memory blocks on the affected servers, which may potentially contain encryption keys, usernames, passwords, etc. More specifically, the vulnerability is discovered inside *OpenSSL* which is an extension of many Web and email server software applications for supporting the *https* connections. In particular, the aforementioned three software applications assumed in our two example websites, namely, two Web server applications (*Apache* and *Nginx*), and one mail server (*Cyrus IMAP server*) are all affected by this vulnerability. In such a case, attackers now have the ability to exploit one vulnerability in order to compromise all three servers.

Figure 1 shows technical details about *Heartbleed* and how it functions in relation to these three software applications. The three software applications will simply hand the encryption tasks to the *OpenSSL* extension. The vulnerability appears after the software applications make external calls to the *OpenSSL* extension. By calling the same functions to establish 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 vulnerability, attackers would construct a specific heartbeat request with a fake length and send it to the servers in order to extract sensitive memory blocks from the servers.

On the other hand, not all the server software applications that use SSL connections are affected by this vulnerability. For example, Microsoft IIS does not use *OpenSSL* to create secure connections and hence is immune to the vulnerability. Therefore, our objective in this use case is to automatically capture and systematically measure the likelihood of such vulnerabilities that are shared by different

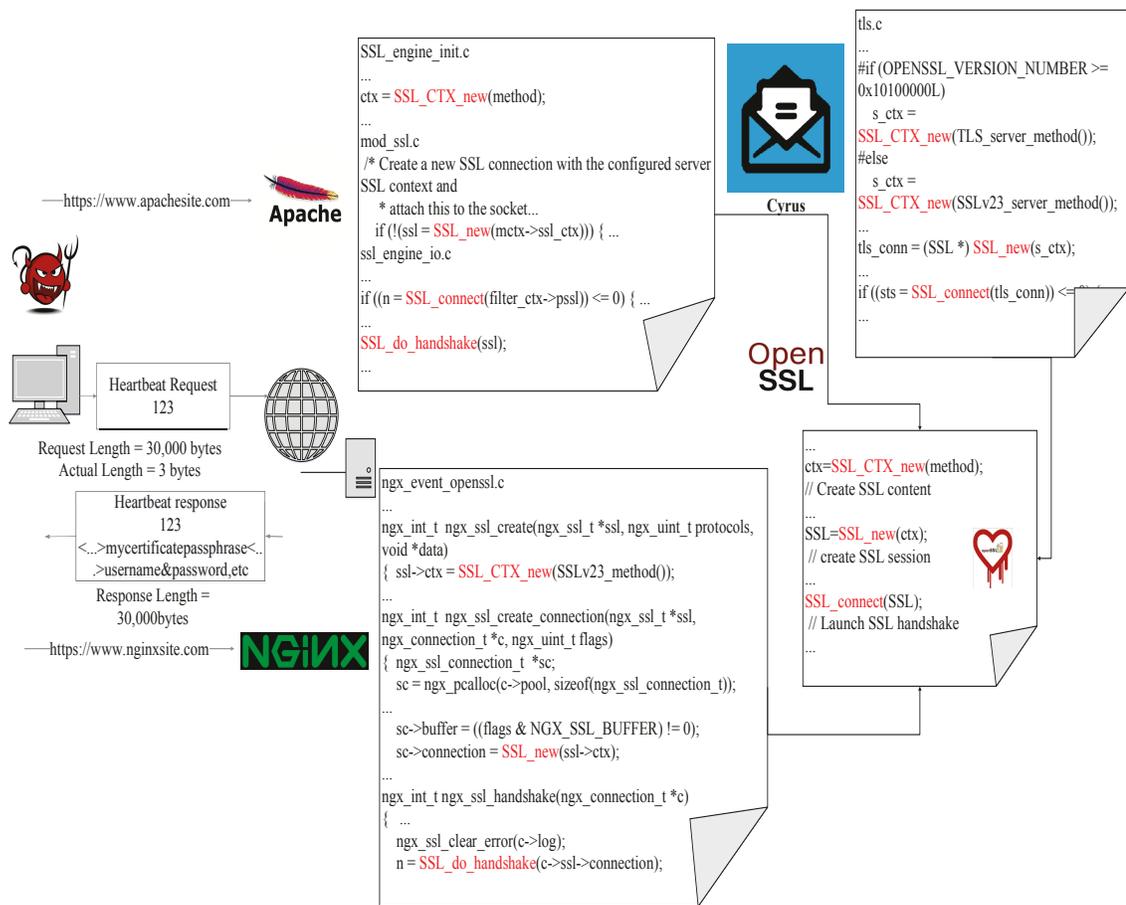


Figure 1: Heartbleed in Apache and Nginx

software applications due to cloned codes. Consequently, the same software patches or fixes could be applied or developed to mitigate those vulnerabilities in a similar manner, which may potentially reduce the time and effort spent in developing vulnerability patches and fixes.

Use Case 2 *Providing Inputs to Diversity Mechanisms:* Different notions of diversity have been used for security purposes, e.g., compile-time diversity [15], run-time diversity [28], automatic patch generation [49], and automatic signature detection [51]. Diversity has been employed to enhance security in terms of preventing attacks and worm propagation for various software applications, e.g., e-mail topologies, client-server file shares, and sensor networks [40]. Opportunistic diversity may already exist among different software systems [52], and diversity can also be automatically generated by breaking the monoculture and adding more features to the applications [33].

However, as the basis of most of those diversity mechanisms, the definition of software diversity remains unclear in the face of code reusing. For example, in our apache site, the two hosts *h1* and *h2* are equipped with different software applications that are remotely accessible, a diversity mechanism will thus consider this website as having sufficient diversity in terms of remote attacks over the network [56]. However, as we have explained, the two seemingly different software applications may in fact share similar vulnerabilities due to code reusing, which means the website is not necessarily diversified enough against remote attacks.

While *Heartbleed* remains a good example, the impact of code reusing upon software diversity is certainly not limited to this specific case. In fact, the practice of code reusing and its impact on software diversity may have become increasingly pervasive today. First, it is very common for different software to import the same popular library functions, e.g., those from C standard library which have a history of including high risk memory-related vulnerabilities, such as buffer overflow. The *HeartBleed* vulnerability is also caused by the fact that the developers of different software applications have essentially all invoked the same library function *memcpy()* without any

boundary check. While the practice of using existing library functions reduces the time and cost for software development, it also increases the chance that the same vulnerability may be introduced to multiple software applications.

The impact of code reusing upon software diversity is not limited to importing common library functions. For example, both the *Apache* and *Nginx* projects are Web servers developed in C language. The similar functionality of both software means there is a high chance that the developers of both projects may reuse the same or similar codes in addition to importing common library functions. In many software development companies, on-board training provides tutorials for new employees to learn how to build codes according to the company's existing practice and guidelines. Such a process reduces the cost of code management and facilitates the transfer of knowledge and information, but at the same time it further increases the chance for passing similar vulnerabilities to all the company's software applications [6].

Therefore, our objective in this use case is to formally model and automatically compute the degree of software diversity from the security point of view, while taking into consideration common library functions and reused codes among different software applications. Such results on software diversity could be taken as inputs by other higher level diversity mechanisms in order to allow them to better deal with the existing similarity between different software.

Use Case 3 *Moving Target Defense*: The moving target defense (MTD) concept is a relatively new approach to security and a potential direction for improving the security of static systems [16]. The main idea of MTD is to continuously change the system configurations which would hopefully interrupt any attacker's process in compromising the system [57]. In our apache site, the host h1 is equipped with Web server applications, and administrators could implement MTD by constructing a resources pool of web server applications, e.g., $\langle Apache, Microsoft IIS, Nginx, Lighttpd, Tomcat, GoogleGWS, LiteSpeed, IBM Web Server, Tengine, Jetty \rangle$ which are all different Web server applications. Assume any of the software from the resource pool

could fulfill the functionality requirements and the currently deployed software application on *h1* is *Apache*. An administrator may change the server's configuration by switching from one software application to another inside the resource pool. In theory, all those different Web servers would have different attack surface and hence rotating among them from time to time will make attackers' job harder. However, we have seen that this may not always be the case as the same or similar vulnerabilities may exist among supposedly different software applications, as demonstrated in Figure 1. In fact, other than *Microsoft IIS* and *Jetty*, all the aforementioned Web servers will be affected by the same vulnerability [13]. In such a case, changing the configuration may not interrupt the attack process, and MTD becomes less effective due to the weakened software diversity.

Therefore, in this use case, our objective is to provide a better foundation to MTD by defining software diversity from the security point of view, while taking into consideration of potential cloned codes among different software. To follow the previous example, there are more than 500 Web server software applications published on *GitHub*, and our study will indicate significant amounts of cloned code segments among these software applications. The effectiveness of MTD mechanisms can be improved by choosing the right software for the resource pool based on the understanding of similarities between software applications.

2.2 BackGround

Before defining the model of common attack surface, we discuss some background concepts. 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 design security measures in order to characterize the security impact of such code fragments.

In order to detect the similarity of source codes, several algorithms and tools have been proposed in existing researches on clone detection. Generally speaking, there

are different types of code clone detection (a detailed review of related works will be given in Section 6) that can be categorized into two classes, those based on textual similarity and those on functional similarity. Tools are developed based on techniques belonging to following categories, text-based techniques, token-based techniques, tree-based techniques, graph-based techniques, metrics based clone techniques and hybrid techniques [47].

We use *CCFinder* [25], a language-based source code clone detection tool, to find cloned code fragments within given software. 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. The first step in the clone detection process is to obtain the language originated preprocessing, which can be seen as a normalization process. Tokens are generated from the source code after the lexical analysis and concatenated into token sequences. Then, the generated tokens are applied to the matching process. During the matching process, the algorithm suffix-tree matching is used to enhance the efficiency of matching, and the location of clone information is represented as a tree with nodes storing information about identical subsequences [25].

However, the result from clone detection tools, including *CCFinder*, only reveals similar code fragments between source codes, without indicating their potential security impact. The primary challenge is therefore to understand and measure the potential impact of clone detection on security in terms of leading to potential vulnerabilities.

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 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) (denoted as $\langle M, C, D \rangle$). Attack surface measures the intrinsic properties of a software application regardless of external factors such as the existence of exploits. Therefore, attack surface may potentially

cover both known and unknown vulnerabilities. In this thesis, we apply the attack surface concept to quantify the cloned code fragments in terms of their likelihood to lead to potentially similar vulnerabilities shared between software applications. We focus on entry and exit points and do not consider channels and untrusted data items for simplicity.

We discuss a key challenge in applying the attack surface concept to characterize cloned code fragments among software applications. As we have described, entry/exit points are the methods calling I/O functions, e.g., in Figure 2, function *handle_response()* is an entry point since it calls *fseek()* and *ftell* (which are I/O functions from standard C library), so is function *quicksand_mime()* in Figure 3.

```
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);
```

Figure 2: /Simple-Webserverche/server.c *handle_response()*

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

Figure 3: /quicksand_lite/libqs.c *quicksand_mime()*

A naive application of attack surface concept here would indicate both functions count as one entry point inside the attack surface, and hence, they have the same security impact. However, such an application is too coarse-grained since the two functions clearly include different number of calls to I/O functions (three calls in *handle_response()* and two in *quicksand_mime()*). The different numbers of I/O function calls should be taken into account while measuring the chance to cause vulnerabilities inside the software applications. In our model, we will refine the entry/exit point concept from security point of view.

Chapter 3

The Model of Common Attack Surface

In this chapter, we model the similarities between two software applications through two security measures, the *conditional common attack surface measure* (*ccas*) and the *probabilistic common attack surface measure* (*pcas*).

First, the *conditional common attack surface measure* (*ccas*) is designed to be asymmetric for use cases in which one software is of particular interest and evaluated against other software. For example, suppose a company has developed a new Web server application and want to understand any similarities between their product and other similar Web servers, e.g., *Apache* and *Nginx*. The developers want to calculate the percentage of attack surface their product may share with any other software applications. In such a case, the developer can apply the measure *ccas* to measure the common attack surface shared by their software application when compared to other software applications.

Second, in a slightly different scenario, an administrator wants to understand the level of software diversity between any two software applications inside the network. In such a case, both software applications in comparison are equally important, so the symmetric measure *pcas* can be applied, which will yield a unique measurement of similarity between two software applications. The following details the *ccas* and

pcas measures.

3.1 Conditional Common Attack Surface

Suppose we want to measure the common attack surface of one software application in comparison to another. We need first to identify the clone segments between the two software applications. As an example, Figure 4 demonstrates an instance of clone segments between two software applications where the left upper corner of the figure depicts a Web server application *SimpleWebserver*, and the remainder of the figure depicts an SSH application *SSHBen*. In the figure, the Clone id is a unique number indicating a group of related clones inside both software applications. For example, the code segments inside the solid line blocks indicate the clone segments of both software applications that belong to the same Clone id 28, and the dashed line blocks are the clone segments that belong to the Clone id 78. The same part of the code may appear in two different clone ids, e.g., line 146 and 147 in *Simple-Webserver* appear in both clone ids.

In 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 clone segments must be referred to in an asymmetric manner as well. We denote by an abstract function $c(\cdot)$ the clone segments between two software applications, with $c(A|B)$ representing the clone code segments in software application A in comparison to software B, which is not necessarily identical to the clone code segments in software application B, $c(B|A)$.

