

Development of Injected Code Attacks in Online Banking Fraud Incidents

An exploratory study for understanding the evolution of code used by cyber criminals in financial malware

Georgia Charalambous



Development of Injected Code Attacks in Online Banking Fraud Incidents

**An exploratory study for understanding the
evolution of code used by cyber criminals in
financial malware**

by

Georgia Charalambous

to obtain the degree of Master of Science
on Complex Systems Engineering and Management
at the Delft University of Technology,
to be defended publicly on Wednesday August 29, 2018 at 15:00.

Student number:	4627121	
Thesis committee:	Prof. dr. M. van Eeten	chairperson, TU Delft
	Dr. ir. C. H. Gañán	1st supervisor, TU Delft
	Dr.-Ing. T. Fiebig	2nd supervisor, TU Delft

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Preface

Firstly, I would like to express my sincere gratitude to my first supervisor Dr.ir. Carlos Hernández Gañán and second supervisor Dr.-Ing. T. Fiebig for their continuous support of my study and related research. Their guidance, feedback and suggestions helped me in all the time of research and writing of this thesis. I would also like to thank Prof. Michel van Eeten who had the role of the chairman. His critical view and comments proved to be a valuable addition to the content of this thesis.

In addition, I would to thank my parents who helped me have this great opportunity to continue my studies at Delft University of Technology. To my friends, thank you for your support even when we are not physically at the same place. Finally, I would like to thank my brother Konstantinos since without his constant support and encouragement I would not be able to complete my master thesis.

*Georgia Charalambous
Delft, August 2018*

Contents

List of Figures	vii
List of Tables	ix
1 Introduction	3
1.1 Research Problem	3
1.2 Research Opportunity	4
1.2.1 Literature.	4
1.2.2 Knowledge Gap	5
1.3 Research Question	5
1.3.1 Research Approach.	5
1.3.2 Sub Questions, Methods and Tools	6
1.4 Structure of the report	6
2 Background	9
2.1 Banking Trojan Malware Families	9
2.1.1 ZeuS	9
2.1.2 Gozi	10
2.1.3 Ramnit	10
2.1.4 Dyre	10
2.2 Injected Code Attacks	10
2.3 Web injects Evolution	12
2.4 Malware Evolution	12
3 Dataset Description	13
3.1 Overview of the Dataset	13
3.1.1 General Info of Dataset.	13
3.1.2 Malware Family Selection	16
3.2 Dataset Pre-processing	16
3.2.1 Code Extraction	16
3.2.2 Multiple-steps attacks	18
4 Descriptive Analysis	19
4.1 Multi-steps attacks computation	19
4.2 Attack Repetition.	20
4.3 Malware Lifespan	21
4.4 Attack Duration	21
4.5 Attacks by different malware families.	22
4.6 Attacks per Botnet	24
4.7 Attacked Domains	24
4.8 Summary of Findings.	27
5 Code Similarity Tools	29
5.1 Clone Detection	29
5.1.1 Clone Detection Methods.	30
5.1.2 Clone Detection Tools	31
5.1.3 Other clone detection tools	33
5.2 Plagiarism Detection	34
5.2.1 Plagiarism Detection Techniques	34
5.2.2 Tools Comparison	34

5.3	Malware Similarity Detection	35
5.4	Selected tools	35
5.5	Scalable Similarity Metrics	37
5.5.1	Similarity tools comparison.	37
6	Code Similarity Analysis	41
6.1	Overall Injected Code Similarity	41
6.2	Case studies on Code similarity.	42
6.2.1	Code similarity between Citadel and Ramnit attacks	42
6.2.2	Self-similarity matrix inside the self-similarity matrix.	43
6.2.3	Code similarity in domains belonging to the same banking group	44
7	Conclusions and Discussion	47
7.1	Conclusions	47
7.2	Discussion	48
7.3	Recommendations to financial institutions	49
7.4	Limitations.	49
7.5	Future Research	49
	Bibliography	51
A	HTML and JavaScript keywords	59
B	Attacks per Botnet	61
C	Similarity Tools Execution	63
C.1	MOSS	63
C.2	Sherlock	66
C.3	JSCPD	67
D	Self-similarity matrixes	71
D.1	Citadel self-similarity.	71
D.2	Attacks against Lloyds banking group	72

List of Figures

2.1	Example of webinject configuration file format Boutin [13]	11
2.2	Example of malicious code that was injected in customer's web browser (Boutin, 2014)	11
3.1	Overview of injected-code attacks in the dataset.	14
3.1	Dataset - Abstract Entity Relationship Diagram	15
3.2	Dataset Pre-processing steps	16
3.3	Code Extraction Steps	17
4.1	Duration of Attacks in weeks time	23
4.2	Attacked domains for ZeuS-P2P and Citadel malware	25
4.3	Attacked domains for Gozi-EQ and Ramnit malware	26
5.1	Domain 1: Code Similarity Metrics Evaluation	39
5.2	Domain 2: Code Similarity Metrics Evaluation	39
6.1	Self-similarity matrix computed from all attacks	41
6.2	Self-similarity matrix computed from comparing Citadel and Ramnit Attacks	42
6.3	Self-similarity matrix inside the self similarity matrix from Citadel attacks	43
6.4	Self-similarity matrix of the domains part of Lloyd banking group from Gozi-EQ attacks	45
B.1	Citadel: Attacks per Botnet	61
B.2	Ramnit: Attacks per Botnet	61
B.3	Gozi-EQ: Attacks per Botnet	62
C.1	MOSS Execution Command	64
C.2	MOSS Results Overview	64
C.3	MOSS Results Files Comparison	64
C.4	MOSS Results Files Source Code Comparison	65
C.5	Python Script for extracting Moss Results	65
C.6	List of the available options for Sherlock pairwise comparisons	66
C.7	List of the results for all Sherlock pairwise comparisons	67
C.8	Detailed results of a Sherlock pairwise tokenized comparison	68
C.9	Overview of JSCPD execution	68
C.10	JSCPD options	69
D.1	Self-similarity matrix inside the self similarity matrix from Citadel attacks	71
D.2	Self-similarity matrix of the domains part of Lloyd banking group from Ramnit attacks	72
D.3	Self-similarity matrix of the domains part of Lloyd banking group from ZeuS attacks	72
D.4	Self-similarity matrix of the domains part of Lloyd banking group from Citadel attacks	73

List of Tables

2.1	Common flags for <i>set_url</i> as described by Boutin [13]	10
3.2	Main variables extracted from the database	16
3.3	Code Extraction results with intermediate steps	17
4.1	Information about Mutli-steps attacks per malware family	20
4.2	Statistical values of attack instances repetition for each threat	20
4.3	Malware Life Span	21
4.4	Citadel's attacks used by different malware families	22
4.5	Ramnit's attacks used by different malware families	22
4.6	Number of Attacks per Botnet for each threat	24
5.1	Type of clones by Koschke (2007)	30
5.2	Source Code Changes Classification	30
5.3	Clone Detection Tools by Roy and R. Cordy (2007)	32
5.4	Summary of comparison experiment by Bellon (2002) as presented by Roy & R. Cordy (2007)	33
5.5	Summary of comparison experiment by Koschke et al. (2006) as presented by Roy & R. Cordy (2007)	33
5.6	Summary of comparison experiment by Burd and Bailey (2000)	35
5.7	Domain 1: Code Similarity Metrics Evaluation	37
5.8	Domain 2: Code Similarity Metrics Evaluation	38
A.1	List of HTML and JavaScript keywords	60

Executive summary

The frequency of online banking fraud incidents has increased over the last years. A method used by different cybercriminals is the injection of malicious code into the targeted web pages. For example, attackers might inject an additional piece code into the webpage of a targeted bank asking users to enter extra personal information (e.g., the PIN of the card).

Over the years and despite the different technical solutions have developed to detect injection code attacks, attackers continue to target different financial institutions using the web injection technique. This suggests the presence of an environment which allows cybercriminals to evolve their code. The majority of studies related to the evolution of malware so far are mostly focused on studying the evolution of different components of the same malware or how similar are malware families which they are variants of the same malware family. Limited information exists on the code similarity of the actual code used in web injected code attacks.

Having this in mind, we used a rich dataset provided by Fox-IT, to formulate the following research question: "*How do cybercriminals evolve the code injected by financial malware in MITB attacks?*". To answer this question, we used a data sample from the Fox-IT dataset which contained injected code attacks from four different malware families. For the extraction of the injected code we used a list of JavaScript, HTML and CSS keywords in order to discard empty strings or string containing only English words. In addition through a literature review, we identified MOSS, Sherlock, JSCPD as the most suitable tools to be used to detect similarity in different inject code attack instances. However, due to their inability to scale the results for all the different attack codes, we decided to use metrics based on string matching.

The findings of the descriptive analysis are:

- Attackers tend to reuse the same attack, all the attack instances in this research are repeated at least four times.
- The lifespan for the majority of the attacks for all malware does not last as long as the malware's lifespan. This point out that attackers updated the content of the configuration file (code) more than once during the malware lifespan, which is a sign of code evolution.
- Attacks of Ramnit and Citadel have been also used by other malware families which proves that cybercriminals not only use the same code in attacks of the same malware family but also in attacks of different malware families.
- The majority of times, attackers do not target a domain more than 500 times.

The use of the Levenshtein distance for measuring the code similarity and the visualization of the results through a self similarity matrix allowed to discover cases where attacks have high code similarity.

The results showed that while attackers are frequently changing the attacks against domains of financial institutions, they tend to develop reusable and no page-specific code which allows them to reuse it in attacks against different targeted domains and in some cases even across different malware families.

Finally we used the findings of our analyses to provide recommendations to different financial institutions on how to improve their current defense measures against malware families that perform web injected attacks.

Table of Terms

A table of terms related to the topic of web injected code attacks is presented below. The definition of each term is provided in the second column.

Term	Definition
Web injected code attack	A malicious technique used by cybercriminals, also known as a Man-in-the-Browser attack (MitB). Cybercriminals are using this technique to target specific web pages and modify their content for the purpose of deceiving users, making them believe that the content they are browsing is the actual content of the webpage [13]. In this research after the second chapter, when we use the term attack we are refereeing to web injected code attacks.
Configuration File	The file which provides parameters and initial settings for the operating system or other computer applications. It is usually written in ASCII encoding and contains all necessary data about the specific application, computer, user or file ¹ . In web injected code attacks, it is the file which contains the target web page and the code which will be injected into that web page. It is saved in the command and control (C&C) server of the botnet which downloads the file in the infected computers (i.e. bots).
Injected code	The code developed by cybercriminals which is injected at the targeted web pages, in order to modify their content. It mostly includes JavaScript code but it can also have HTML and CSS content. It is stored in the configuration file.
Lines of code	In this research lines of code is referred to as the physical lines of code (LOC), which is the number of lines in a file, excluding comment lines [80].

¹Definition Retrieved from: <https://www.techopedia.com/definition/3339/configuration-file-config-file>

Introduction

1.1. Research Problem

Since the mid-1990s, banks have introduced online banking as a new way to offer their services to their customers [84]. Online banking services are divided into three main categories: Internet Banking, Home Banking, and Mobile Banking. Over the years, customers have widely accepted these services [94]. However, the number of online banking fraud incidents reported is increasing. Estimating the cost of cybercrimes has been proved to be very challenging for researchers. Nevertheless, the majority of researchers agree that it affects everyone and that the losses caused by cybercrimes are massive [5, 39, 46, 75, 90]. Only in 2014, 1,467 financial institutions in 86 countries reported that they were the target of financial Trojans attacks. According to a report published by the European Central Bank in 2015, the total fraud for Single Euro Payments Area (SEPA) was around €1.44 billion and the value of 'card-not-present' (CNP) fraud increased to €958 million.

Throughout the years, different methods which can detect online fraud incidents were proposed by Phua et al. [83], van Vlasselaer et al. [113], Wei et al. [117]. Nevertheless, attackers have managed to find their weakness and developed different types of attacks to target financial institutions. One of the most popular ways of attacking financial institutions is phishing attacks. Phishing attacks are executed through emails containing malicious links [101]. This type of attack can extract personal and sensitive information from users including their passwords and credit card information. Evidence have shown that 30,000 phishing incidents are reported every month and around 900 bank accounts are victims of those attacks [4].

Hackers also commit online banking financial fraud, using financial botnets [88]. Cybercriminals utilize the botnets to perform different illegal activities such as Distributed Denial of Service (DDoS) attacks. Using botnets, attackers can take control and form a network of many infected computers [40]. Many financial botnets are developed to act as Man-in-the-Browser (MitB). MitB attacks are activated through Trojan malware, which is installed on the victim's device and can infect Internet applications [95]. MitB attacks are designed with the ability to "intercept and manipulate" any data a user exchanges when using an Internet application. This type of attacks is very dangerous because they can hide their operation from anti-virus software [93]. For example, attackers can use such Trojan malware to insert into a customer's bank web page an additional piece of code [112]. This piece of code can steal additional information from the user without him noticing. These attacks are called 'web-injected code' attacks.

Over the years, law enforcement agencies around the world with the help of the financial sector and security firms have implemented many operations to takedown botnets of different Trojan malware [34, 76, 77]. Still, Trojan banking malware is top a threat to the financial sector. A report by Check Point [16] has listed two Trojan malware families in the top 10 list of the "most wanted malware". The threat of Trojan banking malware is even more evident in the last report by Check Point [17] which recorded a 50% increase on the impact of banking Trojans in the first months of 2018. However, we still do not know why, despite those takedown operations, attackers continue to use web injected code attacks and how they evolve the code used in the attacks.

Relevance

The focus of our research is on financial malware that performs MitB attacks. We aim to study the evolution of the injected code used in those attacks. This is relevant in academic terms because there is a research gap. Moreover, the problem described is a problem that links together society, technology, institutions and in some extend policy. For technology and institutions, the results can be used to evaluate current countermeasures and develop new policies against different cyber attacks. For society, the results can be used to protect users against any cyber threat.

Science

To the best of our knowledge, so far there are no studies focusing entirely on the evolution of those types of attacks. Researchers usually focus on the current design of such malware and how they can mitigate it [86, 88, 100]. The majority of studies, examine different variants of the malware but not the actual code that has been used in the attacks [69]. Additionally, since it is very difficult to obtain the actual source code of such attacks very little is known about the semantics of the attack code and what actually motivates the cybercriminals to evolve their code.

Society

Except for the possibility of filling the knowledge gap by providing insights on how cybercriminals evolve their code in financial attacks, this research can also have a contribution to society. The possible contribution to society is that knowledge about that behaviour can enable implementation of effective mitigation strategies and be a reference point for a number of financial institutions against the threat of cybercrime. These strategies can help different financial institutions to protect not only their assets but also their customer's personal data and privacy and to a greater extent the national security.

Technology

Finally, the understanding of financial malware functionality is also technically relevant. The results of the study can be used to assess how effective are the current countermeasures in the different financial institutions and facilitate the design of new ones that can minimize the threat.

1.2. Research Opportunity

1.2.1. Literature

Banking Trojan Malware

In the previous section, we mentioned that despite the efforts to take down the botnets of different Trojan malware, the threat is still active. It is likely that the threat is active due to the existence of several malware variants. For example, the leak of the source code of the ZeuS malware, one of the most known Trojan banking malware, lead to the development of many variants. Ice IX, Citadel, ZeuS-P2P, and SpyEye are among the most popular and used [32, 108].

To get a better understanding of how much connected are those variants, we briefly review the existing literature. Boutin [13] states that different malware families continue to use the same web injected tools and code when targeting bank web pages. Tajalizadehkhoob et al. [109] studied the evolution of the injected code in the ZeuS financial malware data files. They detected code similarity in the injected codes, which indicates code reuse in the attacks. According to Rahimian et al. [86], the Citadel and the ZeuS malware share a significant proportion of the same code. Their main difference is related to the injected code module: Citadel has "country-specific" features when is targeting online banking accounts. The authors argue that this additional feature makes the Citadel malware more sophisticated than the ZeuS. Criscione et al. [23] discovered that the SpyEye and the ZeuS malware are using the same injected code for the targeted web pages.

Detection Methods

The methods which have been used so far for analyzing the code used in the injected code attacks are mostly reversed engineering and clone-based analysis [12, 86]. On the other hand, Criscione et al. [23] argue that reverse engineering technique is time-consuming and has a low response time to cyber attack incidents. Instead, they propose an automatic method (ZARATHUSTRA) to test whether malware targets a specific web page. They tested their method against a dataset containing ZeuS samples and the URLs of bank web pages. The ZARATHUSTRA method was able to identify all the malicious code which injected into the web pages. Successful malware detection methods using the Zeus and the SpyEye as malware samples have also been developed by Garcia-Cervigon and

Llinas [41], Riccardi et al. [89]. Even though these various detection methods exist, web injected code attacks still occur and the injected code has evolved over the years [13].

1.2.2. Knowledge Gap

In the previous subsection, we briefly describe the problem of injected code malware. Through literature, we found that variants of banking Trojan malware tend to share and reuse the same code. The studies which have analyzed and measure the code similarity are mostly using the binary or assembly code of the malware attacks [12, 86]. We assume that this is the result of the difficulty to obtain the actual source code of the malware. In addition, the majority of the literature focuses on detecting the similarity between the variants of different malware families and not the similarity in the code used in the attacks. Finally, limited information exists on how exactly cyber criminal decide to evolve their code and what changes they perform to new attacks compare to their previous versions.

A rare and great research opportunity was created to study the evolution of the injected code by examining the source code of the attacks. This opportunity is thanks to Fox-IT, a security company, which gave us access to a dataset. This dataset contains a massive amount of information related to cyber attacks including their configuration files. Having access to the configuration file enable the extraction of the source code that was injected into the targeted web pages.

1.3. Research Question

The limitations which we already mentioned and the ability to use a very rich dataset in terms of information provides us, the window to study even further the evolution of the injected codes. Therefore, we formulated the following research question:

How do cybercriminals evolve the code injected by financial malware in MiTB attacks?

The objective of the research is to investigate the evolution of web injected code attacks which are targeting financial institutions. It is intriguing to examine how attackers adjust and evolve their code, how similar it is to previous versions and to different malware families. Moreover, this research can give insights on why attackers have the incentives to use the same code, whether is only related to the cost or additional motivations exist. Finally, it can help banks and other financial institutions to evaluate if they have effective countermeasure against financial malware attacks and propose way they can deal with malware that uses already known code.

1.3.1. Research Approach

In this section, we explain the research approach we followed for the conduction of the thesis. We use a case study research since its aim is to answer “how” research questions and to provide insights and a better understanding of a situation happening in the real word [124]. In this research, we use multiples case studies, with a descriptive/exploratory approach. This approach is based on [124] approach. The three main parts of our research which we used to answer our research question are:

1. Data Collection and Preparation

The conduction of our research required a dataset which contains instances of malware attacks. Fox-IT has provided a dataset which includes raw information about different online financial fraud incidents caused by different malware families. However, not everything in the dataset was considered relevant for this research. Taking into consideration the size of the dataset and the limited time we had for this project, we selected a sample of four malware families to continue our research. Next, we selected the main parameters which we used as inputs for our analysis. In the last stage of the data preparation, we extracted and saved the injected code for easier access in later sections of the research.

2. Data Exploration

In this step, we performed a descriptive analysis. We used our selected parameters and a series of different analyses to discover evidence of code reuse and evolution in our sample set. In addition, we performed a literature review to discover the most suitable tool for comparing the injected code used in the different attacks.

3. Data Analysis

The last step of the thesis is the actual analysis of the data. In this step, we compared the injected code of

the attacks to identify code similarity. Afterwards, we used a self similarity matrix to visualize the results and performed code similarity analysis to draw conclusions regarding the code evolution. The tools we for the data analysis were the programming languages Python and R.

Advantages of the multiple cases approach is that can be considered more robust compared to the single case approach. On the other hand, limitation of the approach, it is the fact that it can be time consuming and that requires more resources than a single case [124].

1.3.2. Sub Questions, Methods and Tools

The list of sub-question that were used to answer the main research question how cyber criminals evolve their code in different attacks instances is provided below. For each sub-question, the tools and methods used to find the answers are also explained.

1. What indicators of code evolution can be observed in the dataset?

We used this subquestion to describe our data and discover evidence of code evolution or reuse in our injected code attacks samples. We conducted a series of descriptive analyses using different levels of analysis. Our main aim was to identify if code evolution actually exists in the injected code attacks.

2a. What is the most suitable tool for detecting code similarity in different inject code attacks?

The literature review has shown that different malware shares a significant portion of the same code. In our descriptive analysis, we identified indications of code evolution in the attacks. However, we did not actually know how much did the code change. In this sub-question, we defined which of the available code similarity tools is suitable for measuring the code similarity in the injected attacks. We conducted a literature review in which we reviewed tools developed for code clone detection, plagiarism detection, and malware similarity detection. Limitation regarding the selection of tools was the availability of the tool and its ability to detect code similarity in JavaScript code which is the programming languages in injection code attacks.

2b. What are the possible limitations of the the tool and how can they impact the result when it is used for injected code comparisons?

The goal of the sub-question was to identify what are the possible limitation of the tools we selected. We answered the sub-question by exploiting the limitations of each. We also provided an alternative solutions to serve as measure of code similarity.

3. How much code re-using, stealing or sharing does exist across different instances of inject code attacks??

The goal of this question is to examine if attackers use the same code in different injected code attacks. In order to answer this question, we first used the tools which we selected in the previous sub-question to measure the code similarity between all the attack samples we extracted. Then we used the results to plot a self similarity matrix which shows the code similarity among all codes in scale. We used the self similarity matrix to identify cases where signs of code reuse was visible. We formulated hypotheses to explain the code similarity. Finally we examined the code used in those cases so we can either accept or reject our hypotheses.

Two data analysis tools were used for answering the sub-question. Python is used for extracting and preparing the datasets for each analysis separately, while R is used for the interpretation and statistical analysis of the results. The advantages of using these tools is their ability to handle large dataset, their flexibility and the different methods they offered for data analysis.

1.4. Structure of the report

This thesis is structured as follows: In the second chapter, we provide background information. We start the chapter by giving examples of malware families which perform MitB attacks, and we briefly explain how they inject the malicious code. Then, we take a look at the literature related to web inject and malware evolution.

In chapter three, we provide a brief overview of the dataset supplied by Fox-IT. We explain how we constructed a dataset which only contains injected code attacks and how we pre-process our data. In chapter fourth, we present the results of our descriptive analysis which allowed us to discover to what extent evidence of code evolution or reuse exist in our attacks samples.

In chapter five we review existing code similarity tools from literature. We select existing tools and we review what it is their possible impact and limitation when they are used to compare the injected code. The chapter ends with a presentation of metrics based on string matching which we used to detect code similarity in different inject code attack instances. Chapter 6 contains all the analyses we performed to detect code similarity in different injected code attacks. First, we present a self similarity matrix which show the code similarity among all injected codes. Then we explore three cases of code similarity by examining the code used in the attacks.

Finally, in chapter seven we summarize the results and answer the main research question based on the answers of the research sub-questions in chapters 4 - 6. Additionally, we discuss results, limitations, and recommendations for future research.

2

Background

In this chapter, we provide background information related to our research problem. In the first section, we provide examples of different banking Trojan malware families. Next, we briefly describe how an injected code attack operates. In the final section, we shift our focus to the malware evolution.

2.1. Banking Trojan Malware Families

2.1.1. ZeuS

ZeuS or Zbot is one of the most known Trojan banking malware, it appeared first in 2017 [32]. It is responsible for almost 90% of the global banking fraud and for the 44% of online malware infections [4]. The ZeuS kit could be bought privately, up to the price of \$4,000. However, the leak of its source code in 2011, reduced its price [89]. As we already mentioned, this leak contributed to the creation of several malware variants. In the following, we explain some of them.

Zeus Peer-to-Peer (P2P)

Zeus peer-to-peer (P2P) or GameOver Zeus is a variant of the Zeus Trojan malware which appeared in September 2011 [32]. Unlike the other variants of Zeus, Zeus P2P uses a P2P network and a domain generation algorithm for C&C [103]. The lack of centralized C&C servers makes taking down the Zeus P2P botnet harder, when using the same countermeasures used against the original Zeus malware [6]. The P2P network of Zeus has two central roles: (i) It allows the bots to exchange binary and configuration updates with each other, and (ii) Bots can share a list of proxy bots that can be used to drop stolen data and retrieve commands [6]. The Zeus P2P malware performs its illegal activities in real time and is usually distributed and disguised as an email invoice [103].

