A Study of Malicious Source Code Reuse Among GitHub, StackOverflow and Underground Forums

Michal Tereszkowski-Kaminski[1] [0000−0002−1141−4159], Santanu Kumar Dash[2] [0000−0002−5674−8531], and Guillermo Suarez-Tangil[1] [0000−0002−0455−2553]

[1] IMDEA Networks Institute, Spain
michal.tereszkowski-kaminski@imdea.org
guillermo.suarez-tangil@imdea.org
[2] University of Surrey, United Kingdom
s.k.dash@surrey.ac.uk

Abstract. To date, most analysis of collaboration between malware authors has been performed on meta-data and compiled binaries, while ignoring artifacts present in the source code. We collect a vast amount of malicious source code from Underground Forums posts, Underground Forum code attachments, and GitHub repositories and devise a methodology that allows us to filter out most auxiliary code, leaving the measurement to focus on malicious code. We leverage this to perform an in-depth measurement of the reuse of malicious code between these malware centers as well as StackOverflow. We find that our methodology has high precision in identifying malicious code (93.1%) and provides a contemporary snapshot of malware code reuse across the Web, offering insights into the manners in which this takes place.

Keywords: Underground forum · source code reuse · malware.

1 Introduction

The proliferation of malware source code has soared in recent years in part due to code-hosting repositories on sites like GitHub [2,12]. This wealth of data means that aside from the analysis of malware binaries, it is possible to focus on the analysis of malware source code. This has numerous advantages: source code contains, for instance, more information about programming style and is more easily human-readable, not requiring binary static analytical tools to understand the mechanisms and dependencies. Existing works studying the dissemination of malware in open-source repositories either focus on the actors and meta-data as opposed to the source code itself [12,4] or present approaches that do not scale to a large corpora [13,9].

In this work, we extract information from open-source software repositories as well as snippets from forums to identify and track malware supply chains.

We gather malware source code from 5 different sources, namely Underground Forum post code snippets, Underground Forum post attachments, GitHub repositories from Underground Forum posts, GitHub repositories collected by [12], and StackOverflow post code snippets. We collect source code in four languages: C, C++, Java, and Python, and extract over 12 million function samples from this corpus.

In an effort to identify malicious code reuse, we establish similarity between sources through a large-scale code clone measurement. To this end, we overcome two specific challenges. The first challenge is that the size of the code corpus impedes scalability of existing clone detection tools for complex clone types. The second challenge is clone detection tools do not allow tuning the tool to focus on malicious parts only, which is the focus of our work. Therefore, existing tools confuse similarities arising from ancillary code, which are small helper functions that are common enough that their detection does not imply code reuse, as malicious reuse.

To improve state-of-the-art in clone detection, and make it useful for measuring malicious source code reuse, we devise a method to filter samples in our corpus to extract malicious code only. This allows us to focus on a subset of our entire corpus improving scalability of the clone detection. The initial filtering step also allows to focus on malicious reuse and ignore ancillary methods.

We present the following contributions:

- We produce an evaluation dataset of 100 malicious repositories (3,314 files and 499,854 LoC) and 100 benign repositories (2,317 files and 434,476 LoC) in order to evaluate our malicious code filter. It is constructed from known malicious and benign GitHub repositories.
- We devise a method to filter out benign source code, with the aim of leaving only samples exhibiting malicious code. This method achieves 93.1% precision in identifying malicious software repositories from a balanced set of malicious and benign samples.
- We perform a measurement of source code reuse in 4 different programming languages from StackOverflow, GitHub and Underground Forums, focusing on the malicious characteristics.

This paper is structured as follows: we present the related work and where this paper fits into the literature in §2. In §3 we explain the background around source code reuse and the taxonomy of code clones. In §4 we present our methodology and data corpus and we evaluate our methodology in §5. In §6 we offer the measurement of source code reuse among the different data sources and a discussion of it. We present some limitations of our work, the key takeaways, and offer conclusions in §7.

2 Related Work

This section makes reference to code clone types 1 through 4, the taxonomy of which is explained in Section 3.

Code clone identification is an active area of research, as shown by Ragkhitwetsagul et al.'s comprehensive study [10]. This research covers same-language and cross-language clones, with significant contributions from Cheng et al.[3], Nafi et al.[6], and Yahya et al.[20]. Recent progress emphasizes large-scale clone detection, primarily utilizing token-based methodologies[16,15,7]. Token-based approaches excel in identifying nearly identical code (Type-1 and Type-2 clones) by analyzing the raw code, without considering higher-level abstractions like abstract syntax trees and control flow graphs, which are necessary for reliable detection of Type-3 and Type-4 clones.

In malware source code analysis, notable works include those by Rokon et al.[12], Islam et al.[4], and Calleja et al. [2]. Rokon et al. identify over 7,000 malware repositories on GitHub, doubling every three years. Islam et al. analyze GitHub code authors' relationships with security forums, focusing on user activity and repository metadata rather than source code. Calleja et al.'s study on code reuse among malware samples is limited to 456 samples, with some dating back to 1975.

Additionally, Rokon et al.'s work [13] achieves high precision and recall (98% and 96%, respectively) in identifying malicious GitHub repositories based on source code. However, due to its reliance on machine learning algorithms, the approach is not scalable to our extensive dataset. This highlights the need for both a scalable clone detection tool and a filtering method to narrow down the candidates for analysts.

In parallel, efforts have been directed towards identifying collaborative communities on GitHub, irrespective of their association with the malware ecosystem [5]. Moradi et al. discovered that the preferred programming language significantly influences collaboration dynamics. However, their study lacks an in-depth analysis of the source code itself. On the other hand, comprehensive studies of code reuse between GitHub and StackOverflow are available [21,1,11], but do not focus on malware as such and they **do not account for the interplay between malware development and the dependencies introduced by underground commoditization** [18].

To the best of our knowledge, this is the first work to study code reuse between public malware sources, underground hacking forums, and the biggest source code Q&A site on the Web, StackOverflow, and the first to compare code reuse patterns in malicious code between different programming languages.

3 Taxonomy of Clones

Software clones represent pairs of code snippets performing identical tasks, spanning a spectrum from outright identity to varied implementations aiming at the same functionality—illustrated by, for instance, using Quicksort instead of Bubblesort for sorting.

