electronics

Article

Function Similarity Using Family Context

Paul Black [1,*], Iqbal Gondal [1], Peter Vamplew [2] and Arun Lakhotia [3]

1 Internet Commerce Security Laboratory (ICSL), Federation University, Ballarat 3353, Australia; iqbal.gondal@federation.edu.au
2 School of Engineering, Information Technology and Physical Science, Federation University, Ballarat 3353, Australia; p.vamplew@federation.edu.au
3 School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, LA 70504, USA; arun@louisiana.edu
* Correspondence: p.black@federation.edu.au

Received: 4 June 2020; Accepted: 13 July 2020; Published: 17 July 2020

Abstract: Finding changed and similar functions between a pair of binaries is an important problem in malware attribution and for the identification of new malware capabilities. This paper presents a new technique called Function Similarity using Family Context (FSFC) for this problem. FSFC trains a Support Vector Machine (SVM) model using pairs of similar functions from two program variants. This method improves upon previous research called Cross Version Contextual Function Similarity (CVCFS) e epresenting a function using features extracted not just from the function itself, but also, from other functions with which it has a caller and callee relationship. We present the results of an initial experiment that shows that the use of additional features from the context of a function significantly decreases the false positive rate, obviating the need for a separate pass for cleaning false positives. The more surprising and unexpected finding is that the SVM model produced by FSFC can abstract function similarity features from one pair of program variants to find similar functions in an unrelated pair of program variants. If validated by a larger study, this new property leads to the possibility of creating generic similar function classifiers that can be packaged and distributed in reverse engineering tools such as IDA Pro and Ghidra.

Keywords: malware similarity; malware evolution; function similarity; binary similarity; Zeus malware; ISFB malware; machine learning

1. Introduction

This paper provides a technique called Function Similarity using Family Context (FSFC). This is a 1-step method using a Support Vector Machine (SVM) model with a low false positive rate for identifying similar pairs of functions from two variants of a program. The primary innovation is to demonstrate that our SVM model abstracts function similarity features in function pairs from two program variants to find similar functions in an unrelated pair of program variants. A second innovation is a new technique that extracts features from each function under consideration as well as specific related functions. These contextual features strengthen the function similarity results and provide a substantial performance improvement over and above training with features taken from individual functions. FSFC is a one-step method for function similarity that performs well without pre-filtering and provides a low false positive rate. FSFC is built on previous research called Cross Version Contextual Function Similarity (CVCFS) [1].

Function similarity techniques are used for malware triage [2], program patch analysis [3], identification of library functions containing known bugs [4,5], malware authorship analysis [6], identification of similar function pairs in detailed malware analysis [7], and for plagiarism analysis [8].

Organisations performing anti-virus or threat intelligence work process large daily feeds of malware samples. Due to packing, these malware samples initially appear to be unique. However, after unpacking malware sample volumes may be reduced by orders of magnitude through the use of malware similarity techniques to reduce the malware feed to a smaller set of unique malware payloads [9]. Function level similarity can then be performed on the unique malware payloads for malware family identification.

Software vendors provide program patches for maintenance and security purposes. High-level details describing the program patches may be provided, but this level of detail may not be sufficient for some organisations. Program patch analysis uses function similarity for the identification of updated functions. This is the first step in a reverse engineering process to identify the vulnerabilities that have been patched.

Internet of Things (IoT) devices, including routers and consumer electronics, utilize open-source software and a wide range of Central Processing Unit (CPU) architectures. When software vulnerabilities are identified in open source libraries, a wide range of IoT devices become vulnerable. Function similarity techniques are used for cross-architecture bug search purposes to search IoT firmware for vulnerable functions.

Law enforcement anti-malware efforts prioritize the identification of malware authors. Function similarity techniques are used to identify code reuse in malware families, which can identify the malware products produced by specific malware authors [10].

Following the identification of new malware variants, detailed analysis is required to determine the new malware capabilities. Function similarity provides the first stage of this analysis and allows the elimination of unmodified functions from the reverse engineering workload [11].

Function similarity methods are also used for the analysis of compiled programs in cases where software plagiarism or breach of licensing conditions is suspected [8].

This paper provides two studies. The first study uses a Zeus malware dataset; the second study uses a dataset from a sample of ISFB banking malware [12]. The distinct and independent codebases of the Zeus and ISFB malware families have both been publicly leaked. The Zeus dataset study highlights the improved machine learning performance. Previous research [1] was re-run using the same testing configuration as the other experiments in this paper. This gave an F1 score of 0.19 for Zeus function similarity using the CVCFS algorithm [1], FSFC gives an average F1 score of 0.44 on the Zeus dataset with the same SVM model.

The ISFB study uses the training data from the Zeus dataset to predict function similarity in a dataset taken from a pair of ISFB variants where the compilation dates differ by approximately one year. In the ISFB study, FSFC gives an average F1 score of 0.34. This study shows that the SVM model abstracts features from the Zeus dataset sufficiently to predict function pairs in an independent malware family.

This paper makes the following contributions:

- An SVM model that can abstract function similarity features from one pair of program variants to identify similar functions in an unrelated pair of program variants.
- A one-step function similarity technique that extracts features from each function under consideration and specific related functions. This contextual similarity technique strengthens the function similarity results and provides a substantial performance improvement over and above training with features taken from individual functions and provides a low false positive rate.
- Improved performance of numeric features through the use of an improved encoding of these features.
- Run time reduction, as due to the higher performance of FSFC, there is no need for an additional step to remove false positives.
- Development of a curated dataset of matched functions in two versions of ISFB malware for use in future research.

- Development of two labelled Interactive Disassembler (IDA) databases for ISFB malware versions 2.04.439, 2.16.861 (The datasets related to this research are available online at http://federation.edu.au/icsl/evolvedsimilarity).

The structure of this paper is as follows: Section 2 presents related work, Section 3 presents the research methodology, Section 4 presents the empirical evaluation of results, and Section 5 presents the conclusion.

2. Related Work

2.1. BinJuice Function Semantics

BinJuice computes abstract semantics for each basic block of all functions in a compiled program, by a process of disassembly, Control Flow Graph (CFG) generation, symbolic execution, algebraic simplification, and the computation of function semantics [13]. The aim of generating abstract semantics is to represent any two equivalent code sequences by the same semantics [14,15].

Existing tools are used to perform Disassembly and CFG extraction. Symbolic execution is used to generate basic block semantics. Symbolic execution does not involve execution on a physical processor; instead, the effects of the program instructions can be represented as a set of simultaneous equations. An algebraic simplifier provides simplification of the symbolic expressions resulting from the symbolic execution. The simplified symbolic expressions are mapped into a canonical form to ensure that sections of equivalent code having equivalent symbolic expressions are sorted in the same order. The juice operation addresses the problem of comparing sections of equivalent code containing different register allocations by renaming register names in the simplified symbolic expressions to consistent logical variables [13].