Citadel

Following the leak of Zeus, source code in 2011 the Citadel malware emerged as a new variant. Citadel is targeting financial institutions and performs MitB attacks. The Citadel kit can be bought in underground Russian forums for \$3,000, compared to \$100 for the SpyEye and leaked Zeus kits [102]. Citadel has kept almost all major features of Zeus [78]. However, the authors of Citadel have added extra capabilities and expanded the abilities of the malware. Some of the capabilities include video capture, denial of services and automated command execution [78, 86]. Moreover, besides web injected code attacks, Citadel can also act as ransomware [32]. A McAfee report, published in 2012, reveals that cybercriminals are using Citadel to target mainly European countries, with the United Kingdom, Germany, and the Netherlands being their main focus [99].

Other Variants

The popularity of mobile banking, lead to the development of the ZeuS-in-the-mobile (ZitMo) malware that has targeted many customers in European countries[32]. Ice IX is a variant of ZeuS family which steals users' bank account details and phone numbers too [59]. The SpyEye variant is a third-generation malware, which hijacks the user's device's audio and video and uses the keylogger mechanism to steal sensitive information [32]. The Bugat malware browses and sends information from the victim's infected machine [72]. The Shylock malware appeared in 2011 and uses the same "configuration architecture" as the original ZeuS malware [120]. The authors of Shylock used the injected code attacks to extract user's bank credential for fraudulent purposes [106].

2.1.2. Gozi

The Gozi malware first appeared in 2007 and was one of the first banking Trojans. It mostly targets English speaking countries. The source code of its first variant was leaked in 2010. The second variant Gozi v2 introduced a new mechanism which allowed the malware to perform injected code attacks when targeting different banks [96]. According to a Kaspersky Lab report, published in 2016, during the 3rd quarter of 2016 the Gozi malware family was the 4th most used malware against online banking users. Later reports show that the developers of Gozi malware have updated the code of web injection mechanism to commit "form grabbing" and web injection in the Edge browser which is the default browser of Windows 10 [96].

2.1.3. Ramnit

The Ramnit malware first appeared in 2010 and was operating as worm. In 2011, due to the leak of ZeuS source code, Ramnit obtained some of ZeuS' modules and started performing MitB attacks [104]. The main difference to the ZeuS web injects module, is that Ramnit does not communicate with the C&C server directly. The Ramnit botnet compromised approximately 350,000 computers worldwide [105]. According to the AV Test report published in 2016, Ramnit was the 4th most used malware on the Windows platform in 2015 and the 3rd for the 1st and 2nd quarter of 2016.

2.1.4. Dyre

The Dyre malware first appeared in 2014, it mostly targets English speaking countries and it is spread using spam emails.[18]. The Dyre malware can execute different types of MitB attacks [107]. The first type is similar to the attacks performed by the ZeuS malware. The Dyre malware injects malicious code to modify the appearance of different websites and steal additional information entered by the victims. The second type of MitB attacks redirects victims to a fake website. The fake web page steals user's information before redirecting him back to the original web page. In contrast with other banking Trojan malware, the Dyre malware has no connection with the ZeuS malware family.

2.2. Injected Code Attacks

The majority of Banking Trojan malware that performs MitB attacks operates as follows. Once the malware infects a user's computer, it becomes a bot and is connected to the malware botnet. Then the configuration file is downloaded to the infected machine from the command and control server (C&C). The configuration file contains the details of the injected code attack including the target and the JavaScript code which is injected into the web pages which are browsed by the infected computer. Any change in the content of the configuration file will only be activated if the infected machine downloads the new version form the C&C server.

Figure 2.1 shows an example of a configuration file. As it can be seen, the `set_url` parameter defines the targeted URL. The following letters are acting like flags which are used to define the action which will be executed when the targeted URL is activated. The most common flags are displayed in Table 2.1 as reported by [13]. Following that, three tags appeared: the `data_before`, which is the content before the injected code, the `data_inject`, which is the content injected in the web page and lastly the `data_after`, which is the content that comes after the injected point.

Table 2.1: Common flags for `set_url` as described by Boutin [13]

<code>set_url</code> flag	Explanation
G	GET requests should be inspected for possible injection.
P	POST requests should be inspected for possible injection.
L	Used for logging purposes
H	Used for logging purposes

According to Boutin [13], developers of well-known malware like ZeuS and SpyEye prefer the web inject technique. This technique allows them full control over what they are stealing from the victim's online transactions. An example for such attack is the following: criminals are using this technique at the login page of a user's bank account and ask the user to add his PIN number or social security numbers, something which is not usually required. Once the customer provides the additional information, the data is stolen by attackers [3, 25, 32].

```

set_url https://[redacted].ca/[redacted].html*command=displayAccountSummary* GP

data_before
<body
data_end

data_inject
style="visibility:hidden"
data_end

data_after
>
data_end

data_before
</body>
data_end

data_inject
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js"></script>
<script>
var chBalansUsd=0;
var chBalansCad=0;

```

Figure 2.1: Example of webinject configuration file format Boutin [13]

Figure 2.2 is an example of such an attack.

Country	<input type="text" value="United States"/>		<div style="border: 1px solid gray; padding: 5px; background-color: #f0f0f0;"> For verification purposes you must update your card details. </div>
First name	<input type="text" value="John"/>		
Last name	<input type="text" value="Doe"/>		
Address line 1	<input type="text" value="The White House"/>		
Address line 2 (optional)	<input type="text" value="1600 Pensylvania"/>		
City	<input type="text" value="Washington"/>		
State	<input type="text" value="DC"/>		
ZIP code	<input type="text" value="20500"/>		
Phone number	<input type="text" value="202-456-1111"/>		
Card number	<input type="text" value="4512123213213213"/>		
Expiration date (mm/yy)	<input type="text" value="12"/> <input type="text" value="12"/>		
CSC	<input type="text" value="123"/>		

Verified by Visa Password is incorrect

Card number	
Name embossed on card (Exactly as on card)	<input type="text" value="John Doe"/>
Date of birth (mm/dd/yyyy)	<input type="text" value="01"/> <input type="text" value="01"/> <input type="text" value="0001"/>
Mother's maiden name	<input type="text" value="DoeMrs"/>
Social security number	<input type="text" value="123"/> <input type="text" value="12"/> <input type="text" value="1323"/>
Driver license number	<input type="text" value="456456456"/>
Credit / Debit card PIN	<input type="text" value="1234"/>
Verified by Visa password	<input type="password" value="....."/>

Figure 2.2: Example of malicious code that was injected in customer's web browser (Boutin, 2014)

2.3. Web injects Evolution

Boutin [13] studies the evolution of web injects malware, he notes that web injects started with simple keyloggers and form grabbing functions and have moved to more advanced techniques like web injected code and Automatic Transfer System (ATS).

In earlier versions of the SpyEye malware, the web injected code was only used in the Internet Explorer browser. Later versions of the malware introduced the technique to the Mozilla Firefox and Netscape browsers [100]. The authors study the evolution of 13 different versions of SpyEye malware. The first version of the malware supported simple operations like form grabbing and auto fill. Throughout the years, the malware has evolved and now supports more sophisticated actions including more advanced web injection and email grabber support.

Wyke [120] investigates the different strategies which are implemented by malicious software when it is in a virtual or 'sandbox' environment. One of his case studies is the Shylock malware. The Shylock malware is able to identify whether it is running in a sandbox environment. In those situations, it returns a fake file which shows that the injected code changes only a little over time. On the other hand, when the malware is running on real conditions the file which contains the injected code looks more complicated and advanced. According to the author, the Shylock malware makes it more difficult for financial institutions to realize that have been victims of cyber attacks and identify how the injected code looks like.

2.4. Malware Evolution

Malware analysis as defined by Walenstein and Lakhotia [115] refers to the problem which aims to understand and manage malicious software. They state that malware is evolving by re-using and sharing code from different malware families. They argue that measuring the similarity between different malware can help the problem of malware analysis. They also claim that the code re-use between different malware is similar to code similarity problems. However, techniques like plagiarism detection were not used so far in the malware analysis problem. Noreen et al. [81] defined malware evolution as the "*process which brings a change in the behavior, functionality or semantics of a given malware*". They also state that malware developers have built "malware creation engines" which they use to create different variations of malware, by changing the code of a given malware. However, the authors mention that despite the variants of malware the new versions maintain the previous code semantics.

Over the years, different methods which can detect malware evolution have been developed. They are classified into two main categories: (i) static code analysis, and (ii) dynamic code analysis. Static analysis is conducted before the execution of the code studying either the source code or the object code. On the other hand, dynamic analysis is done after the execution of the code and where the assembly code or the binary code is studied. Lindorfer et al. [69] developed BEAGLE, a system which can identify code evolution in different malware components over time using different techniques. They tested their tool against 16 malware samples from 11 families, including a variant of the ZeuS malware using the binary code. They found that malware developers reuse parts of the same code and that some malware families evolve more than others. In addition, for the specific malware families, they identified malware components which do not change at all or they change more frequently than others. Comparetti et al. [22] developed Reanimator which is a system capable of identifying malicious code using dynamic analysis. Their system is also able to identify if that specific piece of code is present in other malware samples. BinClone and ScalClone are two tools which can detect similarities in the binary files of malware samples [35, 36]. In contrast, the NiCad tool can detect similarities in the source code of malicious files.

3

Dataset Description

In the following section, we provide a brief overview of the dataset. We also document our approach for selecting a sample from the dataset. As the dataset contains traces which are not relevant to our research questions, we construct a subset of the originally provided dataset which only contains code-injection attacks. Finally, we also document our full pre-processing pipeline for completeness reasons.

3.1. Overview of the Dataset

In this research project we use a dataset of malware attacks provided by Fox-IT [38]. Fox-IT is a Dutch company in the field of computer and network security. Their portfolio includes providing technical solutions in risk mitigation against attacks on digital systems. Their clients include governmental bodies, critical infrastructure providers and financial institutions. The dataset used in this project stems from cyberattacks monitored by the company.

3.1.1. General Info of Dataset

The dataset was collected between 21/02/2014 and 03/11/2017. It contains a total of 123,456,789 entries, of which 1,957,564 relate to injected code attacks. These entries correspond to a total of 41,481 unique attack connected to 372 botnets and 26 different malware families. Table 3.1 provides an overview of the data set. Note that the number of domains includes all unique domains of each malware family during the entire infection period and not the common domains in all four years.

Malware	# of distinct attack_id	# of total entries attacks	# of botnets per malware	# of distinct domains	# of distinct URLs
BokBot	69	276	1	3	9
Citadel	2,345	192,100	3	4,308	18,894
CoreBot	86	344	1	11	58
Dridex-Loader	4,834	158,752	20	2,785	17,022
Gootkit	569	6,460	21	2,784	17,020
GootkitLoader	11	84	1	2,784	17,019
Gozi-EQ	4,431	397,984	55	279	5,378
Gozi-ISFB	7,429	107,540	23	2,821	17,090
Ice9	168	11,784	0	28	2,447
KINS	5,220	101,580	129	4,354	19,080
Kronos	1,309	21,304	13	4,307	18,903
Matrix	11	64	1	3	6
NuclearBot	452	1,908	16	2,512	16,436
Nymaim	157	5,104	1	22	102
Pkybot	10	40	1	5	17
Qadars	1,309	55,824	2	843	4,135
Qakbot	2	12	0	-	-
Ramnit	1,658	27,388	3	131	447
Ramnit-BankerModule	351	1,716	1	46	283
ReactorBot	590	3,824	2	28	357
Tinba-v1	67	268	1	694	1,095
Tinba-v2	5,232	123,432	13	453	4,987
Zeus	651	373,888	34	2,786	17,026
Zeus-Action	216	18,920	4	78	135
Zeus-P2P	1,688	297,428	0	214	1,102
Zeus-Panda	2,616	49,540	26	877	2,266
Total	41,481	1,957,564	372		

Table 3.1: Overview of injected-code attacks in the dataset.

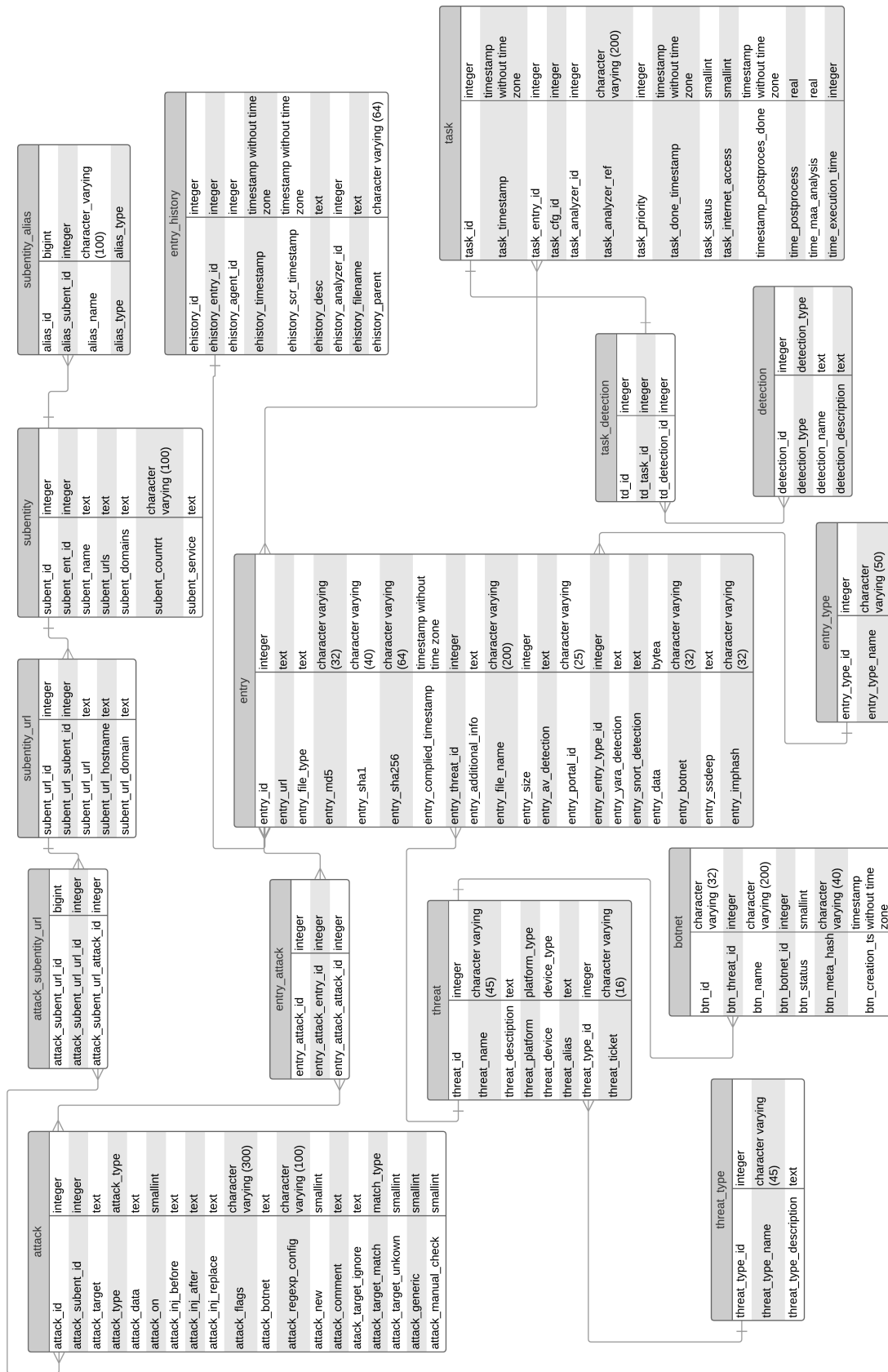


Figure 3.1: Dataset - Abstract Entity Relationship Diagram

Fox-IT provided us with the dataset in a PostgreSQL database. Please see Figure 3.1 for an overview of the Entity Relationship Diagram. We use the **threat** table to identify the malware family of each attack. In the table **attack**, we find all the data of a configuration file. We use the field *attack_type* of the table to filter out the attacks not related to injected code attacks. We select the injected code from the field *attack_data* which we use as main input for our analysis. We also use the field *attack_target* to connect each attack with their corresponding targeted URLs and domains. In the **entry** table and in the field *entry_botnet*, we find the unique key (usually an RC4 key) for each botnet. We use this key to connect each attack with its associated botnet. Finally, we use the **entry_history** to get the life span of the injected code attacks through field *ehistory_timestamp*. Table 3.2 presents these variables which are the inputs for the rest of our research.

Table 3.2: Main variables extracted from the database

Parameter	Explanation
Threat	Malware family responsible for the attack
Injected Code	Code used in the attacks
Botnet	Botnet associated with each attack
Attack Life span	Duration of the attack
Target	Targeted domains and URLs

3.1.2. Malware Family Selection

Due to the size of the dataset and the limited time available for this project, we do have to select a sample of attacks for the remainder of this work. The four malware families we selected are ZeuS-P2P, Citadel, Gozi-EQ, and Ramnit. We selected ZeuS-P2P and Citadel because both are variants of the ZeuS malware, so we can examine how differently code from related families evolve. Gozi-EQ is one of the earliest malware that performs MitB attacks which allows us to observe the earliest developments in code evolution. Finally, Ramnit was a worm malware which only later started performing MitB attacks. Hence, we can investigate how the injected code attacks evolve over time. Note, that this sampling may introduce a sampling bias. Nonetheless, as this study is exploratory, we consider our sampling approach as reasonable. Nevertheless, we note that the results we obtain should be validated in an exhaustive study as well.

3.2. Dataset Pre-processing

In this section, we briefly describe how we pre-process the raw dataset prior to our analysis. The steps are provided in the box in the Figure 3.2.

1. Extract unique attacks and their code using steps of Box 2
2. Extract all entries_attacks of all targeted URLs for all register domains
3. Merge the datasets returned from step 1 + step 2
4. Repeat steps 2-5 for all malware families

Figure 3.2: Dataset Pre-processing steps

3.2.1. Code Extraction

After we selected our sample malware families, we identified unique attack for each malware family. Then we extracted the injected code used in these attacks. Note, that we found several cases where the *attack_data* field was an empty string or merely contained one or several English words, e.g., “access” without any JavaScript or HTML content. The empty fields could be the result of a missing configuration file for the web injection attack. Another possible explanation is that attackers are using more advanced encryption techniques or store the configuration files in parts, making it more difficult to achieve full decryption of the file [13]. Hence, additional aggregation of the data is required to discard these unusable samples. The steps in Figure 3.3 describe this pre-processing process. In short, the process extracts only the contents of the column *attack_data* that are not an empty string and which contain one of the predefined HTML and JavaScript keywords. All the returned injected codes are saved as JavaScript files (*.js*). The keywords used for the extraction process are saved in a table which consists of a collection of JavaScript reserved terms (e.g, `var`, `function`) and HTML tags (e.g, `<body>`, `<html>`). Our

list contains all the JavaScript reserved words which they cannot be used as variables, labels, or function names. In addition, it includes some of the most commonly used HTML Tags. Therefore, we consider our list capable of extracting the code sample which we can use in the research. Table A.1 in Appendix A provides the exhaustive list.

While manually inspecting the contents of column *attack_data*, we noticed that some instances of the attacks linked to Ramnit contained just the word **finj**. Note, that we assumed that the word **finj** is not an actual injected code but a simple word. The keyword list includes the JavaScript operator **in** which it is a substring of the word **finj**. Therefore, we discarded the attacks which contained only this word, by temporally removing the keyword **in** from the list. Nevertheless, we note that the presence of the word **in** should be examined further in future work.

```

1.  Extract distinct attacks for each threat using the attack_id.
2.  Select instances where attack_data is not an empty string.
3.  Check the returned instances against the keywordList.
4.  If the returned instance contains at least one element from the keywordList
    Save the instance as .js file with the name Attack_id.
Else
    Move to the next instance.

where
attack_id and id, the identifier of each attack,
attack_data, the code used in the attack,
threat, one of the following: Ramnit, Zeus P2P, Citadel or Gozi-EQ
keywordList, a list of HTML and JavaScript keywords

```

Figure 3.3: Code Extraction Steps

We found that all malware families had entries which were either empty strings or did not contain any keyword. The total number of attacks dropped from 10,122 to 8,484 which shows that 27% of the initial attacks were discarded. This sample of injected code attacks is the subset used for the rest of the research. In addition, we noticed that more than 400 of the injected code attacks of Citadel malware were empty strings, which resembles the 18.25% of the total number of its attacks. Similarly, almost the same amount of attacks were removed from the Ramnit subset because they did not contain any of the keywords. Discarding these attacks lead to the largest decrease in the final result: 30% for Ramnit and 24.4% for Citadel. Table 3.3 illustrates the total number of web injected code attacks extracted for each malware, along with the intermediate steps of the code extraction process.

Table 3.3: Code Extraction results with intermediate steps

Malware	# of distinct attacks	# of attacks after discarding empty string	# of attacks containing keywords
ZeuS-P2P	1,688	1,643	1,473
Citadel	2,345	1,917	1,772
Gozi-EQ	4,431	4,355	4,080
Ramnit	1,658	1,631	1,159

After we merged the two datasets as described in Figure 3.2, we created a dataset per malware family which contains all the entries of web injection code attacks for all the targeted URLs with their associated domains. An entry in the dataset contains the following information: (1) the *attack_id* which uniquely identifies the injected code, (2) the name of the targeted domain, (3) the URL path which was the target, and (4) the timestamp when the attack appeared in a configuration file.

3.2.2. Multiple-steps attacks

During the examination of the datasets, we noticed cases in which a targeted URL was attacked more than once at the same time. The *attack_ids* of those entries were different, meaning that the URL was most likely injected with different injected codes. We hypothesized that the injected codes of these different attacks are somehow linked and act as one single attack. We defined these attacks as multi-steps attacks. In the next chapter, we provide more information about these attacks.

4

Descriptive Analysis

In the previous chapter, we provided a brief overview of the research dataset. Additionally, we selected the main parameters which we are going to use as input for the rest of our analysis. The parameters are: (1) the injected code of the attacks, (2) the timestamp of the attack, (3) the target of the attack, and (4) the botnet associated with each attack.

Looking back, the main objective of this study is to determine how attackers have evolved the code in the web injected code attack. However, we still do not know if code evolution exists in our selected sample. We use a descriptive analysis approach, and we perform a series of different analyses to answer the following subquestion:

Sub Question 1: What indicators of code evolution can be observed in the dataset?

4.1. Multi-steps attacks computation

In the previous chapter we defined the multi-steps attacks. We determined that we should recognize these attacks as one single attack. However, we still need to link them so we know what is the actual number of those attack.

Once we completed the linking process, we noticed interesting observations for the four malware families. The total number of attacks for ZeuS-P2P dropped from 85,964 to 42,967 (49.6%). Similarly, after we linked together the attacks of the Gozi-EQ malware, we observed that the initial number of attacks (153,232) declined by 36.6% resulting to 139,864 attacks. On the other hand, the initial number of attacks of the Citadel and the Ramnit malware was reduced by only 8.2% and 3.3% respectively. These observations indicate that malware developers of the Citadel and the Ramnit malware have not used the mechanism of multi-steps attacks to the same extent as the cybercriminals of the other two malware families.

The linking process also helped us to count the actual number of multi-steps attacks for each malware family. The ZeuS-P2P attacked the targeted domains with 20,187 multi-steps attacks which resemble the 47.3% of its total attacks. The majority of Gozi-EQ attacks, 80.87% to be exact, were developed using the multi-steps attacks technique. In contrast, only 9.44% (13,208) of the Citadel's attack and 2.59% (760) of the Ramnit's attacks were developed using the same technique. Since not all four malware families follow the same trend, it is difficult to make a coherent conclusion. It is possible that these differences exist because different cybercriminals had different programming skills which allowed them to develop more complex attacks. At the same time, it is also likely that attackers did not use multi-step attacks due to the extra effort and time needed to develop such attacks.