Example 1 *In Figure 4, Clone id 78 contains five clone segments which corresponds to one segment inside SimpleWebserver and four segments in SSHBen. The I/O function calls inside the clone segments are $\langle strcat, fopen, fseek, ftell \rangle$ (here the matching between the two clone segments is inexact [25], since *strcat* does not exist in SSHBen). As to Clone id 28, the I/O function calls inside the clone segments are $\langle strcat, fopen \rangle$.*

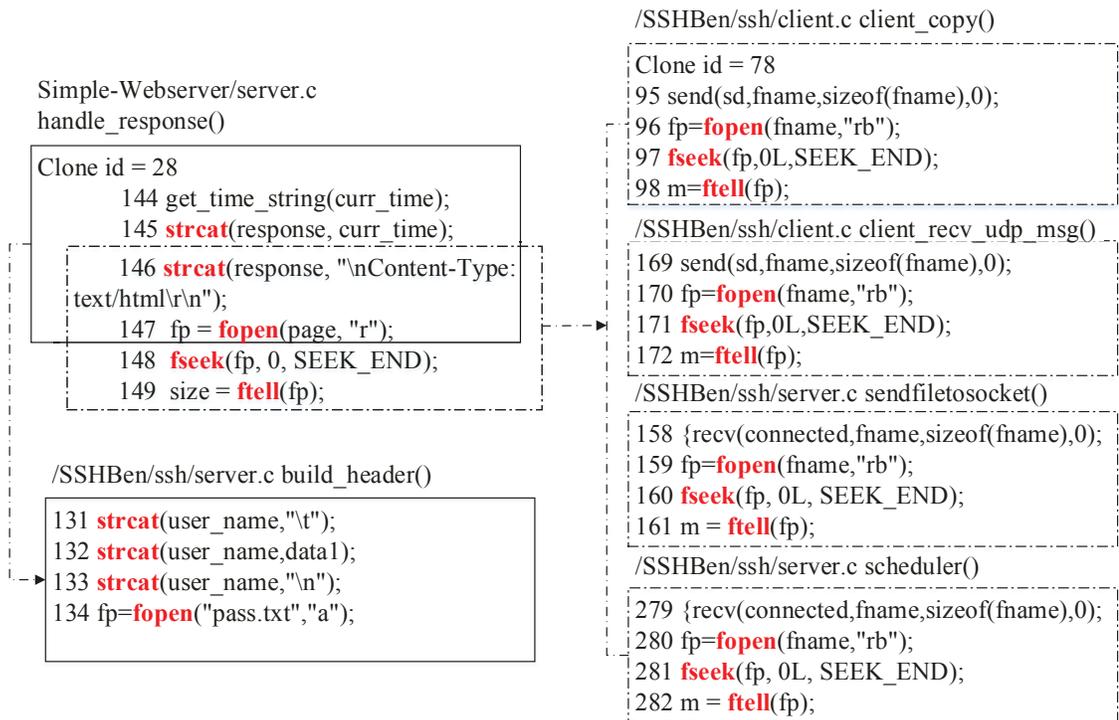


Figure 4: An Example of Clone Segments

Definition 1 (Common Attack Surface) *Given two software applications, the common attack surface is defined as the multi-sets (which preserves duplicates) of I/O function calls that exist inside the clone segments of each software application with the same Clone id i , denoted as $cas_i(B|A)$ and $cas_i(A|B)$, respectively.*

Example 2 *To follow our example, we have $cas_{78}(SimpleWebserver|SSHBen) = \langle f_{open}, f_{seek}, f_{tell} \rangle$ and $cas_{78}(SSHBen|SimpleWebserver) = \langle f_{open}, f_{seek}, f_{tell}, f_{open}, f_{seek}, f_{tell}, f_{open}, f_{seek}, f_{tell}, f_{open}, f_{seek}, f_{tell} \rangle$.*

To measure the similarity between two software applications in terms of common attack surface, we will need to calculate the size of the attack surface, i.e., the number of I/O function calls. As we can see from the above example, the clone segments inside two software applications for the same Clone id are not necessarily a one-to-one mapping, e.g., for Clone id 78 there is one segment inside *Simple-Webserver* and four segments inside *SSHBen*. Therefore, we will count the exact number of I/O function calls inside the common attack surface of each software application.

Example 3 *For Clone id 78, this gives 3 for Simple-Webserver and 12 for SSHBen. As to Clone id 28, both software applications have one clone segment, with 3 I/O function calls ($\langle strcat, strcat, fopen \rangle$) in Simple-Webserver and 4 in SSHBen ($\langle strcat, strcat, strcat, fopen \rangle$). The first *strcat* in Clone id 28 is considered as part of the common attack surface but not in Clone id 78. On the other hand, *fopen* is considered in both Clone ids, and hence we count *fopen* only once between the two Clone ids for Simple-Webserver. Based on those discussions, we can calculate the total number of I/O function calls for both Clone id is 5 for Simple-Webserver and 16 for SSHBen.*

The conditional common attack surface defined below represents the ratio between the size of the common attack surface of a software application (with respect to another software application) and the total number of all I/O function calls inside that software. This ratio roughly indicates the degree of similarity between the two software applications in terms of common attack surface.

Definition 2 (Conditional Common Attack Surface) *Given two software applications, A and B , the conditional common attack surface, denoted as $ccas(B|A)$ and $ccas(A|B)$, assume n is the total number of clone ids between A and B , and AS_A and AS_B the total number of I/O functions inside A and B , respectively. We define the conditional common attack surface as:*

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

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

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 $ccas(SSHBen | SimpleWebserver) = \frac{16}{182} = 0.0879$ and $ccas(SimpleWebserver | SSHBen) = \frac{5}{16} = 0.3125$.*

3.2 Probabilistic Common Attack Surface

In the previous section, the conditional common attack surface measure $ccas$ is designed for evaluating one software application against others. In this section, we take a different approach of defining a symmetric probabilistic common attack surface measure for two software applications. The main purpose of this measure is 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 measure 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 measure in our case, we need to define both the intersection and union of the attack surface of two software applications. In the previous section, the intersection has been defined as the common attack surface. However, as we mentioned, the common attack surface of two software applications are asymmetric in nature, which

may contain different clone segments for each software in comparison. Instead, we will define the intersection between the attack surface of two software applications differently using the standard multi-set intersection operation, which is described below.

Definition 3 (Intersection of Multi-Sets [50]) *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 multi-set) 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 $AS_A \cup AS_B = AS_A + AS_B - cas(B | A) \cap cas(A | B)$. With both the union and intersection operations defined, we can now define the probabilistic common attack surface measure as follows.

Definition 4 (Probabilistic Common Attack Surface Measure) *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:*

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

Example 6 *The size of attack surface in Simple-Webserver and SSHBen is 16 and 182, respectively. From our previous discussions, we have $cas(SSHBen | SimpleWebserver) \cap cas(SimpleWebserver | SSHBen) = \langle strcat, strcat, fopen, fopen, fseek, ftell \rangle$ whose size is 6, and hence $pcas(SSHBen.SimpleWebserver) = \frac{6}{16+182-6} =$*

3.1%. Intuitively, this result indicates, among all the attack surface (i.e., I/O function calls) which may potentially cause either known or unknown vulnerabilities, the percentage of attack surface is shared between the two software applications. Such a result, when applied to all the remotely accessible software applications inside a network, may allow administrators to estimate and consequently improve the degree of software diversity in the network from a security point of view.

Chapter 4

Design and Implementation

To automatically identify and measure the common attack surface between software applications, we design and implement a tool, *DupSec*, to generate and visualize traceable outputs. We first detail the architecture and various components of *DupSec* in Section 4.1, and then discuss the automated analysis using our tool in Section 4.2.

4.1 The Main Components of DupSec

Figure 5 provides a high-level architecture of our tool, *DupSec*, which consists of three components, the clone detection module, the source code labeling and prediction module, and the visualization module. Those modules work together to identify and measure the common attack surface between two given software applications. The following details each module.

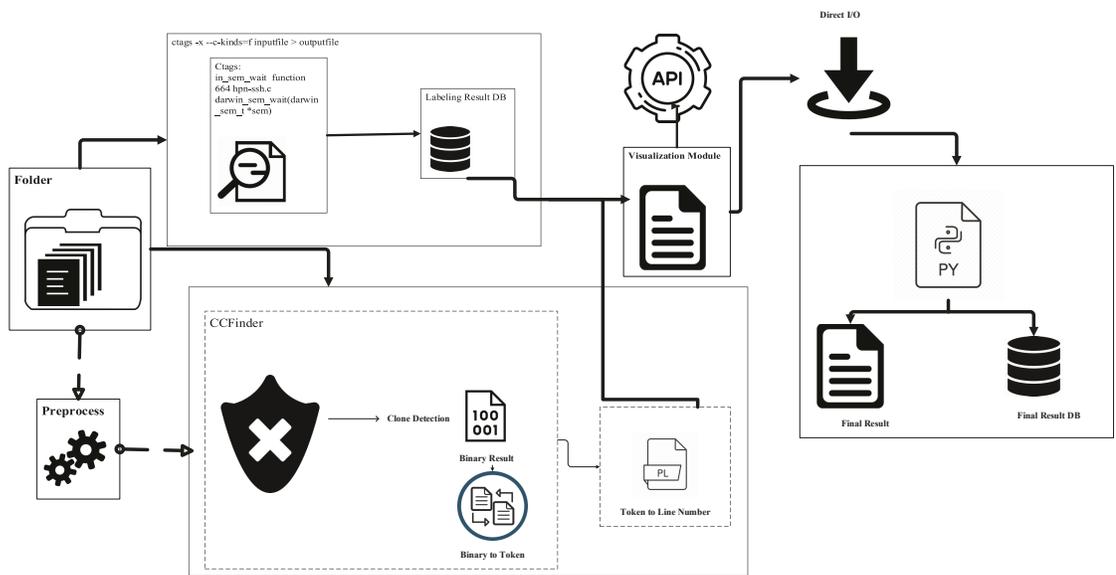


Figure 5: Architecture of DupSec

4.1.1 The Clone Detection Module

As we discussed briefly in Section 2.2, we choose *CCFinder* as the basis of our clone detection module. The following details challenges and solutions for applying *CCFinder*.

Since our tool *Dupsec* is developed and operated under Linux, we apply only the back-end of *CCFinder*. While the Microsoft Windows version of *CCFinder* can be more easily installed, installing the back-end component on Linux is more challenging. Specifically, the default Linux version of *CCFinder* is designed to work on *Ubuntu 9*. With *Ubuntu* upgraded to the version 14 and newer, many libraries are no longer valid for *CCFinder*. Therefore, several libraries need to be installed separately, e.g., *libboost-dev* and *libcicu-dev*. In practice, various libraries may be needed based on the version of the Linux system, which can be determined based on the warnings and errors produced by *CCFinder*. The minimum requirement of *Dupsec* is for the *CCFinder* back-end to normally execute any *CCFinder* command and produce clone detection correctly. Once *CCFinder* is successfully installed, the clone segments can be fetched from the output files.

Various parameters can be fine tuned in *CCFinder* to customize its execution mode [24]. Although default parameters exist, for the purpose of acquiring better results, parameters should be adjusted based on specific applications' needs. Table 1 lists the three options which have been adjusted in our module to achieve the desired clone results.

Option	Description	Parameter
-b	the minimum length of the detected code clones	20
-t	the minimum number of types of tokens involved	8
-w	to specify whether to detect inner-file clones or/and inter-file clones	f-w-g+

Table 1: The Parameters of CCFinder

As shown in the Table 1, the parameter b and t are the two parameters that control the length of the clone code segments, which is important to our application.

Specifically, parameter t represents the type of tokens used in clone detection based on the rules of *CCFinder*. The basic expression in C language, such as `int A;`, consists of three tokens: the integer, identifier, and ‘;’. Therefore, as long as the setting of the parameter t is larger than three, the effect of t upon clone detection will be insignificant for this specific application.

In addition, based on our experimental results, we find that when parameter b is larger than 30, no clone result would be reported. Therefore, for the purpose of obtaining clone segments, parameter b is set to be less than 30. The trend of the error rate for the two software applications with different values of parameter b can be seen in Figure 6. Based on Figure 6, when parameter b is around 18-23, the error rate is 0% and the false negative increases with the incremental increase of the parameter b . When parameter b is between 10 to 22, the false negative remains stable. Based on the results show in Figure 6 and Table 1, we choose the parameters $b=20$, $t=8$ for further experiments.

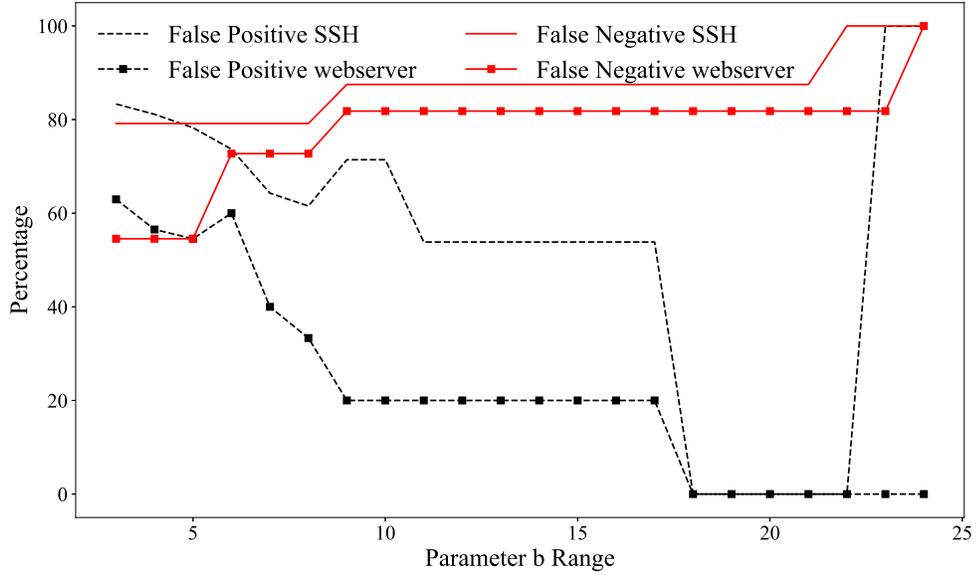


Figure 6: The False Positive and False Negative Rates in Different Parameter Ranges

	b=5 t=3		b=20 t=3		b=50 t=3	
	FP	FN	FP	FN	FP	FN
<i>SSHBen</i>	46.49%	54.59%	0.0%	95.67%	N/A	100%
<i>Simple-Webserver</i>	46.87%	48.48%	0.0%	90.90%	N/A	100%
<i>SSH-Multithread</i>	81.81%	79.16%	0.0%	87.5%	N/A	100%
<i>C-Webserver</i>	54.54%	54.54%	0.0%	81.81%	N/A	100%

Table 2: The Effect of Different Parameters (FP: false positive; FN: false negative)

The parameter w is used to determine whether *CCFinder* will perform inner-file clone whose results contain clones between different parts of the same software application, which is not the focus of this paper. Therefore, the parameter w is set to be $f-w-g+$ for the purpose of focusing on the inter-file clone.