VirusBattle is built on BinJuice and is used to identify relationships between malware samples in a large database [16]. VirusBattle unpacks the submitted malware sample and creates four different representations of the unpacked code. These representations are disassembled code, generalised code, semantics, and generalised semantics (juice). VirusBattle identifies similar functions by comparing the hashes of the function's generalised semantics. This provides a significant performance gain compared with non-approximate methods, e.g., theorem solvers [15]. VirusBattle has been commercialised by Cythereal Inc and is known as Cythereal MAGIC.

Cythereal function similarity aims to identify similar functions using semantic hashes derived from symbolic execution. Cythereal similarity makes use of the abstraction of each function, and if a pair of functions are similar but have different purposes, then Cythereal will still identify these as function pairs. The design goals of FSFC are to be able to identify function similarity in evolved variants of malware families [17] and so it aims to match function pairs, including those that differ as a result of software development. The use of contextual features allows FSFC to differentiate similar functions with different purposes due to their different position in the call tree. A further point of difference is that due to FSFC's use of machine learning, there is the potential that this research could lead to a general-purpose function similarity engine that can identify similar functions in a broad range of programs.

2.2. Cross-Architecture Bug Search

DiscovRE [4] identifies vulnerable functions across a range of IoT devices. IoT devices often use open-source code and a variety of CPU architectures. Vulnerability discovery in an open-source library may impact a range of IoT devices. The identification of functions containing known vulnerabilities requires the capability to search code generated by different compilers, and a range of CPU architectures. DiscovRE uses k-Nearest Neighbors (kNN) machine learning to pre-filter potentially vulnerable functions. The Pearson product-moment correlation coefficient [18] has been used to select numeric features that are robust across multiple compilers, compiler options, and CPU architectures. The following function counts have been used as features: call instructions, logic instructions, redirection instructions, transfer instructions, local variables, basic blocks, edges, incoming calls, and instructions. The final identification of vulnerable functions is performed using maximum common subgraph isomorphism.

In some scenarios, the DiscovRE pre-screening stage is unreliable [19]. CVSkSA performs cross-architecture searching for vulnerable functions in IoT device firmware using the kNN algorithm, followed by SVM machine learning for pre-filtering [19]. Bipartite graph matching [5] is then used to identify vulnerable functions from the pre-screened matches. Although the accuracy of the SVM model for pre-filtering is good, the run time is slow, but kNN pre-filtering reduces the number of functions to be checked by the SVM model, and this reduces execution time by a factor of four with a small reduction in vulnerability identification performance. CVSkSA uses two levels of features; function level features are used in the pre-filtering while basic block level features are used for graph matching. The function level features are call count, logic instruction count, redirection instruction count, transfer instruction count, local variables size, basic block count, incoming call count, and instruction count.

The above research techniques locate software vulnerabilities in one version of code, compiled for different CPU architectures. These techniques are optimised for cross-architecture vulnerability identification within one version of a program. The research in this paper provides identification of similar functions in different versions of a program compiled for a single CPU architecture.

2.3. CVCFS

This paper builds on an earlier research work called CVCFS [1], that identifies similar function pairs in two variants of the same program. There are three areas where FSFC has improved on CVCFS. The first is an improved definition of function context, the second is a rationalisation of the features, and the third is an improved encoding of numeric features. In CVCFS, a function context was defined as the set of non Application Programming Interface (API) functions that can be reached by walking the call graph starting from the selected function. This definition could lead to a large number of functions in a function context. FSFC restricts function context and limits the call graph depth for feature extraction. This use of smaller function contexts results in substantially higher performance. CVCFS makes use of the following local features from individual functions: the set of API calls, the set of constants, stack size, function callers count, and basic block count. The following contextual features were extracted: set of API calls, set of constants, stack size, return instruction release count, and call instructions count. CVCFS uses local and contextual features that are similar but are not fully aligned. For example, CVCFS local features contain function callers count, and the contextual features contain a call count feature. FCFS removes the distinction between local and contextual features and uses six consistent features, as defined in Section 3.3. In CVCFS, numeric features from each function in the context are summed, while API call names and constants are stored in sets. During the testing of CVCFS, the performance of the numeric features was low. In FCFS, numeric features are stored as a set, and each element in the set consists of a value from the function context. In CVCFS, 20,000 training iterations were used, FCFS research found that better performance resulted from 100,000 training iterations. This paper demonstrates the substantial performance improvements obtained in FCFS by using an improved definition of function context, improved encoding of numeric features, and increased training iterations.

3. Function Similarity Using Family Context

The FSFC research is introduced in the following order, experimental design, function context definition, feature definition, feature extraction, feature ratios, ground truth, and the FSFC algorithm.

3.1. Experimental Design

The FSFC method uses an SVM model to classify similar function pairs from a pair of program variants. Both the training and classification steps use features that are calculated from the pairwise comparison of all of the functions in two program variants. These features are calculated using the feature ratios described in Section 3.5. The Function Similarity Ground Truth (FSGT) table (see Section 4.2) contains the Relative Virtual Addresses (RVAs) of each matching function pair in the training and testing datasets.

The FSGT table is used to label the training data. Due to the pairwise feature generation, many of the training features correspond to unmatched function pairs, and a smaller set of features correspond to matching function pairs. As a result, the training dataset is unbalanced, and this is discussed further in Section 4.4. The training features represent exact and approximate function matches. The SVM model training uses the feature labels to identify the features corresponding to matching function pairs. The trained SVM model contains weights and biases that can be used to classify similar functions. The SVM model classification indicates whether the features from each pairwise function pair correspond to a similar function. The FSGT table is used to identify whether the similarity classification is correct for each function pair. The function pairs classified correctly as similar functions are added to the True Positive (*TP*) count. Similarly, the correctly classified dissimilar function pairs are added to the True Negative (*TN*) count. Function pairs that are incorrectly identified as similar are added to the False Positive (*FP*) count, and function pairs that are incorrectly identified as dissimilar are added to the False Negative (*FN*) count. Precision and recall are calculated from the *TP*, *TN*, *FP*, and *FN* counts. *Precision* and *recal*l [20] are defined in Equations (1) and (2), respectively.

$$Precision = TP/TP + FP \tag{1}$$

$$Recall = TP/TP + FN \tag{2}$$

The F1 score (*F1*) [20] defined in Equation (3) is used to assess the quality of the results in terms of precision and recall.

$$F1 = 2 \times (Precision \times Recall)/(Precision + Recall) \tag{3}$$

3.2. Function Context

In FSFC, function context is defined as: Self (*S*), Child (*C*), Parent (*P*), Grandchild (*GC*), Grandparent (*GP*), Great-grandchild (*GGC*) and Great-grandparent (*GGP*). Each of these function contexts is a set of associated functions.