A key feature of code clones is semantic equivalence, even when different statements and control structures are used. Roy et al.'s taxonomy [14] outlines four types. Type-1 clones involve identical code with variations in whitespace,

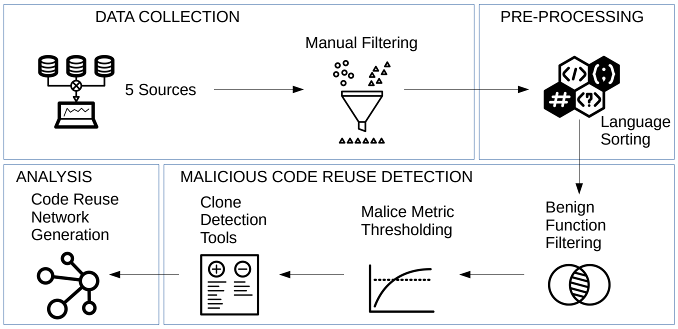
DATA COLLECTION
5 Sources
Manual Filtering
PRE-PROCESSING
Language Sorting
ANALYSIS
Code Reuse Network Generation
MALICIOUS CODE REUSE DETECTION
Clone Detection Tools
Malice Metric Thresholding
Benign Function Filtering

Fig. 1: Methodology outline.

layout, and comments. Type-2 clones extend to syntactically identical fragments with additional variations in identifiers and literals. Type-3 clones include copied fragments with further modifications, often involving alterations, additions, or removals of statements, alongside variations from previous types. Type-4 clones encompass situations where two code fragments perform the same computation but differ in their syntactic manifestations.

Identifying clones does not guarantee code reuse; coincidental clones may occur, where one reproduces code unknowingly, especially with shorter, frequently used snippets. Our analysis focuses on function-blocks rather than single lines of code, making it probable that flagged clones indicate code reuse rather than coincidental similarities. Even when coincidental, code clones are relevant to the analysis, allowing the drawing of links of functional similarity between malware samples.

4 Methodology

We introduce an innovative approach in code clone detection, harnessing a vast dataset to specifically target and isolate malicious code, thereby refining our analysis to focus on its most pertinent aspects.

The measurement aims to capture instances of code reuse both where the person who copies the code modifies it while integrating it into their software and where they do not alter it at all. Thus, we focus on Type-1, Type-2, and Type-3 clones. As a result of the vastness of the data corpus, the first step of our methodology is to filter out benign code from this measurement. We define benign code as software that does not produce actionable malicious actions.

4.1 Data Collection

We collect our data from 5 sources, as described in detail next.

Data Gathering.

We gather source code in 4 different programming languages: Java, C, C++ and Python — collected from the following sources: Underground Forum & SourceFinder Repos, Underground Forum Posts, Underground Forum Source Code Attachments and StackOverflow Posts. All data from Underground Forums is sourced from the CrimeBB dataset, provided by the Cambridge Cybercrime Centre[3], which is available to academic researchers under a legal agreement designed to ensure ethical use.

We next explain all our data in detail:

Source	C/C++	Java	Python
Underground Forum & SourceFinder (UFSF) Repos.	7,151	8,955	122,609
Underground Forum (UF) Posts	7,490	7,587	9,730
Underground Forum (UF) Attachments	1,965	87	5,340
StackOverflow (SO) Posts	1,028,831	2,480,017	3,603,983

Table 1: Dataset breakdown by source. These are the numbers of source code files, before splitting into function-level granularity samples.

GitHub Repositories, from two distinct sources:

- *SourceFinder (SF)*: We acquired a compilation of 6,790 repositories from the SF database, as detailed in the work by Rokon et al. [12]. SF utilizes a supervised learning approach to discern malware repositories, assigning them specific malware types.
- *Underground Forums (UF)*: Our analysis encompassed posts extracted from underground forums, from which GitHub repository links were derived using regular expressions. This effort yielded a total of 9,290 repository links.

The resultant amalgamation, referred to as the UFSF Repositories Dataset, encompasses 15,312 repositories. Notably, a subset of these repositories is common to both SF and UF sources, amounting to 768 instances out of the 15,312. To ensure dataset integrity, duplicate repositories are omitted. Additionally, repositories that are no longer in existence as of the crawling period were excluded, resulting in a refined dataset comprising 14,374 repositories. Among these, 5,704 and 8,522 repositories originated from SF and UF sources, respectively, with 148 repositories shared between both sources.

Underground Forums Snippets.

This collection comprises code snippets extracted from posts on five different Underground hacking Forums from the CrimeBB dataset [8]. The purposes of these posts vary, encompassing requests for debugging assistance, tutorials for forum members on programming specific functionalities, and various coding techniques. Both the original poster of the forum thread and replies may include code snippets, which are typically unable to compile due to being not being

[3] `www.cambridgecybercrime.uk`

intended as complete software. From these forums, we collect 7,125 threads containing Java snippets, 7,513 threads with C or C++ snippets, and 3,504 threads containing Python snippets.

Underground Forums Attachments. These are code attachments that are sometimes included in the posts of the underground forums. These range in nature from sharing malicious projects that the poster wrote themselves to software projects which demonstrate how to code a certain kind of application. They are scraped from two underground forums: MPGH and HackForums. MPGH is a known forum for exchanging knowledge on the creation of video game hacks, commonly known as *cheats*. HackForums is also a known underground forum devoted to hacking in general. From MPGH, we gather 1,926 attachments. From HackForums, we gather 1,779 attachments.

StackOverflow Snippets. These are code snippets scraped from posts on the StackOverflow website. This website is meant to enable programmers of all kinds to get help with their coding endeavors. As with Underground Forums Post snippets, these snippets of code also only rarely are able to compile for similar reasons. These are downloaded from https://archive.org/details/StackOverflow. From these downloaded posts, 2,937,566 Java samples are extracted, 1,252,802 C and C++ samples are extracted and 3,603,983 Python samples are extracted.

Manual Filtering. In this step, we discard those GitHub repositories whose links we scraped from the Underground Forums that are not cybersecurity-related. As an example, a network sniffing tool like Wireshark is retained because its code may be used for malicious purposes. On the other hand, the source code for a video game would be discarded (though the software development kit for one would not be, as these are used in developing hacks). This step is performed by one domain expert. As there are in total 14,374 repositories, an in-depth manual analysis of each was infeasible. Thus, the expert relied on the name of the repository in the majority of cases, sometimes inspecting the repository for the purpose stated in the 'README.md' file.