Finally, we discovered a small number of unique attacks for all malware families: (1) ZeuS-P2P: 308 attacks,(2) Citadel: 484 attacks,(3) Gozi-EQ: 700 attacks, and (4) Ramnit: 198 attacks. The small numbers compared to the total number of web injected code attacks shows that cybercriminals have most likely reused the same attack more than once. Table 4.1, displays the total number of attacks for each malware family along with the intermediates numbers of the linking process. To sum up, through this analysis we counted the number of multi-steps attacks for each malware family. We showed that the malware families do not follow the same pattern. Attacks of the ZeuS-P2P and the Gozi-EQ malware used the technique of multi-steps attacks more, compared to the other two

Table 4.1: Information about Mutli-steps attacks per malware family

Malware	# of attacks before linking	# of attacks after linking	# of multi-step attacks	# unique attacks
Zeus-P2P	85,964	42,967	20,187	308
Citadel	153,232	139,864	13,208	484
Gozi-EQ	760,597	276,376	223,515	700
Ramnit	30,267	29,290	760	198

families (Citadel and Ramnit). The small number of unique attacks for all malware families also shows the tendency of the attackers to reuse the same injected code.

4.2. Attack Repetition

After the linking process, we computed basic statistical metrics for the attacks of the four malware families (see Table 4.2). On average each attack was repeated more than 100 times. However, there are many cases where the average value is affected by outliers. Hence, we also calculated the median value which is more robust to outliers and can minimize such effect. The median amount of times an attack is repeated varies per malware families, i.e., (i) 34 times for the Zeus-P2P malware, (ii) 7 times for the Citadel malware, (iii) 27 times for the Gozi-EQ malware, and (iv) 18.5 times for the Ramnit malware. We considered some attacks as outliers using two possible explanations: (1) the longer lifetime compared to others attacks which this is why they repeated more and (2) the attacks which were repeated multiple times at different parts (URLs) of the top attacked domains. In any case, both median and mean suggest that attack repetition does exist among the injected code for each malware.

We also calculated the standard deviation (sd) for each malware family. We observed large standard deviations for all malware families which confirms the presence of outliers and also the large variation in attack repetition of each attack. We took a closer look at the extremes cases of the Citadel and the Gozi-EQ malware where the sd is very high. In the Citadel's case, we noticed that 81 attacks were repeated only one time but a single attack was repeated 20,330 times. Similarly, in the case of Gozi-EQ, we found that 64 attacks had happened once, but there is one attack which was repeated 66,928 times. The fact that the data is so spread out for these two malware families explains the very large numbers of their sd value. Table 4.2 shows the main statistics.

Table 4.2: Statistical values of attack instances repetition for each threat

Malware	Total Attacks	Distinct Attacks	Average	Median	Standard Deviation
Zeus-P2P	42,697	380	138.62	34	764.41
Citadel	139,864	484	288.97	7	1,273.07
Gozi-EQ	276,376	700	394.82	27	2,976.62
Ramnit	29,290	198	147.92	18.5	345.19

We provide additional details regarding attack repetition. We found that 5 attacks (1.62%) of the Zeus-P2P malware were used only once. Interestingly, 26% of the injected codes of the Zeus-P2P malware were repeated 40 times. Almost 50% of the injected codes for the Ramnit malware was repeated fewer than 12 times, with 34% of those attacks happening only once. We also observed some extreme cases, we found that 14 attacks (2.8%) of the Citadel malware were repeated more than 3,400 times. In addition, we noticed that one attack of the Gozi-EQ malware was repeated 66,928 times, and five attacks were repeated more than 11,000 times. These attacks are the outliers which we identified previously. Finally, almost 30% of the Gozi-EQ attacks were repeated less than 10 times.

In short, we found that all malware families follow a pattern of attack repetition. This finding verifies the results of existing research which states that cybercriminals repeat and reuse the same code. However, we still do not know whether code similarity exists in the code of different attacks. Thus, we have to examine the code of different attacks and find out how much code they share. The answer can provide recommendations on financial institutions on how they can improve their security measures and protect their assets against web injected code

attacks.

4.3. Malware Lifespan

Next, we extracted the lifespan of each malware family, and we compared it with the total number of attacks per malware family (see Table 4.2). Please see Table 4.3 for an overview of the lifetime of each malware family.

Looking at Table 4.3, we noticed that the ZeuS-P2P malware was active for 11 weeks. The multinational efforts

Table 4.3: Malware Life Span

Malware	First Seen Attack	Last Seen Attack	Lifespan (weeks)
ZeuS-P2P	2014-02-27	2014-05-15	11
Citadel	2014-02-21	2017-11-03	193
Gozi-EQ	2014-09-11	2017-01-28	124
Ramnit	2016-09-28	2017-10-26	56

to take down the ZeuS-P2P botnet is the reason why the malware has such sort lifetime. Department of Justice [26] and the FBI [37] coordinated these efforts. The fact that a new attack does not appear after the takedown efforts can lead to some introductory explanations regarding the code re-use in the ZeuS-P2P attacks. Since the takedown efforts were successful, attackers might not have any incentives to improve the code or develop new code for ZeuS-P2P attacks to overpass the newest security measures. Moreover, the fact that all attacks were taken down at the same time could mean two things. Either that the difference between their code was minor, which by itself is another indication of code similarity in different attacks, or that these efforts disrupted the whole infrastructure, so no more configuration files could be captured.

On the other hand, the Citadel, and the Gozi-EQ malware are active almost throughout the entire infection time period which suggests that any effort to take down the botnets of the two malware families has proven insufficient. Also, it could mean that the code of their attacks have changed over the time period and hence, the threat is still active. Therefore, it is important to measure and identify those changes. Finally, Ramnit only appears in 2016 and 2017. The reason behind this might be the takedown operations which took place in 2015 [110]. The comeback of this threat indicates that attackers have developed new attacks which have evolved comparing to the previous ones.

The analysis regarding the lifetime of each of the four malware families has shown evidence of code reuse and evolution for all of them. However, it is interesting to examine why the attacks of Gozi-EQ and Citadel lasted longer compared to ZeuS-P2P and Ramnit.

4.4. Attack Duration

Following the analysis we explained in the previous subsection, here we carried out a more in-depth analysis to estimate the weeks an attack was alive during the infection period. The top plot in Figure 4.1 presents the analysis results. It is intriguing that most of the attacks do not last more than 50 weeks. If we take into consideration that the infection period is 193 weeks in total (21/02/2014 - 03/11/2017), we can hypothesize that attackers change the code of their attacks frequently. We also noticed that for all malware families there is a large number of attacks which lasts less than one week. The ZeuS-P2P's attacks are the ones which last the shortest. This is consistent with the previous analysis which has shown that the ZeuS-P2P malware lasted only 11 weeks during the entire time. The Gozi-EQ's attacks did not last for more than 50 weeks.

On the other hand, the Citadel's and the Ramnit's attacks lasted the longest; in some cases more than 150 weeks. This behavior of Ramnit attacks raised some questions since we previously discovered that its lifetime to be only 56 weeks. We used this finding to perform a more detailed analysis which has shown that different malware families used some of the same attacks. For example, attacks which have been used by Ramnit malware had previously been used by Tinba-v2, Qadars, and Kronos malware families. This behavior is a clear indication that attackers actually reuse the same code in different attacks, even attacks coming from different malware families.

The bottom plot in Figure 4.1 displays once again the duration of the attacks. However this time we dis-

played only attacks of the ZeuS-P2P, the Citadel, the Gozi-EQ, and the Ramnit botnets. We can see the changes through the arrows. Besides the case of Ramnit's attacks, the differences between the two plots are extremely small. In any case, both iterations of the analysis have shown that attacks do not last long. Each malware has average attack duration less than 20 weeks: ZeuS-P2P 8 weeks, Citadel 17 weeks, Gozi-EQ 16 weeks, and Ramnit 3 weeks.

The lower duration compared to the entire infection period indicates once again the attackers' incentives to change the code of their attacks. Either due to low success rate or due to their efforts to improve the attacks in an attempt to achieve better results.

4.5. Attacks by different malware families

In the previous analysis, we identified attacks which have been used by different malware families. In this analysis we want to see to what extent this happens for the four malware families we used in this research. Our findings revealed that none of the attacks of the Zeus-P2P and the Gozi-EQ malware were used by other malware families. In contrast, attacks of the Citadel and the Ramnit malware are also used by different malware families. Before we explain in more detail the cases of the Citadel and the Ramnit malware, it is worth mentioning that 790 of the attacks of the complete dataset were used by more than one malware family.

The number of Citadel attacks which used by different malware families is 82. All the attacks were used by the KINS malware, which is also a Banking Trojan malware. Its architecture is similar to the ZeuS and the Citadel malware. In addition, its source code has been linked with the source code of the Citadel malware [79]. Sixty-three attacks are used by the ZeuS and the Ice9 malware. As we already mentioned, the Citadel and the Ice malware families are variants of the ZeuS malware. Therefore, it is possible that these attacks were first developed for the ZeuS malware and later were reused by cybercriminals for the Citadel and the Ice9 malware families. Table 4.4 displays the break down of the Citadel's attacks which used by different malware families.

We found that thirty-seven of the Ramnit malware attacks were also used by the Ramnit-BankerModule which

Table 4.4: Citadel's attacks used by different malware families

Malware Family	# of Attacks
Citadel	82
KINS	82
Ice9	63
ZeuS	63

is directly linked with the Ramnit malware. A C&C server responsible for stealing user's data was shared by different banking Trojan malware including Ramnit, Tinba, and GootKit [114]. This fact can explain the number of attacks which were used by these three malware families. Finally, the reuse of attacks among Kronos, Qadars, Gootkit, and Ramnit malware can be explained by the fact that those malware families were part of ElTest, a malware distribution network [58, 111]. Table 4.5 shows the number of the Ramnit's attacks which used by different malware families. Through this analysis, we obtained evidence which shows that cybercriminals not only

Table 4.5: Ramnit's attacks used by different malware families

Malware Family	# of Attacks
Kronos	7
Qadars	1
Ramnit	71
Timba-v2	36
Ramnit-BankerModule	37
GootKit	2

use the same code of attacks of the same malware family but they also reuse attacks which have been used by different malware families. The majority of the attacks are coming from the malware families which are variants of each other. Other examples have shown that malware families which are linked in the same malware distribution network or sharing the same C&C server, are more likely to share the same attacks as well.

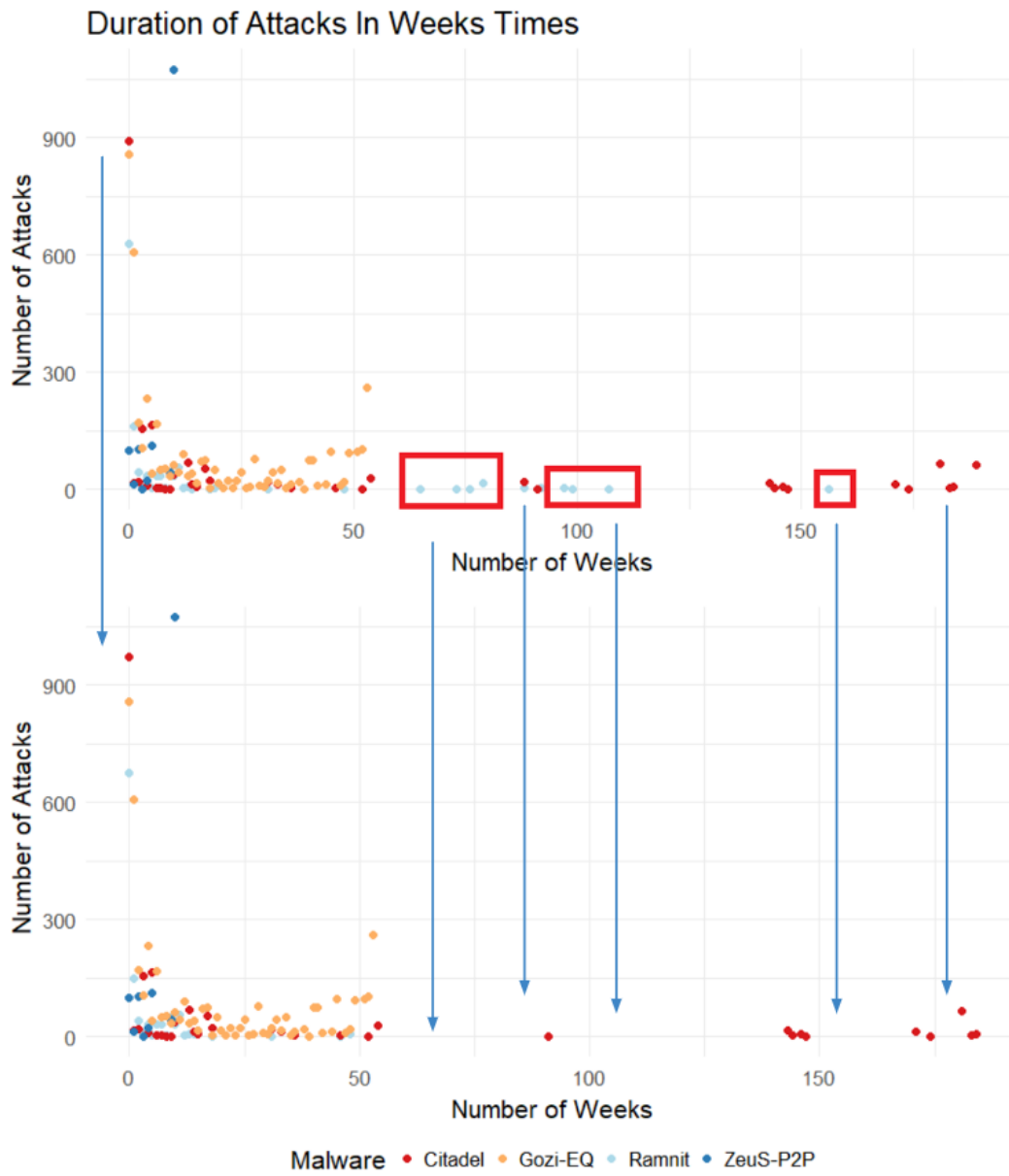


Figure 4.1: Duration of Attacks in weeks time

4.6. Attacks per Botnet

Having in mind the fact that the configuration file(s) of any attack can be linked back to the botnet, we conducted an analysis to connect all attacks with their associated botnet. This analysis can give us an idea of how frequently the configuration files were updated. Note, that we discovered some of the dataset limitations. First, with the current information of FOX-IT dataset, it was impossible to link back any configuration file to the ZeuS-P2P botnets. Regarding the rest of the attacks, still, it was not possible to link all of them with a botnet. For the Citadel and the Ramnit malware, only 2 botnets were captured. In contrast, for the Gozi-EQ malware, we extracted more than 40 samples, linked with 47,528 attacks (with a standard deviation of 208,28 and coefficient of variation of 35,096.37). We made the following assumption to explain why we did not to link all attacks with their associated botnets. It is possible that the decryption of the configuration files was not successful and thus, some attacks are missing their associated botnet. We also extracted the number of unique attacks per botnet. The different numbers reveal the number of times attackers updated the content of the configurations files. Updates in configuration files mean that either the attackers have changed the targeted URL or the code. Regardless, these updates are another indication of code evolution in web injected code attacks. In the following chapter, we provide a more detailed analysis regarding the number of attacks per botnet. In the upcoming analysis, we examined whether different botnets of the same malware family have the same target and how much code similarity exists among their code in such case. The tables in Appendix B break down the exact number of attacks per botnet per malware family.

Table 4.6: Number of Attacks per Botnet for each threat

Malware	# of botnets (RC4 keys)	# of total attacks	# of unique attacks
ZeuS-P2P	n/a	n/a	n/a
Citadel	2	47,528	552
Gozi-EQ	40	9,410	1,527
Ramnit	2	7,946	1,002

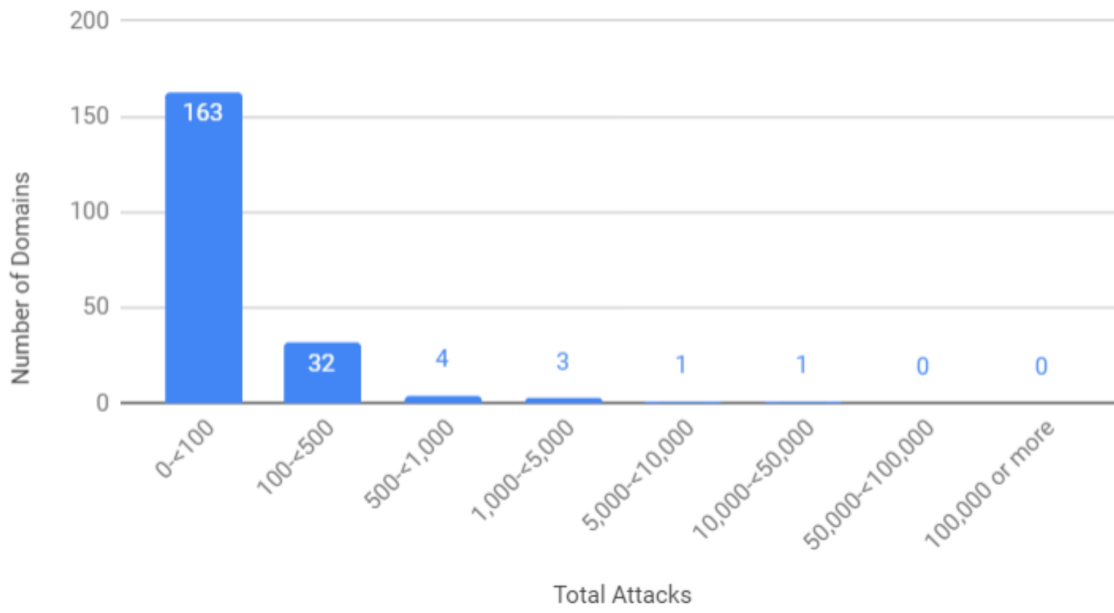
4.7. Attacked Domains

In this analysis, we grouped together attacks based on their targeted domains. Note, that it was not possible to link all attacks with a domain. We explain that using the following reasons. Either the configuration file does not contain a targeted URL or the URLs included in the configuration files are not linked to a domain, due to attackers actions (e.g., use of regular expressions). However, our aim was not to determine the attractiveness of each domain based on the total number of attacks, but rather to discover evidence of code evolution or reuse in the attacks against them.

The four plots in Figures 4.2 and 4.3 display the number of attacked domains for the four malware families. We distributed the domains based on the total number of times which were attacked. Looking at the numbers, we observed that the majority of the domains were attacked less than 500 times during the entire infection time period. The list of those domains includes mostly banks worldwide. The domains which were attacked more than 1,000 times include domains which are among the top financial services sites according to Alexa rating ¹. Therefore, it is possible that attackers are insisting to attack those specific domains due to their popularity and traffic they receive every day.

¹https://www.alexa.com/topsites/category/Top/Business/Financial_Services

ZeuS-P2P



Citadel

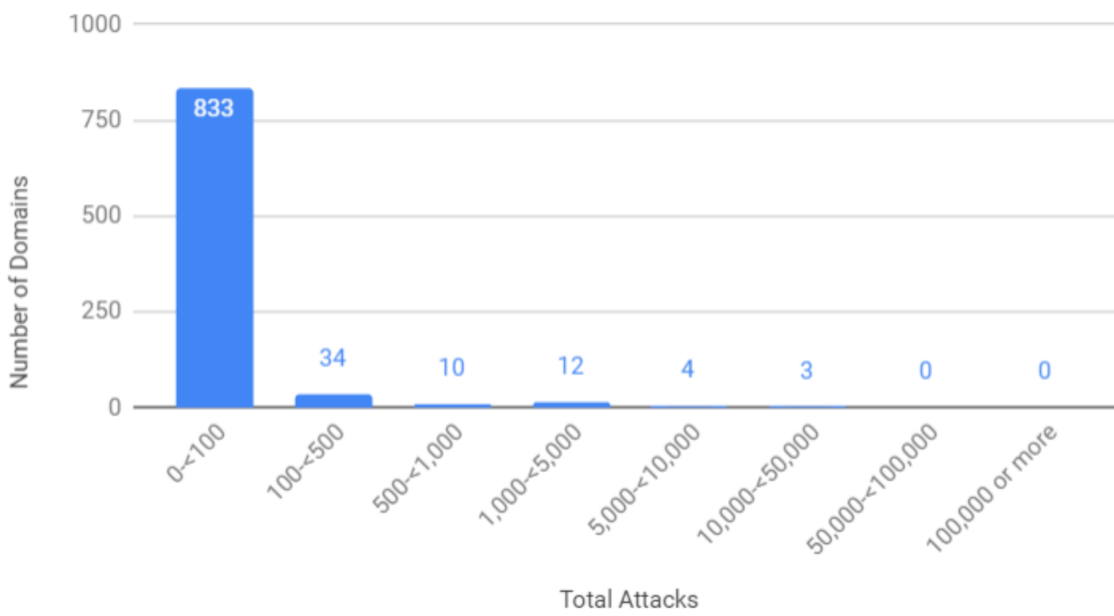


Figure 4.2: Attacked domains for ZeuS-P2P and Citadel malware

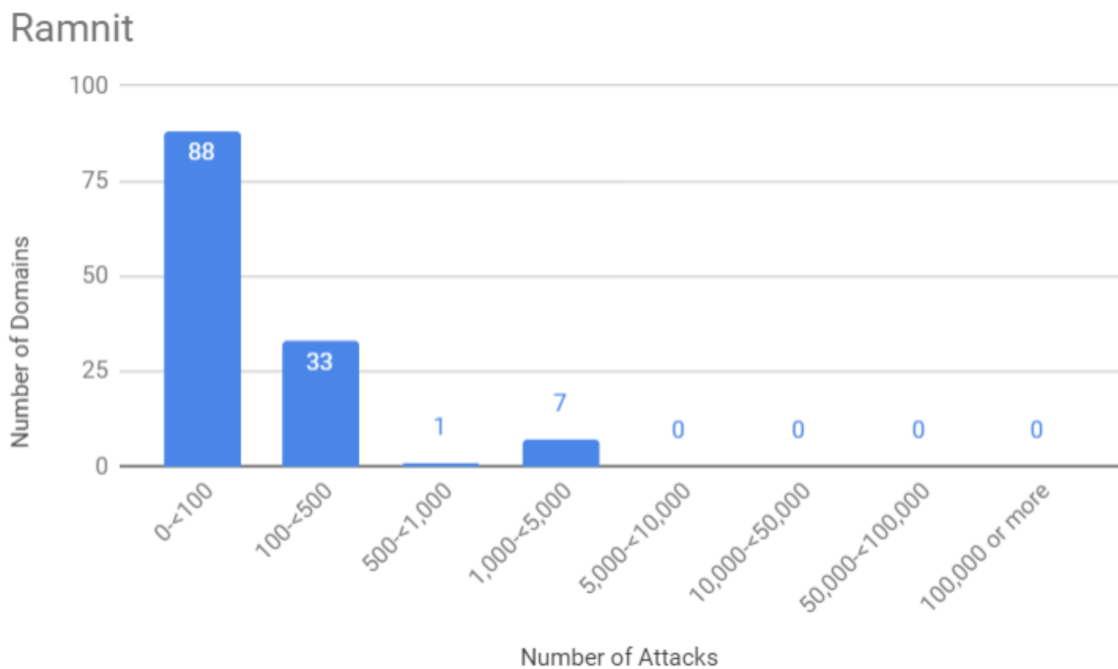
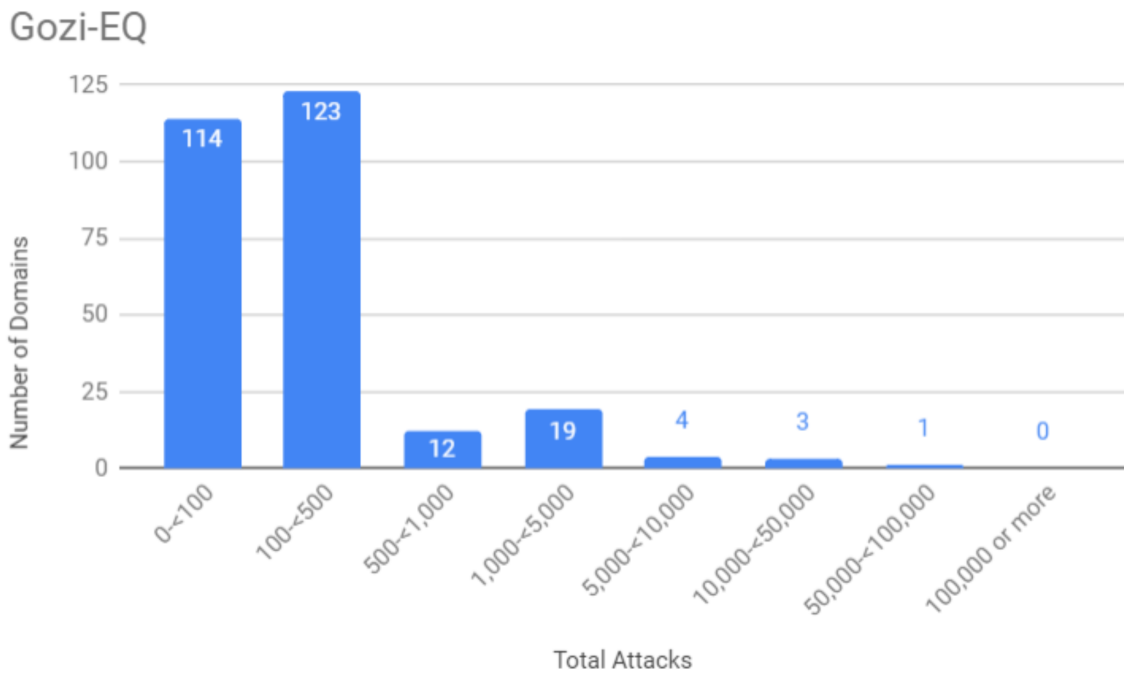


Figure 4.3: Attacked domains for Gozi-EQ and Ramnit malware

In addition, the average time a new attack appears are: for the Zeus-P2P 2 times, for the Citadel 2 times, for the Gozi-EQ 5 times, and for the Ramnit 4 times. We provide two possible scenarios which can explain the numbers: (1) the attacks were successful so attackers did not have any reason to change the injected code, and (2) the attacks were not successful and attackers changed their targets, instead of developing a new attack to overpass

the security measures of the domain. We also noticed cases where domains were attacked with more than 20 unique attacks. From this, we can conclude that attackers frequently change the way (code) they attacked those specific domains. The changes might happen because attackers found those domains extremely attractive and they continued to target them no matter what security measures were taken by the defender side. In any case, the injected code used for these domains have changed over time. Therefore, it is important to identify how much the code has changed and if the different attacks share a portion of the same code.