The Self (*S*) context of function f is the set of the non-API functions that are called from function f.

$$S(f) = \{s_0, s_1, ..., s_\mathrm{i}\} \tag{4}$$

The set of functions in the Child C context of f can be obtained by iterating through each of the functions in the Self context of f and extracting the non-API functions.

$$C(f) = \{s(f') \,\forall\, f' \;in\; S(f)\} \tag{5}$$

The functions in the Grandchild GC context of function f can be obtained by iterating through each of the functions in the Child context of f and extracting the non-API functions.

$$GC(f) = \{s(f') \,\forall\, f' \;in\; C(f)\} \tag{6}$$

The functions in the Great-Grandchild GGC context of f can be obtained by iterating through each of the functions in the Grandchild context of f and extracting the non-API functions.

$$GGC(f) = \{s(f') \,\forall\, f' \;in\; C(f)\} \tag{7}$$

The functions in the Parent P context of f is the set of non-API functions that call function f.

$$P(f) = \{p_0, p_1, ..., p_\mathrm{j}\} \tag{8}$$

The functions in the Grandparent *GP* context of *f* can be obtained by identifying the callers of each of function in the Parent context of *f*.

$$GP(f) = \{P(f') \; \forall \; f' \; in \; P(f)\} \tag{9}$$

The functions in the Great-Grandparent *GGP* context of *f* can be obtained by identifying the callers of each of function in the Grandparent context of *f*.

$$GGP(f) = \{P(f') \; \forall \; f' \; in \; GP(f)\} \tag{10}$$

3.3. Features

FSFC extracts the following features from each function in the context of function *f*:

- API calls,
- Function calls count,
- Return release count,
- Constants,
- Stack size,
- Basic block count.

API calls: The system programming interface for the Windows operating system is provided by the Windows API [21]. The API provides a dynamic function call interface for Windows system services. Windows programs that use the portable executable (PE) format. Let $AC(f,p)$ be the set of API functions called by function *f* in a program *p*.

$$AC(f,p) = \{a_0, a_1, ..., a_k\} \tag{11}$$

Function calls count: Let *FC* be the count of function call instructions *c* within function *f* in program *p*.

$$FC(f,p) = |\{c_0, c_1, ..., c_l\}| \tag{12}$$

Return release count: Let *rc* be the byte release count of any return instruction within function *f*, and let *m* be the count of return instructions within *f*. If *rc* is 0, then *rc* is set to 1. Let *RC*, the return release count, be the sum of the release count *rc* of all return instructions within function *f* in program *p*.

$$RC(f,p) = \sum_{i=0}^{i=m} max(rc_i, 1) \tag{13}$$

Constants: The goal in extracting a set of constants is to extract invariant numerical constants from the operands of instructions in functions. Call and jump instructions were excluded because they have operands that contain program and stack addresses that are not invariant. Let $CS(f,p)$ be the set of constants that are not program or stack addresses within function *f* in program *p*.

$$CS(f,p) = \{c_0, c_1, ..., c_n\} \tag{14}$$

Stack size: Let $SS(f,p)$ be the stack size of function *f* in program *p*.

$$SS(f,p) = s_0 \tag{15}$$

Basic block count: A basic block is defined as the maximal sequence of consecutive instructions that begin execution at the first instruction, and when the block is executed, all instructions in the basic block are executed sequentially without halting or branching, except for the last instruction in the block [22]. Let *BB* be the count of each basic block *b* within function *f* in program *p*.

$$BB(f,p) = |\{b_0, b_1, ..., b_q\}| \tag{16}$$

3.4. Feature Extraction

CVCFS [1] summed numeric features, e.g., stack size, function callers count, function calls count, and basic block count to an individual number. In FSFC, all features are extracted from the context of a function and are stored in sets, with one value from each function in the context. For example, a context of three functions with stack sizes of 12, 0, and 24 bytes, in the earlier research, the stack size feature would be summed to 36 bytes, but in FSFC, the stack size feature would be {12, 0, 24}. Jaccard indexes provide an efficient measure of set similarity and have been used extensively in previous similarity research [2,23]. Jaccard indexes are calculated from the contextual features for each function in the pair being compared. This calculation of Jaccard indexes for all features has simplified FSFC implementation.

3.5. Feature Ratios

Feature ratios use the Cartesian product of all functions in program $p1$ and all functions in program $p2$. It is noted that function similarity is commutative, and the same function pairs will be identified by comparing programs $p1$ and $p2$ as would be identified by comparing program $p2$ and $p1$. The following features definitions provide names for the features that are used in subsequent algorithms. Let $F(p)$ be the set of all functions in program p.

$$F(p) = \{f_0, f_1, ..., f_1\} \tag{17}$$

The set of function pairs $FP(p1, p2)$ of programs *p1* and *p2* are defined as follows:

$$FP(p1, p2) = F(p1) \times F(p2) \tag{18}$$

Each element of the Cartesian product *FP* is a function pair *fp* consisting of one function $f1$ from program $p1$ and one function $f2$ from program $p2$.

$$fp = (f1, f2) \tag{19}$$

API Ratio: Let AC_1 and AC_2 be the sets of API calls extracted from the context of each function in function pair *fp*. Let the API ratio *ACR* be the Jaccard index of AC_1 and AC_2.

Function Calls Ratio: Let FC_1 and FC_2 be the sets of the call instructions counts extracted from the context of each function in function pair *fp*. Let the function calls ratio *FCR* be the Jaccard index of FC_1 and FC_2.

Return Release Count Ratio: Let RC_1 and RC_2 be the sets of return release counts extracted from the context of each function in function pair *fp*. Let the return release count ratio *RCR* be the Jaccard index of RC_1 and RC_2.

Constants Ratio: Let CS_1 and CS_2 be the sets of constants extracted from each of the context of each function in the function pair *fp*. Let the constants ratio *CSR* be the Jaccard Index of CS_1 and CS_2.

Stack Ratio: Let SS_1 and SS_2 be the sets of stack sizes extracted from the context of each function in the function pair *fp*. Let the stack ratio *SSR* be the Jaccard index of SS_1 and SS_2.

Blocks Ratio: Let BB_1 and BB_2 be the sets of basic block counts extracted from the context of each of the functions in the function pair *fp*. Let the blocks ratio *BBR* be the Jaccard index of BB_1 and BB_2.

3.6. Ground Truth

As a supervised machine learning method, FSFC requires labelled data both for training the SVM model, and for the evaluation of the system performance. To support this, the FSGT table was constructed via manual reverse engineering of the malware samples against leaked malware source code. The FSGT table contains data defining the pairing of the functions of the programs being compared. The FSGT table contains the RVAs of the function pairs.