The default output of the *CCFinder* is stored in a file with *.ccfd* as the extension. Normally, the GUI GemX, which is the front-end of *CCFinder*, translates the result into a human-friendly format, such as a clone-set table, scatter plot, or source text. In our module, only the back-end of *CCFinder* has been used for the purpose of accessing the outputs directly. Therefore, we apply the command `./$PATH/ccfx -p name.ccfd` to translate the *.ccfd* binary file into a human-readable version. Figure 7 shows a small example of the translation process.

The translated file, in the first part of Figure 7, contains only the token information, which cannot be directly mapped back to the source code files. To obtain the corresponding source code clone segments, a second translation is needed. For this purpose, we have applied a script, *post-prettyprint.pl* [43], to read the translated results and to convert the token information into corresponding line numbers in the source code. The second translated output is saved in an *XML* file containing the clone identification(ID), the name of the source code file, and the line numbers of the cloned segments. All the information produced by *CCFinder* that is relevant to our application is obtained after this second translation. However, this result does not automatically map back to the clone segments in the source code. The next module focuses on obtaining the necessary information from source code to automatically trace back to the cloned segments.

4.1.2 The Source Code Labeling and Prediction Module

As mentioned above, the second translated output from *CCFinder* provides only the file name and line number of the clone segments, without information needed for mapping them back to the original source codes. For the purpose of generating traceable output with source code fragments, a mapping between the line number of

```

^Q^@^@^A^@^@^@/PATH/Tachftp/privparent.c
^R^@^@^c^B^@^@/PATH/Tachftp/privsock.c
^S^@^@^A^B^@^@/PATH/Tachftp/tunable.h
^T^@^@^A^A^@^@/PATH/Tachftp/parseconf.c
^U^@^@^@^A^@^@^@/PATH/Tachftp/str.h
^V^@^@^@^A^@^@^@
^@^@^@^@^@^@^@
^@^@^@^@^@^@^@f^A^@^@^@^@M^@^@^@=8d>^@^@^@M^@^@^@Q^@^@^@Q^@^@^@
^@^@^@^@^@^@^@f^A^@^@^@^@^@^@M^@^@^@^@n^A^@^@^@=85>^A^@^@^@Q^@^@^@^@
^@^@^@^@^@^@^@y^D^@^@^@=84>^D^@^@^@M^@^@^@y^@^@^@M^@^@^@. ^@^@^@^@^@^@^@
^@^@^@^@n^D^@^@^@=84>^D^@^@^@M^@^@^@n^A^@^@^@=85>^A^@^@^@Q^@^@^@^@^@^@
M^@^@^@y^@^@^@M^@^@^@^@^@^@^@^@y^D^@^@^@=84>^D^@^@^@. ^@^@^@^@^@^@^@M^@
^@^@^@=8d>^@^@^@M^@^@^@^@^@^@f^A^@^@^@^@^@^@Q^@^@^@^@M^@^@^@
n^A^@^@^@=85>^A^@^@^@^@^@^@f^A^@^@^@^@^@^@Q^@^@^@^@M^@^@^@n^@
A^@^@^@=85>^A^@^@^@^@^@^@M^D^@^@^@=84>^D^@^@^@Q^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@

```

(a)

```

19      /PATH/Tachftp/privsock.c          705
20      /PATH/Tachftp/tunable.h          1
21      /PATH/Tachftp/parseconf.c       508
22      /PATH/Tachftp/str.h              1
}
source_file_remarks {
}
clone_pairs {
81      1.358-381          13.141-164
81      1.358-381          13.366-389
46      1.1113-1156       13.121-164
81      1.1133-1156       13.366-389
46      13.121-164        1.1113-1156
81      13.141-164        1.358-381
81      13.366-389        1.358-381
81      13.366-389        1.1133-1156
}

```

(b)

```

2  <clone id="46">
3  <file>
4  <path>/PATH/Tachftp/sysutil.c</path>
5  <startline>18</startline>
6  <endline>22</endline>
7  </file>
8  <file>
9  <path>/PATH/2016-ITCN-FTP-Proxy-Template/ftp_proxy.c</path>
10 <startline>214</startline>
11 <endline>218</endline>
12 </file>
13 </clone>

```

(c)

Figure 7: .ccfd file (a),the first translated file(b), and the second translated file(c))

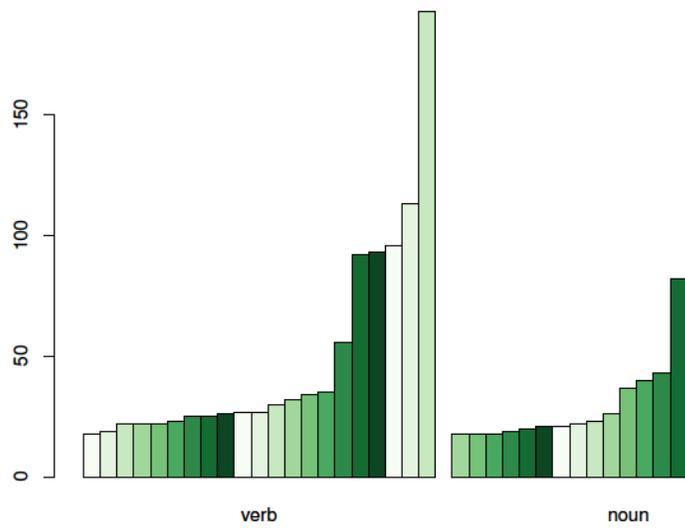
the clone segments and the source code needs to be established. This second module is designed for this purpose by automatically retrieving a clone code segment from the source code according to the result of *CCFinder*.

In addition, this module also provides probable predictions of potential vulnerabilities involved in the clone segments based on the function names, as detailed in the following. Through examining the *CVE*(Common Vulnerabilities and Exposures) database, we have discovered that software developers tend to use meaningful function names, either by combining several English words together with underscore or by applying Camel Case [20] standard to name the parameter. Certain keywords inside the function names may indicate the type of potential vulnerabilities to which the function is prone. Therefore, the function names will provide clues to the types of potential vulnerabilities introduced inside the clone segments, which is the useful information to users.

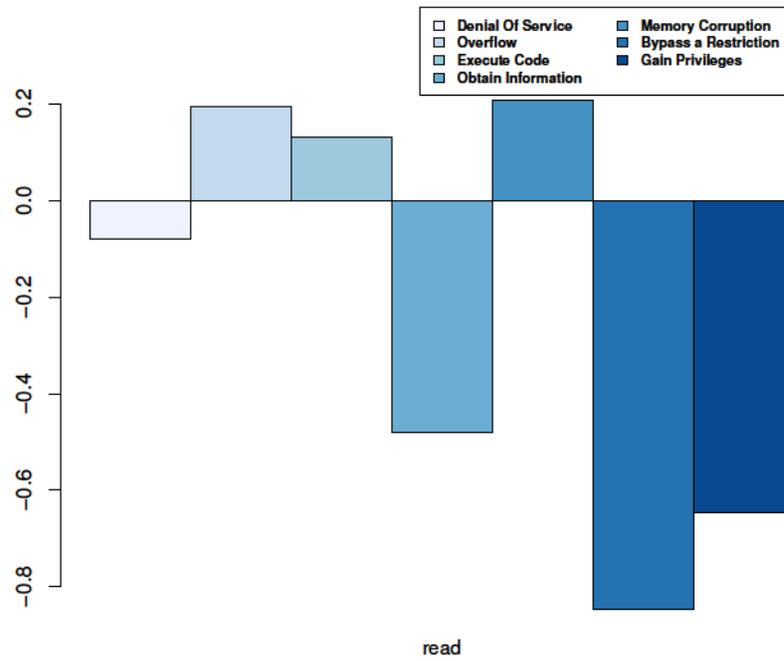
Therefore, we have obtained all the function names related to the C library from *CVE* [10] entries dated from 2009 to 2017. Figure 8a presents the top 36 most frequent keywords (minimum 18 repetitions) from the vulnerable functions. Figure 8b depicts the vulnerability distribution of the most-repeated keyword *read*, which has been repeated up to 221 times among 1867 functions; the x-axis indexes are ordered by the number of vulnerabilities in each category.

Let N and n be the total number of vulnerabilities among all experiments and that for one keyword, respectively, we call $p_k = \frac{n}{N}$ the zero-preference probability. Meanwhile, let N_c and n_c be the total number of vulnerabilities in one category among all experiments and that for one keyword, respectively, and let $p_c = \frac{n_c}{N_c}$. We use the difference between p_k and p_c to indicate the preference for the keyword where a positive number means the keyword represents a positive preference with respect to this type of vulnerabilities, and vice versa with negative number.

Figure 8b indicates that the *read* keyword is prone to be associated with buffer overflow, executing arbitrary code, and leaking information-related vulnerabilities. This is reasonable since *read*-related functions intuitively indicate certain relationship



(a)



(b)

Figure 8: Keywords Distribution (a) and The *Read* keyword Vulnerability Distribution (b)

Keywords	Variation
read	read:193→ readbody:10→ readwrite:2→ readcdir:2→ readdir:2→readbox:1→ reader:1→ readexter:1→ readfp:1→ readgifim:1→ readline:2→ readme:1→ readobject:1→ reqdrequ:1→ readstr:1
decode	decode:113→ dec:11→ decompress:11→ decrypt:7→decoder:6→decomp:2→decoded:1→ decodepkt:1→ decodeupdate:1
get	get:93→getaddrinfo:3→ get32/64:2→ get8bim:1→ getarena:1→ getautomntbyname:1→ getbands2:1→ getbits:1→ getbuf:1→ getcmap:1→ getcode:1→ getcompparms:1→ getdata:3→ getline:1→ gets:1→ gettoken:1→ getvaluebyclass:1→ getword:1
parse	parse:96→ parser:2→pareswf:2→parse8bim:1→ parsectrl:1→ parseicon:1→ parserr:1
dissect	dissect:92→ disst:2
process	process:57→ proc:3
write	write:34→ writel:1→ writeprolog:1→ wrlck:1→ ws:1→ wstr:1→ wstring:1
create	create:32
load	load:30→ loadcode:1→ loadexponentialfunc:1→ loadit:1

Table 3: The Keywords and Variations

	Required			Output			
	Source Code	Main Function	Pre-process	Function Name*	Function Line Number*	Call Graph	Function Definition
cflow	✓	✓		✓		✓	
pycparse	✓		✓	✓	✓		✓
Ctag	✓			✓	✓		✓

Table 4: The Comparison of Different Tools

with buffer overflow and information leakage caused by performing I/O actions. In addition, the code execution in C language is more likely related to vulnerable input validation, thus resulting in buffer overflow. Some vulnerability types which contain only small amounts of results, e.g., directory traversal, http response splitting, SQL injection, and cross site scripting, are not considered in our experiments to avoid misleading results. A more detailed discussion of the findings is given in Section 5.5.

To extract the function names from source code, we have compared several tools for generating and capturing such information in Table 4, as detailed below.

- GNU *cflow*, a flow graph generator that is part of the GNU Project [55], analyzes C source files and outputs the dependencies within functions with the *main()* function as the entry source [55]. The disadvantages of applying this tool in our case are threefold. First, the call graph from *cflow* reveals only the functions statically invoked either directly or indirectly via the *main()* entry. This means the runtime functions are missed completely. Second, the tool cannot parse customized libraries without the entry point of *main()* function inside. Third, the line numbers of functions are missed in the result. The final

call graph generates the dependency relationships between functions without any line information, which is needed in *Dupsec*.

- *Pycparser* is a parser for the C language, written in pure Python [14]. *Pycparser* works under Linux and is easy to install. It is also frequently recommended for parsing C programs. This tool simply parses the source code and generates the result with function names, line numbers, and definitions. However, *Pycparse* requires the compliable source code as the input. In this case, the C compiler needs to be invoked as the preprocessor before applying *Pycparse*. *Pycparse* requires the C compiler to handle the directives, remove comments, and complete other tasks related to the compiling process before parsing the source code. It is evident during our experiments that this requirement for preprocessing significantly reduces the applicability of *DupSec*, which is designed to process any code segments, whether compliable or not.
- *Ctags* parses the source code and generates complete results with function names and the corresponding line numbers [11]. The advantages of *Ctags* in our application are twofold. First, *Ctags* allows one to take non-compliable code segments as inputs and a pre-processing step is not required, which increases the applicability of *DupSec*. Second, the results obtained from *Ctags* contain function names, function line numbers, and the function definition, which provide a perfect collection of information for generating function names corresponding to clone segments in *DupSec*.

In the end, we have chosen *Ctags* and integrated it into *DupSec*. We parse the output of *Ctags* and store them in the MySQL database under columns ‘line number’, ‘function name’, ‘file path’, etc. Those information is stored for later analysis purposes. The function names will later be used to establish statistical correlation between keywords and vulnerability types, which may provide helpful clues for developers who may be more interested in specific types of vulnerabilities. Additionally, this module with its outputs including function names could be used as an API to

support call-graph related clone detection. Finally, we have used function names as an indicator for common vulnerabilities across software with different functionalities in our experiments as detailed in later sections.

4.1.3 The Visualization and CAS Calculation Module

The visualization module generates the results of clone segments while mapping them back to source code. The results include clone ID, file path, function name, clone segment, start line number, and end line number. Figure 9 provides an example output from *DupSec* corresponding to the source code segments mentioned in Section 3. The visualized output has been 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 *CAS* (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 [46] (*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. Vulnerabilities are usually associated with such input and output functions, e.g., function *memcpy()*, which could take user inputs as the source, and copy them directly to the memory block pointed to by destination. This function has caused many serious security flaws, such as CVE-2014-0160, i.e., the Heartbleed bug [9]. Thus, this kind of functions are considered as entry points in our experiments. As another example, in Figure 9, the vulnerable I/O function being shared among software applications is *strcpy*. The final *CAS* value is calculated based on the I/O functions shared among all software applications. To support further analysis, we have implemented an additional feature to store all the results into a database.

```

1 <clone id="8">
2   <file>
3     <path>./ssh-multithread/src/machine.c</path>
4     <startline>11</startline>
5     <endline>14</endline>
6     <funcname>machine_create_new</funcname>
7     <content>
8       strcpy(machine-&gt;host, host);
9       machine->user = malloc(strlen(user) + 1);
10      strcpy(machine-&gt;user, user);
11    </content>
12  </file>
13  <file>
14    <path>./C-Webserver/Webserver/reqhead.c</path>
15    <startline>142</startline>
16    <endline>144</endline>
17    <funcname>Parse_HTTP_Header</funcname>
18    <content>
19      else if (!strcmp(temp, "REFERER")) {
20        reqinfo->referer = malloc(strlen(buffer)+1);
21        strcpy(reqinfo->referer, buffer);
22      </content>
23    </file>
24    <file>
25      <path>./ssh-multithread/src/machine.c</path>
26      <startline>19</startline>
27      <endline>21</endline>
28      <funcname>machine_create_new</funcname>
29      <content> {
30        machine->
31        pass=malloc(strlen(pass)+1);
32        strcpy(machine->pass, pass);
33      </content>
34    </file>
35    <io>strcpy</io>
36  </clone>

```

Figure 9: An Example of DupSec Output

4.2 Automated Analysis Using *DupSec*

As we have mentioned before, *DupSec* is an automated tool to identify the common attack surface among software applications, which may lead to potential vulnerabilities shared among software applications. The analysis using *DupSec* is implemented through a procedure for systematically performing pair-wise comparison between given software applications.