4.2 Pre-Processing

We sort the corpus of samples by programming language. We distinguish two different scenarios based on the type of data source:

- In the case of UFSF Repositories and UF Attachments, we label each sample by their file extension: for example, ".java" would be classed as a Java source file, whereas ".cpp" would be classed as a C++ source file.
- On the other hand, samples from SO Posts and UF Posts do not have an inherent file extension, as they are text scraped from forum threads. As such, we employ the machine-learning tool Guesslang [22] to identify the language they are written in.

Once we perform the language sorting, we then apply the core component of our method to identify malicious code reuse as we describe next.

4.3 Malicious Code Reuse Detection

In this phase, we employ a couple of filtering steps in order to discard code samples that do not contain code interesting to a malware analyst. We then pass the remaining samples to a code clone detection tool in order to produce a code reuse network.

Benign Function Filtering. We conduct Type-1 code clone detection between our corpus and a subset of benign samples, detailed in Appendix A. This subset varies depending on the language under study, representing a portion of the total benign dataset for each language. If a function sample in our corpus is identified as a Type-1 clone with a sample in the benign set, we remove it from the corpus. We assume that Type-1 clones of methods in our dataset that are also present in the subset of benign samples, are a reliable indicator of benign behaviour. We discuss the impact of this assumption in subsection 7.1. We only consider Type-1 clones at this stage for several reasons. First, our study diverges from exclusive scrutiny of malware functions, encompassing a broader scope. Our primary goal is to substantially narrow the search space, but the inclusion of benign functions in our dataset does not alter or compromise our outcomes. Second, the distinction and prioritization of Type-1 clones are fast, facilitating efficient identification and filtering of identical code fragments.

Finally, it ensures we retain samples that originated as benign code but were altered to perform malicious functions.

Malice Metric Thresholding. We construct a metric that can suggest whether a sample belongs to a malicious project. We compute the malice metric as follows.

Let W_f be the set of function calls in the allowlist with normalized incidence rates, and W_i be the set of import statements in the allowlist with normalized incidence rates.

Let S be a sample, and S_f be the set of function calls found in sample S, and S_i be the set of import statements found in sample S.

Then, the malice metric $M(S)$ for sample S is calculated as follows:

$$M(S) = \varSigma_{f \in W_f \cap S_f} f + \varSigma_{i \in W_i \cap S_i} i \tag{1}$$

Where:

$W_f \cap S_f$ denotes the intersection of the allowlist function calls and the function calls found in the sample.

$W_i \cap S_i$ denotes the intersection of the allowlist import statements and the import statements found in the sample.

f represents the normalized incidence rate of a function call in the allowlist W_f.

i represents the normalized incidence rate of an import statement in the allowlist W_i.

These two steps result in a malice metric score.

Known Malicious Function Calls.

Function calls are extracted by parsing with regular expressions and using heuristics to discard non-matching strings. Any string which ends in an open brace (‘(’) is matched. Then, a series of heuristics are employed to discard those strings which are identified to be control statements, and the line context of the string is checked to ensure that the string is not a declaration. An example of an extracted C++ function call is *get_new_cred(*.

Known Malicious Import Statements.

We compile a list of import statements used in malicious projects, identified by their GitHub repositories labeled by [12]. We parse all samples from all the data sources and discard those samples that do not contain any of the import statements in the whitelist.

Clone Detection Tools.

At this point in the process, we pass the data to clone detection tools, which help us to identify clone links between the different samples. To do this, we utilize CloneWorks, which is a highly advanced and scalable clone detection tool [17]. The scalability of this tool is particularly important, given the large volume of data that we are dealing with. In addition, it achieves high accuracies, outperforming among others the popular tool SourcererCC [16].

4.4 Analysis

Code Reuse Network Generation.

The output of the clone detection tool is used to produce a graph. The nodes are projects that contain samples: what this is varies by malware source as we explain next. (1) In UF Posts, a node is a thread from Underground Forums. (2) In UF Attachments, it is a source code attachment from Underground Forums. In UFSF Repositories, it is a GitHub repository. (3) In StackOverflow Posts, it is a thread from the StackOverflow website. The edges between nodes denote the existence of detected code clones. The resulting network for our data corpus is shown in Figure 2.

5 Evaluation

In this section, we first evaluate the steps in our methodology, and then more specifically evaluate the method used to extract function calls from our samples. We include an evaluation of an attempt to use GPT3.5 for this latter task.

5.1 Evaluation Dataset

This consists of 100 malicious repositories and 100 benign repositories, dubbed *SFKC Dataset*, all of these containing varied numbers of source code files. The 100 malware repositories are randomly selected from the SourceFinder repository list, while the 100 benign repositories are randomly selected from the 50KC dataset, explained in Appendix A. These are then verified by a domain expert to ensure they belong in their respective categories. We select 100 of each category as manually inspecting a greater number was judged infeasible.

5.2 Evaluation of Methodology

We test our methodology on the SFKC evaluation dataset described above. The results are shown in Table 2.

We withhold 12.5% of the SourceFinder repositories as a slice of the dataset used to fine-tune and evaluate our malice metric. This number is an attempt to strike a balance between a sufficiently representative sample of the malicious repositories while also preserving as many repositories as possible for the measurement itself. This evaluation slice amounts to 713 repositories, of which 60% (427, SF Training) are used for populating the Allowlist, and 40% (286, SF Testing) for testing. Using this slice, we experiment with multiple malice metric mechanics, of which the following (Malicious Function Call Allowlisting and Malicious Import Allowlisting) had positive results and were chosen to be included in the method.

Benign Function Filtering.

Using the CloneWorks tool, Type-1 clone detection is performed among the SFKC evaluation dataset and a set of repositories from the 50KC dataset which were not used for constructing the SFKC evaluation dataset. There are 3,510 of these, we dub them KC Training. Any source file containing a Type-1 clone to a sample in the KC Training set is removed on suspicion of being a benign sample.

Malicious Function Call Allowlisting.

We use the 427 SF Training repositories to populate the Allowlist with function calls. Here, all samples in the SFKC evaluation dataset are checked for the presence of these collected function calls. Any samples that do not contain any of these function calls are removed.

Malicious Import Allowlisting.

We extract import libraries from the 427 SF Training repositories. Thus, all samples in the SFKC evaluation dataset are checked for the presence of these same imports. Any samples which do not contain any of these imports are removed.