3.7. FSFC Algorithm

The FSFC algorithm only operates on static non-API function calls. Figure 1 illustrates the relationships between the functions being processed. The function currently being processed is referred to as the current function. The self context contains those functions called by the current function. The child functions are those functions called from the self context functions. The child context consists of the function calls made from each of the child functions. The callers of the current function are referred to as parent functions. The parent context consists of the callers of each of the parent functions. The other relationships and contexts are identified in a similar manner and are defined in Section 3.2.

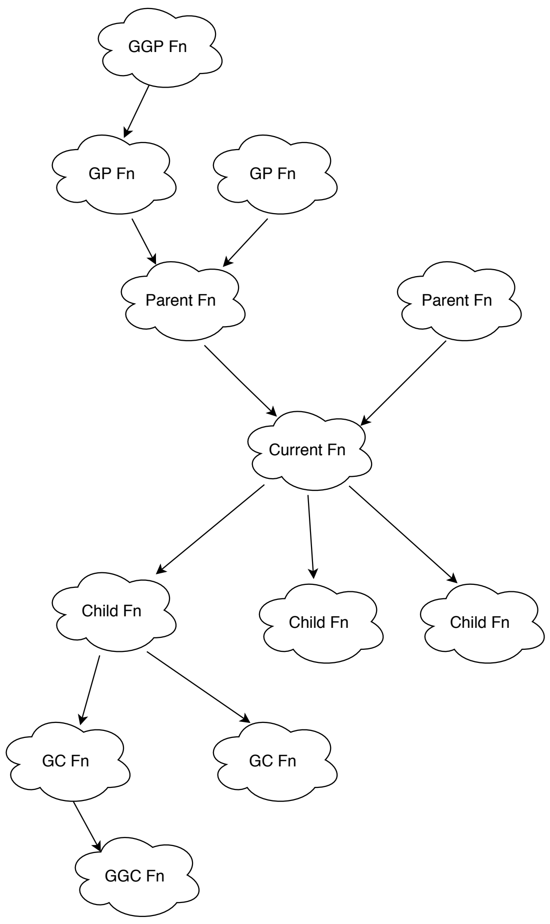

Figure 1. Function Call Relationships.

The function context extraction algorithm is shown in Figure 2. For each function *f* in the program being analysed, the self context is the set of static non-API function calls made by *f*. The child context is the set of all static non-API function calls made by each function in the self context. The grandchild and great-grandchild contexts are calculated in a similar manner. The parent contexts are built in a similar manner by walking up the call tree. The parent context $p1$ is the set of all static non-API function calls made to each function in the self context. The grandparent and great-grandparent contexts are calculated from the parent and grandparent contexts in a similar manner.

```
For each function f in program
  Build function callers map
For each function f in program
  S context c0 = static calls in f
  For each function f1 in self context c0
      C context c1 += static calls in f1
  For each f2 in child context c1
     GC context c2 += static calls in f2
  For each f3 in grandchild context c3
      GGC context c3 += static calls in f3
  From function callers map
      Build parent context P from f
  From p1
      Build grandparent context GP from P
  From p2
      Build great-grandparent context GGP from~GP
```

Figure 2. Function Context Extraction Algorithm.

The Cythereal output includes unpacked malware, disassembled code, generalized code, semantics, and generalized semantics. The FSFC algorithm reads the disassembly of the unpacked malware, extracts the functions from the disassembly, builds a map of function callers, and then iterates through each function in the program and builds the function context sets.

The algorithm for feature extraction is shown in Figure 3. This algorithm reads the Cythereal disassembly of each of the programs being compared. Next, it extracts the function contexts for each function in programs p1 and p2, and then calculates the features, AC, FC, RC, CS, SS, BB for the context of each function. For each function in program 1, and for each function in program 2, calculate the Jaccard Index of the features to produce the feature ratios ACR, FCR, RCR, CSR, SSR, BBR , then label each function combination using the FSGT table.

```
p1 = baseline version of program
p2 = updated version of~program

  for each function in p1
    extract function context for~function

  for each function in p2
    extract function context for~function

  for each function in p1
    calculate features AC1, FC1, RC1, CS1, SS1, BB1 for function~context

  for each function in p2
    calculate features AC2, FC2, RC2, CS2, SS2, BB2 for function~context

  for each f1 in p1
    for each f2 in p2
      #calculate feature ratios
      ACR = jaccard(AC1, AC2)
      FCR = jaccard(FC1, FC2)
      RCR = jaccard(RC1, RC2)
      CSR = jaccard(CS1, CS2)
      SSR = jaccard(SS1, SS2)
      BBR = jaccard(BB1, BB2)
      if f1, f2 in FSGT table
        set label = ‘‘1’’
      else
        set label = ‘‘0’’

  rva1 = rva(f1)
  rva2 = rva(f2)
  features = map(rva1, rva2, ACR, FCR, RCR, CSR, SSR, BBR)
```

Figure 3. Feature Extraction Algorithm.

4. Empirical Evaluation

The FSFC research is presented as two case studies. The first case study uses the Zeus dataset from the previous CVCFS research and provides a measure of the improvement in FSFC performance. The second case study uses the Zeus SVM model to predict similarity in a pair of ISFB malware variants. In the first experiment, the optimal number of training iterations is determined; the second experiment determines the context set that provides the best performance. The third experiment tests the performance of individual features. The fourth experiment evaluates all feature combinations to identify the highest performing feature combinations. A fifth experiment is performed using the CVCFS algorithm with the same iteration count as the FSFC research. This provides a comparison of the performance of the CVCFS and FSFC algorithms. A final experiment is performed to compare the performance of the numeric feature encoding method from CVCFS with the new numeric feature encoding in FSFC. These tests are run for the Zeus dataset and the ISFB dataset. This section concludes with a discussion of future research.

4.1. Datasets

Two malware datasets are used to evaluate FSFC; the first dataset is a Zeus dataset, and the second dataset is an ISFB dataset. Training features were extracted from Zeus samples 1 and 2. Function similarity prediction was performed on the features from Zeus samples 2 and 3 to show that the SVM model can abstract function similarity features from one pair of program variants to identify similar functions from later versions of the same malware family. Testing features were then extracted from ISFB Samples 4 and 5, and function similarity prediction was performed using the training features from the Zeus samples 1 and 2. This test was performed to show the ability of the SVM model to identify similar functions in a pair of malware variants from a different malware family. Table 1 gives the Secure Hash Algorithm 1 (SHA1) hashes identifying the malware samples used in this paper, and Table 2 shows the malware version numbers associated with these malware samples.

Cythereal provided the unpacked Zeus samples, and Malpedia provided the unpacked ISFB samples [24]. These samples were disassembled for manual analysis using IDA. The linker date in the Portable Executable (PE) header was used to indicate the time the malware samples were created. Although the linker date can be modified by the malware author, there are no inconsistencies that suggest that the linker date has been modified. The linker dates associated with the malware samples used in this paper are shown in Table 3. The linker dates provide the following development timeline: sample 2 was compiled one month after sample 1, sample 3 was compiled six months after sample 2, and sample 5 was compiled one year after sample 4.