To summarize, in this section we grouped the targeted domain of each malware family. We showed that the majority of times, attackers do not target a domain more than 500 times. We determined that the top-attacked domains are among the top financial services sites.

4.8. Summary of Findings

In this chapter, we conducted a series of descriptive analyses to describe the data and find indications of code reused or evolution in the injected code of the attacks. We summarized our results as follows:

- The ZeuS-P2P and the Gozi-EQ malware used the technique of multi-steps attacks more, compared to the other two families (Citadel and Ramnit).
- Attackers tend to reuse the same attack; the majority of attacks in this research were used more than once during the time period in which the malware was active.
- The lifespan for the majority of the attacks for all malware does not last as long as the malware's lifespan. This point out that attackers updated the content of the configuration file (code) more than once during the malware lifespan, which is a sign of code evolution.
- Attacks of Ramnit and Citadel have been also used by other malware families which proves that cyber-criminals not only use the same code in attacks of the same malware family but also in attacks of different malware families. The majority of those attacks are from malware families which are variants of each other. Other examples have shown that malware families which are linked in the same malware distribution network or share the same C&C server, are more likely to share the same attacks as well.
- The different numbers of attacks per malware botnet, reveal the number of times attackers updated the content of the configurations files which can be used as another indication of code evolution in web injected code attacks.
- The majority of times, attackers do not target a domain more than 500 times.

Following the sub-question mentioned in the introduction of the chapter, we extracted enough pieces of evidence regarding code evolution and reuse. It is clear that all attacks have either changed or reused over time. However, it is still unknown to what extent they have changed and how similar they are to previous versions. In the next chapter, we conduct a literature review to identify which is the most suitable tool for measuring the code similarity among these attacks.

5

Code Similarity Tools

Recalling the literature review, different banking malware reuse a significant portion of the same code. However, variations are also introduced. In this chapter, we aim at assessing different tools that allow to quantify to what extent attackers recycle code, i.e., tools to measure the similarity among different attack codes. To reach this goal, first we start by identifying which of the existing tools can detect better the similarity among the injected code. In order to address this, we formulate the following sub-questions in the chapter:

Sub Question 2a: How can we measure code similarity among different injected code attacks?

Sub Question 2b: What are the limitations of the available similarity tools and how do they impact the similarity metrics?

The task of detecting code similarity in software programs is a problem that has been the subject of research for many years. Researchers have developed many tools that can detect code similarity. Walenstein and Lakhoria [115] argue that malware is just software that contains malicious code. Therefore, they claim that methods used for comparing software similarity can also apply to malware code. In the following sections, we compare and review existing code similarity tools from existing literature. The review will determine the sensitivity of the resulting similarity metric to the tool. We limited our measurement to those tool that were available at the time of this research and were able to measure similarity of JavaScript code.

5.1. Clone Detection

Cloned or duplicated code is defined as the code which is copied and probably modified in a new file. The process of finding such code is called clone detection. In the survey of Koschke [63] on software clones and in the literature review by Djurić and Gašević [27], different types of code clones and code modification techniques are reviewed. In Table 5.1, we present the three types of code clones based on the work of Koschke [63].

Table 5.1: Type of clones by Koschke (2007)

Type 1: exact copy without modifications (except for white-spaces and comments)
Type 2: syntactically identical copy (changes to variables, types or function identifiers)
Type 3: copy with further modifications (statements modifications, additions, deletions)

Djurić and Gašević [27] use the same concepts as in Ciere et al. [21], Jones [50], Joy and Luck [51], Kustanto and Liem [66], Liu et al. [70], Roy and R. Cordy [92] to classify source code modifications in two main categories: (1) *lexical* and (2) *structural*. They also provide examples for each category. In the current research, we merge the three types of code clones and the classification of source code modifications in one taxonomy which is visible in Table 5.2. We extracted the following conclusions:

- Type 1 clones are all lexical modifications and require less knowledge and programming skills compare to the other two types.
- Type 2 clones are a mix of lexical and structural modifications.
- The majority of the examples for type 3 clones are structural modifications with more advance changes which will probably be more difficult to detect.

Table 5.2: Source Code Changes Classification

Clone Type	Example	Modification
Type 1	modification of source code formatting addition, modification or deletion of comments language translations modification of program output	Lexical
Type 2	renaming of identifier modification of constant values	Lexical
	changing of data types and modification of data structures	Structural
Type 3	addition, modification or deletion of modifier split or merge of variable declarations	Lexical
	changing the order of variables in statements changing the order of statements within code block reordering of code blocks addition of redundant statements or variables modification of control structures method inlining and method re-factoring redundancy temporary variables and sub-expressions structural redesign of code modification of scope	Structural

5.1.1. Clone Detection Methods

Koschke [63] and Roy and R. Cordy [92] in their research surveys on Software Clones define five main clone detection methods that have been used to detect code clones.

Methods based on text comparison

The code similarity result is dependent completely on the comparison of the text and strings in the code files. These methods do not take into consideration any structural changes in the code which is their main weakness. Using the classification (see Table 5.2), we hypothesized that these methods are detecting better the Type 1 clones. Johnson [48] and Johnson [49] use the Karp-Rapin fingerprint algorithm [55, 56] to detect similar substrings of the code. Manber [73] also uses fingerprints to detect similarity. Ducasse et al. [29] and Ducasse et al. [30] develop a language-independent simple line-based string matching approach to detect duplicate code. Their approaches use dot-plots to visualize the results of the duplicated code. Wettel and Marinescu [118] also use the dot-plots approach. Baker [8] compares the code, line by line but he also uses tokens which is a different method which we explain below.

Methods based on token comparison

In this method first, the code is converted to sequences of tokens. Then, the sequences are checked to identify the same sub-sequences of tokens. These sub-sequences represent the cloned code. As we mentioned earlier, the Baker [8] approach is a combination of text and token based comparison. The code is firstly tokenized and then is compared line by line using a suffix tree. Kamiya et al. [53] use the same approach. They use a suffix tree matching algorithm to find and return the sub-sequences of tokens that are more likely to be the duplicated code. Before the comparisons, the tokens are normalized. Basit et al. [9] develop an approach which uses a suffix array instead of a suffix tree. They perform experiments and according to their results, their approach has more flexibility, precision and perform better than Kamiya et al. [53] approach.

Methods based on tree comparison

Tree-based methods first convert the code into a parse tree or an abstract syntax tree (AST). Then tree matching algorithms are applied in order to find similar sub-trees. The similar sub-trees correspond to the cloned code. Baxter et al. [10] use annotate parse tree and hash function to find and return similar sub-trees. A similar approach is also developed by Raza et al. [87]. Koschke [63] develop an approach in which first the AST sub-trees are serialized, and then a suffix tree is created to compare these AST sub-trees. This approach enables finding code clones more quickly by avoiding comparing all sub-trees. On the other hand, Yang [123] uses a dynamic programming approach for comparing and detecting the most similar sub-trees.

Methods based on program dependency graph comparison

Graph-based methods are using a higher level of abstraction to represent the code using its semantic information. The code is presented as graph taking in consideration control flow and data dependencies. Then using algorithms developed for finding isomorphic sub-graphs, the most similar sub-graphs are returned which correspond to the cloned code. Komondoor and Horwitz [61] use program slicing to find isomorphic sub-graphs. Krinke [65] propose an approach to find the maximal similar sub-graphs using the k-limiting technique.

Methods based on metric comparison

Metrics-based methods are using different metrics for code fragment to detect code similarity instead of comparing the code itself. Mayrand et al. [74] use metrics like the number of lines of the code to measure the functions that have similar metric values, these functions are considered cloned code. Metrics such as the number of variables is used by Patenaude et al. [82] to detect similar methods. Kontogiannis et al. [62] measure the similarity of the code by using a pattern matching tool that uses Markov models. Metrics-based methods have also been proposed to detect duplicated web pages by Lucca et al. [71] and Lanubile and Mallardo [67].

5.1.2. Clone Detection Tools

Different tools have been developed using the aforementioned methods. The survey by Roy and R. Cordy [92] presents an extensive comparison and evaluation of these tools. We present some of the tools in Table 5.3 using the same information as Roy and R. Cordy [92]. In subsection 5.1.2, we summarize the main findings of the survey. The authors evaluate the tools using existing literature and their own experimental evaluation.

Table 5.3: Clone Detection Tools by Roy and R. Cordy (2007)

Tool	Proposed by	Supported Languages	Approach	Use
DupLooc	Ducasse et al. [30, 31]	Language In-dependent	Line / Exact string matching	Academic
Dup	Baker [8]	C, C++, Java	Line-based/text-based	Academic
CCFinder	Kamiya et al. [53]	C, C++, Java and others	Token comparison with suffix tree	Academic
RTF	Basit et al. [9]	C	Token with suffix array	Academic
CloneDr	Baxter et al. [10]	C, C++, Java and others	AST/Tree matching	Commercial
ccdmil (Bauhaus)	[87]	C, C++	AST/Tree matching	Academic
PDG-DUP	Komondoor and Horwitz [61]	C, C++	PDG/Slicing	Academic
Duplix	Krinke [65]	C	PDG, graph matching	Academic
Covet	Mayrand et al. [74]	C and others	Metrics	Academic

Tools Evaluation

As we mentioned earlier, in this research we use information from Roy and R. Cordy [92] regarding the evaluation of the different clone detection tools. In their survey they provide different ways and level of comparison to evaluate the different approaches and tools. They characterize their first comparison approach as a high-level comparison and use different parameters. The findings of this first approach can be summarized as follows:

- Text-based approaches are easily adaptable to new programming languages. They cannot detect many clones because they find only duplicated code. On the other hand, they have high precision due to the textual similarity. Their scalability level depends on the comparison algorithm they use.
- Token-based approaches are language dependent. They can detect many clones, but their precision is low due to the transformation process to tokens. Their scalability level can be high when they combined with suffix tree algorithms.
- Tree-based approaches cannot support different programming languages. In general, they cannot detect many clones because they take into consideration the structure of the code, but they have high precision. Their scalability level depends on the comparison algorithm they use.
- Graph-based approaches have the same characteristics as the tree-based approach, but their scalability is very low.
- Metrics-based are language dependent, cannot detect many clones but they have a good precision level with high scalability.

For the evaluation of the tools Roy and R. Cordy [92] use existing literature. In this research, we present a summary of their findings. The authors present the experiment conducted by Bellon [11]. The author compares the tools in terms of precision, recall and on how well they detect the three types of clones. Bellon [11] uses an interval scale (++ is the best and – the worst) to measure and rank the different tools. The results show the same behavior as the high-level comparison provided by Roy and R. Cordy [92] which conclude that if a tool has good precision, its recall is not very good and vice versa. However, this is not the case for the PDG approach where both precision and recall are bad. In terms of detecting the clone types, it seems that the token-based approach has better results, but none of the tools examined in the experiment can detect clones type 3. We summarize the results in the Table 5.4.

Koschke et al. [64] use the same approach to compare their tool, cpdetector. They develop the tool using an abstract syntax suffix tree approach (ASST). Roy and R. Cordy [92] presents the results of Koschke et al., in

Table 5.4: Summary of comparison experiment by Bellon (2002) as presented by Roy & R. Cordy (2007)

Tool	Approach	Precision	Recall	Clone Type		
				Type 1	Type 2	Type 3
Duploc	Text	-	+	+	-	-
Dup	Token	-	+	++	+	-
CCFinder	Token	-	+	++	++	-
CloneDr	AST	+	-	+	-	no
Duplix	PDG	-	-	-	-	-
CLAN	Metrics	+	-	+	-	-

terms of precision, recall, and clone types using an interval scale (- is the worst and +++ the best). The results are similar to the results of Bellon. It seems that token based tools have on average better recall overall, but their precision is not as good as AST and metrics tools. Text-based and token-based tools can detect better clone types 1 and 2, than type 3. The PDG tool is the only one that can detect well the clone type 3 but it not good for detecting type 1 and 2 and again its precision and recall are bad. We summarize the results in Table 5.5.

Table 5.5: Summary of comparison experiment by Koschke et al. (2006) as presented by Roy & R. Cordy (2007)

Tool	Approach	Precision	Recall	Clone Type		
				Type 1	Type 2	Type 3
Duploc	Text	+	-	++	-	-
Dup	Token	+	+-	++	+	-
CCFinder	Token	+	++	++	++	-
CloneDr	AST	++	-	++	-	no
Duplix	PDG	-	-	-	-	++
CLAN	Metrics	+++	-	++	+	-
cpdetector	ASST	+	+-	++	+	-

5.1.3. Other clone detection tools

Kim et al. [57] design the VUDDY tool which detects code clones Type 1 and Type 2. The authors develop abstraction rules that are using along with normalization before creating the fingerprint dictionary of the code file. Due to the abstraction rules, the VUDDY is able to detect all Type 2 clones. According to the authors, the fingerprint dictionary is important for the development of VUDDY because it reduces the comparisons between files during the clone detection phase. They evaluate the tool's efficacy and effectiveness in terms of scalability and accuracy. They compare VUDDY with other code clone detection tools, and they find that VUDDY has better results. VUDDY is available for use ¹, so far it only supports the C/C++ programming languages. ReDeBug is one of the tools has worse performance than VUDDY. ReDeBug is a syntax based clone detection tool which is able to detect unpatched clones [47]. The files before the comparison are normalized (e.g., remove white-spaces) and tokenized. The evaluation of the tool shows that ReDeBug has good results in terms of scale, speed and false detection rate, but detects fewer code clones compare to other tools.

Sajnani et al. [97] develop SourcererCC ², which follows a token-based approach for clone detection. According to the authors, the tool can detect Type 3 code clones which most of the tools are unable to do it. The SourcererCC tool divides the code to code blocks and then uses a bag-of-token technique for the code blocks comparison. This technique is the reason that is able to detect Type 3 clones. The code similarity is measured by comparing the different code blocks. The authors evaluate the tool in terms of scalability, execution time, recall, and precision. They use two clone benchmarks tools, the Mutation Framework and BigCloneBench to measure the recall rate of the tool. They find that it has good recall rate and compare to the Mutation Framework has perfect recall even for Type 3 clones. They measure the precision manually and the results show good precision rate. The authors make the tool available, but users have to implement their own parser to run successfully the clone detection process.

In summary, we reviewed different clone detection methods and tools. The information from existing literature

¹<https://iotcube.net/>

²<http://mondego.ics.uci.edu/projects/SourcererCC/>

proves that there is no tool that can outperform the others. On average token based approaches have better precision and recall rate. They can detect very well Type 1 and Type 2 clones, but they are unable to detect well Type 3. However, no tool can support JavaScript language which is the one used in injected code attacks. Therefore, we did not select any of those tools.

5.2. Plagiarism Detection

Plagiarism detection is similar to clone detection. The main difference is that in plagiarism detection the duplicated code is deliberately added in a way that is more difficult to detect. The code used in different malware is probably modified so it cannot be detected easily. Thus, it is possible that a tool developed for plagiarism detection is able to detect better any changes in injected code. In this section, we review existing plagiarism detection tools.

5.2.1. Plagiarism Detection Techniques

Several tools have already been developed which can detect plagiarism between code files. The tools are developed based on three approaches.

Text-Based

The text-based approach is similar to the text-based approach for clone detection in the sense that it does not take into consideration the syntax of the code when calculating the similarity. It just compares the code based only on the use of the English language.

Attribute-Based

The attribute-based approach uses the properties of the code (e.g., the number of unique operators [28]) when comparing two code files. The values of those properties determine the similarity between the two files.

Structure-Based

The structure-based approach uses the structure of the code when comparing code files. Tokenization and string matching algorithms are also used to determine the similarity between code files. Two of the most known and used plagiarism detection tools: JPlag [85] and MOSS [98] are structure-based.

5.2.2. Tools Comparison

Hage et al. [44] compare the performance of five plagiarism detection tools: JPlag, MOSS, Marble [43], Plagie [1], and SIM [42]. They perform a sensitivity analysis to examine which tool performs better when different plagiarism techniques applied to the code. Moreover, they examine the precision of each tool by comparing their results in a top-n comparison. Their results show that no tool outperform the other, but all of them have strong and weak points. Heres and Hage [45] compared the performance of 9 plagiarism detection tools, including JPlag and MOSS, using quantitative metrics: the F-measures and the area under the precision-recall curve. They find that overall all the tools show similar results but MOSS have on average the best performance on both metrics. Chen et al. [20] used SID (Software Integrity Diagnosis system) and showed that it performs better over other plagiarism detection tools including MOSS.

Burd and Bailey [14] evaluates three clone detection tools and two plagiarism detection tools in terms of precision and recall. The CloneDr, an AST based approach tool has the best results in terms of precision (100%) meaning that does not return any false positives. However, the tool has the worst recall rate (9%) meaning that the tool cannot find correctly all clones. Moreover, CloneDr cannot detect clone Types 2 and 3. It is only able to detect Type 1. On the other hand CCFinder, a token-based approach has the best recall rate (72%) indication that can find a good percentage of the clones. The two plagiarism detection tools have similar precision and recall rates as the clone detection tools. We present the results in Table 5.6.

In summary, we presented different code plagiarism tools. The information from existing literature proves that similar to clone detection tools there is no tool that can outperform the others. A study has shown that plagiarism detection tools have pretty similar precision and recall rates compare to the clone detection tools. From the tools analyzed in that research, only MOSS can support JavaScript language. Therefore we selected the MOSS tools as one of the tools that we will use for comparing the code similarity. We also selected the Sherlock tool which is a language-independent plagiarism tool. We provide more information regarding the two tools in a later section of this chapter.

Table 5.6: Summary of comparison experiment by Burd and Bailey (2000)

Tool	Precision	Recall
CCFinder	72%	72%
CloneDr	100%	9%
Covet	63%	19%
JPlag	82%	12%
Moss	73%	10%

5.3. Malware Similarity Detection

Karim et al. [54] use techniques from bioinformatics to build malware phylogeny models which can be used to measure the evolution and similarity of malware. They have n-grams and n-perms as similarity measures and they use the code assembly opcode sequences form. They find that n-perms have better similarity scores. Limitation of their work is the fact that the model measures evolution relate only with the order of the code. In their white paper, Walenstein et al. [116] adapt techniques from text-based search to develop the Vilo tool. Vilo is a method which searches and returns from a database, malware which has similar code as the malware used in the search query. They also use n-grams and n-perms features to compare the disassembled code of the malware.

BinClone is a clone detection tools which can detect code clones in malware binaries files developed by Farhadi et al. [35]. The same authors also develop ScalClone which is based on their previous work. ScalClone also detects code clone in the assembly form of the code with high recall rate and can support large-scale assembly code [36]. The work by Charland et al. [15] is also a clone detection tool which detects similarities in the binary form of the code.

Chen et al. [19] use the NiCad clone detection tool to find malware code clone which is used in Android applications and uses a text-based approach. NiCad applies static analysis, so the comparison uses the source code of the Java applications. The authors use training and test set of applications which contain malicious code. The evaluation of their work shows that the tool can detect 95% of already known malware, with low false positive and accuracy rate of 96.88%. NiCad so far doesn't support JavaScript source code files by default. If a user wishes to use the tool with JavaScript it is necessary to develop TXL parser for the JavaScript language and the granularity file for the language.

5.4. Selected tools

In this section, we explain in detail how the selected tools work, and we also use the JSCPD tool. In Appendix C we provide the steps which are required for running each tool.

MOSS

MOSS or *Measure Of Software Similarity* is the work by Schleimer et al. [98]. The tool is mainly used by teachers for detecting plagiarism in programming courses. MOSS is offered as web-service and can be used for free by everyone who wishes to obtain a MOSS account³. The MOSS subscription script is running on Linux platform, and with Cygwin, it may work for Windows too.

Currently, MOSS supports 25 programming languages, but it is extensible only by the authors. Additionally, MOSS supports file to file comparison and submission comparison. It also allows the use of a base file code which is ignored during the comparison. MOSS compares the standardized version of the document when measuring their similarity by using *winnowing*, a fingerprint algorithm [98]. The fingerprinting algorithm firstly divides the document into *k*-grams and then hashes each *k*-gram. *k* is selected by the user. After the hashing of the *k*-grams, a subset is selected as the document's fingerprints. The difference between MOSS and other fingerprints applications is the use of the *winnowing* algorithm which selects with an efficient and more robust way the *k*-gram subsets.

³<http://theory.stanford.edu/aiken/moss/>

Sherlock

Sherlock⁴ is a plagiarism detection tool developed by the University of Warwick and available for use by everyone. Sherlock is language independent, can detect plagiarism in source code files and natural language documents [119]. The tool performs pairwise incremental comparisons. It compares each pair of files three times: (1) in the original form of the files, (2) by removing the white-spaces and all comments of the files and (3) by transforming the files in a sequence of tokens [52]. Then it summarizes the results in a list, which contains all the pair comparisons along with their similarity percentage. Moreover, Sherlock can visualize the similarity results in the form of a Kohonen-self-organizing map [60]. In the map, nodes are the code files, a line between two nodes exist only if the two file are similar. The length of the line is based on the length of similarity (short line – strong similarity). In the cases where a similar pair of files do not share any similarity with the rest of the files, a separate cluster is created [52].