Thresholding.

We compute the malice metric as described in §4.

Results.

All the steps combined produce an output where 93.1% of the samples are malicious repositories, up from 50% at the outset. We find that Malicious Import Allowlisting results in the highest precision in identifying malicious code.

We experimentally find the best place to draw the threshold for keeping samples. To do this, we test thresholds from

1×10^{-5} to 2×10^{-4} in intervals of 1×10^{-5}. We find that the threshold of 4×10^{-5} results in the best balance between precision and recall (results in the highest F1 score). If the scores of samples are calculated in a repository-wise manner, this F1 score from using the metric is 0.76. This lowers to 0.54 if the scores of samples are calculated in a function-wise manner. Thus, it appears that using the metric to identify which repositories are malicious is significantly more effective than identifying which lone code functions are malicious.

Step of methodology applied	No. Malicious	No. Benign	% of Samples Malicious
Original Dataset	100	100	50.0%
(1) Benign Function Filtering	43	13	76.8%
(2) Mal. Function Call Allowlisting	72	37	66.1%
(3) Mal. Import Allowlisting	47	4	92.2%
Combined (1, 2, 3) w/ Thresholding	27	2	93.1%

Table 2: Effect on the numbers of malicious and benign code sample identification in the SFKC evaluation dataset of steps in the methodology.

5.3 Evaluation of Function-call Extraction Techniques

In order to evaluate function call extraction techniques, we take a stratified sample across our data sources, so that there are 5 samples per source. Each sample is a source code file. In total, 5 samples are from Underground Forum snippets, with the same number from Underground Forum attachments, GitHub repositories and StackOverflow Posts. We perform this evaluation on the C++ programming language.

We extract a ground truth of function calls in each of the 20 samples through manual analysis. We then use each evaluated method to produce a list of function calls from the samples, and this is compared against the ground truth to produce the performance metrics of recall and precision. We find 321 function calls total within the 20 samples.

We compare the performance of our method with a function-call extraction method derived from prompting a popular Large Language Model. For GPT3.5, the model was given the following prompt: *"Please have a look at the following snippet of code and reply with ONLY a comma-separated list of the function calls found in the snippet, in the order that they appear. For each call, do not include the parameters passed to it. In the list, include each function call even if it is a duplicate. If there are no function calls, return the word 'NULL'. The snippet is:"* followed by the snippet of code. We see that our method outperforms the baseline by a factor of 2.

Method	Regex + Heuristics	GPT3.5
Recall	0.798	0.377
Precision	0.834	0.515
F1	0.816	0.435

Table 3: Evaluation of different function call extraction methods.

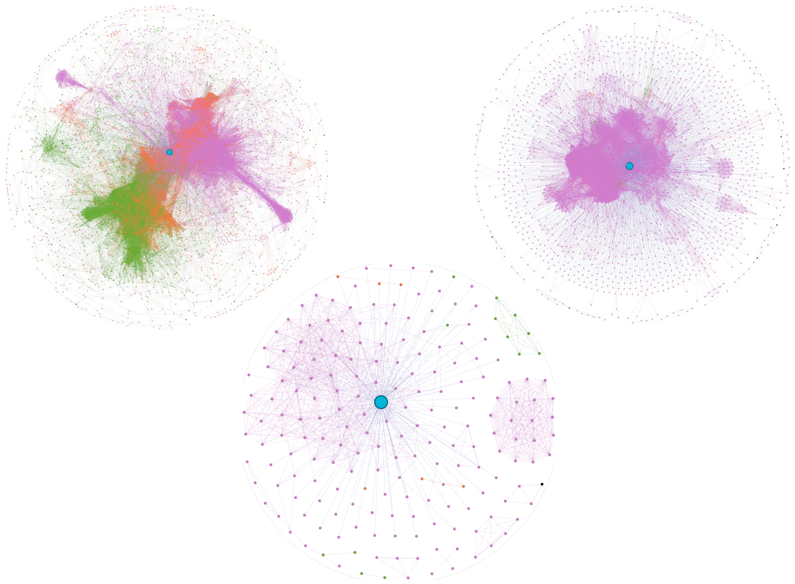

Fig. 2: Code reuse networks after methodology filtering, C/C++ top left, Python top right, Java bottom. Pink nodes represent UFSF GitHub repositories, green nodes represent UF posts, orange nodes represent UF attachments, and the blue node represents StackOverflow posts.

6 Code Reuse Measurement

We next present the measurement of code reuse in 4 different languages, presenting C and C++ in the same section as the samples for both are processed together. For each language, we first give an overview of the data extracted from the corpus using our methodology, and then present statistical details of the code reuse. The full code reuse networks for these categories are shown in Figure 2.

6.1 C & C++

The number of source files in these two languages in UFSF repositories is 395,885, the number of source files in UF Snippets is 18,387, the number of source files in UF Attachments is 67,313 and the number of source files from SO Snippets is 1,252,802. Out of these, 24,530 (6.2%) of the source files from UFSF repositories do not contain any API calls, 6,794 (36.9%) of the source files from UF Snippets do not contain any API calls, 4,330 (6.4%) of the source files from UF Attachments do not contain any API calls, and 174,725 (13.9%) of the source files from SO Snippets do not contain any API calls.

Code Reuse.

There are 491,975 function-granularity snippets gathered from all the sources in C and C++, which are found to be Type-3, Type-2 or Type-1 clones with some other snippet in the corpus. The majority of these, 448,653 are from UFSF repositories. 24,889 are from UF Attachments, 10,753 are from StackOverflow posts and 7,680 are from UF posts. The number of these which are cloned with a snippet from UFSF repositories is 463,424 (94.2%). The number of snippets that are cloned with a snippet from UF Attachments is 64,970 (13.2%). The number of snippets that are cloned with a snippet from StackOverflow is 13,796 (2.8%). The lowest source with which snippets are cloned is UF Posts, with 12,543 (2.5%) snippets being cloned with snippets from UF Posts.

A breakdown of the percentages of reused samples from each source that are reused with particular sources is shown in Table 4.