Algorithm 1 Automated Analysis

Input: A set of software applications S

Output: The *XML* reports containing detailed *CAS* information

- 1: **Let** $S^2 = S \times S$ (the set of all pairs in S)
 - 2: **for** each $\langle s_1, s_2 \rangle \in S^2$ **do**
 - 3: Let $Clone(s_1.s_2)$ be the result of the **clone detection module**
 - 4: Let $T(Clone(s_1.s_2))$ be the result of the **translation module**
 - 5: **Generate** intermediate *XML* report
 - 6: **end for**
 - 7: **for** each $T(Clone(s_1.s_2))$ **do**
 - 8: **Extract** cloned code and function names from the source code
 - 9: **Compare** common I/O functions
 - 10: **Compute** *CAS* values
 - 11: **Append** the result to the *XML* report
 - 12: **end for**
 - 13: **return** all *XML* reports
-

The procedure shown in Algorithm 1 shows how *Dupsec* will apply the three modules to each pair of software applications in order to generate the corresponding *CAS* results. With N software applications given as the input, the procedure will be performed over $N * (N - 1)/2$ pairs and generate the same number of *XML* reports with detailed *CAS* results also saved into the database for further analysis.

To make the tool more flexible and user friendly, we have implemented two additional modules, a configuration module and a logging module. The configuration module is designed to store all the configuration information, e.g., the input and output paths and the directories of different components and tools, in a *.ini* file, which is used to control the way the tool will perform various tasks. This centralized configuration module allows any change made to the directory structures or files to be easily accommodated without having to modify the tool itself.

Another useful module is the logging module. In this module, we imported the Python package *logging*, which is a standard library that enables other Python modules to participate. By formatting the output of logs, the normal operation of the tool can be maintained without interruption even if it encounters errors or warnings. In addition, the operator can trace back to the problems later on, by searching the log files. The log entries may have five different levels of severity, including *DEBUG*, *INFO*, *WARNING*, *ERROR* and *CRITICAL*, which are detailed in Table 5.

DEBUG	Detailed information
INFO	Information that the software is working as designed
WARNING	Something wrong, but the system still working
ERROR	Some functionality cannot be performed as designed
CRITICAL	The software cannot continue processing

Table 5: Logging Parameters of *DupSec*

In addition to the level of severity, our log module also records the processing time and the current status of the processing, which is helpful when running large scale experiments. By reading entries at the *INFO* level, the operator can understand the status of the tool, e.g., whether the tool is still running normally or it has run into some infinite loops.

Chapter 5

Experimental Results

This chapter presents experimental results on the common attack surface of real world 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 experiments. At the beginning, we manually downloaded software applications from several open-source repositories, such as *SourceForge* and *Github*. This approach was time consuming and the number of software applications obtained was limited.

To address this challenge, we decide to develop a script to automatically parse the download links from the open-source software hosts. Our research shows that *GitHub* [18] 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 information such as the author name, updating date, size, and categories. The result of each query is returned on multiple pages with each page containing 30 result entries. Based on this information, we have developed a script using the API. 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. For example, our customized search link for database servers is `https://api.github.com/search/repositories?q=db+language:c&sort=stars&order=desc` which returns the database servers written in C language sorted by the stars, i.e., the popularity of the software applications, in descending order. 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. We complete the collection of software applications after the script finishes parsing all the download links and storing them in the local directory.

Our experimental environment is a virtual machine running Ubuntu 14.04, with the Intel core i3-4150 central processing unit and 8.0GB of RAM. We have applied our tool to totally 293 different software applications belonging to seven categories. The number of software applications in each category is as follows: 32 Databases, 62 Web servers, 25 SSH servers, 79 FTP servers, 41 TFTP servers, 6 IMAP servers, and 48 Firewall servers. Those amount to totally $\binom{293}{2} = 42778$ pairs of software applications tested using our tool in our experiments.

In the remainder of this chapter, we first validate our models through the study of correlation between common attack surface and vulnerabilities in Section 5.2. Then, we apply *DupSec* to study the existence of common attack surface between different categories of software applications in Section 5.3, and the trend of common attack surface within the same category in Section 5.4. In the end, we study the implication of function names in Section 5.5.

5.2 Common Attack Surface and Vulnerabilities

Although the concept of attack surface is not intended as a one-to-one mapping to actual vulnerability, the size of attack surface can provide a rough indicator for the abundance of vulnerabilities, since entry and exit points represent the interfaces exposed by the software for accepting inputs from (or sending outputs to) the outside environment. Consequently, the existence of common attack surface between two software applications may also indicate shared vulnerabilities. Therefore, we study the correlation between the two in this section. Seven categories of software applications are downloaded and used in the experiment, and our results reveal that common attack surface exists not only inside each category of software applications but also between those with different functionalities.

To evaluate the correlation between common attack surface and vulnerabilities, we examine pairs of software applications with respect to the results of a vulnerability scanner called *Flawfinder* [54]. *Flawfinder* is an open-source tool that can be used to scan C and C++ source code and report potential vulnerabilities [54]. It is regarded as an effective tool for detecting misused functions with ranked risks. For the purpose of verifying the relationship between common attack surface and vulnerabilities, we manually compare the *Dupsec* outputs with the results of *Flawfinder*.

Our findings indicate that common vulnerabilities may indeed be correlated with common attack surface. For example, we examined the two software *SSH* and *simple-webserverche*, which are of the category *SSH* and *Webserver* applications, respectively. In *SSH*, the main file uses function *strcat* to copy data to an internal parameter *user_name*, and those data are applied without any boundary check. The same thing happens in the application *simplewebserverche* where a file named *server.c* calls function *handle_response()* to apply function *strcat*. The source parameter *curr_time* is applied before the boundary checking. Our tool *Dupsec* detects these code fragments as a common attack surface, while *flawfinder* reports that both have the potential to lead to similar *buffer overflow* vulnerabilities.

Servers	Percentage
FTP	4.07%
FireWall	3.74%
DB	3.40%
WebServer	4.08%
SSH	3.37%
TFTP	3.10%
IMAP	6.52%

Table 6: Percentage of Detected Vulnerabilities Which Correlate to Reported Common Attack Surface

The above example demonstrates that common attack surface reported by *Dupsec* may indeed correlate with similar vulnerabilities detected by *Flawfinder*. To evaluate the extent of such correlation, we compare the outputs of *Flawfinder* and *Dupsec*, and Table 6 shows the degree of correlation between common attack surface and vulnerabilities.

As revealed in Table 6, in every category of software applications, there exist certain percentage of vulnerabilities which correlate to the common attack surface reported by our tool. The results also show that, although not every attack surface is vulnerable, they correspond to a superset of the detected vulnerabilities. In the following sections, we show studies on the common attack surface of software applications between different categories and within the same category, and we show how the common attack surface relates to various factors, such as the category, the publish time, and the size of the software.

5.3 Cross-Category Common Attack Surface

In this section, we study the two-proposed common attack surface measures through a study of real world software applications which covers 42778 distinct software application combinations (pairs) using our tool, *Dupsec*. 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 *Dupsec* by the size of the software applications. The normalization factor used for the cases where software application A is to be compared to software application B (the common attack surface to be normalized is $cas(A|B)$) is denoted as A^B . On the contrary, when software application B is to be compared with software application A to calculate $cas(B|A)$, then the normalization factor is denoted as B^A . Since the sizes of both software are typically quite large, we have used $\log_{1000}(A^B)/1000$ and $\log_{1000}B^A/1000$ in our experiments based on our experiences (except for cases where we explicitly mention the absolute value of common attack surface is used).

Results and Implications: Figures 10 through 16 show 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 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.

In summary, the results across all categories are shown in the heat map in Table 7 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.

After understanding the general existence of common attack surface among the seven categories of software applications, we aim to study more specific trends in

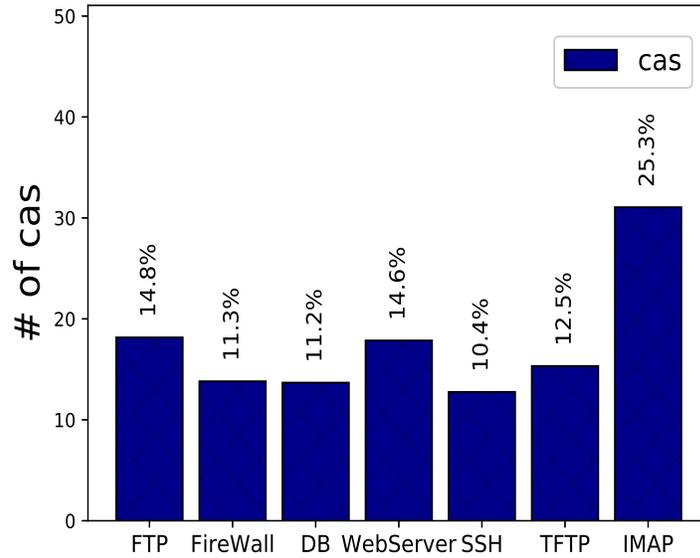


Figure 10: The Common Attack Surface between FTP and All the Categories

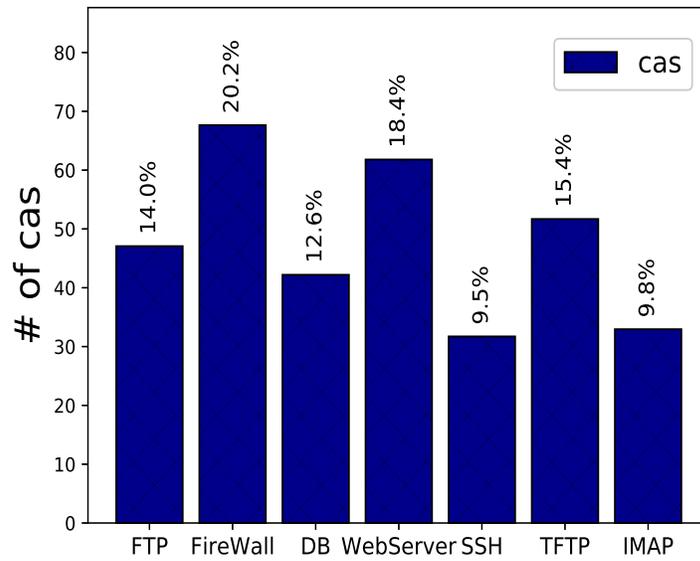


Figure 11: The Common Attack Surface between FireWall and All the Categories

our second sets of experiments. We first examine the distribution of absolute values of common attack surface to study the effect of normalization. Second, we use the

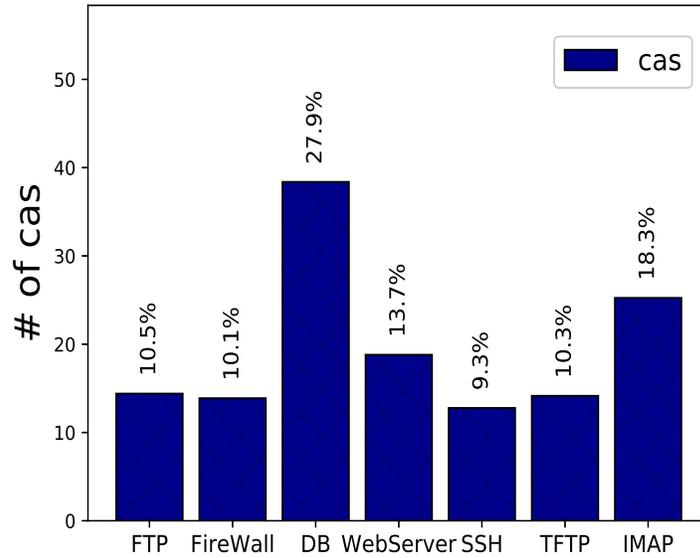


Figure 12: The Common Attack Surface between DB and All the Categories

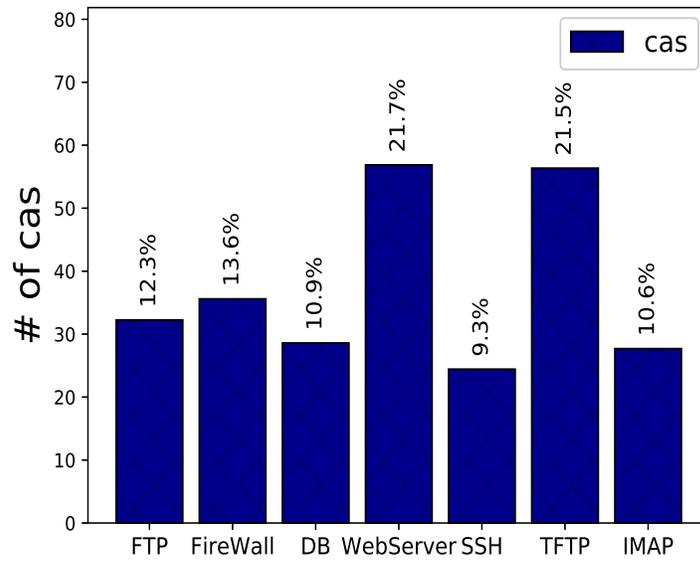


Figure 13: The Common Attack Surface between WebServer and All the Categories

normalized results to obtain the trend of common attack surface in the sizes and time (i.e., the year of publishing) of the software applications, respectively. Finally, we