Table 1. Malware Sample Details.

Sample SHA1 Hash	**Version**
8a7faa25f23a0e72a760075f08d22a91d2c85f57	Zeus 2.0.8.7
706bf4dcf01b8eceedf6d05cf5b55a27e4ff8ef0	Zeus 2.0.8.9
30c6bb2328299a252436d2a3190f06a6f04f7e3f	Zeus 2.1.0.1
ee0d9a91f52de9b74b8de36e25c2de91d008cee5	ISFB 2.04.439
3ec5a5fe4c8e6b884b2fb9f11f9995fdaa608218	ISFB 2.16.861

Table 2. Malware Sample Versions.

Sample Identifier	**Version**
Zeus Sample 1	Zeus 2.0.8.7
Zeus Sample 2	Zeus 2.0.8.9
Zeus Sample 3	Zeus 2.1.0.1
ISFB Sample 4	ISFB 2.04.439
ISFB Sample 5	ISFB 2.16.861

Table 3. Malware Linker Dates.

Sample Identifier	Linker Date
Zeus Sample 1	14 September 2010
Zeus Sample 2	15 October 2010
Zeus Sample 3	24 March 2011
ISFB Sample 4	17 September 2015
ISFB Sample 5	14 September 2016

4.2. Ground Truth

The FSGT table identifies the function pairs, function names, and the function RVA for each malware sample used in this research. Manual analysis of the unpacked malware samples, using the leaked Zeus [25] and ISFB [26] source code was performed to identify the function pairs for the FSGT table. Function identification was performed based on the API calls, constants, and CFG structure. Function RVAs are used to identify functions in the Cythereal disassembly and in the research code. A summary of the function name labelling is shown in Table 4. Cythereal identified 1035 and 1133 functions in the ISFB samples, while manual analysis identified 1007 and 1098 functions in the ISFB samples. This difference arises from Cythereal's identification of code fragments that perform a jump to a system call as functions. In this research, these code fragments are regarded as compiler artifacts and are excluded from the analysis. A total of 998 function pairs were identified that were common to the two ISFB samples. Of these 998 functions, 985 function pairs were present in both the Cythereal disassembly and in the manual function pair identification.

Table 4. Function Counts and Manual Match Count.

Tool	Version	Count	Version	Count
IDA	Zeus 2.0.8.7	577	Zeus 2.0.8.9	553
Cythereal	Zeus 2.0.8.7	577	Zeus 2.0.8.9	539
IDA	Zeus 2.0.8.9	553	Zeus 2.1.0.1	601
Cythereal	Zeus 2.0.8.9	539	Zeus 2.1.0.1	601
IDA	ISFB 2.04.439	1007	ISFB 2.16.861	1098
Cythereal	ISFB 2.04.439	1035	ISFB 2.16.861	1133

4.3. Features

FSFC uses the following features:

- Set of API calls
- Set of constants
- Stack size
- Call instruction count
- Return release count
- Basic block count

The above features are extracted from the context of each function. Constants are extracted from mov, push, add, cmp, and sub instructions in each function and were stored in the set of constants feature. Ad-hoc analysis showed that these instructions contained a significant proportion of invariant operands. Program and stack addresses were further filtered by excluding values greater than the program base address. As this research is based on static analysis, the non-API function call counts used in this research are a count of static function calls. Stack size is taken from the function prologue when it is present; otherwise, it is zero. The stack size SS is taken from the sub instruction in the function prologue shown in Figure 4. It is noted that some compilers may not use the same idiom for their function prologue.

```
push(ebp)
mov(ebp,esp)
sub(esp,SS)
```

Figure 4. Function Prologue.

4.4. SMOTE Oversampling

The function similarity training dataset in FSFC is imbalanced due to the use of the Cartesian product comparison, which results in an unstable performance of the SVM model. Assume that the two versions of the same program are being compared, and each program contains 500 functions. The maximum number of matching function pairs is 500. The number of function pairs generated by the Cartesian product is 250,000, and the minimum number of non-matching function pairs is 249,500. The use of a Cartesian product in the generation of training features inherently leads to an imbalanced dataset. The SVM model's performance was improved using the Synthetic Minority Oversampling Technique (SMOTE) [27] to rebalance the training dataset.

4.5. SVM Model Training

Zeus samples 1 and 2 are used for SVM model training, as these two Zeus samples are similar but exhibit minor differences. Function similarity ratios are calculated for the Cartesian product of all functions in the training samples. These features are labelled as matching or not matching using the FSGT table. The SVM model is used to predict function similarity between Zeus samples 2 and 3, and to predict function similarity between ISFB samples 4 and 5.

Tests are performed for the Zeus and ISFB function similarity datasets to identify the following machine learning parameters:

- Iteration count,
- Best context combination,
- Individual feature performance,
- Best feature combination.

The features in each of these tests were assigned a binary identifier, e.g., the first feature is identified as 000001, and the second feature was identified as 000010. The use of binary identifiers allowed the numbering of individual tests, and these feature identifiers are shown in Table 5.

Table 5. Numbering of Feature Combination Tests.

Feat #	Vector	Description
1	000001	API Ratio
2	000010	Calls Ratio
4	000100	Return Release Ratio
8	001000	Constants Ratio
16	010000	Stack Ratio
32	100000	Blocks Ratio

4.6. Context Sets

A function's context is the set of functions associated with a specific function. This paper tests the question of whether it is possible to improve performance by creating sets composed of the sum of different contexts. The context sets used in this research consist of the sum of the functions from the context sets shown in Table 6.

Table 6. Naming of Context Sets.

Name	Components
First Context Set	Self
Second Context Set	Self, Child, Parent
Third Context Set	Self, Child, Parent, Grandchild, Grandparent
Fourth Context Set	Self, Child, Parent, Grandchild, Grandparent, Great-Grandchild, Great-Grandparent

4.7. Statistical Significance

Due to the stochastic nature of machine learning, the function similarity prediction results vary. To address this variation, each test in this paper is repeated 20 times, unless stated otherwise, and an average F1 score is calculated. There are cases where we need to determine whether a specific test configuration provides better results than another. To answer this question, we need to determine whether the test results are normally distributed. The three sets of Zeus function similarity results from Table 7 were tested to see if they followed a normal distribution. A Shapiro-Wilk test [28] was performed, and a *p*-value of 0.0173 indicated that we must reject the null -hypothesis and conclude that the results were not normally distributed. As a result, a two-tailed Wilcoxon signed-rank test [29] was used to determine whether selected test results differ at a 95% level of confidence.