	UFSF Rep.	UF Posts	UF Attach.	SO Posts
UFSF Rep.	99.7%	3.6%	25.3%	2.7%
UF Posts	7.9%	94.6%	27.1%	8.4%
UF Attach.	59.1%	6.2%	83.8%	1.6%
SO Posts	16.7%	10.9%	3.5%	95.9%

Table 4: Percentage of C/C++ samples within source found to contain clones with each source. For example, 3.6% of the samples within UFSF Repositories are found to be cloned with some sample from UF Posts.

There are 4,138 nodes in the C/C++ code reuse network shown in Figure 2, split into 152 distinct clusters. The largest cluster contains 3,706 nodes (89.6%), with 92 clusters being pairs of nodes. We next discuss the most prominent similarity matches seen in C/C++ code reuse and we refer the reader to Appendix B for a comprehensive analysis:

- **Cluster #2:** These are 12 hacks for a videogame called DayZ. All 12 of these projects are code attachments from Underground Forums. They reuse a specific piece of malicious code which is a thread callback function.
- **Cluster #4:** These are 9 Linux malware, 8 of them with Rootkit in the name, and all 9 GitHub repositories. The name of one of them suggests it was a homework assignment for a class. They reuse a piece of code that initializes the rootkit.
- **Cluster #5:** These are 8 threads from an Underground Forum. Their code reuse centers around low-level memory manipulation with function names such as *ModifyMemory()* and *WriteToMemory()*.
- **Cluster #6:** These are 7 ransomware samples, all GitHub repositories, one of them being a collection repository for ransomware. The reuse present are functions that encrypt and decrypt files, as well as getting directory listings of files on the host system.

6.2 Java

Overview.

The number of source files in UFSF repositories is 3,460,516, the number of source files in UF Snippets is 24,583, the number of source files in UF Attachments is 25,038 and the number of source files from SO Snippets is 3,161,918. Out of these, 360,598 (10.4%) of the source files from UFSF repositories do not contain any API calls, 5,141 (20.9%) of the source files from UF Snippets do not contain any API calls, 1,223 (4.9%) of the source files from UF Attachments do not contain any API calls, and 331,515 (10.5%) of the source files from SO Snippets do not contain any API calls.

Code Reuse.

There are 97,212 function-granularity snippets gathered from all the sources in Java, which are found to be Type-3, Type-2 or Type-1 clones with some other snippet in the corpus. The majority of these, 95,084 are from StackOverflow. 1,891 are from UFSF repositories, 28 are from UF snippets and 209 are from UF attachments. The number of these which are cloned with a snippet from StackOverflow is 95,474 (98.2%). The number of snippets which are cloned with a snippet from UFSF Repositories is 12,951 (13.3%). The number of snippets which are cloned with a snippet from UF Posts is 321 (0.3%). The lowest source with which snippets are cloned is UF Attachments, with only 270 snippets being cloned with snippet from UF Attachments.

A breakdown of the percentages of reused samples from each source which are reused with particular sources is shown in Table 5.

	UFSF Rep.	UF Posts	UF Attach.	SO Posts
UFSF Rep.	95.9%	0.01%	0.5%	95.6%
UF Posts	7.1%	92.8%	0.0%	25.0%
UF Attach.	31.1%	0.0%	73.2%	4.3%
SO Posts	11.8%	0.3%	0.05%	99.8%

Table 5: Percentage of Java samples within each source found to contain clones with each source. For example, 0.01% of the samples from UFSF Repositories are found to contain clones with samples from UF Posts.

The Java code reuse network seen in Figure 2 consists of 200 nodes, split into 10 independent clusters. The largest cluster makes up 77.5% nodes in the network and it is mainly held together by the StackOverflow posts node, pictured in the figure as a large blue node. Upon manual inspection, it appears that a substantial amount of this largest cluster is ancillary code that can be used for a wide range of malicious activities. For example, we see code from StackOverflow related to getting user keyboard input, and this is used in the keyloggers connected to the StackOverflow node in the figure. We next discuss the most prominent similarity matches seen in Java code reuse (cf. Appendix B for a comprehensive analysis):

- **Cluster #1:** There are 6 nodes in the network that come from Underground Forum snippets, making up 3.1% of the network. All of them are within the largest cluster. They deal with socket connections and user login functionality. Underground Forums and SourceFinder repositories on the other hand make up 89.8% of the network. These variants vary greatly in the malicious functionality they reuse, with some reusing socket connectivity code, and others network vulnerability scanning code. Still another sample obtained code from StackOverflow which would fake the working of threads using *Thread.sleep()* calls.

- **Cluster #2:** The second largest cluster consists of 19 cryptocurrency miner GitHub repositories. The reuse centers around various functionality, from blockchain protocol implementations encryption implementations.
- **Cluster #3:** The third largest cluster consists of 7 hacks for the video game Call of Duty: Modern Warfare 3, all existing in the network as source code attachments from Underground Forums.
- **Cluster #4-5:** The fourth and fifth largest clusters both contain 4 projects each. One consists of 3 keyloggers and a Remote Administration Tool trojan with the code reused centered around keylogging activity. The other cluster consists of 4 blockchain implementations.

6.3 Python

Overview. The number of source files in UFSF repositories is 122,609, the number of source files in UF Snippets is 9,730, the number of source files in UF Attachments is 5,340 and the number of source files from SO Snippets is 3,603,983. Out of these, 11,605 (9.5%) of the source files from UFSF repositories do not contain any API calls, 3,785 (38.9%) of the source files from UF Snippets do not contain any API calls, 360 (6.7%) of the source files from UF Attachments do not contain any API calls, and 845,294 (23.5%) of the source files from SO Snippets do not contain any API calls.

Code Reuse.

There are 84,533 function-granularity snippets gathered from all the sources in Python which are found to be Type-3, Type-2 or Type-1 clones with some other snippet in the corpus. The majority of these, 66,091 are from StackOverflow. 18,096 are from UFSF repositories, 94 are from UF snippets and 252 are from UF attachments. The number of these which are cloned with a snippet from StackOverflow is 70,722 (83.7%). The number of snippets which are cloned with a snippet from UFSF Repositories is 29,712 (35.2%). The number of snippets which are cloned with a snippet from UF Attachments is 4,245 (5.0%). The lowest source with which snippets are cloned is UF Posts, with only 4,020 (4.8%) snippets being cloned with snippet from UF Posts.

A breakdown of the percentages of reused samples from each source which are reused with particular sources is shown in Table 6.