SourcererCC

The SourcererCC tool is not designed to detect code clones in the source code of JavaScript files. However, it can be extended to this goal with an external parser and a tokenizer. The output of the tokenizer has to follow a specific form so not every JavaScript tokenizer which is publicly available is suitable. In his thesis Žitný [125], studies JavaScript files which are hosted in Github repositories to identify possible duplicates. He uses the SourcererCC and a tokenizer which he developed. This tokenizer is publicly available in the author's repository⁵. We conducted tests to investigate if the tokenizer can successfully be used for the JavaScript code of the web injected code attacks. The tokenizer is unable to successfully tokenize all the injected codes, due to the different level complexity. Therefore, we decided to do not use SourcererCC for the code comparisons.

JSCPD

JSCPD⁶ is an open source tool under the MIT license, which is used to detect duplicate lines in the source code. It supports 28 programming languages and it is extensible only by the author upon a request. JSCPD is executed through the terminal by a command client line and offers some options for the users. Some of its options, it is to set the minimum size of the duplication in tokens(or lines) or to ignore the comments in the code during the comparison. The output of the comparison is offered to the users with two options, either as an XML report or as a JSON file.

Tools Limitations

Unfortunately, the selected tools have some limitations that could potentially impact the final results. This subsection explains these limitations.

Moss does not perform the pairwise comparison for all the possible pairs available in the subset. Studies showed that this lead to low recall rate which is one of the tool limitations. The file to file comparison option, instead of submission comparison can solve this problem. However, the file to file comparison can be very time consuming when the number of code files is very large. Moreover, in the cases of the file to file comparison the extraction of the results requires extra effort. We have to create a pipeline which (1) creates and executes the command for the Moss script for all available file pairs, (2) receives the URL which gives access to the comparison results, and (3) parses the result page and saves the percentage of similarity. At the same time, there are cases where the injected code includes not only JavaScript code but also HTML or CSS content. Moss supports comparisons using one programming language each time. Hence, the similarity percentage of those case is impacted by this limitation.

Even though JSCPD scan all code files submitted for comparison, it only performs comparisons to the code files which have duplicate lines. Hence, the final results are based only on the degree of duplication exist in the files. This weakness of the tool to overlook popular code modification methods like variables renaming or code reordering makes its result less accurate compared to the other selected tools.

The main limitation of the Sherlock tool is its inability to extract or parse the final results which contain the similarity of all pairs. This forces to a manual interpretation of the results which makes the automation of the whole process impossible. In the cases where the number of code file is relatively large, the calculation of the similarity for all the returned pairs is notably time-consuming. Another limitation of Sherlock is its dependency

⁴<https://warwick.ac.uk/fac/sci/dcs/research/ias/software/sherlock/>

⁵<https://github.com/jakubzitny/SourcererCC>

⁶<https://www.npmjs.com/package/jscpd>

on the length of the string used for the comparison. However, since it is almost impossible to predict beforehand the correct value of the string length for the different comparisons, the final percentage can be easily influenced by this option.

To conclude, the main weakness of all selected tools is their inability to be used in an automated fashion. Considering the large number of code files per malware family which are compared, the use of these tools would make the calculation, extraction, and interpretation of the similarity results extremely time-consuming. As a result, this limitation can decrease the scalability of our research.

5.5. Scalable Similarity Metrics

In the previous section, we determined that the selected tools are unable to scale which can limit the external validity of our results. To overcome this scenario, we decided to use metrics based on string matching as a replacement for the tools. We decided to use the Levenshtein distance and the `diff`⁷⁸ command. The Levenshtein distance measures the distance between two strings [68]. The distance between two strings is defined as the minimum number of steps required per character (insertions, character substitutions, or deletions) to change one string into the other. The `diff` command compares two files and returns the lines which are different.

5.5.1. Similarity tools comparison

The examination of the selected tools identified their main weakness to be their inability to automate the process of code comparison. Having in mind the large number of code files per malware family, the use of the selected tools would make the calculation, extraction, and interpretation of the similarity results extremely time-consuming. Therefore, and to overcome this scenario, this thesis examines if the already implemented Levenshtein can be used as a replacement of the tools⁹.

For the evaluation of DL we compare DL's results against the three tools we selected earlier. We used injected code samples for two domains in these tests. Moss and DL follow the same trend. Once the code of the following attack changes, either by increasing or reducing the percentage of similarity the two tools are capable to identify it with very similar results in 80% of the cases. At the same time, Sherlock is not able to detect such changes. JSCPD has mixed results. First, its results are completely different from the other tools in 70% of the cases. Furthermore, there are cases where the tool was not able to detect changes in the code. You can find an overview of our results in Table 5.7 and 5.8.

Table 5.7: Domain 1: Code Similarity Metrics Evaluation

AttackID	MOSS	Sherlock	JSCPD	Damerau–Levenshtein
7026	-	-	-	-
7434	0	0	14.63	0
10528	0	0	27.9	30
13352	0	0	0	0
13353	96	0	0	100
17636	0	0	29.06	0
17774	0	0	29.06	0
216922	0	0	0	0
228386	0	0	0	0
228387	0	0	0	0
228397	0	0	14.71	0
232968	0.5	0	5.76	2
232969	5	0	0.41	0
243591	0	0	0	0

Looking at the two tables, there are some examples which are worth examining in more detail, since the behavior

⁷<http://savannah.gnu.org/projects/diffutils/>

⁸<https://www.computerhope.com/unix/udiff.htm>

⁹While in the rest of the thesis we will use the Levenshtein distance (LD) to measure the distance between two strings, in this section we will use the Damerau–Levenshtein (DL) which is like the Levenshtein distance, but transposition of adjacent symbols is allowed. [24, 68]

of the tools is completely different. In Table 1 for the first domain, the code comparison between the code of the *attack_13352* and *attack_13353* returns 96% similar for Moss and 100% for DL which means the DL distance is 0. In contrast, Sherlock and JSCP show that the code is completely different. We examined the code manually, and found that it is indeed exactly the same source code. In this specific case Moss and DL estimated the similarity correctly, and DL even more accurately. The code of those files contains only commands with the script tag (i.e. `<script>`). JavaScript code uses these command by default, and this is why JSCP ignored the duplication. Sherlock fails to detect the similarity, due to the default length used for the string comparison. In this specific case Sherlock's dependency on the string length lead to misleading results. Other examples worth mentioning are the code comparison between the code of the (1) *attack_13353* and *attack_17636* and (2) *attack_17636* and *attack_17774* where JSCP returned a percentage of duplication, and the three tools return zero similarity. Again, we manually validated these results, and found that the code was indeed completely different. The reason why JSCP detected a difference is that it found duplication in the same file, but not between the two files. These examples show that JSCP and Sherlock are not suitable tools for code comparison among different web injected code attacks. The analysis shows that Moss and DL follow the same trend which is also visible in Figures 5.1 and 5.2.

Table 5.8: Domain 2: Code Similarity Metrics Evaluation

AttackID	MOSS	Sherlock	JSCP	Damerau-Levenshtein
15545	-	-	-	-
218458	0	0	0	0
221268	96	99	47	96
221284	96	99	47	96
221724	96	99	47	97
227192	99	100	69	99
227207	96	99	47	97
230295	96	99	47	96
231775	4	0	0	37
238832	99	100	32	99
238936	0	0	0	0
240366	0	0	0	0
240374	99	99	38	99
243300	99	98	26	99
258413	99	99	38	99
259326	4	0	0	36
295836	99	99	64	99

To summarize, the aim of this section was to determine if a traditional distance metric such as the Levenshtein distance can be used as a similarity metric instead of the three aforementioned selected tools. The reason behind the decision to find an alternative tool, is the fact that the three tools do not offer the ability to automate the code comparison process. The implementation of the DL distance can enable such automation. In order to determine if the use of the DL distance has good enough results to be the one used for the code comparisons, experiments were conducted in which the performance of DL was compared against the three other tools. The injected code used in the attacks against two domains was used as a test sample. The results show that Moss and DL follow the same trend and are able to detect changes in the code. On the other hand, Sherlock and JSCP have mostly negative results and thus are not suitable to be used for the injected code comparisons. Based on these findings it can be concluded that Moss is the most suitable tool among the original three, and since Moss and DL followed the same trend, DL can replace the Moss tool since it offers the ability to automate the comparison process.

Summary of Findings

In this chapter first, we introduced the notion of clone detection, including methods and tools which have been developed for this task. The literature did not provide a definite answer on which tool is the most suitable for malware code comparison. We reviewed the state-of-the-art tools for clone detection. The tools have very good results in terms of accuracy and efficiency, but they don't support JavaScript code by default. We introduced the

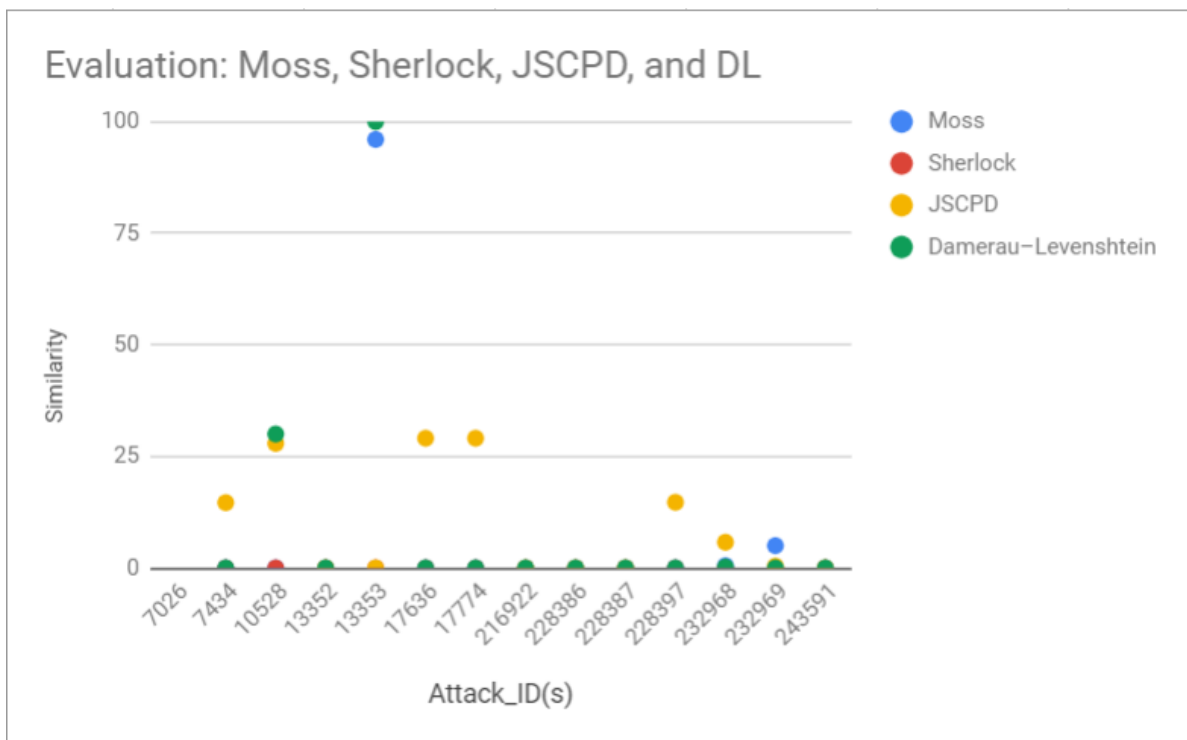


Figure 5.1: Domain 1: Code Similarity Metrics Evaluation

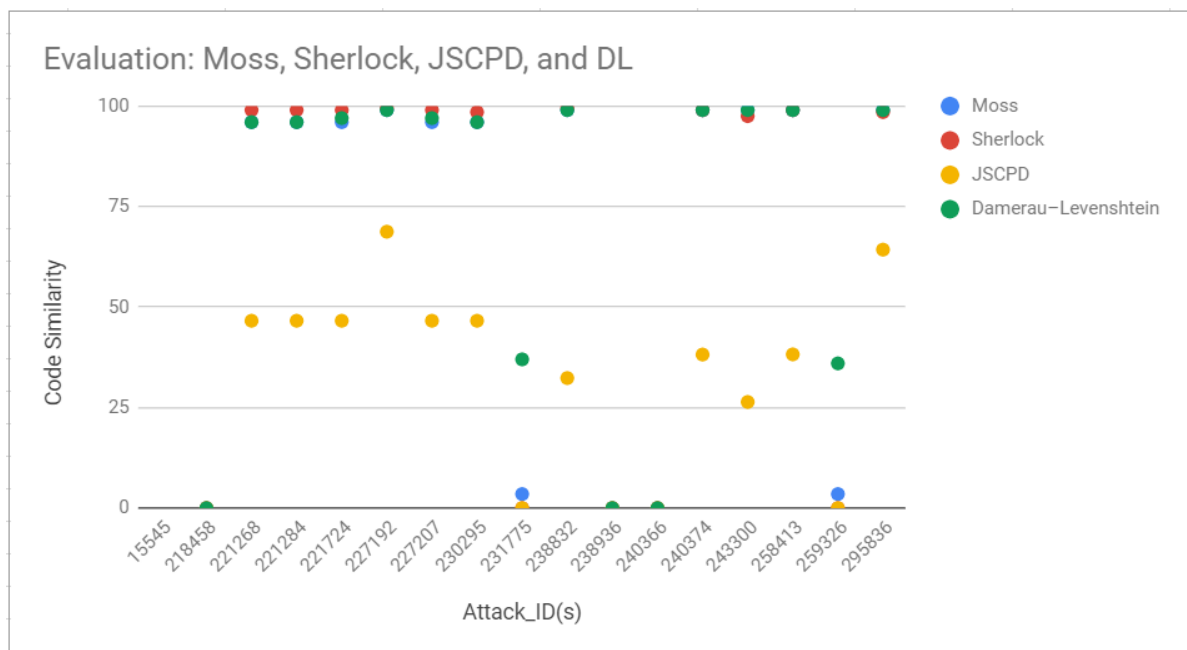


Figure 5.2: Domain 2: Code Similarity Metrics Evaluation

plagiarism detection problem. The literature review showed that there is not a plagiarism tool which outperforms all the others. Their performance depends heavily on the comparison experiments. However, from all the available tools only MOSS and Sherlock support the JavaScript language. We also reviewed methods for detecting code similarity among different malware. We selected the MOSS, the Sherlock, and the SourcererCC as the most suitable tools to use to detect similarity among different attack instance of injected code. Finally, we also presented simple metrics based on string matching that can serve as measures of code similarity while being scalable.

6

Code Similarity Analysis

In this chapter, we answered the following research question: **Sub Question 3:** How much does code re-using, stealing or sharing exist across different instances of injected code attacks?

We started our analysis by comparing all pairwise combinations of our code samples using the Levenshtein distance. We visualized the resulting similarity in a self-similarity matrix. We sorted all attacks in the horizontal and vertical axis according to the first time they targeted a domain. The main diagonal of the matrix indicates where the self-similarity between the same code samples.

6.1. Overall Injected Code Similarity

Figure 6.1 shows the similarity matrix computed from all the injected codes in our sample. Each pixel corresponds to the normalized Levenshtein between 2 attacks. The pixels get brighter as the code similarity increases. Therefore, yellow pixels in the matrix represent attacks with similar code. We used labels to divide the horizontal and vertical axis of the matrix into four segments. Each segment represents one of the four malware families we used in this research, i.e., from top-bottom/left-right Citadel, Gozi, Ramnit and ZeuS.

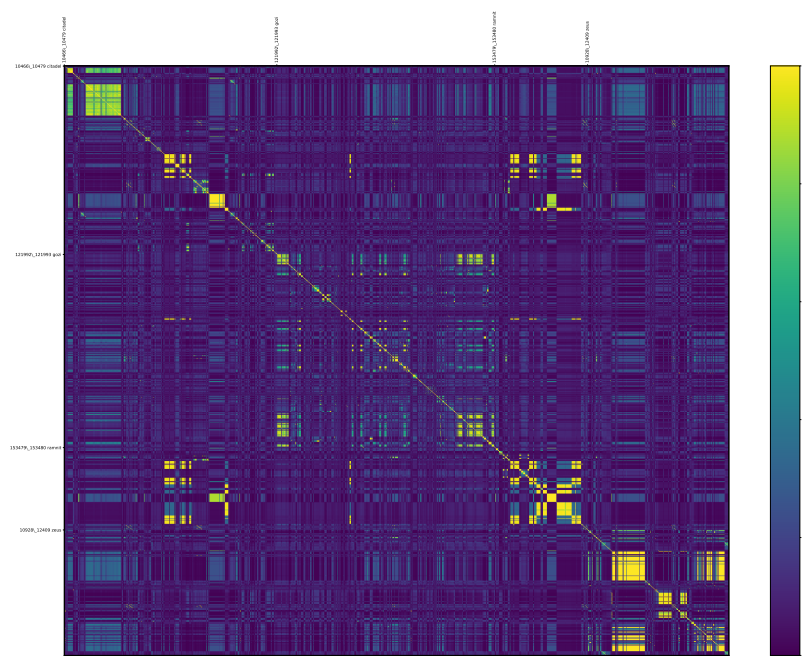


Figure 6.1: Self-similarity matrix computed from all attacks

A visual inspection of the self-similarity matrix as seen in Figure 6.1 already shows different levels of code similarity among inject codes. For instance, within the Citadel malware family we can observe a cluster of code reuse as well as within the attack codes of Ramnit. To gain further insights into code reuse, we further investigate the similarity of three different cases.

6.2. Case studies on Code similarity

The overall self-similarity matrix as shown in Figure 6.1 evidenced the existence of code similarity across different attacks. We selected 3 different study cases based on the clusters of high similarity.

6.2.1. Code similarity between Citadel and Ramnit attacks

From Figure 6.1 we observed clusters of code similarity between attacks of Citadel and Ramnit. This high similarity appeared just at the same time of the creation of the Ramnit family. This seems to indicate that the criminals behind each malware family reuse the code from the other family. Thus, we formulate the following two hypotheses.

H0: Citadel malware borrowed code from Ramnit.

H1: Ramnit malware borrowed code from Citadel.

First, we identified and plotted in full scale the clusters containing the attacks with high similarity (see Figure 6.2). Afterwards we used the attack labels to create the subset of the attacks which were part of the clusters. Then, we used the first attack of each malware family in the cluster to find the first time it attacked a domain during the entire time the malware families were active. We found that the first attack of the Citadel malware was used on 18-10-2016, while the one of the Ramnit malware was used on 21-11-2016. Since the Citadel's attacks appeared first, we **rejected H0**.

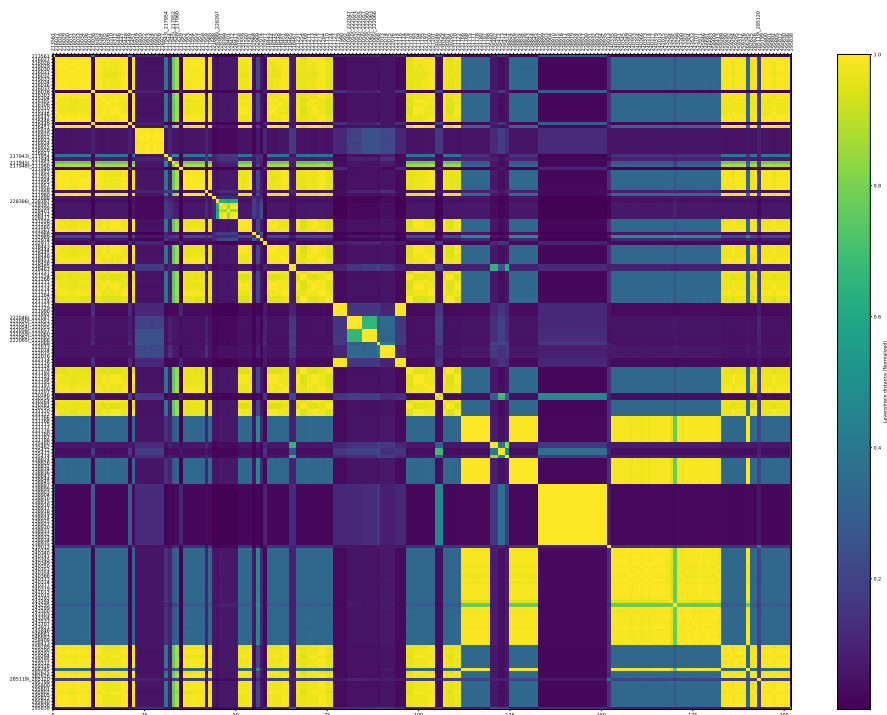


Figure 6.2: Self-similarity matrix computed from comparing Citadel and Ramnit Attacks

However, we still need to verify H1. We tested the hypothesis by comparing the code used in Ramnit attacks

against the code used in Citadel attacks. Using the results of the Levenshtein distance, we computed the similarity of all attacks in the clusters. The resulting similarity is more than 90% for all the attacks. We then manually examined the injected code in each attack to identify differences and the reason why it was possible to be reused by Ramnit attacks.

Our findings showed that the code of all Ramnit attacks is almost identical to the code used in the Citadel attacks. Their main differences are related to the URLs which are injected for the URL redirection mechanism. We also found cases where additional code was included (e.g., injection of unknown function calls). Nevertheless and despite those minor modifications, the semantics and the main functionality of the code were the same across all attacks. Finally, we noticed that the codes were not developed to modify the targeted web pages but mostly to redirect the users to either another part of the targeted domain or to unknown web pages.

In sum, the examination of the code showed that Ramnit attacks were similar to Citadel attacks. The code used in the attacks was not developed to be page specific, which makes it easier to reuse. Thus, we can conclude that attackers of Ramnit have borrowed code used in Citadel attacks. As a result, **H1 is accepted**.

6.2.2. Self-similarity matrix inside the self-similarity matrix

During the examination of the overall self-similarity matrix, we noticed cases where self-similarity matrixes were formed inside the global self-similarity matrix in the attacks of the Citadel and the ZeuS-P2P malware families. Using the self similarity matrix, we identified those cases. We found 5 different cases for the Citadel malware family and only one for the ZeuS-P2P. Figure 6.3 illustrates an example of such case (see Appendix D for other examples). Since we used each attack only once, those cases indicate the reuse of the same code in different attack files. In order to examine if this is actually true we introduce the following hypothesis:

H3: Attackers reused the same exact code in different attacks of Citadel and ZeuS-P2P malware families.

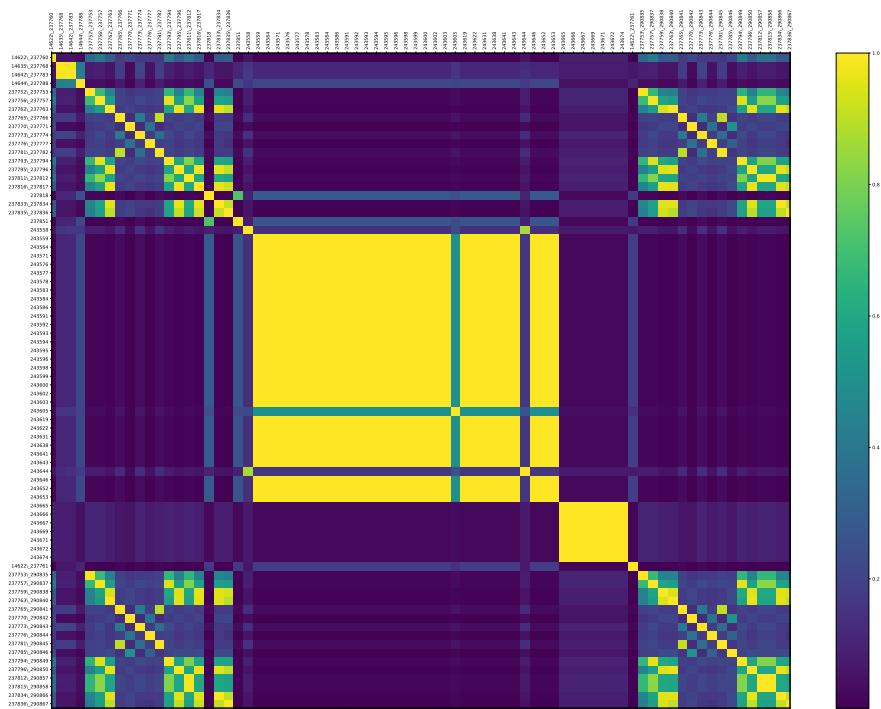


Figure 6.3: Self-similarity matrix inside the self similarity matrix from Citadel attacks

We tested our hypothesis by comparing the injected code of those attacks and their targeted domains. First, we discovered that all attacks which had high code similarity in the self-similarity matrix inside the self-similarity matrix had the same targeted domains and URLs. The “newest” attacks were used to replace the old ones which in some cases lasted only one day. Also for the attacks in which it was possible to link with their botnets, we identified that they belonged to the same botnet. We examined the injected code of the attacks with high similarity and identified two cases: (1) cases where the exact same code was reused in different attacks and (2) cases where the code was almost identical. The differences in the injected codes of the second case were: (i) the use of an IP address instead of a domain name when a URL was assigned as a value of a variable and (ii) minor differences in the `eval()` function which is used for data encryption.