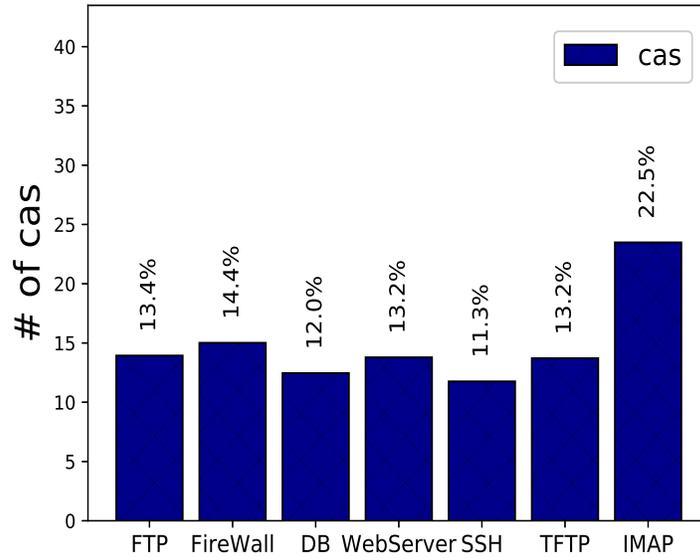


Figure 14: The Common Attack Surface between SSH and All the Categories

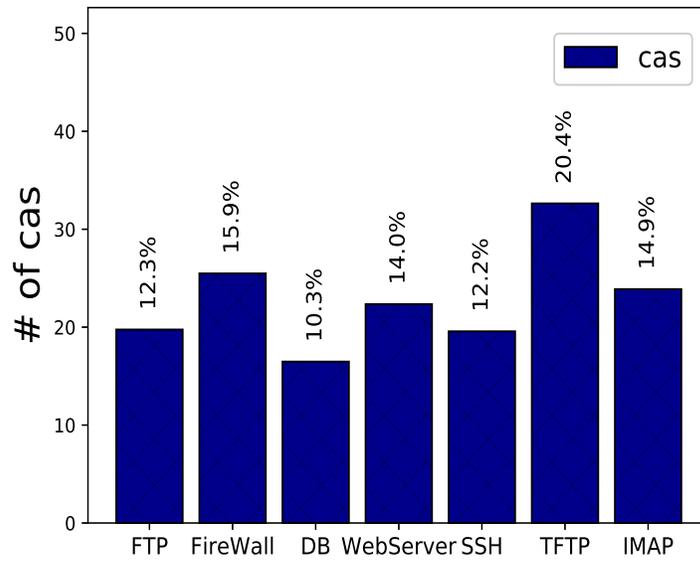


Figure 15: The Common Attack Surface between TFTP and All the Categories

study the trend of our second measure, the probabilistic common attack surface.

Results and Implications: Figure 17 shows the accumulated number of pairs of

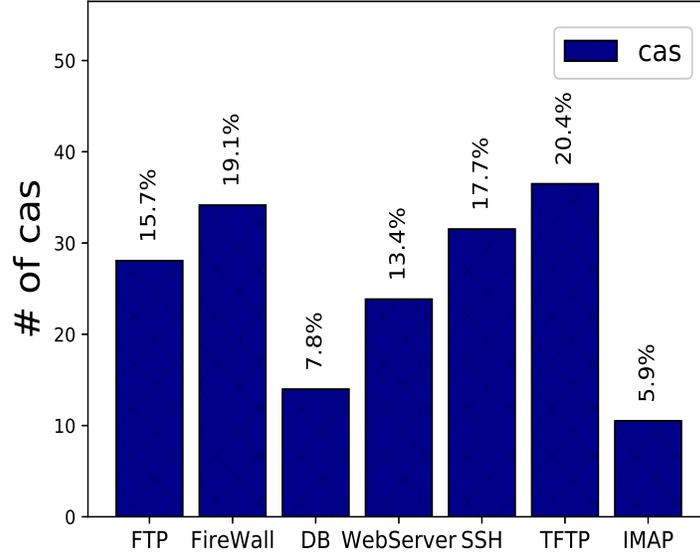


Figure 16: The Common Attack Surface between IMAP and All the Categories

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. Afterwards the accumulation of software applications increases slowly until the common attack surface reaches about 27, and then the accumulation flattens out. Those results lead to following two implications.

	FTP	FireWall	DB	Web-Server	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

Table 7: HeatMap for Common Attack Surface in Different Categories

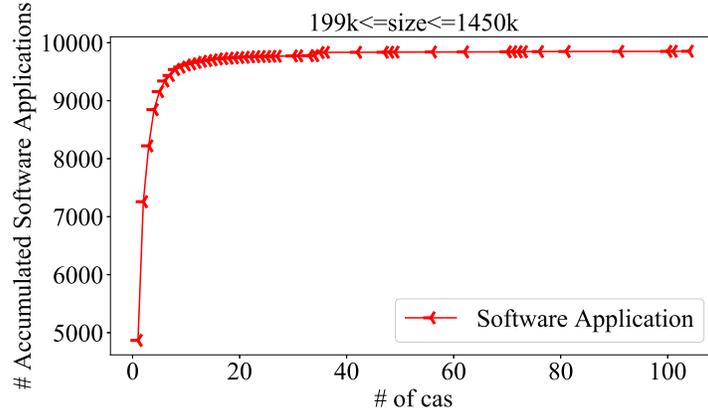


Figure 17: The Common Attack Surface in Accumulated Pairs of Software Applications

First, the existence of common attack surface between different software applications seem pervasive. Among all the combinations of software applications, about 20% share common clone segments, and 56% of the clone segments contain at least one common attack surface. The reason for this could be that I/O functions are commonly used by developers in the same or similar manner, either because those are established coding practices or due to code reusing. For example, between the two software applications, *cwebserver* and *simplewebservercart*, there exist a common function, *bind* invoked by other common functions, e.g., *open_clientfd*, *open_listenfd* and *prepare_socket*, which all contribute to the common attack surface.

Second, on the other hand, we should not expect to see a large common attack surface value between different software applications. From our experimental results, it is clear that the absolute value of common attack surface between different software applications remains relatively small, with the majority of the results to be under ten per pair of software applications. We believe this is reasonable since in most cases two different software applications are not supposed to share a lot of code. We note this result may be influenced by the parameters chosen in *CCFinder* to identify clone segments, e.g., a smaller size for the clone segments will result in more clone segments to be identified.

Figure 18 depicts the relationship between common attack surface and sizes of the

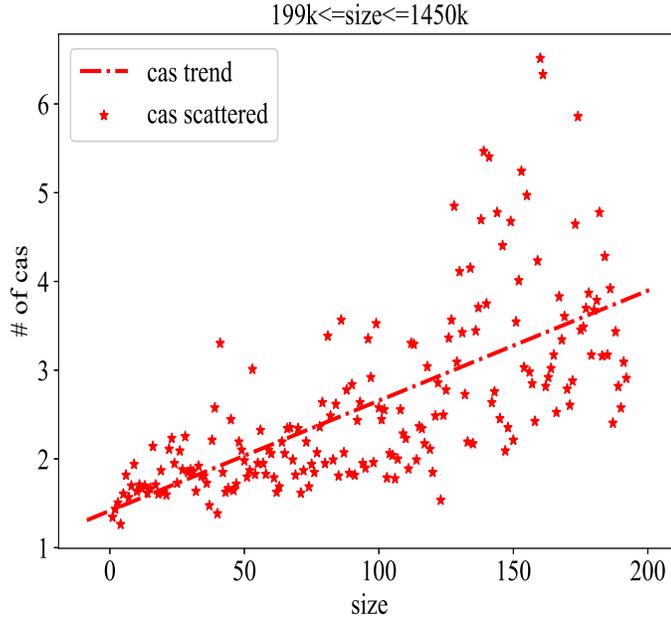


Figure 18: Common Attack Surface in Accumulated Software Application Pairs vs Size

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. The X-axis depicts the indices of sizes ordered by the normalization factor; index 1 indicates the smallest sizes, and index 200 indicates the largest sizes. We have presented both scattered and trending results in the figure. We can observe that, with increasing sizes of the software, the value of common attack surface generally increases. This is as expected since the attack surface is based on the number of I/O functions, which is roughly proportional to the size of the software.

Figure 19 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. Although the average number of I/O functions per software application does not have a simple trend over time (as far as the software applications tested in our experiments are concerned), we can use this as a baseline for comparing with

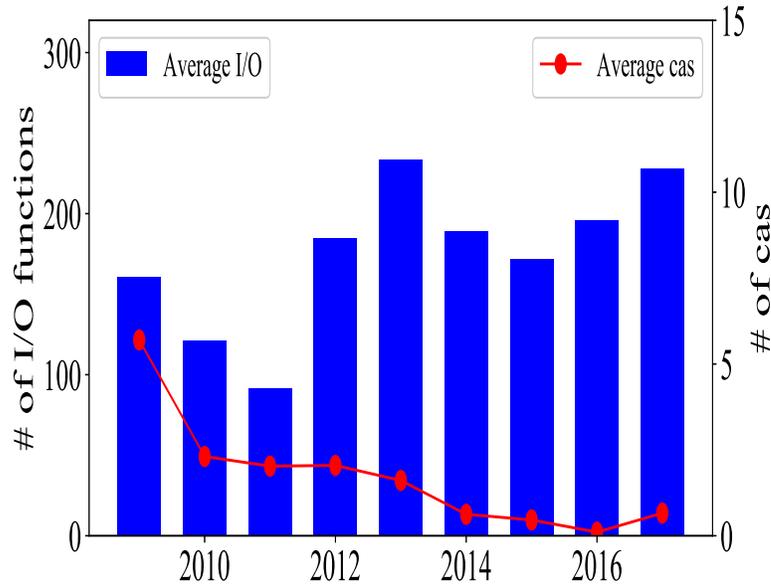


Figure 19: Common Attack Surface Trend in Years

the average value of common attack surface over time. 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 the code that has been written at an earlier time) which means older software will more likely share common attack surface with other software.

Figure 20 explores the trend of the probabilistic common attack surface measure versus the total number of I/O functions (called the size in the figure). Recall that the probabilistic common attack surface measure reveals the diversity between two software applications where a larger value indicates a lower diversity level. Although previously we have seen that the absolute value of common attack surface increases with the increase of the size, as shown in Figure 18, the value of the probabilistic

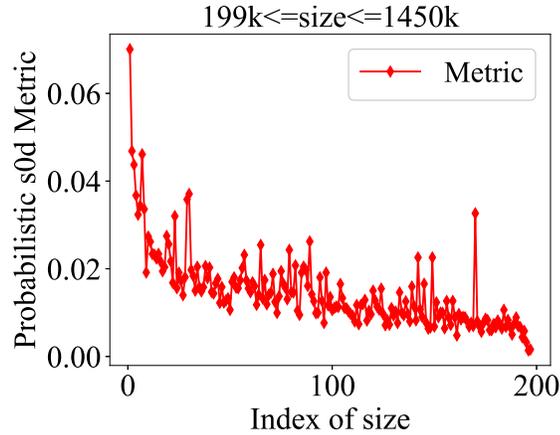


Figure 20: The Probabilistic cas measure

common attack surface measure actually decreases in Figure 20. The reason for this counter-intuitive result is that, the increase of the number of I/O functions in software applications is faster than the increase of common attack surface, so the value of the probabilistic common attack surface measure decreases with the increase of the number of I/O functions.

In fact, both Figure 18 and Figure 19 match the results of existing vulnerability discovery models (as reviewed in the related work section), 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 [1]) has 1,453 vulnerabilities over nine years [10], while Apache (with 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).

In the third set of the experiments, we aim to study the relationship between functionalities and common attack surface. Previously we have seen that common attack surface not only exist between software in the same category but also across different categories, as shown in Figures 10 through 16 and Table 7. Since the exact functionality of software applications is not easy to characterize automatically, we

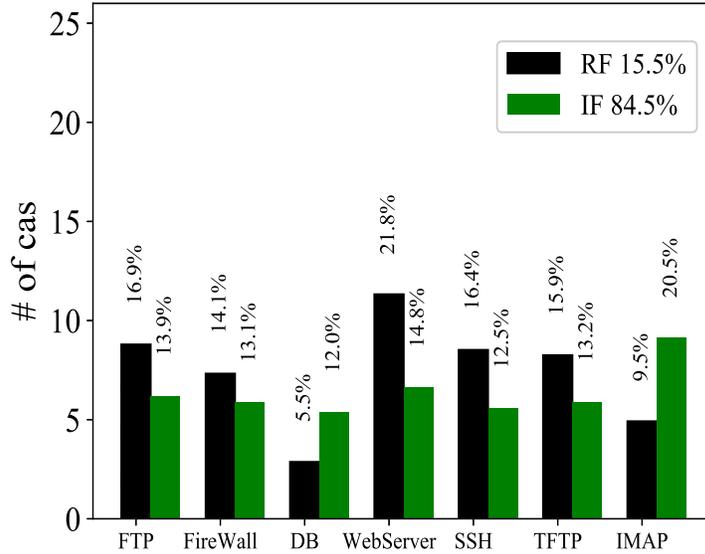


Figure 21: cas vs Functionalities in FTP with All the Categories

rely on the simple method of inferring it from keywords inside function names, as described in Section 4.1.2. Although this method may be inaccurate and misclassify the functionality of some software, it allows us to automatically extract results from a large number of software applications and correlate the results with common attack surface.

Results and Implications: Figures 21 through 27 shows the common attack surface of each category of software applications. In the figure, RF indicate the results for what we call software applications with relevant functionalities, which means the function names inside each pair of software applications contain some common keywords implying similar functionality (see Section 4.1.2), and IF means irrelevant functionalities (no common keyword exists in the function names). The percentages on top of each bar represents the average ratio of common attack surface in each category. We can see from the results that common attack surface does exist both between software with similar functionalities and between those with different functionalities. Although in most cases common attack surface is more likely to exist between software with

similar functionalities, the difference is not significant. Therefore, we can conclude that common attack surface is not necessarily related to similarity in functionality.