The Python code reuse network seen in Figure 2 consists of 1,775 nodes, split into 59 independent clusters. The largest cluster makes up 1,633 (92%) nodes in the network and it is mainly held together by the StackOverflow posts node, pictured in the figure as a large blue node. Instead, 39 of the 59 clusters consist of only a single pair of nodes. Samples from UFSF repositories make up 98.2% of the nodes in the network. The other sources, UF Attachments, UF

	UFSF Rep.	UF Posts	UF Attach.	SO Posts
UFSF Rep.	99.7%	0.2%	52.0%	86.3%
UF Posts	11.8%	20.6%	58.8%	26.5%
UF Attach.	66.3%	10.7%	69.4%	26.6%
SO Posts	18.9%	0.2%	5.0%	98.8%

Table 6: Percentage of Python samples within source found to contain clones with each source. For example, 0.2% of the samples from UFSF Repositories are found to contain clones with a sample from UF Posts.

Posts, and StackOverflow posts, account for 0.99%, 0.74%, and 0.05% of the nodes respectively. We next discuss the most prominent similarity matches seen in Python code reuse (cf. Appendix B for a comprehensive analysis):

- **Cluster #2:** These are 7 Remote Administration Tool GitHub repositories, 5 of them being versions of one and 2 of their versions of another. There is a big overlap between this cluster and Cluster #10 of C/C++ code. The code reuse includes writing and reading files and preparing to execute shellcode.
- **Cluster #5:** These are 3 GitHub repositories that contain spamming programs for the app Instagram. They reuse a lot of code that performs the critical functionality.
- **Cluster #6:** These are 2 GitHub repositories and 1 source code attachment from an Underground Forum. They reuse code that performs UDP flooding. This is particularly interesting as it is not obvious that they are related projects.

6.4 Findings

We next summarize the key findings we extract from our analysis.

C/C++ miscreants favor more reuse. Even though there are fewer C and C++ samples than Python and Java (1,045,437 versus 3,741,662 and 2,496,646 respectively), many more samples are found to contain code reuse with some other samples in the corpus, this being 491,975 for C/C++, 84,533 for Python and 97,212 for Java. This suggests that source code reuse is significantly more prevalent in C/C++ malware development than in the other two languages.

Java miscreants favor StackOverflow. There are many more Python samples than Java samples (3,741,662 and 2,496,646 respectively) yet over 2.8 million Java methods are found to be Type-1 clones with another method in the corpus, while under 1 million Python functions are found to be Type-1 clones with another function. As well, 95.8% of Java methods which are found to be Type-1 clones are examples of reuse from StackOverflow, while the proportion of Python functions reused with this source is only 38.3%. Code written in Python is overall reused less with StackOverflow both in identical (Type-1) clones and more distant clone types.

Java miscreants reuse less from other miscreants. The Java reuse network compared to the Python network is very small (200 nodes versus 1,775 nodes) even as the two languages have similar numbers of samples which are reused (97,212 and 84,533 respectively) due to the reuse in Java being strongly focused with StackOverflow posts which are represented by a single node in the network. On the other hand in Python, while the majority of reuse of Type-1, Type-2 and Type-3 clones still comes from StackOverflow (78%), this leaves 1,774 other nodes in the network. This suggests that Java malware developers, while relying on StackOverflow to aid in developing their products, reuse code from other malware authors relatively very little compared to the other languages studied in this work.

Java miscreants almost never reuse the least between UF code attachments and other sources. Tied to the above, only 0.5% of samples from UFSF Repositories which are reused, 0.05% of SO Posts and 0 samples from UF Posts are reused with samples from UF Attachments. The figures for being reused with UF Posts are similarly extremely low. Actors who write Java malware primarily reuse code between and within GitHub repositories and StackOverflow.

C/C++ miscreants favor Underground Forums. Notably, both the Java and Python reuse networks feature relatively very little reuse with UF attachments and UF posts, while the C/C++ network reuse is balanced between all sources, with the clusters from UF attachments and UF posts being particularly closely related.

C/C++ miscreants reuse code with StackOverflow significantly less. Among C/C++ language samples, only samples from StackOverflow are reused to a high degree (95.9%) with StackOverflow. The other sources exhibit uncharacteristically low incidences of reuse with StackOverflow. As shown in Table 4, only 2.7% of UFSF Repository samples which are reused, 8.4% of UF Post samples and 1.6% of UF Attachment samples are reused with StackOverflow. These numbers for Java and Python are much higher, with both samples from UF Posts from Java and Python being reused with StackOverflow at rates approximating 25%, and samples from UFSF Repositories being reused with StackOverflow at rates approximating 90%.

7 Discussion & Conclusions

7.1 Limitations

Benign dataset. Our method's ability to identify malicious source code relies on the characteristics of the benign datasets used for benign source code removal. It is crucial to use a dataset with a diverse range of benign code from various projects. While it is expected that our "benign datasets" contain an amount of malware in line with the incidence rate of malware on GitHub, the high precision of our tool suggests that this contamination is not problematic.

Underlying tools. For code reuse identification, we depend on CloneWorks. Conducting function-wise clone detection, especially with code from Underground Forum posts, may be incomplete due to uncompilable samples.

Heuristics. In constructing the malice metric, we pinpoint sets of function calls and libraries in malware repositories. A larger and more varied malware dataset enhances effectiveness by providing representative function calls and libraries. Although the SF repositories should be representative, the dataset composition method claims 89% accuracy, making our ground-truth not perfectly reliable.

Novel malware. Given our method's reliance on existing malware, it may have limited efficacy in detecting novel malware types that use different techniques. However, it is commonplace not to start from scratch when developing commodity malware, even when devising a novel campaign [2,19].

Code reuse origins. In our analysis, we do not take into account the timestamp information for each sample. As such, when two samples exhibit code reuse, we are unable to say whether the code is copied from the first or the second sample. An analyst using our method would likely benefit from investigating the temporal information to trace the direction of code reuse.

7.2 Key Takeaways

Reuse of malicious source code is significantly more prevalent in C and C++ languages than it is in Python and Java. The code reuse network for C and C++ also shows a seemingly more mature ecosystem with a large degree of interlinking of projects from all sources (GitHub repositories, UF posts, and UF attachments) while the other languages exhibit much less inter-source linking. Future efforts are better served in these two languages.