4.8. Zeus Dataset Tests

The functions in Zeus samples 1 and 3 exhibit more differences due to software development than the training dataset. A testing set of function similarity features were created from the Cartesian product of all functions in samples 1 and 3. The SVM model was used to predict the matching function pairs from the testing feature set. The results of this prediction were evaluated using the FSGT table. Due to the stochastic nature of machine learning, the results from individual tests vary. The F1 score is used to assess the accuracy of the results.

The function similarity tests shown in Table 7 show the effect of varying the training iteration count. The column titled “S.D” indicates whether the current row was significantly different from the preceding row. The “*p*-value” column gives the *p*-value from the two-tailed Wilcoxon signed-rank test for this comparison. These tests show that performance increased with increasing training iterations and reached a maximum at 100,000 training iterations, which gave an average F1 score of 0.44. The two-tailed Wilcoxon signed-rank test indicated that the F1 score with 100,000 training iterations was not significantly different from the test using 50,0000 iterations.

Table 7. Zeus Dataset—Training Iteration Performance.

Iterations	Average F1	S.D	*p*-Value
50,000	0.31	-	-
100,000	0.44	N	0.05238
150,000	0.44	N	0.72634

The function similarity tests are shown in Table 8 and test the effect of varying combinations of function context. The tests used 100,000 training iterations and different combinations of function contexts. These results show that the best results were obtained using the second context set, giving an average F1 score of 0.44. The F1 score of the second context set test was significantly different from the first context set test.

Table 8. Zeus Dataset—Training Context Combinations.

Context Set	Average F1	S.D.	p-Value
First	0.14	-	-
Second	0.44	Y	0.00008
Third	0.29	Y	0.0091
Fourth	0.26	N	0.5619

The performance of individual features was assessed by performing function similarity classification using SVM models trained for each feature. These tests used 100,000 training iterations and the second context set. The results of this evaluation are shown in Table 9.

Table 9. Zeus Individual Feature Performance.

Test	Feature	Average F1
1	API Ratio	0.23
2	Calls Ratio	0.14
4	Return Release Ratio	0.08
8	Constants Ratio	0.10
16	Stack Ratio	0.03
32	Blocks Ratio	0.21

The best performing feature combinations are shown in Table 10. The function pair prediction results in this research vary from run to run due to the stochastic nature of machine learning. Owing to the time taken to run the SVM model for all 64 feature combinations of both datasets, the assessment of all feature combinations was only run five times.

Table 10. Zeus Dataset—Highest Performing Feature Combinations.

Test	Vector	Average F1
15	001111	0.47
37	100101	0.43
41	000111	0.40
44	101001	0.43
46	101110	0.43
47	101111	0.43
51	110011	0.44
53	110101	0.44
54	110110	0.40
63	111111	0.45

4.9. ISFB Dataset Tests

The second set of tests performed in this paper determines whether ISFB function similarity can be predicted from the Zeus SVM model. The functions in ISFB samples 4 and 5 exhibit differences due to approximately one year of software development. The SVM model was used to predict the matching function pairs in the ISFB data set. The function pair matches in the FSGT table were used to evaluate the prediction results.

The function similarity tests are shown in Table 11 and test the effect of varying the training iteration count. The tests were repeated 20 times using the second context set. The F1 score of 0.34 from 100,000 training iterations was significantly different (S.D.) from the F1 score of 0.22 for 50,000 training iterations and was not significantly different from the F1 score of 0.43 for 150,000 training iterations.

Table 11. ISFB Dataset—Training Iteration Performance.

Iterations	Average F1	S.D.	*p*-Value
50,000	0.22	-	-
100,000	0.34	Y	0.0455
150,000	0.43	N	0.0703

The function similarity tests shown in Table 12 show the effect of varying combinations of function context. The tests were repeated 20 times using 100,000 training iterations and different combinations of function contexts. The second context set gives the best performance for both the Zeus and ISFB datasets.

Table 12. ISFB Dataset—Training Context Combinations.

Context Set	Average F1	S.D.	*p*-Value
First	0.11	-	-
Second	0.38	Y	0.00008
Third	0.26	Y	0.0096
Fourth	0.16	Y	0.02642

The performance of each feature was assessed by performing function similarity classification using SVM models trained for each feature. The function pair prediction was run with 100,000 iterations and the second context set. The results of this evaluation are shown in Table 13.

Table 13. ISFB Individual Feature Performance.

Test	Feature	Average F1
1	API Ratio	0.10
2	Calls Ratio	0.12
4	Return Release Ratio	0.11
8	Constants Ratio	0.14
16	Stack Ratio	0.01
32	Blocks Ratio	0.20

The feature combinations that provided the best performance are shown in Table 14. The function pair prediction was run with 100,000 iterations and the second context set. Owing to the time taken to run the SVM model for all 64 feature combinations of both datasets, the assessment of all feature combinations was only run five times.

Table 14. ISFB Dataset—Highest Performing Feature Combinations.

Test	Vector	Average F1
7	000111	0.46
44	101100	0.49
46	101110	0.45
47	101111	0.44
54	110110	0.41
58	111010	0.40
59	111011	0.41
60	111100	0.41
61	111101	0.42
62	111110	0.43
63	111111	0.45

4.10. FSFC Evaluation

The F1 score for function similarity performance for the Zeus dataset shows an average value of 0.44, and for the ISFB dataset, an average value of 0.34. For comparison, a random classifier would result in a low F1 score due to the unbalanced distribution of the classes within these datasets. If we consider Zeus dataset 2, a random classifier would classify half of the 539 matching function pairs as matching and half as non-matching, and would similarly split the 323400 non-matching function pairs between these two classes. This would give a precision score of 269.5/(269.5 + 161700) = 0.0017, and a recall of 269.5/539 = 0.5. Combining these gives an F1 score of 0.0034. For the ISFB dataset with 1035 matching function pairs, a random classifier would give a precision score of 0.0009 and a recall score of 0.5, giving an F1 score of 0.0018. Thus, the F1 scores obtained by FSFC are well above those that would be expected using random classification.

The FSFC F1 scores provide a 57–65 percent improvement over the CVCFS algorithm. Table 8 shows the Zeus experiment with the second context set, resulting in an average of 480 true positive matches and 1384 false positive matches. Table 12 shows the ISFC experiment with the second context set, resulting in an average of 943 true positive matches and 4075 false positive matches. In the manual generation of the FSGT table, the time required to identify the function pairs in these two datasets was approximately two days for the Zeus dataset and four days for the ISFB dataset. The use of this research to generate a list of candidate similar function pairs is potentially of substantial benefit to a malware analyst.

Next, we examine the execution time performance of FSFC and show that the execution time is proportional to the product of the number of functions in the two malware variants. FSFC research made use of representative historical malware samples. Therefore, the timings reported are indicative of computational costs likely to occur with further malware analysis.