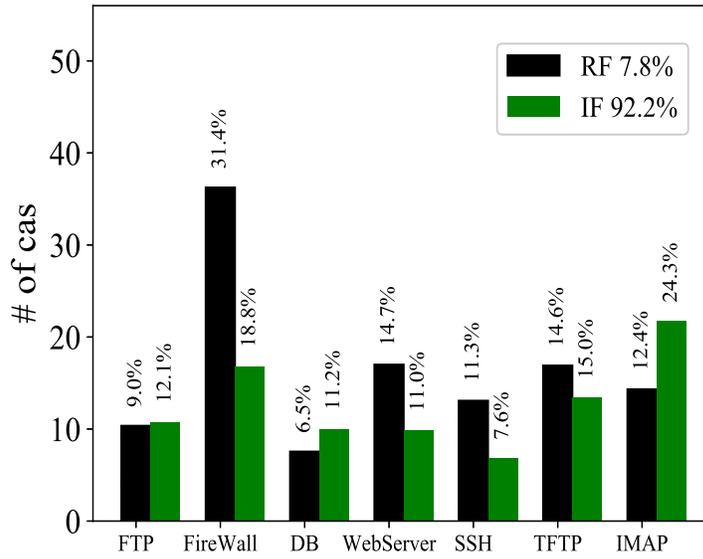


Figure 22: cas vs Functionalities for FireWall with All the Categories

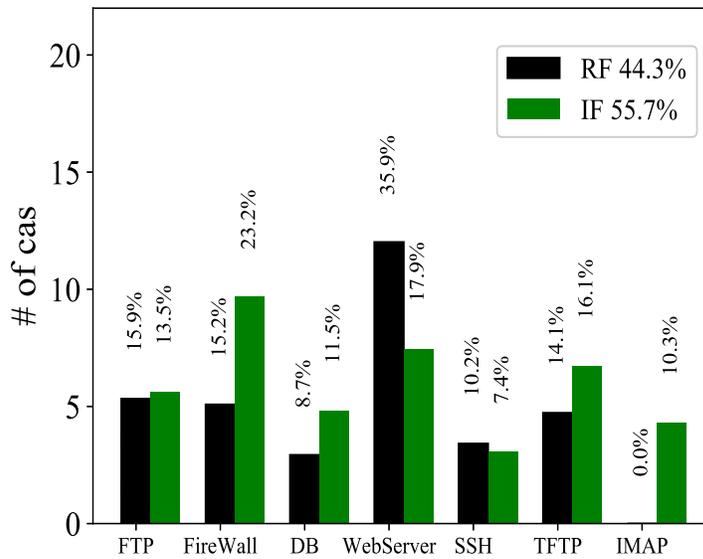


Figure 23: cas vs Functionalities for DB with All the Categories

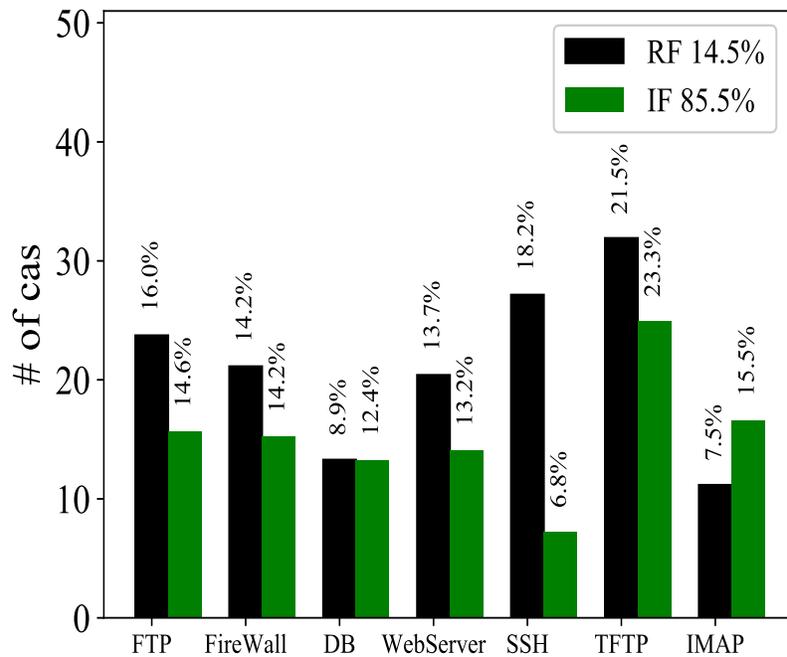


Figure 24: cas vs Functionalities for WebServer with All the Categories

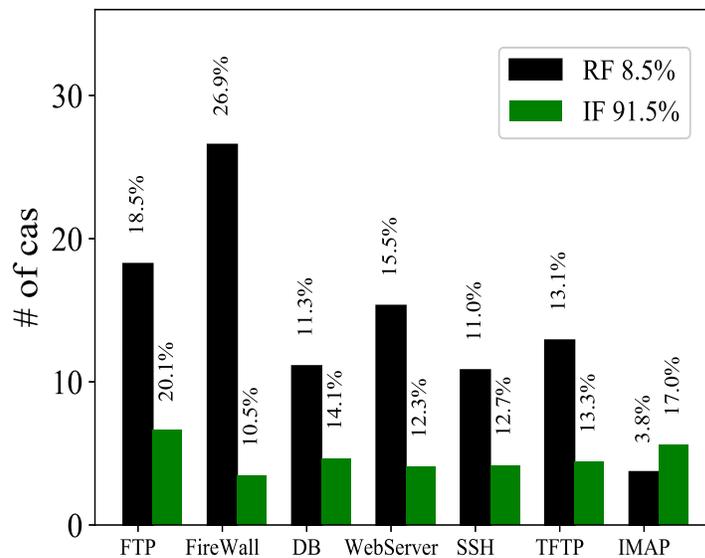


Figure 25: cas vs Functionalities for SSH with All the Categories

We have also studied the relative popularity of I/O functions among common attack surface to see which of those functions are most likely to result in common

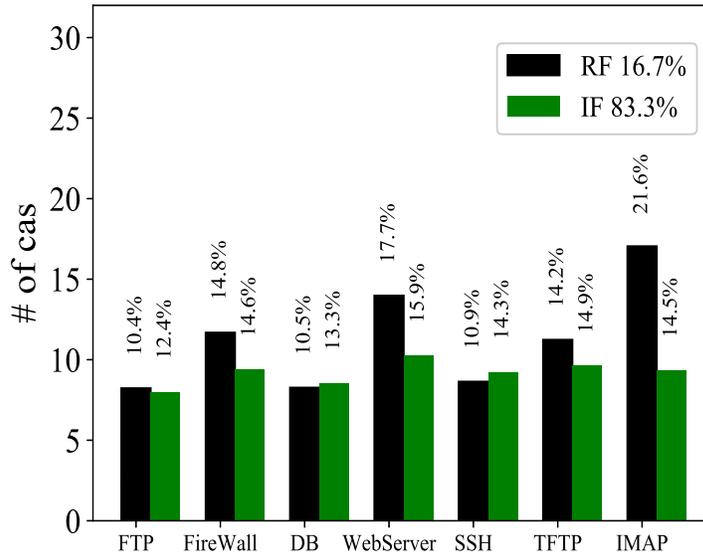


Figure 26: cas vs Functionalities for TFTP with All the Categories

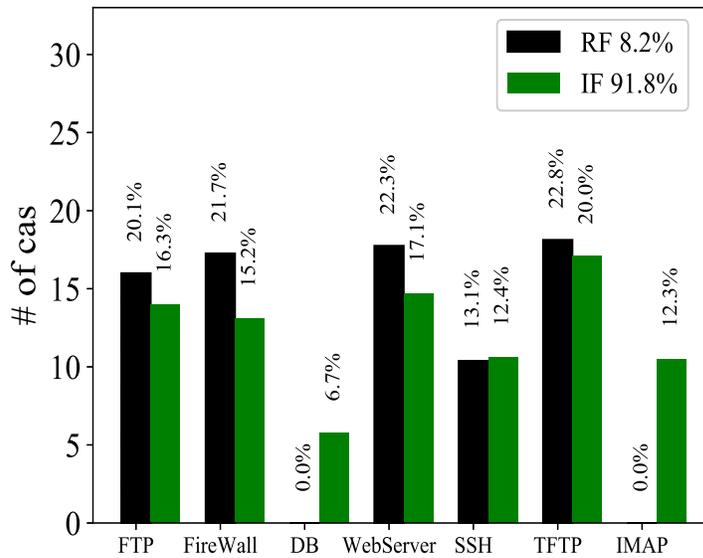


Figure 27: cas vs Functionalities for IMAP with All the Categories

attack surface. First, we rank for the I/O functions among all common attack surface to determine the top ranked I/O functions. Second, we show the I/O functions

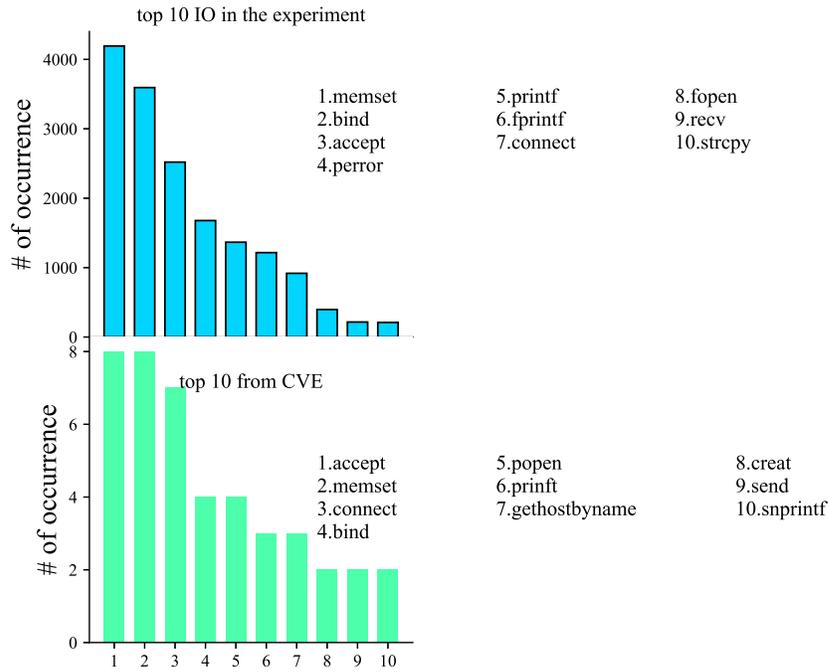
corresponding to different keywords to observe the relationship between I/O functions and various functionalities.

Results and Implications: As seen in figure 28, the I/O functions most frequently appearing in common attack surface are *memset*, *bind*, and *accept*. Among these, *memset* occurs most often because most of our software are server-related applications, which typically involves lots of copying and setting. For example, in the software application *webserma*, the function *start* contains code *memset(&serv_addr, 0, sizeof(serv_addr))*, which initializes the server addresses with 0s, which has the same pattern as the code in another software *dbasepeclphp7* and produces the common attack surface between the two software. The second and third most frequently appearing I/O functions, *bind* and *accept*, are the functions related to the socket; the *bind* assigns an address to a socket, and *accept* accepts a connection on a socket; *bind* is also corresponding to the most frequently used keyword, *create* 28(b). In Figure 28(b), the two representative keywords, *read* and *create* are chosen to show the corresponding I/O functions that are top ranked based on their occurrence in the common attack surface. It can be seen that different keywords have different correlated I/O functions.

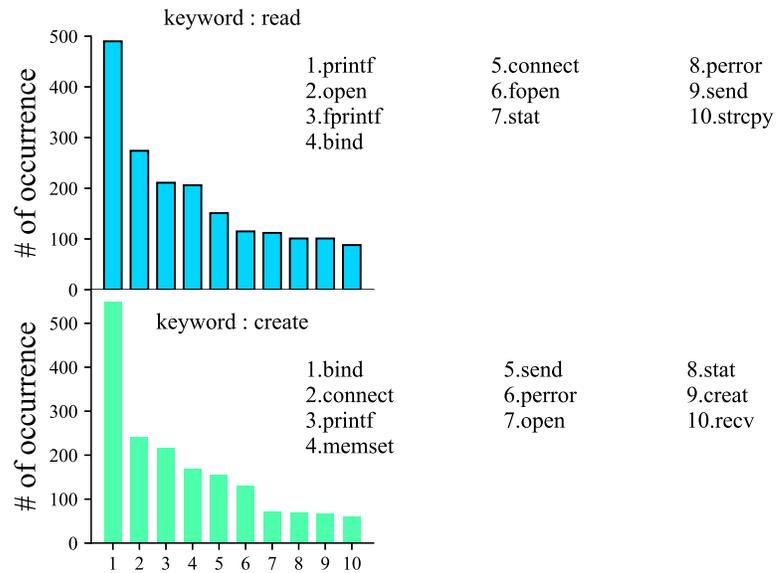
5.4 Common Attack Surface in the Same Category of Software Applications

We study the trend of common attack surface between software within the same category in this section. We study common attack surface in terms of sizes for the two categories *WebServer* and *FTP* in the first set of experiments. We compare the trend over time and the trend of the probabilistic common attack surface measure inside the same category. In the end, we study the common attack surface against functionalities inside the same category.

Results and Implications: Figure 29 depicts the common attack surface for different sizes of software in the category *WebServer*, represented in both scattered and



(a)



(b)

Figure 28: Top 10 I/O Functions (a) and CVE (b) for Different Functionalities

trending results. The orange scattered points and the dotted line indicate the result and the red dotted line is the same trend borrowed from Figure 18 for comparison, which corresponds to the overall common attack surface. The scattered purple points and the dotted line in Figure 30 is the result inside the *FTP* category. 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 7, i.e., *WebServer* has the highest percentage of common attack surface inside its own category, 27.1%, which is significantly higher than the common attack surface cross categories. On the other hand, in the figure 30, 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* (which can be seen from Table 7).