Source code reuse in Java is more highly concentrated around reuse with StackOverflow posts, suggesting that Java malware developers copy from each other less and are more likely to base the code their reuse off common functionality rather than specialized malware features. This could suggest less sophistication on the part of Java malware developers, owing to the fact that most experienced developers use languages that do not have to interact with intermediary layers (the JVM).

Much of the code on StackOverflow is ambiguous in its degree of malice. Although our methodology is aimed at removing benign source code, much reuse remains flagged with StackOverflow. Under manual inspection, this appears to be because the code that is flagged is ambiguous in its function: it could be used in a benign manner while also possibly being used in a malicious project. For example, many keyloggers exhibit reuse with StackOverflow due to users on this website providing examples on how to hook keyboard input. As such, the removal of malicious code from StackOverflow is likely to be unfeasible.

Our approach makes inroads into scalable malicious code reuse detection. The recent works that make code reuse measurements in malware [12,4,13,9] do not take steps to filter ancillary or benign code and as such do not mitigate at all the risk of finding incidental or benign commonalities between projects, potentially leading to wasted person-hours for the malware analyst should their methodologies be adopted. Additionally, their scalability is limited. When attempting a measurement with a huge collection of data, such as from Stack-Overflow, these are not feasible. Despite the limitations of this work (cf. §7.1, we believe our findings can offer valuable insights to the research community on how to remove goodware from their measurements.

7.3 Conclusion

We have presented a methodology complete with a malice metric that achieves 93.1% precision in identifying malicious GitHub repositories based on their source code. Using this, we have conducted a first-of-its-kind measurement of the malware ecosystem as written in 4 languages, C, C++, Java, and Python.

The key takeaways from this work provide valuable insights into the code reuse patterns among malware developers in C/C++ and Java, shedding light on their preferences for specific programming languages, online platforms, and forums. Understanding these patterns is crucial for the development of effective countermeasures and threat detection strategies within the malware research community.

Acknowledgements

This project was funded by TED2021-132900A-I00, from the Spanish Ministry of Science and Innovation, with funds from MCIN/AEI /10.13039/501100011033, and the European Union-NextGenerationEU/PRTR; and by PID2022-143304OB-I00 funded by MCIN/AEI /10.13039/501100011033/ and the ERDF "A way of making Europe." M. Tereszkowski-Kaminski's work was supported by "Programa Investigo" grant 2022-C23.I01.P03.S0020-0000038, funded by the European Union NextGeneration-EU/PRTR and MITES/SEPE. G. Suarez-Tangil has been appointed as 2019 Ramon y Cajal fellow (RYC-2020-029401-I) funded by MCIN/AEI/10.13039/501100011033 and ESF Investing in your future.

References

1. Baltes, S., Diehl, S.: Usage and attribution of stack overflow code snippets in github projects. Empirical Software Engineering **24**(3), 1259–1295 (2019)
2. Calleja, A., Tapiador, J., Caballero, J.: The malsource dataset: Quantifying complexity and code reuse in malware development. IEEE Transactions on Information Forensics and Security **14**(12), 3175–3190 (2018)
3. Cheng, X., Jiang, L., Zhong, H., Yu, H., Zhao, J.: On the feasibility of detecting cross-platform code clones via identifier similarity. In: Proceedings of the 5th International Workshop on Software Mining, KDD. pp. 39–42 (2016)
4. Islam, R., Rokon, M.O.F., Darki, A., Faloutsos, M.: Hackerscope: The dynamics of a massive hacker online ecosystem. arXiv preprint arXiv:2011.07222 (2020)
5. Moradi-Jamei, B., Kramer, B.L., Calderón, J.B.S., Korkmaz, G.: Community formation and detection on github collaboration networks. In: Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, KDD. pp. 244–251 (2021)

6. Nafi, K.W., Kar, T.S., Roy, B., Roy, C.K., Schneider, K.A.: Clcdsa: cross language code clone detection using syntactical features and api documentation. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1026–1037. IEEE (2019)
7. Nakagawa, T., Higo, Y., Kusumoto, S.: Nil: large-scale detection of large-variance clones. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 830–841 (2021)
8. Pastrana, S., Thomas, D.R., Hutchings, A., Clayton, R.: Crimebb: Enabling cybercrime research on underground forums at scale. In: Proceedings of the 2018 World Wide Web Conference. pp. 1845–1854 (2018)
9. Qian, Y., Zhang, Y., Chawla, N., Ye, Y., Zhang, C.: Malicious repositories detection with adversarial heterogeneous graph contrastive learning. In: Proceedings of the 31st ACM International Conference on Information & Knowledge Management. pp. 1645–1654 (2022)
10. Ragkhitwetsagul, C., Krinke, J., Clark, D.: A comparison of code similarity analysers. Empirical Software Engineering **23**(4), 2464–2519 (2018)
11. Ragkhitwetsagul, C., Krinke, J., Paixao, M., Bianco, G., Oliveto, R.: Toxic code snippets on stack overflow. IEEE Transactions on Software Engineering **47**(3), 560–581 (2019)
12. Rokon, M.O.F., Islam, R., Darki, A., Papalexakis, E.E., Faloutsos, M.: Sourcefinder: Finding malware source-code from publicly available repositories in github. In: 23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020). pp. 149–163 (2020)
13. Rokon, M.O.F., Yan, P., Islam, R., Faloutsos, M.: Repo2vec: A comprehensive embedding approach for determining repository similarity. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 355–365. IEEE (2021)
14. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Queen's School of Computing TR **541**(115), 64–68 (2007)
15. Saini, V., Farmahinifarahani, F., Lu, Y., Baldi, P., Lopes, C.V.: Oreo: Detection of clones in the twilight zone. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 354–365 (2018)
16. Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., Lopes, C.V.: Sourcerercc: Scaling code clone detection to big-code. In: Proceedings of the 38th International Conference on Software Engineering. pp. 1157–1168 (2016)
17. Svajlenko, J., Roy, C.K.: Cloneworks: a fast and flexible large-scale near-miss clone detection tool. In: ICSE (Companion Volume). pp. 177–179 (2017)
18. Thomas, K., Huang, D.Y., Wang, D., Bursztein, E., Grier, C., Holt, T.J., Kruegel, C., McCoy, D., Savage, S., Vigna, G.: Framing Dependencies Introduced by Underground Commoditization. In: Proceedings of the Workshop on the Economics of Information Security (WEIS) (June 2015)
19. Weaver, N., Paxson, V., Staniford, S., Cunningham, R.: Large scale malicious code: A research agenda (2003)
20. Yahya, M.A., Kim, D.K.: Clcd-i: Cross-language clone detection by using deep learning with infercode. Computers **12**(1), 12 (2023)
21. Yang, D., Martins, P., Saini, V., Lopes, C.: Stack overflow in github: any snippets there? In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). pp. 280–290. IEEE (2017)
22. yoeo: Guesslang. `https://github.com/yoeo/guesslang` (2020)

A Benign Datasets

These are used for finding code reuse between them and our corpus samples of that language during the Benign Function Filtering step of our methodology (refer to §4). Thus, we have 3 benign code datasets, one for each language category in the measurement. There is not a separate benign dataset for C on its own as the C and C++ measurements are done on the same samples.