Next, we examined the injected code of the preceding and succeeding attacks to check what has changed in the code. We identified two different cases: (1) attacks targeting the same domains but different URLs and (2) different targeted domains and URLs. When the preceding or succeeding attack is targeting the same domain, the main differences in the code are related to the injection of HTML and CSS elements while the JavaScript code which is injected has similar semantics. In the case where the injected code includes only JavaScript, the similarity is higher within the same domain. On the other hand, when the attacks targeted different domains, we noticed higher similarity in preceding and succeeding attacks when attacks are linked with the same botnet or the injection code it includes only JavaScript. We discovered that the similarity drops in preceding and succeeding attacks of different domains when the injected code contains HTML and CSS elements which have to be page specific.

To conclude, the examination of the code of the attacks which formed a self-similarity matrix inside the self-similarity matrix proved that attackers have developed reusable code which they used in different attacks in the same targeted domain. Thus **H3 is accepted**.

6.2.3. Code similarity in domains belonging to the same banking group

We also computed the self-similarity matrixes for the top-attacked domains based on the amount of unique attacks per malware family. We observed high code similarity in different domains within the same malware family. We extracted those domains and observed that all of them are part of the Lloyds¹ banking group which we hypothesized it allowed attackers to reuse similar code. Thus, we formulated the following hypothesis:

H4: Attackers have used similar code within the same malware family when they targeted domains of the Lloyds banking group.

To test our hypothesis, we first created a list which contains 8 domains which are part of the Lloyds banking group (lloydstsb.co.uk, halifax-online.co.uk, lloydstsb.com, lloydsbank.co.uk, bankofscotlandhalifax-online.co.uk, bankofscotland.co.uk, bankofscotlandbusiness.co.uk, lloydsbank.com). We used the list to filter out all the attacks per malware family which did not target domains of our list. Next, we plotted all the remaining unique attacks per domain ordered by the first time an attack was used at the domain. We used the attack id as the label in the horizontal and vertical axes of the matrix. Figure 6.4 illustrated the self similarity matrix for the Lloyds domains for the Gozi-EQ attacks (see Appendix D for the cases of ZeuS and Ramnit).

The first thing we discovered just by examining the self similarity matrix was the use of the same exact attack against different domains within the same malware family. Using the self-similarity matrix as a reference point, we also extracted the different attacks with high similarity used against different domains within the same malware family. Our comparisons have found that those attacks used have more than 90% similar code. Next, we manually examined the code of the attacks to discover why it was possible for the attacker to reuse it in attacks against different domains.

First we examined the code which was possible to be reused in attacks against different domains. We discovered that the code was either injecting URL for redirection or it had only JavaScript commands alerting messages to extract additional information from the users. When there were differences in the injected code between attacks of different targeted domains, then most changes in the code are related to : (1) the use of the name of the targeted domain as variable when they alerted messages for extracting additional information from the users (2) the URL used for the URL redirection mechanism. However and despite those small changes, the semantics of the code still remain the same. In any case, we can conclude that the code was not developed to be

¹<https://www.lloydsbankinggroup.com/>

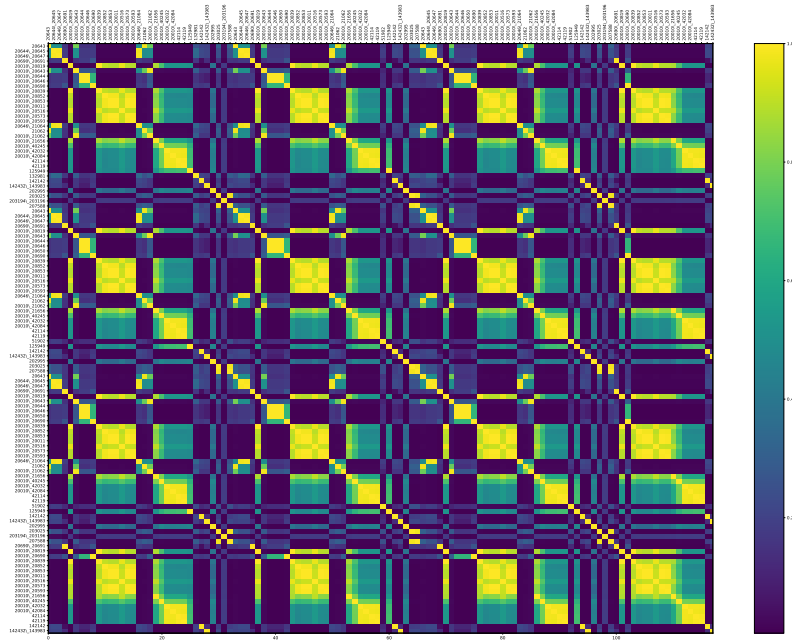


Figure 6.4: Self-similarity matrix of the domains part of Lloyd banking group from Gozi-EQ attacks

page specific which allowed the attackers to reused it in attacks against different targeted domains within the same financial group. Thus **H4 is accepted**.

Conclusions and Discussion

In this section, we use the results of the analyses to answer our overarching research question. We also provide recommendations, limitations, and suggestions for future research.

7.1. Conclusions

In this thesis, we investigated the evolution of web injected code attacks which are targeting financial institutions. Previous studies mostly focused on studying the evolution of the malware components code through different variant and not the evolution of the actual injected code. Those studies mainly analyzed and measured the code similarity in the binary or assembly form of the code due to the difficulty to obtain the actual source code. For this thesis, we obtained access to the source code used in web injected code attacks. This led to the formulation of the following research question to investigate the evolution of injected code:

RQ. How do cybercriminals evolve the code injected by financial malware in MiTB attacks?

Overall we found evidence that cybercriminals develop their inject code to make it reusable across targets. To reach this conclusion we conducted our analyses using a subset of the most prevalent malware families, i.e., (1) ZeuS-P2P, (2) Citadel, (3) Gozi-EQ and (4) Ramnit malware. We provide an answer to the research question through the following research sub-questions:

RQ1. What indicators of code evolution can be observed in the dataset?

We performed a descriptive analysis to discover indications of code reuse or evolution during the infection time period in which our malware families were active. Here we summarize the findings of our analysis.

- **Frequent attack repetition.** We found that attackers tend to reuse the same attack; the majority of attacks in this research were used more than once during the time period in which the malware was active.
- **Short lifetime of attack instances.** The lifespan for the majority of the attacks for all malware families does not last as long as the malware's lifespan. This indicates that attackers updated the content of the configuration file (code) more than once during the malware lifespan, which in turn evidences code evolution.
- **Attack reuse by different malware families.** We found that attacks related to Ramnit and Citadel have been also used by other malware families. This finding demonstrates that cybercriminals not only use the same code in attacks of the same malware family but also in attacks of different malware families.
- **Attacked reuse across different targets.** We found that identical injected code was used to target different financial institutions.

RQ2a. What is the most suitable tool for detecting code similarity in different inject code attacks?

Through a comprehensive scientific literature review, we identified multiple code similarity tools that can be used for code comparison. Some of these were language-independent tools (or not suitable for comparing JavaScript code) and presented poor precision and recall results. As the injected code was mainly developed in JavaScript, MOSS and Sherlock plagiarism tools were the only ones that had native support to run JavaScript-specific comparisons while being publicly available.

RQ2b. What are the limitations of the available similarity tools and how do they impact the similarity metrics?

The main limitation of the selected similarity tools is their inability to compare code at scale. Considering the large number of code files per malware family at hand, the use of these tools would make the calculation, extraction, and interpretation of the similarity results unfeasible within the time frame of this thesis. To overcome this limitation we decided to use the Levenshtein distance which is a simple metric based on string matching that can serve as measure of code similarity.

RQ3. How much does code re-using, stealing or sharing exist across different instances of injected code attacks?

The computation of the pairwise similarity across all the attacks showed that there exists a great portion of code reuse. By analyzing three specific case studies, we concluded that:

- **There exists code similarity between Citadel and Ramnit attacks.** Attackers of the Ramnit malware have borrowed code used in Citadel attacks.
- **Websites were targeted in multiple occasions with the similar attacks** Attackers develop reusable code which they use in different attacks against the same targeted domains.
- **Different domains within a banking group were targeted with the same code** Attackers develop reusable code which they use in different attacks against different targeted domains within the same banking group.

Recalling the main research question introduced at the beginning of this section, we conclude that:

Attackers tend to develop non-target specific code which allows them to reuse it in attacks against different domains and in some cases even across different malware families.

7.2. Discussion

This thesis contributes to the scientific literature in several ways:

- This is the first study to compare inject code similarity over time for different malware families. As mentioned before, only a handful of studies exist which examine the evolution of the actual code used in MitB attacks.
- We assessed the limitations of different code comparison tools and evaluated the advantages and disadvantages. The results showed that distance metric is able to capture the same trends in code similarity as specific tools.
- We demonstrated the existence of code reuse in a great portion of inject attacks and exemplify several cases of code sharing.

While these contributions are bounded to a dataset whose completeness cannot be assessed, we believe that these results can serve better understanding of the cybercriminal's incentives and modus operandi. However, it must be noted that the inject codes just refer to targeted entities and does not necessarily imply that those attacks were successful.

7.3. Recommendations to financial institutions

Financial institutions can play a major role disrupting the MitB attacks analyzed in this thesis. Our results have shown that the majority of the attack code is generic so that cybercriminals can reuse it across multiple targets and during large periods of time. This indicates that attackers do not have enough incentives to develop target specific attacks at scale. In this sense, small modifications of the targeted website will have major impact on the deception capabilities of the MitB attacks. Hence, financial institutions could reduce the likelihood of successful MitB attacks by simply performing periodic small changes on their websites.

More specific countermeasures could also be taken depending on the type of injected code. Our analysis allowed us to identify different types of injected code against which defenders could deploy specific countermeasures. In the list below, we briefly provide some mitigation strategies against the three types of injected code we observed:

- *URL Redirection.* Financial institutions could benefit from deploying countermeasures that prevent URL redirection. Different countermeasures have already been proposed in the literature which are also applicable to this case [2].
- *Code which modifies the web page.* Countermeasures against these attacks could aim at signalling the attack to the website users by counteracting the deception techniques of the attack. For instance, frequent changes in the appearance of the web pages could be very useful.
- *Code with obfuscation techniques.* Obfuscation code is not easy to detect as shown in Xu et al. [121], thus the implementation of tools which are able to detect such code can be a good countermeasure against injected obfuscation code (e.g., Xu et al. [122]).

7.4. Limitations

In this section we explain the limitations of the current research in terms of threats against its validity.

External validity. The main limitation of the research is related to what extent can the findings be generalized. The analyses were conducted using the attack code of only four case studies, although the Fox-IT dataset contained information of higher number of web injected malware. The external validity may also have been affected by the decision to use a subset of our sample in the hypothesis testing in the code similarity analysis.

Construct Validity refers to the degree to which what was supposed to be measured is what actually was measured. A bias in the way the hypotheses were verified may exist from manually analyzing the code of the attacks due to subjective examination of the injected code.

Internal validity is the extend to which causal conclusions exist within the data used in the study. The measurement of code similarity among different attack instances was done using different external tools. However, it was still not possible to verify 100% accuracy for neither of the tools since they were not designed to perform code similarity comparisons. Finally, the task of interpreting the source code to test our hypotheses as indicated earlier is subjective.

Conclusion validity. The small number of malware that was used as case studies combined with the small number of attack codes per case study is a limitation of the research since it may have affected the credibility of the results).

7.5. Future Research

The current research can be the starting point of future research. A list of possible improvements and suggestions for future research are provided below:

- **Analyzing more malware families.** The current research was limited by examining the code similarity of only a subset of malware families. Analyzing more malware families can improve the validity and reliability of the current findings. During the analysis it was found that code reuse exists among different families. This can be further investigated in future research.
- **Development of injected code taxonomy.** Using the injected codes of our sample and key characteristics of their code functionality, a taxonomy can be developed which can tie certain types of injected codes together.

- **Statistical Analysis.** Use of different independent variables (e.g., duration of attack, lines of code) to discover if they can have an impact on when a change to the injected code occurred. The use of other external variables like banks regulations or changes in bank's webpages as independent variables can also show if their impact on code evolution or reuse.

Bibliography

- [1] Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, Baltic Sea '06, pages 141–142, New York, NY, USA, 2006. ACM. doi: 10.1145/1315803.1315831. URL <http://doi.acm.org/10.1145/1315803.1315831>.
- [2] Mitsuaki Akiyama, Takeshi Yagi, Takeshi Yada, Tatsuya Mori, and Youki Kadobayashi. Analyzing the ecosystem of malicious url redirection through longitudinal observation from honeypots. *Computers & Security*, 69:155 – 173, 2017. ISSN 0167-4048. doi: <https://doi.org/10.1016/j.cose.2017.01.003>. URL <http://www.sciencedirect.com/science/article/pii/S016740481730007X>. Security Data Science and Cyber Threat Management.
- [3] Ammar Alazab, Jemal Abawajy, Michael Hobbs, Robert Layton, and Ansam Khraisat. Crime toolkits: the productisation of cybercrime. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 1626–1632. IEEE, 2013.
- [4] Mamoun Alazab, Sitalakshmi Venkatraman, Paul Watters, Moutaz Alazab, and Ammar Alazab. Cybercrime: The case of obfuscated malware. In Christos K. Georgiadis, Hamid Jahankhani, Elias Pimenidis, Rabih Bashroush, and Ameer Al-Nemrat, editors, *Global Security, Safety and Sustainability & e-Democracy*, pages 204–211, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33448-1.
- [5] Ross Anderson, Chris Barton, Rainer Böhme, Richard Clayton, Michel JG Van Eeten, Michael Levi, Tyler Moore, and Stefan Savage. Measuring the cost of cybercrime. In *The economics of information security and privacy*, pages 265–300. Springer, 2013.
- [6] D. Andriesse, C. Rossow, B. Stone-Gross, D. Plohmann, and H. Bos. Highly resilient peer-to-peer botnets are here: An analysis of gameover zeus. In *2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, pages 116–123, Oct 2013. doi: 10.1109/MALWARE.2013.6703693.
- [7] AV Test. The AV-TEST Security Report, 2016. URL https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2015-2016.pdf.
- [8] Brenda S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 1992.
- [9] Hamid Abdul Basit, Simon J. Puglisi, William F. Smyth, Andrew Turpin, and Stan Jarzabek. Efficient token based clone detection with flexible tokenization. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers, ESEC-FSE companion '07*, pages 513–516, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-812-1. doi: 10.1145/1295014.1295029. URL <http://doi.acm.org/10.1145/1295014.1295029>.
- [10] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant’Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.
- [11] Stefan Bellon. Vergleich von techniken zur erkennung duplizierten quellcodes. Master’s thesis, Institut für Softwaretechnologie, Universität Stuttgart, Stuttgart, Germany, 2002.
- [12] Hamad Binsalleeh, Thomas Ormerod, Amine Boukhtouta, Prosenjit Sinha, Amr Youssef, Mourad Debbabi, and Lingyu Wang. On the analysis of the zeus botnet crimeware toolkit. In *Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on*, pages 31–38. IEEE, 2010.
- [13] Jean-Ian Boutin. The evolution of webinjects. In *Virus Bulletin Conference*, 2014.

- [14] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 36–43, 2002. doi: 10.1109/SCAM.2002.1134103.
- [15] Philippe Charland, Benjamin CM Fung, and Mohammad Reza Farhadi. Clone search for malicious code correlation. In *NATO RTO Symposium on Information Assurance and Cyber Defense (IST-111)*, Koblenz, 2012.
- [16] Check Point. October’s ‘Most Wanted’ Malware List Shows Attacks on the Rise, 2016. URL <https://blog.checkpoint.com/2016/11/21/octobers-wanted-malware-list-reveals-attacks-rise/>.
- [17] Check Point. June’s Most Wanted Malware: Banking Trojans Up 50% Among Threat Actors, 2018. URL <https://blog.checkpoint.com/2018/07/05/junes-most-wanted-malware-banking-trojans-crypto-mining/>.
- [18] Check Point and Europol. Banking Trojans: From Stone Age to Space Era, 2017. URL <https://www.europol.europa.eu/publications-documents/banking-trojans-stone-age-to-space>.
- [19] Jian Chen, Manar H. Alalfi, Thomas R. Dean, and Ying Zou. Detecting android malware using clone detection. *Journal of Computer Science and Technology*, 30(5):942–956, Sep 2015. ISSN 1860-4749. doi: 10.1007/s11390-015-1573-7. URL <https://doi.org/10.1007/s11390-015-1573-7>.
- [20] Xin Chen, B. Francia, Ming Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, 50(7):1545–1551, July 2004. ISSN 0018-9448. doi: 10.1109/TIT.2004.830793.
- [21] Michael Ciere, Carlos Gañán, and Michel van Eeten. Partial device fingerprints. In Michelangelo Ceci, Jaakko Hollmén, Ljupčo Todorovski, Celine Vens, and Sašo Džeroski, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 222–237, Cham, 2017. Springer International Publishing. ISBN 978-3-319-71246-8.
- [22] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying Dormant Functionality in Malware Programs. In *2010 IEEE Symposium on Security and Privacy*, pages 61–76, May 2010. doi: 10.1109/SP.2010.12.
- [23] Claudio Criscione, Fabio Bosatelli, Stefano Zanero, and Federico Maggi. Zarathustra: Extracting webinject signatures from banking trojans. In *Privacy, Security and Trust (PST), 2014 Twelfth Annual International Conference on*, pages 139–148. IEEE, 2014.
- [24] Fred J Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.
- [25] Mark de Bruijne, Michel van Eeten, Carlos Gañán Gañán, and Wolter Pieters. Towards a new cyber threat actor typology. Report, Delft University of Technology, Delft, 2017.
- [26] Department of Justice. U.S. Leads Multi-National Action Against “GameOver Zeus” Botnet and “Cryptolocker” Ransomware, Charges Botnet Administrator, 2014. URL <https://www.justice.gov/opa/pr/us-leads-multi-national-action-against-gameover-zeus-botnet-and-cryptolocker-ransomware>.
- [27] Zoran Djurić and Dragan Gašević. A source code similarity system for plagiarism detection. *The Computer Journal*, 56(1):70–86, 2012.
- [28] John L Donaldson, Ann-Marie Lancaster, and Paula H Sposato. A plagiarism detection system. In *ACM SIGCSE Bulletin*, volume 13, pages 21–25. ACM, 1981.
- [29] Stéphane Ducasse, Matthias Rieger, and Serge 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.

- [30] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. Lightweight detection of duplicated code: a language-independent approach. *Institute for Applied Mathematics and Computer Science. University of Berne*, 2004.
- [31] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching: Research articles. *J. Softw. Maint. Evol.*, 18(1):37–58, January 2006. ISSN 1532-060X. doi: 10.1002/smr.v18:1. URL <http://dx.doi.org/10.1002/smr.v18:1>.
- [32] N. Etaher, G. R. S. Weir, and M. Alazab. From ZeuS to Zitmo: Trends in Banking Malware. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 1386–1391, Aug 2015. doi: 10.1109/Trustcom.2015.535.
- [33] European Central Bank. Fourth report on card fraud, 2015. URL https://www.ecb.europa.eu/pub/pdf/other/4th_card_fraud_report.en.pdf.
- [34] Europol. Global action targeting shylock malware, 2014. URL <https://www.europol.europa.eu/newsroom/news/global-action-targeting-shylock-malware>.
- [35] Mohammad Reza Farhadi, Benjamin C. M. Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *Proceedings of the 2014 Eighth International Conference on Software Security and Reliability, SERE '14*, pages 78–87, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4296-1. doi: 10.1109/SERE.2014.21. URL <http://dx.doi.org/10.1109/SERE.2014.21>.
- [36] Mohammad Reza Farhadi, Benjamin C.M. Fung, Yin Bun Fung, Philippe Charland, Stere Preda, and Mourad Debbabi. Scalable code clone search for malware analysis. *Digit. Investig.*, 15(C):46–60, December 2015. ISSN 1742-2876. doi: 10.1016/j.diin.2015.06.001. URL <http://dx.doi.org/10.1016/j.diin.2015.06.001>.
- [37] FBI. GameOver Zeus Botnet Disrupted Collaborative Effort Among International Partners, 2014. URL <https://www.fbi.gov/news/stories/gameover-zeus-botnet-disrupted/gameover-zeus-botnet-disrupted>.
- [38] Fox-IT. About Fox-IT, 2018. URL <https://www.fox-it.com/en/>.
- [39] Carlos H. Gañán, Michael Ciere, and Michel van Eeten. Beyond the Pretty Penny: The Economic Impact of Cybercrime. In *Proceedings of the 2017 New Security Paradigms Workshop, NSPW 2017*, pages 35–45, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-6384-6. doi: 10.1145/3171533.3171535. URL <http://doi.acm.org/10.1145/3171533.3171535>.
- [40] Carlos Gañán, Orcun Cetin, and Michel van Eeten. An empirical analysis of ZeuS C&C lifetime. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 97–108. ACM, 2015.
- [41] Manuel Garcia-Cervigon and Manel Medina Llinas. Browser function calls modeling for banking malware detection. In *Risk and Security of Internet and Systems (CRiSIS), 2012 7th International Conference on*, pages 1–7. IEEE, 2012.
- [42] Dick Grune. The software and text similarity tester sim, 2006. URL https://dickgrune.com/Programs/similarity_tester/.
- [43] Jurriaan Hage. Programmeerplagiaatdetectie met marble. Master’s thesis, UU WINFI Informatica en Informatiekunde, 2006.
- [44] Jurriaan Hage, Peter Rademaker, and Nike van Vugt. A comparison of plagiarism detection tools. *Utrecht University. Utrecht, The Netherlands*, 28, 2010.
- [45] Daniël Heres and Jurriaan Hage. A quantitative comparison of program plagiarism detection tools. In *Proceedings of the 6th Computer Science Education Research Conference, CSERC '17*, pages 73–82, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-6338-9. doi: 10.1145/3162087.3162101. URL <http://doi.acm.org/10.1145/3162087.3162101>.