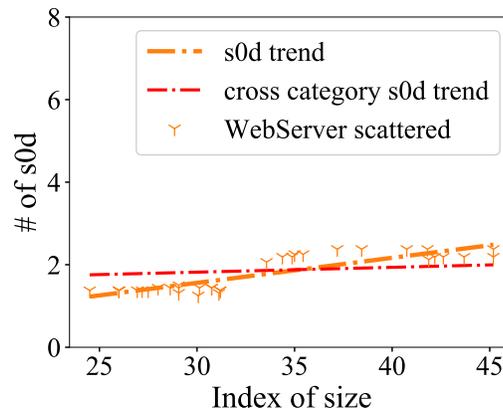


Figure 29: Common Attack Surface(s0d) vs Size for WebServers

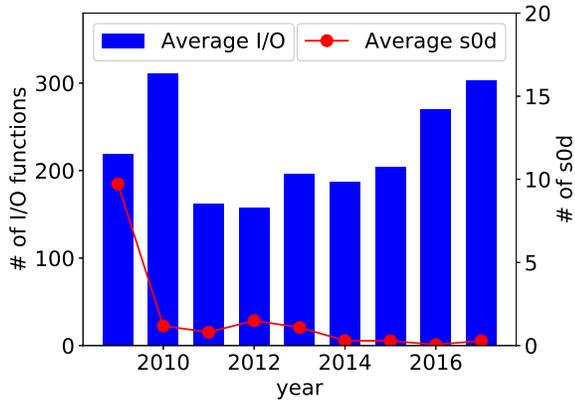


Figure 31: Common Attack Surface(s0d) vs Size over Time

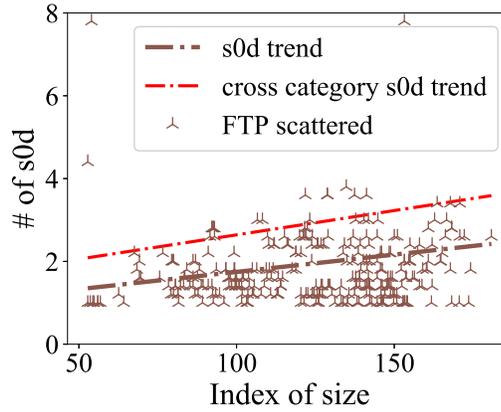


Figure 30: Common Attack Surface(s0d) vs Size for FTP

Figure 31 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 Figure 19, the common attack surface in the same category has higher values, which also match the previous observations. The trend over time matches the trend we have obtained cross categories, i.e., older software applications tend to have more opportunities to share common attack surface with other software.

Figure 32 reveals the trend of the probabilistic common attack surface measure versus the size in the same category, which shows a similar trend as the cross-category

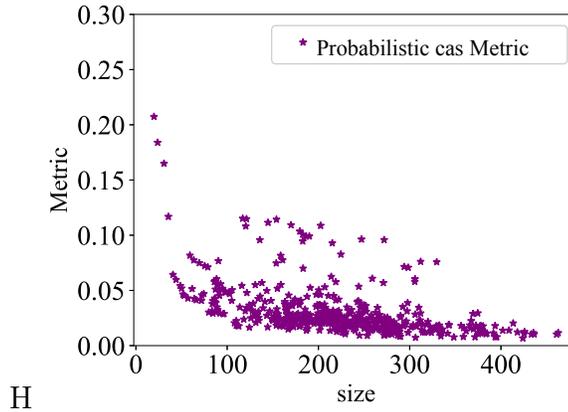


Figure 32: Common Attack Surface vs Size over Time in the Same Category

result as shown in Figure 20, although the trend within the same category starts from a higher value around 0.20 (in contrast, the cross-category measure starts from 0.06).

In Table 8, for categories such as *FTP*, *FireWall*, *WebServer*, *SSH*, and *TFTP*, the percentage value of common attack surface for software with similar functionalities is lower compared with the result for software with different functionalities. However, in Figure 33, we can see the absolute value of common attack surface for similar functionalities is much larger than that for different functionalities. To explain this, we use *FireWall* as an example. The reason is that several software applications in the *FireWall* category contain functions with random names, and they contain also lots of common I/O functions with software applications in other categories, which contributes significantly to the number of common attack surface.

5.5 The Distribution of Function Names

From the results obtained from *CVE*, we have discovered that different verbs in function names are related to different vulnerabilities. Figure 34 is generated based on the *CVE* data from year 2009 to year 2017. We have focused on the vulnerabilities related to *C* language. Seven most popular vulnerabilities are identified, with *Dential of Service*, *Overflow* and *Execute Code* as the top three. From the data we parsed, the

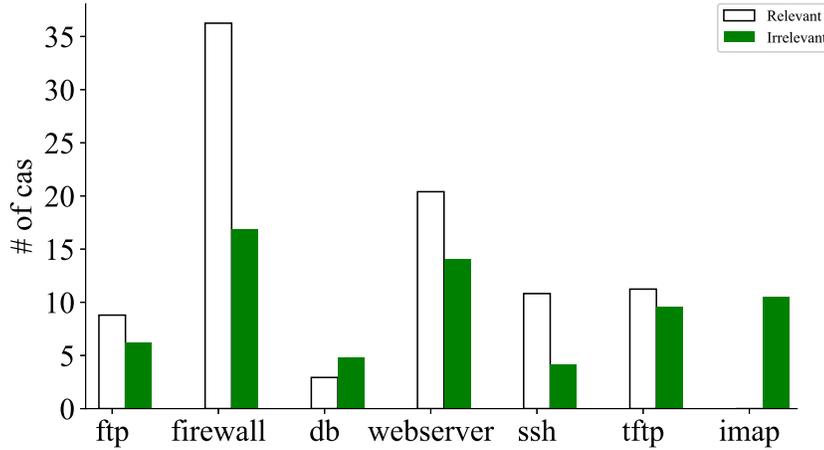


Figure 33: Common Attack Surface vs Funtionalities in Different Categories

	RFP	IFP
FTP	22.9%	77.1%
FireWall	12.5%	87.5%
DB	85.3%	14.7%
WebServer	24.3%	75.7%
SSH	4.1%	95.9%
TFTP	25.7%	74.3%
IMAP	0%	100%

Table 8: Percentage Value of Common Attack Surface

most frequently appearing keywords in function names are *read*, *decode*, and *parse*. For bars with positive values, it means that the verb has a probability to be related to the given vulnerability, and a higher bar indicates a larger probability. In addition, bars with negative values indicate the function containing this verb is irrelevant to the vulnerability. For example, the functions that contain *read* have a greater chance of having *Memory Corruption* vulnerability and less chance of having *bypass Restriction*. This is reasonable since the functions containing *read* works more with input

and output which can potentially cause *Memory Corruption*, such as *readbody* and *readdir*, as show in Table 3.

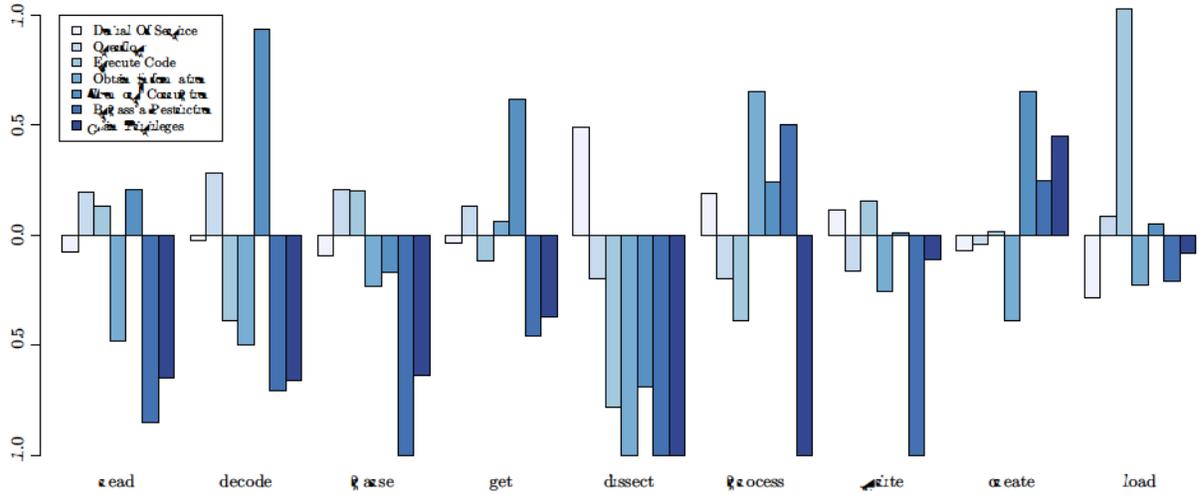


Figure 34: The Distribution of Keywords in Function Names

Chapter 6

Related Work and Background Knowledge

This chapter reviews related work and provides a brief introduction to some background knowledge.

6.1 Clone detection

Source code clone detection tools aim to find the cloned code segments between two given software applications. Existing research has led to extensive developments in the field of both clone detection tools and approaches; it should be noted that many of these tools are mainly for research purposes [47].

There are four main types of clone detection approaches as follows [47].

- Type I identical code fragments
- Type II structurally or syntactically identical fragments
- Type III copied fragments with further modifications
- Type IV detecting code fragments perform same function but implemented syntactic differently

The aforementioned categories usually correspond to different detection techniques, which will be briefly introduced in the following section.

Text-based clone detection Text-based clone detection is mainly based on analyzing source code as text or string. Thus, if two segments of string or text sequences are detected to be similar, the fragments are reported as clone codes. Because of the natural feature of the text-based approach, the structural characteristics of the given programming language is not taken into consideration. The clone detection process usually can be performed without normalizing the source code, as the process deals with the raw code of application. But there are still certain preparatory steps that must be applied prior to undergoing the detection process as follows. The first is removing the comments based on the rule of different languages, and the second is removing whitespaces including all forms of blanks.

For example, we will discuss one of the popular tools in text-based clone detection, *Dup* [3]. If two lines of code are identical after removing all whitespaces and comments, they are recognized as clone codes. The longest line that is detected to be matched are generated as the output, but the minimum length of the reported code can be customized according to different needs. For the purpose of obtaining the longest matches, *Dup* applies following two steps.

First, in order to assign identical integers, *Dup* uses lexical analysis to hash the lines. It then applies the string to the scatter plot; if the diagonals are found to be equivalent, the strings are detected. In order to implement this process, two algorithms are applied. The first requires a longer run-time, but takes up less space. The second one, based on suffix tree data structure, is more effective with a shorter run time; however, it requires three times more space than the previous one. *Dup* has two main problems. First, it fails to detect parameter names within code of the same logic; second, it cannot detect different coding styles, like the *for* statement and *do-while* statement.

Another well-known approach [23] is applying the fingerprint to identify the redundancy of a substring in the source code. There are two steps, first, calculating the fingerprint of substrings within the source code; then, sort the fingerprint value and remove the one that occurs only once. The fingerprinting calculation uses KARP-Rabin's string matching approach [26, 27] to calculate the length of all n substrings. In order to detect consecutive lines, a single, random letter are applied to replace each maximal sequence of alphanumeric characters. The entirety of the matching process consists of three steps: transforming source code to discard unimportant codes, dividing the source file into substrings, and identifying the matching substrings. If there are any near-missing clone codes, all whitespace must be removed, and the alphanumeric characters must be replaced by the letter 'i'.

Consider the following example:

```
for (parama=0; parama<bound; ++parama)
```

is transformed to

```
for(i=0;i<i;++i)
```

Also for the macro definition:

```
#define ABEF121
```

becomes:

```
#iii
```

These preprocessing steps are served to reduce many false positives. By applying these approaches, the file-level structure changes can be detected in different versions.

Ducasse developd [12] *duploc* which was designed to be a parsing-free, language-independent tool. It first reads the source file and sequences of the lines, second 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. The authors apply the Dynamic Pattern Matching

algorithm. Still, since *duploc* is a language-independent tool, it lacks the ability to obtain meaningful clone resolutions. While normalization does not usually apply in text-based detection, Ducasse [12] applies some transformation to the code, not limited to removal of whitespace and comments but removing any of the uninteresting language fragments. For example:

```
static int item =1;
void main(void){
    int count=1;
}
```

Might be transformed to

```
staticintitem=1
intcount=1
```

As indicated in the transformation process, the white spaces and uninteresting parts in the *main* function have been removed.

The last tool, DuDe [53] is another language-independent detection tool based on lines. It parses duplication chains to combine small exact clones. The detection process consists of three steps, which are code preprocessing, the scatter-plot population, and the duplication chains building. The program generates the strings by considering whether a current chunk can be considered valid, and an assessment made by measuring if the size is larger than the minimum threshold. If so, the program continues searching for the next piece, which is added under the condition it does not exceed the maximum limit.

Token-based clone detection Token-based clone detection is also one of the widely applied methods. In token-based clone detection, the process can be summarized into three steps.

- First, performing a lexical analysis in order to generate a sequence of tokens

- Second, using the sequence of tokens to make a comparison and find the duplicated subsequence [21]
- Third, matching the tokens back into the text and output the cloned result. The matching algorithms may be either suffix-tree or suffix-array matching algorithms.

One of the representative tools in token-based detection is *CCFinder* [25], which is applied in our work. The first step of the clone-detecting process is running a lexical analysis in accordance with the lexical rules of the corresponding language. Afterward, every line of the source file is translated into tokens. *CCFinder* detects clones by concatenating all the tokens into a token sequence, removing all whitespaces in the process. Once the sequence is generated, it is transformed by the rules of the programming language, and the identifiers are replaced by individual tokens. Upon completion of the transformation, the substrings are sent for matching-based detection. In order to achieve an efficient matching, *CCFinder* applies the suffix-tree algorithm. The clone location information is then expressed as a tree with nodes of shared identical subsequence. Clone pairs can then be identified by searching tree. Once the clone tokens are detected, a mapping process from token to clone code is required to obtain the source code. *CCFinder* can support multiple languages including *C++*, *C*, *JAVA*, *COBOL*, *VB*, *C#*. The following codes demonstrate the transforming process.

```

void print (const set<int>& num){
    int count =0;
    set<int>::const_iterator a= num.begin();
    for (;a!=num.end();a++){
        cout<<*a<<" ";
        cout <<count++<<endl;
    }
}

```

After applying the transformation rule, the code becomes:

```
void print (const set& num){
    int count =0;
    const_iterator A = num.begin();
    for (;a!=num.end();a++){
        cout<<*a<<" ";
        cout <<count++<<endl;
    }
}
```

Next, algorithms are applied to replace the parameters with tokens; in this example, the identifiers are replaced by the token $\$p$. Finally, the suffix-tree algorithm is applied to identify the token sequence of each substring.

```
\$p $p ($p $p& $p){
    $p $p =$p;
    $p $p = $p.$p();
    for (;$p!=$p.$p();$p++){
        $p<<$p<<$p;
        $p <<$p<<$p;
    }
}
```

Baker's 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 [32] develops a tool called *iclones* [19], which uses suffix-trees to find clones in abstract syntax trees that can operate in linear time and space. The detection approaches are completed by parsing codes, generating AST, and using the serialized AST as an input in the suffix tree detection.