50KC. This is a dataset of compilable Java projects from GitHub. 3,624 of these are included in our benign function filtering step.

Wild C++. This is a dataset of C++ function source code files gathered from GitHub repositories which contain C++ source code, queried for projects with at least 10 stars. 1,000,000 samples are included in our benign function filtering step. This is a fraction of the entire amount available, however we are limited by computational resources in clone detection.

Py150k. This is a dataset of 150k Python source files gathered from GitHub repositories. All 150,000 samples are included in our benign function filtering step.

B Prominent Measurement Clusters

B.1 C/C++ Clusters

We next describe the most prominent similarity matches in C/C++ code reuse:

- **Cluster #1:** There are 1,781 UFSF repositories, 1,161 UF post threads, and 764 UF code attachments in the main supercluster. Within this supercluster, the nature and amount of reuse varies substantially. In places, there are localized subgraphs that have stronger connections to nodes within themselves than to the rest of the supercluster.
- **Cluster #2:** These are 12 hacks for a videogame called DayZ. All 12 of these projects are code attachments from Underground Forums. They reuse a specific piece of malicious code which is a thread callback function.
- **Cluster #3:** These are 11 hacks for a videogame called Grand Theft Auto 5. 10 of these are code attachments from Underground Forums, with the one remaining being a GitHub repository. They reuse a DLL-loading snippet.
- **Cluster #4:** These are 9 Linux malware, 8 of them with Rootkit in the name, and all 9 GitHub repositories. The name of one of them suggests it was a homework assignment for a class. They reuse a piece of code that initializes the rootkit.
- **Cluster #5:** These are 8 threads from an Underground Forum. Their code reuse centers around low-level memory manipulation with function names such as *ModifyMemory()* and *WriteToMemory()*.

- **Cluster #6:** These are 7 ransomware samples, all GitHub repositories, one of them being a collection repository for ransomware. The reuse present are functions that encrypt and decrypt files, as well as getting directory listings of files on the host system.
- **Cluster #7:** These are 6 code attachments from Underground Forums, all 6 being hacks for the videogame Counter Strike: Global Offensive.
- **Cluster #8:** These are 6 GitHub repositories dedicated to "hacking Windows memory".
- **Cluster #9:** These are 6 GitHub repositories which contain rootkits.
- **Cluster #10:** These are 5 code attachments from Underground Forums which are videogame hacks. It appears they are different versions of the same hack, but we are unable to ascertain which videogame they target.

B.2 Java Clusters

We next describe the most prominent similarity matches seen in Java code reuse:

- **Cluster #1:** There are 6 nodes in the network that come from Underground Forum snippets, making up 3.1% of the network. All of them are within the largest cluster. They deal with socket connections and user login functionality. Underground Forums and SourceFinder repositories on the other hand make up 89.8% of the network. These variants vary greatly in the malicious functionality they reuse, with some reusing socket connectivity code, and others network vulnerability scanning code. Still another sample obtained code from StackOverflow which would fake the working of threads using *Thread.sleep()* calls.
- **Cluster #2:** The second largest cluster consists of 19 cryptocurrency miner GitHub repositories. The reuse centers around various functionality, from blockchain protocol implementations encryption implementations.
- **Cluster #3:** The third largest cluster consists of 7 hacks for the video game Call of Duty: Modern Warfare 3, all existing in the network as source code attachments from Underground Forums.
- **Cluster #4-5:** The fourth and fifth largest clusters both contain 4 projects each. One consists of 3 keyloggers and a Remote Administration Tool trojan with the code reused centered around keylogging activity. The other cluster consists of 4 blockchain implementations.
- **Cluster #6:** This cluster consists of 3 forks of the same repository which contains miscellaneous hacking scripts by a hobbyist malware writer.
- **Cluster #7-9:** The remaining clusters are pairs of nodes. We see a pair of hacks for the video game Realm of the Mad God, a pair of hacks for the video game Counter-Strike: Global Offensive, and a pair of GitHub repositories that collect malware samples.

B.3 Python Clusters

We next describe the most prominent similarity matches seen in Python code reuse:

- **Cluster #1:** There are 1,615 UFSF repositories, 6 UF post threads and 12 UF code attachments in the main supercluster. 5 of these 12 are videogame hacks for the game PUBG. In places, there are localized subgraphs that have stronger connections to nodes within themselves than to the rest of the supercluster.
- **Cluster #2:** These are 7 Remote Administration Tool GitHub repositories, 5 of them being versions of one and 2 of their versions of another. There is a big overlap between this cluster and Cluster #10 of C/C++ code. The code reuse includes writing and reading files and preparing to execute shellcode.
- **Cluster #3:** These are 4 versions of one video game hack for the game Minecraft. They reuse code that deals with socket connections, among others.
- **Cluster #4:** These are 4 GitHub repositories containing bitcoin miners.
- **Cluster #5:** These are 3 GitHub repositories that contain spamming programs for the app Instagram. They reuse a lot of code that performs the critical functionality.
- **Cluster #6:** These are 2 GitHub repositories and 1 source code attachment from an Underground Forum. They reuse code that performs UDP flooding. This is particularly interesting as it is not obvious that they are related projects.
- **Cluster #7:** These are 3 GitHub repositories that contain tools for performing DOS attacks.
- **Cluster #8:** These are 3 video game hacks for the game Apex Legends. They reuse code which gathers information from the game screen as well as aids the player in aiming.
- **Cluster #9:** These are 3 GitHub repositories that contain ransomware and reuse code that encrypts files.
- **Cluster #10:** These are 3 GitHub repositories which contain botnets. They reuse code that executes shellcode.