The experiments in this research were conducted on a workstation using an Intel i7-3770 3.40 GHz CPU with 32 GB of RAM. The SVM machine learning software was TensorFlow v1.15. Training features were generated using the Cartesian product of all functions in both training programs. The function counts in Table 4 from the Cythereal disassembly were used to calculate the total function pairs. All features were included in the training, 100,000 training iterations, and the second context set were used to obtain the timing details shown in Table 15. The time for SMOTE oversampling was 59 s. A summary of the feature extraction and machine learning times for the FSFC experiments is shown in Table 15. The column headings provide the following information: “Op” shows whether training or classification is being performed, “Dataset” shows the dataset, “Fn Pairs” shows the function pair count, “Feat Extr (s)” shows the feature extraction time in seconds, “Extr/Pair (μs)” shows the feature extraction time per function pair in microseconds, “SVM (s)” shows the total SVM execution time in seconds, and “SVM/Pair (μs)” shows the total SVM execution time per function pair in microseconds. The FSFC Operation times in Table 15 show that the FSFC training time corresponds to 91% of the execution time for the Zeus Dataset 2 classification and 77% of the execution time for ISFB Dataset 1 classification. The training and classification use features created by the pairwise combination of functions in the two malware variants. This use of the pairwise combination of functions results in execution times that are proportional to the product of the number of functions in the two malware variants.

Table 15. Operation Times.

Op	Dataset	Fn Pairs	Feat Extr (s)	Extr/Pair (μs)	SVM (s)	SVM/Pair (μs)
Train	Zeus Dataset 1	311,003	13	42	1330	4276
Classify	Zeus Dataset 2	323,939	14	43	112	346
Classify	ISFB Dataset 1	1,172,655	37	32	357	304

4.11. Comparison With Previous Research

The Zeus and ISFB similarity experiments were re-run using the algorithm from the CVCFS research [1]. These tests used 100,000 training iterations, all features, and were repeated 20 times. The results of these tests are shown in Table 16. These test results are compared with the Zeus iterations test from Table 11 that gave an average F1 score of 0.44, and the ISFB Iterations test from Table 11 that gave an average F1 score value of 0.34. The difference between these results is statistically significant and shows that the techniques used in this paper provide a significant improvement over the CVCFS research.

Table 16. Zeus and ISFB Similarity Using Previous Research.

Dataset	Average F1	S.D.	*p*-Value
Zeus	0.19	Y	0.00038
ISFB	0.12	Y	0.00030

4.12. Numeric Feature Encoding

This paper encodes numeric features as a set of values taken from each function in the context; this allows the use of a Jaccard Index for the comparison of numeric features. For example, consider a context containing three functions with 5, 3, and 6 basic blocks. The updated numeric feature encoding encodes the blocks feature as the set 5, 3, 6, while the CVCFS method represented the blocks feature as the sum of the basic block counts, which is 14. Individual feature performance using an earlier summed numeric encoding method is shown for the Zeus dataset in Table 17, and for the ISFB dataset in Table 18. These tests used 100,000 training iterations and the second training context; 20 tests were run for each dataset. These results were compared with the individual feature performance using the new numerical feature encoding in Table 9. This comparison shows that excepting the Stack Ratio feature, the new numerical feature encoding performs significantly better than the CVCFS method. In the case of the Stack Ratio feature, the Wilcoxon test was unable to calculate an accurate *p*-value.

Table 17. Zeus Previous Numeric Feature Encoding

Feature	Average F1	S.D.	*p*-Value
Calls Ratio	0.03	Y	0.00008
Return Release Ratio	0.02	Y	0.00008
Stack Ratio	0.02	-	-
Blocks Ratio	0.02	Y	0.00008

Table 18. ISFB Previous Numeric Feature Encoding.

Feature	Average F1	S.D.	*p*-Value
Calls Ratio	0.01	Y	0.00014
Return Release Ratio	0.01	Y	0.00008
Stack Ratio	0.01	-	-
Blocks Ratio	0.01	Y	0.00008

The combined feature performance using the summed numeric encoding method for the Zeus and ISFB datasets is shown in Table 19. This test used 100,000 training iterations and the second training context; all features enabled, 20 tests were run for each dataset. The "Delta F1" column shows the difference from the corresponding FSFC F1 scores. A comparison of these results with the results using the improved numeric feature encoding in Tables 7 and 11 shows that the new numeric feature encoding method substantially improves the identification of similar function pairs.

Table 19. Previous Numeric Feature Encoding.

Malware	Average F1	Delta F1	S.D.	p-Value
ISFB	0.22	−0.11	Y	0.00906
Zeus	0.27	−0.17	Y	0.01046

4.13. Future Work

Work presented in this paper can be extended as follows:

- Investigate the limits of the generality of this research,
- Further improvement of features and feature encoding,
- Test this research on datasets exhibiting a higher degree of software evolution.

5. Conclusions

The ability to match compiled functions with similar code is important for malware triage, program patch analysis, identification of library functions containing known bugs, malware authorship analysis, identification of similar function pairs in detailed malware analysis, and for plagiarism analysis. This paper uses function similarity features from one pair of program variants (Zeus malware) to find function pairs in another unrelated program (ISFB malware). SVM models are trained on contextual features that are extracted not just from the function itself, but also, from a limited set of other functions with which it has a caller and callee relationship. These new contextual features improve the accuracy in detecting function pairs and substantially reduce the false positive rate, removing the need for an additional pass to remove false negatives. The improved numerical feature representation in FSFC results in an improvement in function similarity accuracy, and allows the use of a Jaccard Index for feature ratio comparison, and simplifies the FSFC algorithm.

A major finding of this research is that the SVM model produced by FSFC can abstract function similarity features from one pair of program variants to find function pairs in another unrelated program. This new property leads to the possibility of creating generic similar function classifiers. The Zeus training dataset is relatively small, consisting of approximately 550 function pairs. However, the SVM model trained on features from this dataset was able to predict similarity in the ISFB malware family, with an average F1 score of 0.34 for the ISFB function pair identification. The same training iteration count and context set gave the best results for both the Zeus and ISFB datasets. This result indicates that the FSFC method can abstract function similarity features from Zeus malware and successfully detect similar function pairs in a pair of ISFB malware variants. Future work will examine the ability of this approach to generalise across multiple independent programs.

Author Contributions: P.B. is the main author of this paper. P.B. contributed to the development of the ideas, design of the study, theory, analysis, and writing. P.B. designed and then performed the experiments I.G., P.V., and A.L. reviewed and evaluated the experimental design and the paper. All authors read and approved the final manuscript.

Funding: This research was performed in the Internet Commerce Security Lab (ICSL), which is a joint venture with research partners Westpac, IBM, and Federation University Australia.