CP-Miner [34] as a well-designed token-based clone detector, uses frequent subsequence mining algorithms to detect tokenized segments. The aforementioned tools, *CCFinder* and *Dup*, failed to identify the inserted code, since their code fragments break down the token sequence. *CP-Miner* does not have this problem, since its mining algorithm is able to identify clone codes with interleaving sequence gaps. RTF [5] 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.

Tree-based clone detection The tree-based detector usually transfers the code to an abstract syntax tree (AST) by applying the language-oriented parser. Because the tree contains the complete information of the source code, certain sized subtrees are applied into the comparison process. Similar subtrees are detected, and corresponding source code is returned as clone pairs. The incapability of detecting large-scale source code is the biggest shortcoming of tree-based detector. But it is capable of detecting inserted and deleted codes.

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 algorithm of this tool can be summarized into three steps; first, parsing and generating AST tree by the compiler generator tool; second, using the sequence detection algorithm to find the sub-tree clones; third, to locate near missing clones by a combination of other clones. The *ccdimpl* [44] developed by Bauhaus is similar to the *CloneDR* in the way of dealing with hash and code sequences. And instead of using AST, it applies IML algorithm in the comparing process.

David and Nicholas [17] 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.

Graph-based clone detection Program Dependency Graph(PDG)-based algorithm is a higher level of abstracting the code representation compared with other algorithms. The algorithm generates the semantic information through the process of data flow. One of the leading PDG-based tools is PDG-DUP presented by Komondoor and Horwit [29]

The PDG-based detector considers semantic information. Komondoor and Horwitz's PDG-DUP [29] is the leading PDG-based detection tool, which identifies clones together and keeping the semantics of the source code to reflect software. The algorithm has three steps. First, ignoring the name and the value of the parameter, separating and grouping the PDG nodes into equivalent classes; second, removing the subsumed clones; the last step is to combine those clones and use transitive closure to combine small groups of clone code into larger groups. The most significant limitation of the PDG-based tools is that they don't support large sized programs.

Metric-based clone detection In the Metric-based clone detection algorithms, instead of comparing the code, it compares metrics applied to code fragments. One of the techniques used is applying fingerprinting to generate metrics of classes or the functions within the source code. The clone codes are detected through the value of metrics.

In [37] Mayrand uses the tool *Darix* to generate the metric. The clone identification is based on four values, which are name, layout, expression and control flow [37]. In order to support different languages, after generating the AST, the information is translated into the Intermediate Representation Language (IRL) [2] which contains four categories of information. The first deals with the software architecture, the second about the software data type, the third the control flow of the software, and the last the flow of data. The clone is detected in pairs only.

Kontogiannis [30] uses Markov models to compute the dissimilarity of the code by applying the abstract pattern matching. To improve it, in [31] Kontogiannis proposes two methods for clone detection. The first is the direct comparison of metric values,

the comparison is at the level of *begin-end*. The second is based on a dynamic programming algorithm; the comparison is on the statement level of *begin-end* block, and the result is more accurate with less false positives. Five widely used metrics are applied in the direct comparison [31], which are the number of functions called (fanout); the ratio of input/output variables to the fanout; the McCabe cyclometric complexity; the modified Albrecht's function point metric; the modified Henry-Kafura's information flow quality metric. The similarity is calculated based on those five-dimensional vectors. On the other hand, the dynamic programming works on the abstract feature sets. The features are consisted of 1. variables and its definition; 2. definitions of data types; 3. the previous five metrics.

Hybrid clone-based clone detection Besides the detection technique mentioned above, there are some other approaches that using hybrid clone detections.

In [32], the authors apply the suffix trees to find clones in AST; this approach can find clones in linear time and space. The algorithm they applied can be summarized into four steps [32]: first, parsing the source code to generate the AST; second, serializing the AST tree; third, applying the suffix tree detection algorithm on the clone code; forth, decomposing the result into the syntactic units. The comparison process is based on the tokens of the AST-nodes.

6.2 Attack Surface

The attack surface is the subset of a system's resources that can be used by attackers to exploit the system [36]. Since it is challenging to measure security on a quantitative and qualitative basis, the authors formalize a notion of the system's attack surface in order to more accurately indicate a system's security.

The attack surface can be accessed through three different sources: entry/exit points, channels, and untrusted data items. Thus, the subset of attack surfaces remains the most likely point of potential attacks, as actions can only be launched by

sending data to the system or receiving data from the system. However, not all the attack surface can be used by the attacker, thus the contribution of resources cannot be measured equally. The definition of the resource damage potential defines how serious an attacker can damage a system or how much effort it takes for the attacker to get the corresponding rights to do the attack.

In addition, damage is measured from several aspects, including method privilege, access right, channel protocol, and data item type [36]. For example, consider this basic formula when measuring the attack surface: if A has a larger attack surface than B (with other condition equivalent), then any attack working for B would also work for A. The quantitative measure can be used to measure the absolute attack surface. The detailed method is to estimate the total contribution of all the three vectors, which are the methods, the channels, and the data items.

6.3 Software Diversity

Software diversity has long been employed for security. The model in [41] gives a threshold to measure the software diversity's capability of resisting the viral propagation. The authors in [42] studies the enhancement of the computer system's reliability by exploitation of software diversity, because a diversified system which composed of multiple alternative versions is more reliable than the system with any single version alone, although it is not clear how the diversity among a collection of versions can be optimally achieved. The authors in [35] define and formalize the concept of software diversity from four different points of view which are: structural diversity, fault diversity, tough-spot diversity, and failure diversity; they proposed a way to quantify software diversity and applied it during the life cycle of NVS. The authors in [40] present the distributed algorithms which can be used to assign software packages to different systems thus to analyzes their performances. Their goal is to put limitation on a malicious node from compromising its neighboring nodes by using the same approach, thus to protect the whole network. Unlike those excising works, in our work

we propose a new approach to defining software diversity from the security point of view by measuring diversity in terms of common attack surface.

Chapter 7

Conclusions and Future Work

In this thesis, we have made four main contributions.

- First, we defined software diversity based on the novel concept of common attack surface.
- Second, we have implemented a new tool *Dupsec* which can perform the common attack surface calculation automatically.
- Third, we have conducted experiments on real software applications and examined the common attack surface from different dimensions, and analyzed its relationship with vulnerabilities.
- In the end, we have also analyzed and found that function names have certain indications towards vulnerabilities.

However, our work still has some limitations which will lead to our future work. We briefly summarize some future directions as follows.

- First, we borrow the clone code detection tool *CCFinder*, which requires significant time and resources when applied to large software. Due to this limitation, we have not extended our experiments to very larger software applications, which limits our experimental scope. In the future, we would like to extend our experiments to cover more software.

- Second, the software applications we use are fetched from the public open source host *Github*. It provides us a good source of software applications, but many of those are developed by individuals and not very widely used, so the information about vulnerabilities are usually not available in *CVE*. Also, we are not able to apply our tool to the major software applications appearing in *CVE* due to the size limitation of our tool and the fact many such software applications are not open-sourced. Our future direction is to address this issue and expand the scope of our experiments.
- Third, since we only cover the entry and exit functions of the *c* standard libraries, *DupSec* is limited to the *c* programmed projects for the time being. For *C++* and *Java*, since we have APIs for those two languages, once the entry and exit libraries of these two languages are generated the tool can also be applied. In fact, as long as the I/O libraries are provided, *Dupsec* can be extended to any given language.
- Fourth, all of our experiments are done on a single machine, which is not powerful enough to apply to a large amount of software applications. The next step is to apply it based on a distributed system to perform the tasks in parallel.

Bibliography

- [1] Discover, track and compare open source. <https://www.openhub.net/>, 2017.
- [2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 3–14. IEEE, 1995.
- [3] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 49–49, 1993.
- [4] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95. IEEE, 1995.
- [5] H. A. Basit and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 513–516. ACM, 2007.
- [6] J. Bau, F. Wang, E. Bursztein, P. Mutchler, and J. C. Mitchell. Vulnerability factors in new web applications: Audit tools, developer selection & languages. *Stanford, Tech. Rep*, 2012.
- [7] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.
- [8] J. Caballero, T. Kampouris, D. Song, and J. Wang. Would diversity really increase the robustness of the routing infrastructure against software defects? *Department of Electrical and Computing Engineering*, page 40, 2008.
- [9] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler. Heartbleed 101. *IEEE security & privacy*, 12(4):63–67, 2014.
- [10] C. Community. Common vulnerabilities and exposures. <https://cve.mitre.org/>, 1999.
- [11] U. Ctags. A maintained ctags implementation. <https://ctags.io/>, 2016.

- [12] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 109–118. IEEE, 1999.
- [13] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.
- [14] Eliben. Complete c99 parser in pure python. <https://github.com/eliben/pycparser>, 2010.
- [15] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 67–72. IEEE, 1997.
- [16] A. Ghosh, D. Pendarakis, and W. Sanders. Moving target defense co-chair's report-national cyber leap year summit 2009. *Tech. Rep., Federal Networking and Information Technology Research and Development (NITRD) Program*, 2009.
- [17] D. Gitchell and N. Tran. Sim: a utility for detecting similarity in computer programs. In *ACM SIGCSE Bulletin*, volume 31, pages 266–270. ACM, 1999.
- [18] GitHub.Inc. A web-based hosting service for version control using git. <https://github.com>.
- [19] N. Göde and R. Koschke. Incremental clone detection. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 219–228. IEEE, 2009.
- [20] Google. Google C++ Style Guide. http://google.github.io/styleguide/cppguide.html#Function_Names.
- [21] D. Gusfield. Algorithms on strings, trees, and sequences. *Computer Science and Computational Biology (Cambridge, 1999)*, 1997.
- [22] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang. *Moving target defense: creating asymmetric uncertainty for cyber threats*, volume 54. Springer Science & Business Media, 2011.
- [23] J. H. Johnson. Substring matching for clone detection and change tracking. In *ICSM*, volume 94, pages 120–126, 1994.
- [24] T. Kamiya. Tutorial of cli tool ccfx. <http://www.ccfinder.net/doc/10.2/en/tutorial-ccfx.html>, 2008.

- [25] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
- [26] R. M. Karp. Combinatorics, complexity, and randomness. *Commun. ACM*, 29(2):97–109, 1986.
- [27] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [28] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.
- [29] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*, pages 40–56. Springer, 2001.
- [30] K. Kontogiannis, M. Galler, and R. DeMori. Detecting code similarity using patterns. In *Working Notes of 3rd Workshop on AI and Software Engineering*, volume 6, 1995.
- [31] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1-2):77–108, 1996.
- [32] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 253–262. IEEE, 2006.
- [33] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 276–291. IEEE, 2014.
- [34] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.
- [35] M. R. Lyu, J.-H. Chen, and A. Avizienis. Software diversity metrics and measurements. In *Computer Software and Applications Conference, 1992. COMP-SAC'92. Proceedings., Sixteenth Annual International*, pages 69–78. IEEE, 1992.
- [36] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.
- [37] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *icsm*, volume 96, page 244, 1996.

- [38] S. Neti, A. Somayaji, and M. E. Locasto. Software diversity: Security, entropy and game theory. In *HotSec*, 2012.
- [39] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM, 2007.
- [40] A. J. O’Donnell and H. Sethu. On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 121–131. ACM, 2004.
- [41] A. J. O’Donnell and H. Sethu. Software diversity as a defense against viral propagation: Models and simulations. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 247–253. IEEE Computer Society, 2005.
- [42] D. Partridge and W. Krzanowski. Software diversity: practical statistics for its measurement and exploitation. *Information and software technology*, 39(10):707–717, 1997.
- [43] Petersenna. Cfinder core. <https://github.com/petersenna/ccfinderx-core>.
- [44] A. Raza, G. Vogel, and E. Plödereder. Bauhaus-a tool suite for program analysis and reverse engineering. In *Ada-Europe*, volume 4006, pages 71–82. Springer, 2006.
- [45] K. Ren, C. Wang, and Q. Wang. Security challenges for the public cloud. *IEEE Internet Computing*, 16(1):69–73, 2012.
- [46] T. Rothwell. The gnu c reference manual. <https://www.gnu.org/software/gnu-c-manual/>, 2006.
- [47] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Queen’s School of Computing TR*, 541(115):64–68, 2007.
- [48] A. Saidane, V. Nicomette, and Y. Deswarte. The design of a generic intrusion-tolerant architecture for web servers. *IEEE Transactions on dependable and secure computing*, 6(1):45–58, 2009.
- [49] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security & Privacy*, 3(6):41–49, 2005.
- [50] A. Syropoulos. Mathematics of multisets. In *Workshop on Membrane Computing*, pages 347–358. Springer, 2000.

- [51] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous payload-based worm detection and signature generation. In *International Workshop on Recent Advances in Intrusion Detection*, pages 227–246. Springer, 2005.
- [52] L. Wang, M. Zhang, S. Jajodia, A. Singhal, and M. Albanese. Modeling network diversity for evaluating the robustness of networks against zero-day attacks. In *European Symposium on Research in Computer Security*, pages 494–511. Springer, 2014.
- [53] R. Wettel and R. Marinescu. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Symbolic and Numeric Algorithms for Scientific Computing, 2005. SYNASC 2005. Seventh International Symposium on*, pages 8–pp. IEEE, 2005.
- [54] D. A. Wheeler. A program that examines c/c++ source code and reports possible security weaknesses. <https://www.dwheeler.com/flawfinder/>.
- [55] Wikipedia. Gnu cflow — wikipedia, the free encyclopedia, 2016. [Online; accessed 1-June-2017].
- [56] M. Zhang, L. Wang, S. Jajodia, A. Singhal, and M. Albanese. Network diversity: a security metric for evaluating the resilience of networks against zero-day attacks. *IEEE Transactions on Information Forensics and Security*, 11(5):1071–1086, 2016.
- [57] R. Zhuang, S. A. DeLoach, and X. Ou. Towards a theory of moving target defense. In *Proceedings of the First ACM Workshop on Moving Target Defense*, pages 31–40. ACM, 2014.