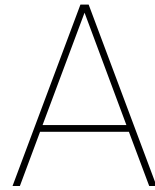
- [46] Paul Hyman. Cybercrime: it's serious, but exactly how serious? *Communications of the ACM*, 56(3):18–20, 2013.
- [47] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 48–62, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0. doi: 10.1109/SP.2012.13. URL <http://dx.doi.org/10.1109/SP.2012.13>.
- [48] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pages 171–183. IBM Press, 1993. URL <http://dl.acm.org/citation.cfm?id=962289.962305>.
- [49] J Howard Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 32. IBM Press, 1994.
- [50] Edward L. Jones. Metrics Based Plagiarism Monitoring. *J. Comput. Sci. Coll.*, 16(4):253–261, April 2001. ISSN 1937-4771. URL <http://dl.acm.org/citation.cfm?id=378613.378727>.
- [51] M. Joy and M. Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133, May 1999. ISSN 0018-9359. doi: 10.1109/13.762946.
- [52] Mike Joy and Michael Luck. Plagiarism in programming assignments. *IEEE Transactions on education*, 42(2):129–133, 1999.
- [53] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.1019480. URL <http://dx.doi.org/10.1109/TSE.2002.1019480>.
- [54] Md. Enamul. Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1):13–23, Nov 2005. ISSN 1772-9904. doi: 10.1007/s11416-005-0002-9. URL <https://doi.org/10.1007/s11416-005-0002-9>.
- [55] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987. ISSN 0018-8646. doi: 10.1147/rd.312.0249.
- [56] Richard M Karp. Combinatorics, complexity, and randomness. *Commun. ACM*, 29(2):97–109, 1986.
- [57] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, May 2017. doi: 10.1109/SP.2017.62.
- [58] Jeremy Kirk. Huge Malware Distribution Network Crippled, 2018. URL <https://www.bankinfosecurity.com/huge-malware-distribution-network-crippled-a-10805>.
- [59] Amit Klein. Security Intelligence - Ice IX Malware Redirects Bank Phone Calls to Attackers, 2012. URL <https://securityintelligence.com/ice-ix-malware-redirects-bank-phone-calls-to-attackers/>.
- [60] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.
- [61] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42314-1. URL <http://dl.acm.org/citation.cfm?id=647170.718283>.
- [62] 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.
- [63] Rainer Koschke. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

- [64] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 253–262. IEEE, 2006.
- [65] Jens Krinke. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301–309. IEEE, 2001.
- [66] C. Kustanto and I. Liem. Automatic Source Code Plagiarism Detection. In *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, pages 481–486, May 2009. doi: 10.1109/SNPD.2009.62.
- [67] Filippo Lanubile and Teresa Mallardo. Finding function clones in web applications. In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pages 379–386. IEEE, 2003.
- [68] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.
- [69] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of Malicious Code: Insights into the Malicious Software Industry. In *Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC '12*, pages 349–358, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1312-4. doi: 10.1145/2420950.2421001. URL <http://doi.acm.org/10.1145/2420950.2421001>.
- [70] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '06*, pages 872–881, New York, NY, USA, 2006. ACM. ISBN 1-59593-339-5. doi: 10.1145/1150402.1150522. URL <http://doi.acm.org/10.1145/1150402.1150522>.
- [71] Giuseppe A. Di Lucca, Massimiliano Di Penta, and Anna Rita Fasolino. An approach to identify duplicated web pages. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment, COMPSAC '02*, pages 481–486, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1727-7. URL <http://dl.acm.org/citation.cfm?id=645984.675896>.
- [72] Robert W Ludwig Jr, Salvatore Scanio, and Joseph S Szary. Malware and Fraudulent Electronic Funds Transfers: Who Bears the Loss? *Fidelity Law Journal*, 16, 2010.
- [73] Udi Manber. Finding similar files in a large file system. In *USENIX WINTER 1994 TECHNICAL CONFERENCE*, pages 1–10, 1994.
- [74] J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *1996 Proceedings of International Conference on Software Maintenance*, pages 244–253, Nov 1996. doi: 10.1109/ICSM.1996.565012.
- [75] Mike McGuire and Samantha Dowling. Cyber crime: A review of the evidence. *Summary of key findings and implications. Home Office Research report*, 75, 2013.
- [76] Jeffrey Meisner. Microsoft and Financial Services Industry Leaders Target Cybercriminal Operations from Zeus Botnets, 2012. URL <https://blogs.microsoft.com/blog/2012/03/25/microsoft-and-financial-services-industry-leaders-target-cybercriminal-operations-from-zeus-botnets/>.
- [77] Jeffrey Meisner. Microsoft works with financial services industry leaders, law enforcement and others to disrupt massive financial cybercrime ring, 2013. URL https://blogs.technet.microsoft.com/microsoft_blog/2013/06/05/microsoft-works-with-financial-services-industry-leaders-law-enforcement-and-others-to-disrupt-massive-financial-cybercrime-ring/.
- [78] Jason Milletary. Citadel trojan malware analysis. *Luettavissa: http://botnetlegalnotice.com/citadel/files/Patel_Decl_Ex20.pdf. Luettu*, 13:2014, 2012.

- [79] Michael Mimoso. KINS Banking Trojan a Successor to Citadel?, 2013. URL <https://threatpost.com/kins-banking-trojan-a-successor-to-citadel/101440/>.
- [80] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A SLOC counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16. Citeseer, 2007.
- [81] Sadia Noreen, Shafaq Murtaza, M. Zubair Shafiq, and Muddassar Farooq. Evolvable malware. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO '09*, pages 1569–1576, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-325-9. doi: 10.1145/1569901.1570111. URL <http://doi.acm.org/10.1145/1569901.1570111>.
- [82] J-F Patenaude, Ettore Merlo, Michel Dagenais, and Bruno Laguë. Extending software quality assessment techniques to java systems. In *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*, pages 49–56. IEEE, 1999.
- [83] Clifton Phua, Vincent Lee, Kate Smith, and Ross Gayler. A comprehensive survey of data mining-based fraud detection research. *arXiv preprint arXiv:1009.6119*, 2010.
- [84] Tero Pikkarainen, Kari Pikkarainen, Heikki Karjaluo, and Seppo Pahlila. Consumer acceptance of online banking: an extension of the technology acceptance model. *Internet research*, 14(3):224–235, 2004.
- [85] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. *J. UCS*, 8(11):1016, 2002.
- [86] Ashkan Rahimian, Raha Ziarati, Stere Preda, and Mourad Debbabi. On the reverse engineering of the citadel botnet. In *Foundations and Practice of Security*, pages 408–425. Springer, 2014.
- [87] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus—a tool suite for program analysis and reverse engineering. In *International Conference on Reliable Software Technologies*, pages 71–82. Springer, 2006.
- [88] Marco Riccardi, David Oro, Jesus Luna, Marco Cremonini, and Marc Vilanova. A framework for financial botnet analysis. In *ECrime Researchers Summit (ECrime), 2010*, pages 1–7. IEEE, 2010.
- [89] Marco Riccardi, Roberto Di Pietro, Marta Palanques, and Jorge Aguila Vila. Titans’ revenge: Detecting Zeus via its own flaws. *Computer Networks*, 57(2):422–435, 2013.
- [90] Markus Riek, Rainer Boehme, M Ciere, Carlos Gañán, and MJG van Eeten. Estimating the costs of consumer-facing cybercrime: A tailored instrument and representative data for six eu countries. In *Proceedings of the 15th Annual Workshop on the Economics of Information Security, WEIS 2016*, pages 12–45, 2016.
- [91] Denis Makrushin Fedor Sinitsyn Alexander Liskin Roman Unuchek, Maria Garnaeva. IT threat evolution Q3 2016. Statistics, 2016. URL https://securelist.com/files/2016/11/KL_Q3_Malware_Report_ENG.pdf.
- [92] Chanchal Roy and James R. Cordy. A survey on software clone detection research. In *School of Computing TR 2007-541*, 01 2007.
- [93] RSA. Making sense of man-in-the-browser attacks: Threat analysis and mitigation for financial institutions, 2011. URL https://www.rsa.com/content/dam/rsa/PDF/Making_Sense_of_Man_in_the_browser_attacks.pdf.
- [94] Bahman Saeidipour, Hojat Ranjbar, and Saeed Ranjbar. Adoption of Internet banking. *IOSR Journal of Business and Management (IOSR-JBM)*, 11(2):46–51, 2013.
- [95] Safenet Security Guide, 2010. URL <http://www2.gemalto.com/email/2010/MITB-2010/Man-in-the-Browser-Security-Guide.pdf>.
- [96] Or Safran. Security Intelligence - Gozi Banking Trojan Upgrades Build to Inject Into Windows 10 Edge Browser, 2018. URL <https://securityintelligence.com/gozi-banking-trojan-upgrades-build-to-inject-into-windows-10-edge-browser/>.

- [97] Hitesh Sajjani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1157–1168, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884877. URL <http://doi.acm.org/10.1145/2884781.2884877>.
- [98] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.
- [99] Ryan Sherstobitoff. Inside the world of the citadel trojan. *Emergence*, 9, 2012.
- [100] Aditya K Sood, Richard J Enbody, and Rohit Bansal. Dissecting SpyEye—Understanding the design of third generation botnets. *Computer Networks*, 57(2):436–450, 2013.
- [101] Aditya K Sood, Sherali Zeadally, and Richard J Enbody. An Empirical Study of HTTP-based Financial Botnets. *IEEE Transactions on Dependable and Secure Computing*, 13(2):236–251, 2016.
- [102] Symantec. Citadel’s Defenses Breached, 2013. URL <https://www.symantec.com/connect/blogs/citadel-s-defenses-breached>.
- [103] Symantec. International Takedown Wounds Gameover Zeus Cybercrime Network, 2014. URL <https://www.symantec.com/connect/blogs/international-takedown-wounds-gameover-zeus-cybercrime-network>.
- [104] Symantec. Ramnit cybercrime group hit by major law enforcement operation, 2015. URL <https://www.symantec.com/connect/blogs/ramnit-cybercrime-group-hit-major-law-enforcement-operation>.
- [105] Symantec. W32.Ramnit analysis, 2015. URL <https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/w32-ramnit-analysis-15-en.pdf>.
- [106] Symantec Security Response. All That Glitters Is No Longer Gold - Shylock Trojan Gang Hit by Takedown, 2014. URL <https://www.symantec.com/connect/blogs/all-glitters-no-longer-gold-shylock-trojan-gang-hit-takedown>.
- [107] Symantec Security Response. Dyre emerges as main financial Trojan threat, 2015. URL <https://www.symantec.com/connect/blogs/dyre-emerges-main-financial-trojan-threat>.
- [108] Samaneh Tajalizadehkhoob, Carlos Gañán, Arman Noroozian, and Michel van Eeten. The Role of Hosting Providers in Fighting Command and Control Infrastructure of Financial Malware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 575–586. ACM, 2017.
- [109] ST Tajalizadehkhoob, Hadi Asghari, Carlos Gañán, and MJG Van Eeten. Why them? Extracting intelligence about target selection from Zeus financial malware. In *Proceedings of the 13th Annual Workshop on the Economics of Information Security, WEIS 2014, State College (USA), June 23-24, 2014*. WEIS, 2014.
- [110] Trend Micro. Ramnit: The comeback story of 2016, 2017. URL <https://blog.trendmicro.com/trendlabs-security-intelligence/ramnit-comeback-story-2016/>.
- [111] Liam Tung. Researchers hijack huge network of hacked sites that spread ransomware, banking trojans, 2017. URL <https://www.cso.com.au/article/639756/researchers-hijack-huge-network-hacked-sites-spread-ransomware-banking-trojans/>.
- [112] Nattakant Utakrit. Review of browser extensions, a man-in-the-browser phishing techniques targeting bank customers. Master’s thesis, Security Research Centre, School of Computer and Security Science, Edith Cowan University, Perth, Western Australia, 2009.
- [113] Véronique van Vlasselaer, Cristián Bravo, Olivier Caelen, Tina Eliassi-Rad, Leman Akoglu, Monique Snoeck, and Bart Baesens. APATE: A novel approach for automated credit card transaction fraud detection using network-based extensions. *Decision Support Systems*, 75:38–48, 2015.

- [114] Doron Voolf. Ramnit goes on a holiday shopping spree, targeting retailers and banks, 2018. URL <https://www.f5.com/labs/articles/threat-intelligence/ramnit-goes-on-a-holiday-shopping-sprees-targeting-retailers-and-banks>.
- [115] Andrew Walenstein and Arun Lakhotia. The software similarity problem in malware analysis. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.
- [116] Andrew Walenstein, Michael Venable, Matthew Hayes, Christopher Thompson, and Arun Lakhotia. Exploiting Similarity Between Variants to Defeat Malware "Vilo" Method for Comparing and Searching Binary Programs. In *BlackHat DC 2007*, page 12, 01 2007.
- [117] Wei Wei, Jinjiu Li, Longbing Cao, Yuming Ou, and Jiahang Chen. Effective detection of sophisticated online banking fraud on extremely imbalanced data. *World Wide Web*, 16(4):449–475, 2013.
- [118] Richard Wettel and Radu 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.
- [119] Daniel R White and Mike S Joy. Sentence-based natural language plagiarism detection. *Journal on Educational Resources in Computing (JERIC)*, 4(4):2, 2004.
- [120] James Wyke. Duping the machine—malware strategies, post sandbox detection. In *Proceedings of the 24th Virus Bulletin International Conference*, pages 91–97, 2014.
- [121] Wei Xu, Fangfang Zhang, and Sencun Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software (MALWARE)*, MALWARE '12, pages 9–16, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-4880-5. doi: 10.1109/MALWARE.2012.6461002. URL <http://dx.doi.org/10.1109/MALWARE.2012.6461002>.
- [122] Wei Xu, Fangfang Zhang, and Sencun Zhu. Jstill: Mostly static detection of obfuscated malicious javascript code. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy, CODASPY '13*, pages 117–128, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1890-7. doi: 10.1145/2435349.2435364. URL <http://doi.acm.org/10.1145/2435349.2435364>.
- [123] Wu Yang. Identifying syntactic differences between two programs. *Software: Practice and Experience*, 21(7):739–755, 1991.
- [124] Robert K Yin. *Case study research and applications: Design and methods*. Sage publications, 2017.
- [125] Jakub Žitný. Analysing JavaScript and NPM at scale. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2017.



HTML and JavaScript keywords

The Table A.1 presents a list of keywords for the JavaScript and the HTML languages. The information was obtained from w3schools ¹ ². For the JavaScript, all they keywords are reserved words that cannot be used as variables, labels, or function names. The words for the HTML is a collection of the most used Tags and are ordered in different categories:

- Basic HTML (i.e <head>, <body>)
- Forms and Input (i.e <form>, <button>)
- Frame (<frame>)
- Links (<link>)
- Tables (<table>)
- Styles and Semantics (i.e <style>, <div>)
- Programming (<script>)

Additionally some keywords like background, width were also included to cover the cases that the content is purely CSS. The * next to a word are reserved words for ECMAScript 5 and 6.

¹JavaScript: https://www.w3schools.com/js/js_reserved.asp

² HTML: <https://www.w3schools.com/Tags/default.asp>

Table A.1: List of HTML and JavaScript keywords

abstract	argument	boolean
break	byte	case
catch	char	const
continue	debugger	default
delete	do	double
else	eval	false
final	finally	float
for	function	goto
if	implements	in
instanceof	int	interface
let*	long	native
new	null	package
private	protected	public
return	short	static
switch	synchronized	this
throw	throws	transient
true	width	type
typeof	var	volatile
while	with	yield
class*	enum*	export*
extends*	super	import*
await*	DOCTYPE	html
head	body	title
div	form	output
input	select	button
option	frame	href
link	table	span
style	header	footer
script	window	alert
position	height	background

B

Attacks per Botnet

The figures below illustrate in more detail the information regarding the attacks per botnet for each of the threat separately. For each threat the Table presents the name of the botnet (RC4 key), the total number of attacks are associated with each botnet and finally the number of distinct attacks (configuration files) that are linked back to each botnet. For some cases, it can be seen that only one configuration file is linked with the botnet. This means that attackers have not updated at all the configuration file and thus it is "responsible" for all the attack instances linked back to this specific botnet. On the other hand, as can be seen there are more attacks than configuration files which was used as an preliminary conclusion that attackers change the content of the configuration files. This preliminary conclusion was used as an indication that attackers possibly use the same code in different attacks.

<i>Botnet</i>	Total Attacks	Distinct Attacks
1A71DEE26B437C8B9F7CA854908483FA	44,578	485
30DC0C542EB5F9EA46793DA33BE0E02E	2,950	67
Grand Total	47,528	552

Figure B.1: Citadel: Attacks per Botnet

<i>Botnet</i>	Total Attacks	Distinct Attacks
4EA11102614E258220C2C58F4AAC62B8	5,016	338
F6B7F04BC27020D98074B6C3EF844020	2,930	664
Grand Total	7,946	1,002

Figure B.2: Ramnit: Attacks per Botnet

Botnet	Total Attacks	Distinct Attacks
072B030BA126B2F4B2374F342BE9ED44	2,714	165
088F603470C73729A2E1B53BC2A5F712	30	15
0F28B5D49B3020AFEED95B4009ADF4C	22	4
1AFA34A7F984EEABDBB0A7D494132EE5	276	69
1F0E3DAD99908345F7439F8FFABDFFC4	830	123
202CB962AC59075B964B07152D234B70	16	8
2838023A778DFAE CDC212708F721B788	2	1
37693CFC748049E45D87B8C7D8B9AACD	2	1
38B3EFF8BAF56627478EC76A704E9B52	364	44
3C59DC048E8850243BE8079A5C74D079	344	68
47E36F89BC2ABF42E75257C6C695A253	2	1
4C56FF4CE4AAF9573AA5DFF913DF997A	228	77
577BCC914F9E55D5E4E4F82F9F00E7D4	308	154
5F93F983524DEF3DCA464469D2CF9F3E	652	42
6364D3F0F495B6AB9DCF8D3B5C6E0B01	6	2
6512BD43D9CAA6E02C990B0A82652DCA	114	20
65DED5353C5EE48D0B7D48C591B8F430	14	7
68D30A9594728BC39AA24BE94B319D21	68	34
698D51A19D8A121CE581499D7B701668	312	60
6F4922F45568161A8CDF4AD2299F6D23	26	6
7CBBC409EC990F19C78C75BD1E06F215	14	7
7F6FFAA6BB0B408017B62254211691B5	96	42
8C19F571E251E61CB8DD3612F26D5ECF	4	2
9778D5D219C5080B9A6A17BEF029331C	226	38
98F13708210194C475687BE6106A3B84	210	50
9B8619251A19057CFF70779273E95AA6	96	48
A0A080F42E6F13B3A2DF133F073095DD	4	2
A1E807703F05998232EE5FC0E22A46EE	52	26
A684ECEEE76FC522773286A895BC8436	96	24
AAB3238922BCC25A6F606EB525FFDC56	8	2
BD4C9AB730F5513206B999EC0D90D1FB	22	11
C0C7C76D30BD3DCAEFC96F40275BDC0A	134	44
C20AD4D76FE97759AA27A0C99BFF6710	538	95
C51CE410C124A10E0DB5E4B97FC2AF39	16	2
C74D97B01EAE257E44AA9D5BADE97BAF	1,004	63
C8FFE9A587B126F152ED3D89A146B445	118	59
C9B289621485AE96E1924E1BE66984E1	4	2
D3D9446802A44259755D38E6D163E820	164	33
F033AB37C30201F73F142449D037028D	204	41
FE9FC289C3FF0AF142B6D3BEAD98A923	70	35
Grand Total	9,410	1,527

Figure B.3: Gozi-EQ: Attacks per Botnet

C

Similarity Tools Execution

C.1. MOSS

MOSS Execution

Once the codes of the attacks are extracted, the MOSS script can be executed and compare the code of the different files. The script is provided with a unique user id and the execution of the script is only done through a terminal. The input files are submitted to the system's server through a query and the results are available for 14 days and then are deleted. An example of the execution of the script for comparing the files of a directory is shown in Figure C.1, the command is highlighted in the red rectangle.

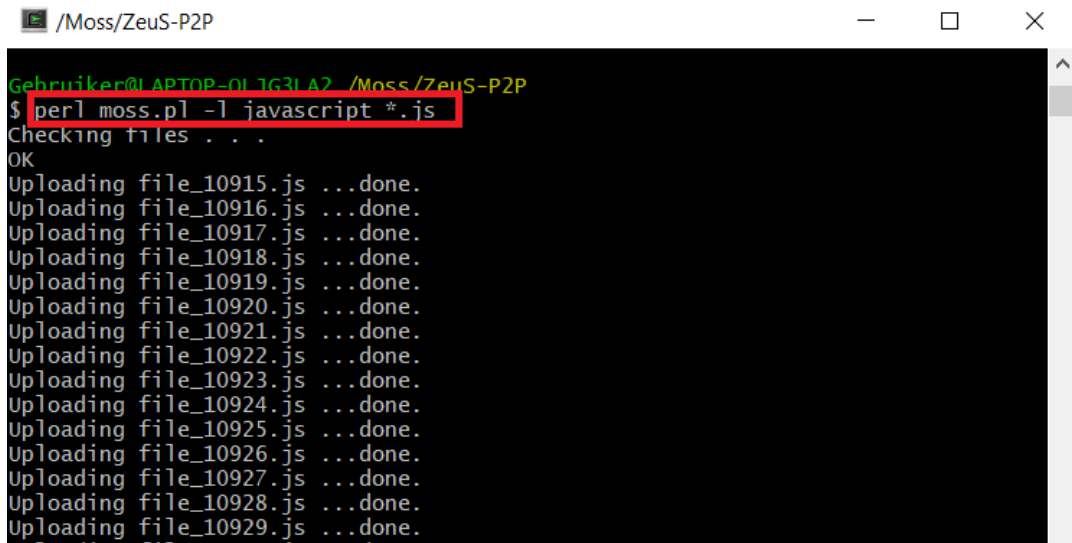
MOSS Results

If the submission of the files is done correctly, a URL is provided which gives access to the results of the comparison. The results are presented in the form of pair comparison and for each pair the percentage of similarity and the lines of code that are matched are provided. An example of the results presentation is shown in Figure C.1. Each pair of files is a hyperlink which link to a page with more information about the matching code. The page is divided in three parts, the first part is a table which show for each files a range of code lines that are considered to be identical, along with a bar that indicates how much of all the code in the file is matched by this range of lines. For example in Figure C.3 a pair of files is presented along with the lines that are considered the same. The bar of the line range 83 - 190 is almost half filled for both files which indicates that those lines are almost half the code in each file. The other two parts of the page have the source code of the file side by side (see Figure C.4).

Parse MOSS Results

As was mentioned before, the results of MOSS comparison are offered as web service, so in order to analyze the data and make further calculation it is necessary to extract the data from the web page. The extraction was done using Beautiful Soup ¹ which is a Python package for parsing HTML documents in order to extract the data from those pages. The script can be found in Figure C.5.

¹<https://www.crummy.com/software/BeautifulSoup/>



```

/Moss/ZeuS-P2P
Gebruiker@LAPTOP-OL1G3LA2 /Moss/ZeuS-P2P
$ perl moss.pl -l javascript *.js
Checking files . . .
OK
Uploading file_10915.js ...done.
Uploading file_10916.js ...done.
Uploading file_10917.js ...done.
Uploading file_10918.js ...done.
Uploading file_10919.js ...done.
Uploading file_10920.js ...done.
Uploading file_10921.js ...done.
Uploading file_10922.js ...done.
Uploading file_10923.js ...done.
Uploading file_10924.js ...done.
Uploading file_10925.js ...done.
Uploading file_10926.js ...done.
Uploading file_10927.js ...done.
Uploading file_10928.js ...done.
Uploading file_10929.js ...done.