Acknowledgments: The authors would like to thank Cythereal (Cythereal has licensed VirusBattle from the University of Louisiana at Lafayette) for providing access to Cythereal MAGIC [30] and to the Zeus malware dataset used in this research. Paul Black is supported by an Australian Government Research Training Program (RTP) Fee-Offset Scholarship through Federation University Australia.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Black, P.; Gondal, I.; Vamplew, P.; Lakhotia, A. *Identifying Cross-Version Function Similarity Using Contextual Features*; Technical Report; Federation University, Internet Commerce Security Lab.: Ballarat, Australia, 2020. Available online: https://federation.edu.au/icsl/tech-reports/icsl_techrep_2020_01.pdf (accessed on 15 July 2020).
2. Jang, J.; Brumley, D.; Venkataraman, S. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In Proceedings of the 18th ACM Conference on Computer And Communications Security, Chicago, IL, USA, 21 October 2011; pp. 309–320.
3. Flake, H. Structural comparison of executable objects. In Proceedings of the 2004 Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2004), Dortmund, Germany, 6–7 July 2004.
4. Eschweiler, S.; Yakdan, K.; Gerhards-Padilla, E. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In Proceedings of the 2016 Network and Distributed System Security (NDSS 2016), San Diego, CA, USA, 21–24 February 2016.
5. Feng, Q.; Zhou, R.; Xu, C.; Cheng, Y.; Testa, B.; Yin, H. Scalable graph-based bug search for firmware images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 480–491.
6. Alrabaee, S.; Debbabi, M.; Wang, L. On the feasibility of binary authorship characterization. *Digit. Investig.* **2019**, *28*, S3–S11. [CrossRef]
7. LeDoux, C.; Lakhotia, A.; Miles, C.; Notani, V.; Pfeffer, A. Functracker: Discovering shared code to aid malware forensics. In Proceedings of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats, Washington, DC, USA, 12 August 2013.
8. Luo, L.; Ming, J.; Wu, D.; Liu, P.; Zhu, S. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16-22 November 2014; pp. 389–400.
9. Osorio, F.C.C.; Qiu, H.; Arrott, A. Segmented sandboxing-A novel approach to Malware polymorphism detection. In Proceedings of the 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), Fajardo, Puerto Rico, 20–22 October 2015; pp. 59–68.
10. Hong, J.; Park, S.; Kim, S.W.; Kim, D.; Kim, W. Classifying malwares for identification of author groups. *Concurr. Comput. Pract. Exp.* **2018**, *30*, e4197. [CrossRef]
11. Jilcott, S. Scalable malware forensics using phylogenetic analysis. In Proceedings of the 2015 IEEE International Symposium on Technologies for Homeland Security (HST), Waltham, MA, USA, 14–16 April 2015; pp. 1–6.
12. Black, P.; Gondal, I.; Layton, R. A survey of similarities in banking malware behaviours. *Comput. Secur.* **2018**, *77*, 756–772. [CrossRef]
13. Lakhotia, A.; Preda, M.D.; Giacobazzi, R. Fast location of similar code fragments using semantic 'juice'. In Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, Rome, Italy, 26 January 2013; p. 5.
14. Ng, B.H.; Prakash, A. Expose: Discovering potential binary code re-use. In Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference (COMPSAC), Kyoto, Japan, 22–26 July 2013; pp. 492–501.
15. Lakhotia, A.; Black, P. Mining malware secrets. In Proceedings of the 2017 12th International Conference on Malicious and Unwanted Software (MALWARE), Fajardo, Puerto Rico, 11–14 October 2017; pp. 11–18.
16. Miles, C.; Lakhotia, A.; LeDoux, C.; Newsom, A.; Notani, V. VirusBattle: State-of-the-art malware analysis for better cyber threat intelligence. In Proceedings of the 2014 7th International Symposium on Resilient Control Systems (ISRCS), Denver, CO, USA, 19–21 August 2014; pp. 1–6.
17. Black, P.; Gondal, I.; Vamplew, P.; Lakhotia, A. Evolved Similarity Techniques in Malware Analysis. In Proceedings of the 2019 18th IEEE International Conference On Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), Rotorua, New Zealand, 5–8 August 2019; pp. 404–410.
18. Puth, M.T.; Neuhäuser, M.; Ruxton, G.D. Effective use of Pearson's product–moment correlation coefficient. *Anim. Behav.* **2014**, *93*, 183–189. [CrossRef]

19. Zhao, D.; Lin, H.; Ran, L.; Han, M.; Tian, J.; Lu, L.; Xiong, S.; Xiang, J. CVSkSA: Cross-architecture vulnerability search in firmware based on kNN-SVM and attributed control flow graph. *Softw. Qual. J.* **2019**, *27*, 1045–1068. [CrossRef]
20. Lipton, Z.C.; Elkan, C.; Naryanaswamy, B. Optimal thresholding of classifiers to maximize F1 measure. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 225–239.
21. Russinovich, M.E.; Solomon, D.A.; Ionescu, A. *Windows Internals*; Pearson Education: New York, NY, USA, 2012.
22. Lam, M.; Sethi, R.; Ullman, J.; Aho, A. *Compilers: Principles, Techniques, and Tools*; Pearson Education: New York, NY, USA 2006.
23. Bayer, U.; Comparetti, P.M.; Hlauschek, C.; Kruegel, C.; Kirda, E. Scalable, behavior-based malware clustering. In Proceedings of the NDSS 2009, San Diego, CA, USA, 8–11 February 2009; Internet Society: Reston, VA, USA, 2009; Volume 9, pp. 8–11.
24. Plohmann, D.; Clauss, M.; Enders, S.; Padilla, E. Malpedia: A Collaborative Effort to Inventorize the Malware Landscape. *J. Cybercrime Digit. Investig.* **2018**, *3*, 1–19
25. Zeus Author. Zeus Source Code. 2011. Available online: https://github.com/Visgean/Zeus (accessed on 15 July 2020).
26. ISFB Author. ISFB Source Code. 2010. Available online: https://github.com/t3rabyt3/Gozi (accessed on 15 July 2020).
27. Chawla, N.V.; Bowyer, K.W.; Hall, L.O.; Kegelmeyer, W.P. SMOTE: Synthetic minority over-sampling technique. *J. Artif. Intell. Res.* **2002**, *16*, 321–357. [CrossRef]
28. Shapiro, S.S.; Wilk, M.B. An analysis of variance test for normality (complete samples). *Biometrika* **1965**, *52*, 591–611. [CrossRef]
29. Wilcoxon, F.; Katti, S.; Wilcox, R.A. Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test. *Sel. Tables Math. Stat.* **1970**, *1*, 171–259.
30. Cythereal Inc. Cythereal MAGIC. Available online: http://www.cythereal.com (accessed on 15 July 2020).