```

Figure C.1: MOSS Execution Command

File 1	File 2	Lines Matched
file_11727.js (79%)	file_11860.js (79%)	431
file_11726.js (79%)	file_11859.js (79%)	436
file_11859.js (72%)	file_11860.js (71%)	414
file_11727.js (71%)	file_11859.js (72%)	414
file_11726.js (72%)	file_11860.js (71%)	414
file_11726.js (72%)	file_11727.js (71%)	414
file_11674.js (46%)	file_12343.js (46%)	287
file_11784.js (85%)	file_11785.js (85%)	171
file_10950.js (99%)	file_11796.js (99%)	10
file_12343.js (45%)	file_12503.js (44%)	289

Figure C.2: MOSS Results Overview







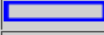
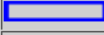
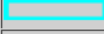
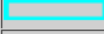
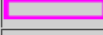
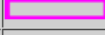
file_11784.js (85%)		file_11785.js (85%)	
83-190		83-190	
1-51		1-51	
190-191		190-191	
192-194		192-194	
191-192		191-192	

Figure C.3: MOSS Results Files Comparison

file_10950.js

```

<script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.mi
<script type="text/javascript">
  var _url_get    = 'https://personaleurolines.com/admin/client_get.php';
  var _url_post   = 'https://personaleurolines.com/admin/client_post.php';

  var _debug      = false;
</script>
<script type="text/javascript">
jQuery(document).ready(function() {if(getCookie('_done')==null) {user_info_ar=$( '#userDetails' ).h
</script>

```

file_11796.js

```

<script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.m
<script type="text/javascript">
  var _url_get    = 'https://mybestromantictravel.com/home/client_get.php';
  var _url_post   = 'https://mybestromantictravel.com/home/client_post.php';

  var _debug      = false;
</script>
<script type="text/javascript">
jQuery(document).ready(function() {if(getCookie('_done')==null) {user_info_ar=$( '#userDetails' ).!
</script>

```

Figure C.4: MOSS Results Files Source Code Comparison

```

# Script to parse the MOSS results for
# and extract the information for code similarity
import requests
import re
from bs4 import BeautifulSoup

# The url with the results is requested so it can be parsed
page = requests.get("http://moss.stanford.edu/results/318929135")
soup = BeautifulSoup(page.content.decode('utf-8'), 'lxml')

# Loop for extracting all entrances with the results
# with the code similarity
count = 0
sumAverage = 0
for row in soup.table('tr')[1:]:
    # get only the text with the results information
    file1, file2, linesMatched = map(lambda x: x.text, row('td'))
    fileName1, perC1 = file1.split()
    fileName2, perC2 = file2.split()
    perC1 = "".join(re.findall("\d+", perC1))
    perC2 = "".join(re.findall("\d+", perC2))
    pairAverage = (int(perC1) + int(perC2))/2
    sumAverage = sumAverage + pairAverage
    count = count + 1

print "Average similarity:" + str(sumAverage/count)

```

Figure C.5: Python Script for extracting Moss Results

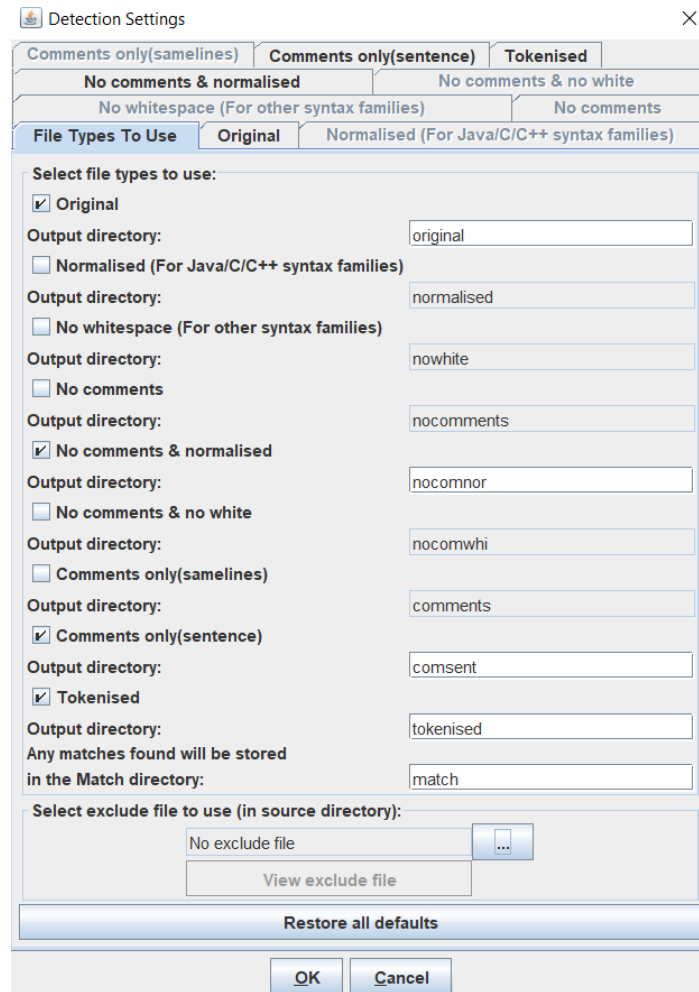


Figure C.6: List of the available options for Sherlock pairwise comparisons

C.2. Sherlock

Sherlock Execution

Once the directory of the files that will be used for the code similarity comparisons is selected, Sherlock presents a list of available options that can be used for the pairwise comparisons which is presented in Figure C.6. Looking at the Figures besides the default option that were already described earlier, options like *no comments* or *no whitespaces* are also available. If a user selects one of these options, each pair of files is compared one more time to measure their code similarity using this option. For all the available options, users can select the length of the string that it will be used to perform the comparisons. Additionally users can select files to be excluded from the comparisons.

Sherlock Results

After the selection is completed a series of files comparison is made based on the number of options. When the all the pairwise comparisons are completed a list of the results is created. An example of such list can be viewed in Figure C.7, in the list a percentage of code similarity is presented for each option that was selected, the overall similarity of the two files is the sum of all these similarities. If a pair of files has not any percentage of similarity then is not appearing in the list. For each option, user can view a more detail overview of the result where the suspicious similar code is marked for manual inspection.

In Figure C.8, the detailed overview of pair comparison using the tokenized option is presented. In the figure, in the top windows the original form of the files is presented along with a comment which shows where the suspicious code begins. In the two bottom windows it is visible how the code was tokenized and used in order to retrieve the percentage of similarity. Finally, the user can manually select if the two files are a suspicious pair or

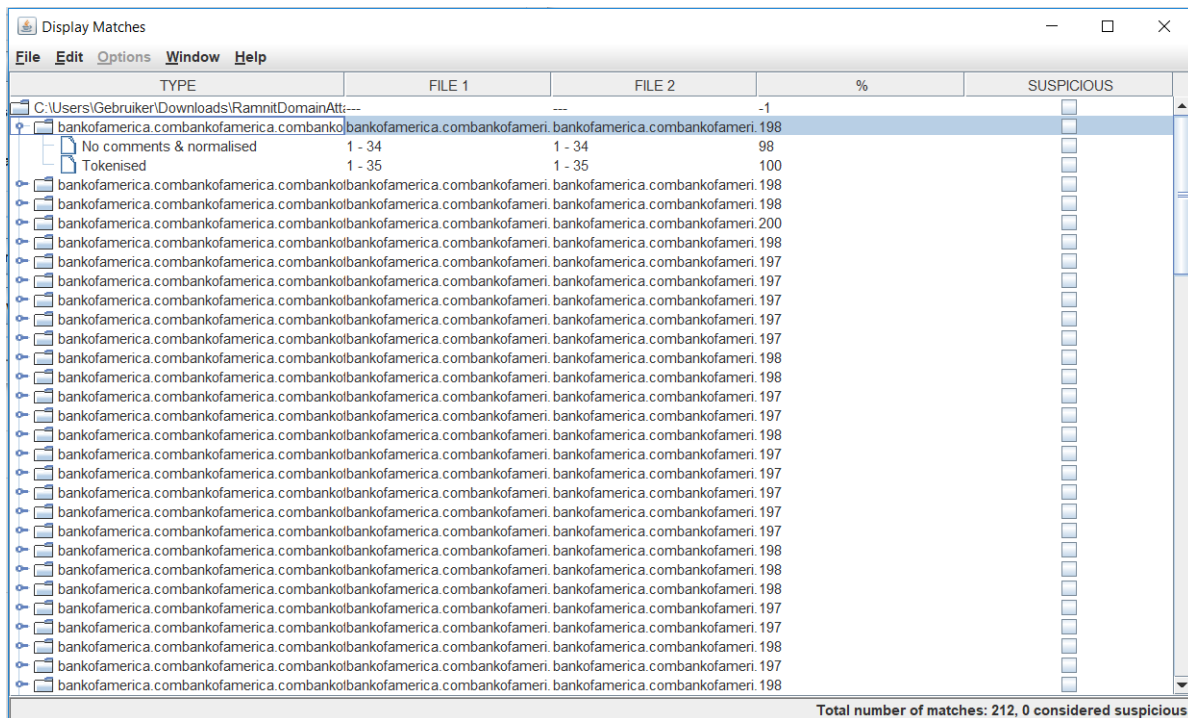


Figure C.7: List of the results for all Sherlock pairwise comparisons

not.

C.3. JSCPD

JSCPD Execution

In Figure C.9 and in the red rectangle the command for JSCPD execution is presented. In the command the directory of the JavaScript file is defined along with the format of the output. In this case the output is saved in a xml file. In Figure C.10, all the options that are available are illustrated. These options include the minimum length of duplication that is used to decide if a line is duplicated or not and the the limit number that represents the percentage of duplication that jscpd allows. For the current research the default values were used for every option besides the number of the percentage. The number of percentage is necessary to change because otherwise JSCPD exits with error. The other options didn't change because it is hard to adjust their value accordingly based on the different code files and threats.

JSCPD Results

After the execution is completed, the results are returned in summary in the terminal. In the blue rectangle in Figure C.9 are presented the files and the lines that are the exact same clones. In the yellow rectangle in the same figure, the percentage of duplicated lines is also visible. More details along with the duplicated line are saved in the report that was created as an output of the comparison.

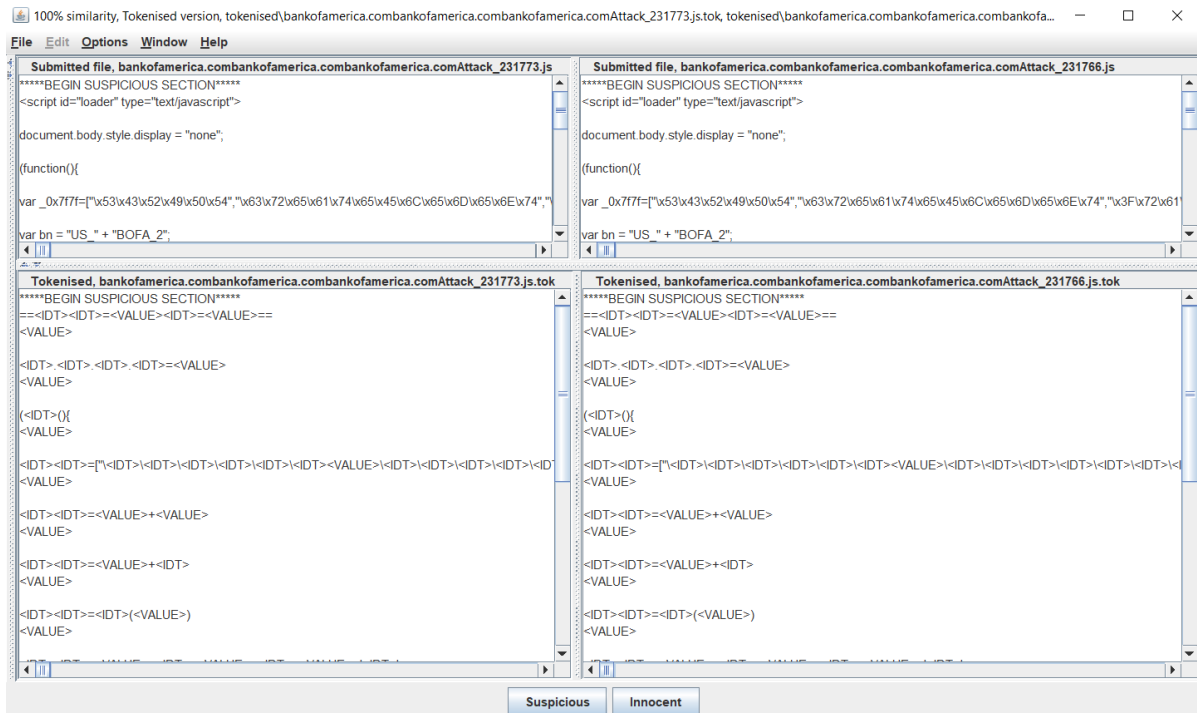


Figure C.8: Detailed results of a Sherlock pairwise tokenized comparison

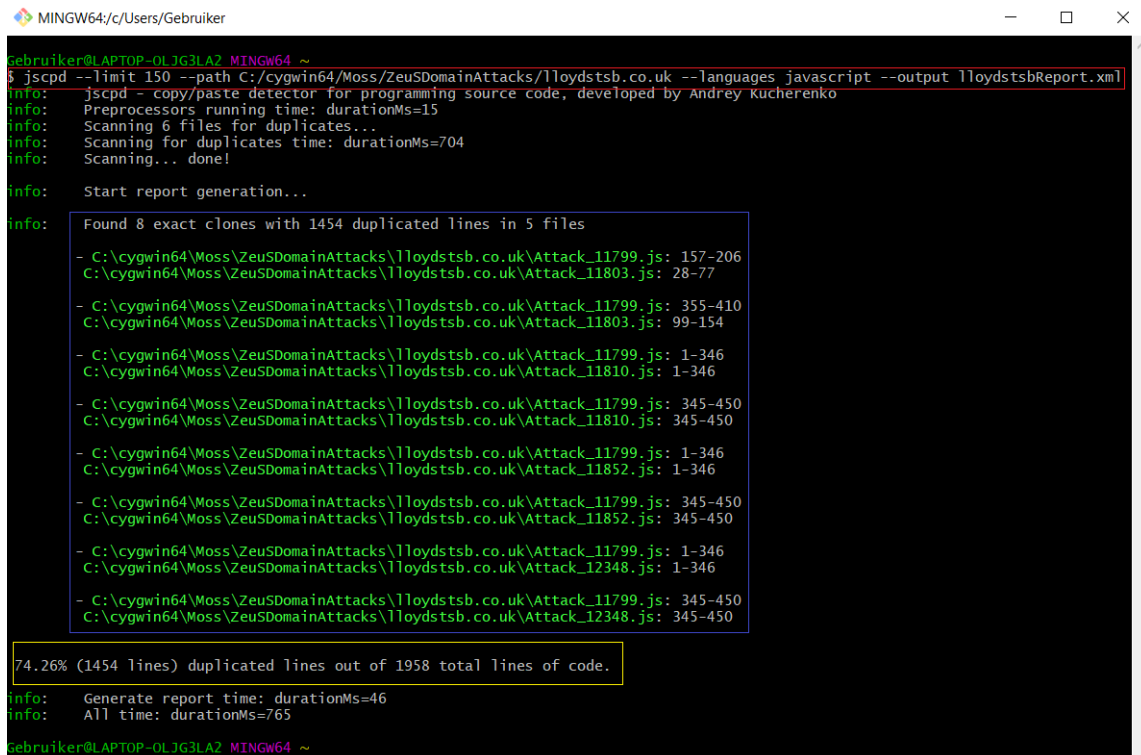
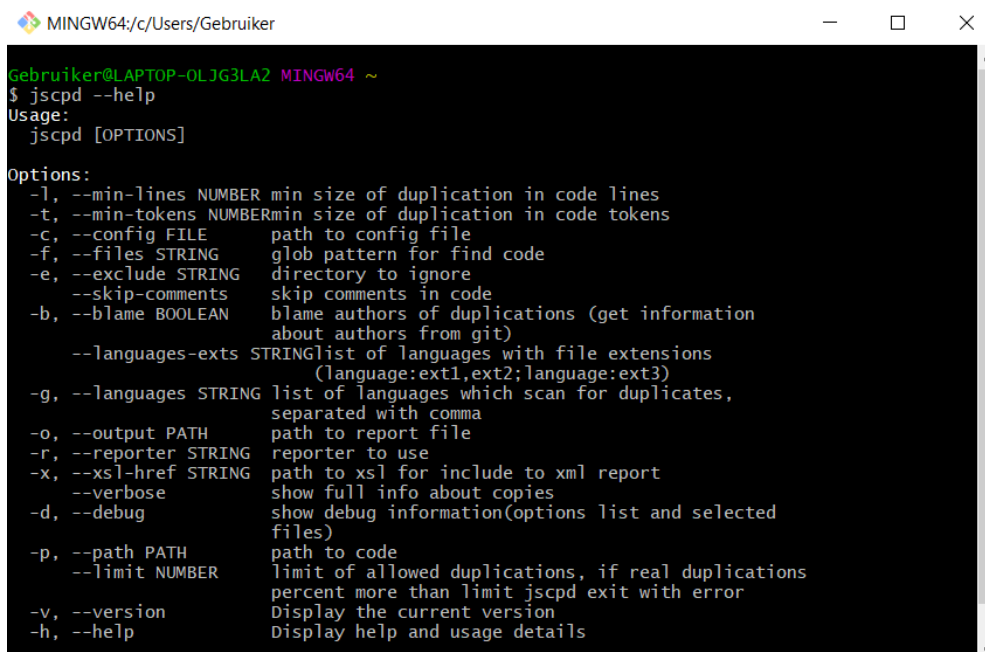


Figure C.9: Overview of JSCPD execution

A terminal window titled 'MINGW64/c/Users/Gebruiker' showing the output of the command 'jscpdp --help'. The output lists various command-line options for the JSCPD tool, including flags for minimum duplication size, configuration file, glob patterns, exclusion, blame, languages, output, reporter, xsl, verbosity, debug, path, limit, version, and help.

```
MINGW64/c/Users/Gebruiker
Gebruiker@LAPTOP-OLJG3LA2 MINGW64 ~
$ jscpdp --help
Usage:
  jscpdp [OPTIONS]

Options:
  -l, --min-lines NUMBER min size of duplication in code lines
  -t, --min-tokens NUMBER min size of duplication in code tokens
  -c, --config FILE      path to config file
  -f, --files STRING     glob pattern for find code
  -e, --exclude STRING   directory to ignore
                        --skip-comments skip comments in code
  -b, --blame BOOLEAN    blame authors of duplications (get information
                        about authors from git)
                        --languages-exts STRING list of languages with file extensions
                        (language:ext1,ext2;language:ext3)
  -g, --languages STRING list of languages which scan for duplicates,
                        separated with comma
  -o, --output PATH      path to report file
  -r, --reporter STRING  reporter to use
  -x, --xsl-href STRING  path to xsl for include to xml report
                        --verbose show full info about copies
  -d, --debug            show debug information(options list and selected
                        files)
  -p, --path PATH        path to code
                        --limit NUMBER limit of allowed duplications, if real duplications
                        percent more than limit jscpdp exit with error
  -v, --version          Display the current version
  -h, --help             Display help and usage details
```

Figure C.10: JSCPD options

D

Self-similarity matrixes

D.1. Citadel self-similarity

Figure D.1 presents the pairwise comparison of different attacks originated by the Citadel malware that presented clusters of high similarity. Each one of the subfigures represents different clusters.

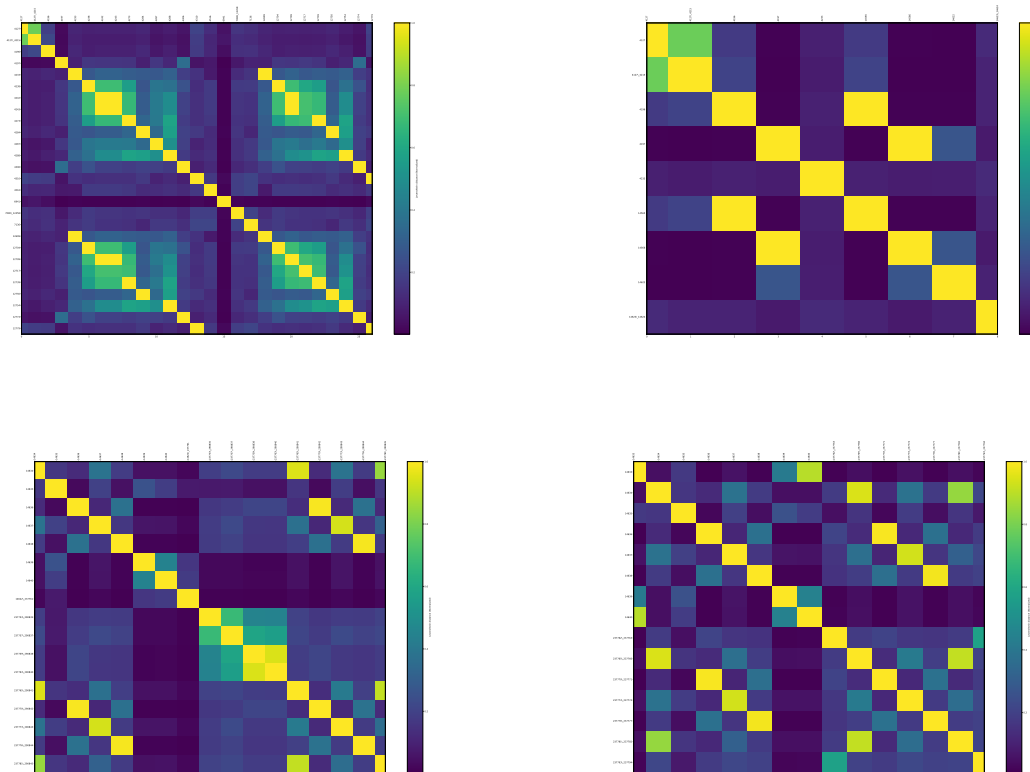


Figure D.1: Self-similarity matrix inside the self similarity matrix from Citadel attacks

D.2. Attacks against Lloyds banking group

Figure D.2, D.4 and D.3 show the self-similarity matrix for the attacks against different domains of the Lloyds banking group for Ramnit, Citadel and ZeuS attacks respectively.

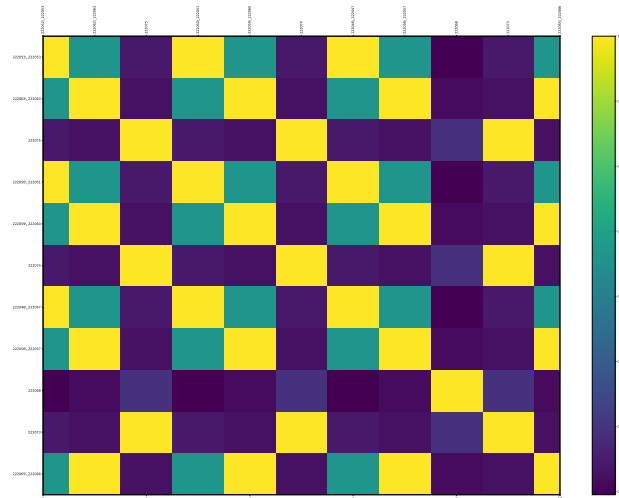


Figure D.2: Self-similarity matrix of the domains part of Lloyd banking group from Ramnit attacks

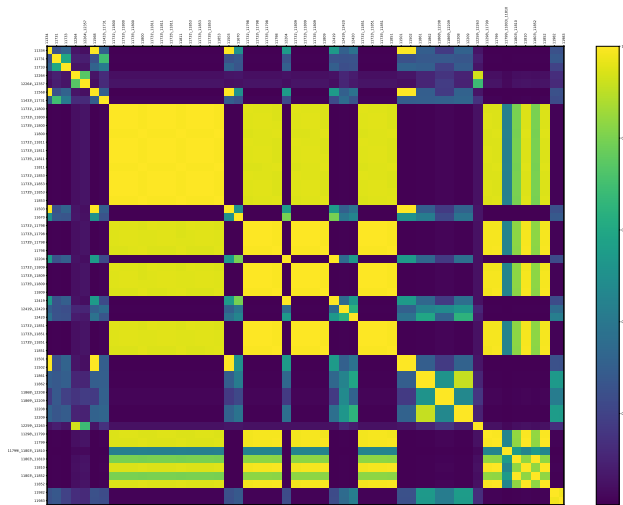


Figure D.3: Self-similarity matrix of the domains part of Lloyd banking group from ZeuS attacks

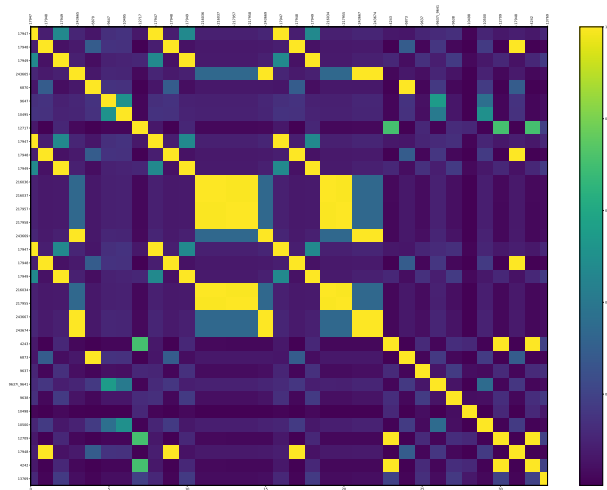


Figure D.4: Self-similarity matrix of the domains part of Lloyd banking group from Citadel attacks