Internet Commerce Security Lab

Technical Report

Identifying Cross-Version Function Similarity Using Contextual Features

Internet Commerce Security Lab (ICSL)
Federation University Australia
PO Box 663
University Drive, Mount Helen
Ballarat, Victoria, Australia 3353

June 2020

Identifying Cross-Version Function Similarity Using Contextual Features

Paul Black* Iqbal Gondal† Peter Vamplew§ Arun Lakhotia‡
*p.black@federation.edu.au, †iqbal.gondal@federation.edu.au, §p.vamplew@federation.edu.au,
‡arun@louisiana.edu

***Abstract*—The identification of similar functions in malware assists analysis by supporting the exclusion of functions that have been previously analysed, allows the identification of new variants, supports authorship attribution, and the analysis of malware phylogeny. A function's context is a set comprising the function itself, and all the program functions that may be executed when this function is called. Contextual features consist of data that is extracted from the functions contained in the function context. This paper presents a novel technique called Cross Version Contextual Function Similarity (CVCFS) to identify function pairs in two programs using features based on both individual functions and function context. The CVCFS technique uses Support Vector Machine (SVM) machine learning of function similarity features to pre-filter function pairs and then applies an edit distance technique using function semantics to reduce false-positives. A case study is provided where individual and contextual features are extracted from three versions of Zeus malware. The SVM pre-filtering, followed by the use of an edit distance technique to filter false-positives, gives a function pair identification accuracy of 85 percent.**

***Index Terms*—malware similarity, malware evolution, function similarity, binary similarity, Zeus malware, machine learning**

I. Introduction

The identification of similar functions in malware assists analysis by supporting the exclusion of functions that have been previously analysed, enables faster identification of new variants, supports authorship attribution, and the analysis of malware phylogeny [1]. Similar functions in a pair of malware samples may be due to the samples being from the same malware family, the samples being produced by the same malware author, use of shared code, a common library, or a well-known malware technique [2].

The organisations performing malware analysis face several challenges. The first is the large number of packed malware samples that must be processed daily. Unpacking of these malware samples reveals the original malware program. A comparison of the unpacked malware programs allows the elimination of exact matches, thereby reducing the volume of new malware samples by several orders of magnitude [3], [4]. Clustering may be used to segregate the unique malware samples into malware variants [5]. Previously analysed variants are removed from the clustered samples using malware similarity, some of which use function comparison [6]. Function similarity can be used to exclude functions that are unchanged and allow analysis to focus on changed functions.

Software evolution occurs in most malware families as a result of malware development efforts. As new malware versions are released, the machine code in the malware functions diverges from that of previous versions. Modeling of the generation and evolution of malware has been performed [2]; however, matching compiled functions across multiple malware versions remains a significant research challenge.

This paper presents Cross Version Contextual Function Similarity (CVCFS), a 2-step method for the identification of similar pairs of functions in two variants of a program. In the first step, a Support Vector Machine (SVM) model is used to pre-filter candidate pairs from a set of all pairs of functions from the two variants. The innovation in this step lies in the use of features that stay reasonably invariant even under refactoring and program evolution. These features termed – contextual features – capture the calling context of a function. In the second step, false positives from the first step are weeded out by using edit distance between over function semantics.

A case study is performed where function similarity features for training were extracted from Zeus version 2.0.8.7 and Zeus 2.0.8.9 malware samples. The compilation dates of these training samples differ by approximately one month, with minor variations between the functions. The malware samples were labelled with ground truth function matching labels that were obtained by manual reverse engineering. An SVM model was trained using the labelled features. Function pair prediction was tested using Zeus version 2.0.8.9 and 2.1.0.1 malware samples. The compilation dates of these malware samples differ by approximately six months with significant variation in some of the functions. A comparison of the SVM model results demonstrates the higher performance of contextual features. The SVM pre-filtering is followed by the use of an edit distance technique to filter false-positives. This resulted in a function pair identification accuracy of 85 percent, this was verified against the baseline function similarity obtained by manual reverse engineering.

This paper makes the following contribution:

- Introduces a new class of features called contextual features that strengthen features from individual functions. These are well suited for capturing similarity between malware samples exhibiting evolutionary changes.
- Cross Version Contextual Function Similarity (CVCFS) technique for finding similar functions in pairs of programs, malware, or benign.

The following additional contributions are provided:

- A curated dataset of pairs of matched functions in three versions of Zeus malware for use in future research.
- A set of three labelled IDA databases of Zeus malware versions 2.0.8.7, 2.0.8.9, and 2.1.0.1. [1].
- Creation of labelled Interactive Disassembler (IDA) databases of Zeus versions 2.0.8.7, 2.0.8.9, 2.1.0.1 malware samples.

The structure of this paper is as follows: Section II presents related work, Section III presents the research methodology, Section IV presents the empirical evaluation of results, and Section V presents the conclusion.

II. Related Work

A. 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 [7]. The aim of generating abstract semantics is to represent any two equivalent code sequences by the same semantics [8], [9].

Disassembly and CFG extraction are performed by existing tools. Basic block semantics are generated using symbolic interpretation. 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 [7].

VirusBattle is built on BinJuice and is used to identify relationships between malware samples in a large database [10]. 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 [9]. VirusBattle has been commercialised by Cythereal Inc and is known as Cythereal MAGIC.

B. Machine Learning In Software Similarity

The proliferation of IoT devices using open source code and a variety of CPU architectures has led to a research interest in the identification of known defects in the same source code compiled for several different CPU architectures. Examples of this research are provided by Gemini [11] and SAFE [12]. Early approaches to function binary similarity made use of graph isomorphism. These techniques were effective, but performance becomes an issue with increasing graph size. Neural networks have been used for cross-architecture bug-search as these techniques exhibit better performance characteristics than graph isomorphism.

Gemini performs cross-architecture, binary, software defect search using Control Flow Graph (CFG) features. Gemini uses a neural network to extract features from the CFG of the compiled functions [11].

Cross-architecture function similarity is also addressed by SAFE, which extracts the instruction sequence from compiled functions and models them as a natural language [12]. The interaction of the instruction sequence is captured using a Gated Recurrent Unit Neural Network (GRU-RNN). An attention mechanism is used to automatically focus on the instructions with the best performance for the identification of similar functions.

The techniques in the above research use neural networks to locate software vulnerabilities in the same version of code that has been compiled for different CPU architectures. These techniques are optimised for cross-architecture vulnerability identification within a program of the same software version. The techniques provided in this paper provide identification of similar functions in different versions of the same program compiled for a single CPU architecture.

C. Dominator Tree Features

Features taken from the dominator tree of Application Program Interface (API) calls in Android malware are used for malware detection [13]. The nesting structure of the API calls in the Android program is captured by the dominator tree of API calls; this allows identification of malware modules. The Term Frequency/Inverse Document Frequency (TF-IDF) weighting method [14], [15] is used to assign weights to each node in the dominator tree. The weighted nodes are used to build a database of malware dominator tree signatures. Classification is performed using RandomForest, RandomTree, ADTree, AdaBoost, and NaiveBayes models [13]. This research on the identification of Android malware is the first to use an API call dominator tree for detecting Android malware. In graph theory, the definition of dominator states that node d of a flow graph dominates node n if every path from the entry node to node n must go through node d [16].

In this Android research, features are extracted from the dominant nodes of the dominator tree. The CVCFS function similarity technique presented in this paper calculates the context of each function in the program under consideration, features are then extracted and summed across all of the functions within the function context.

III. Research Methodology

This research provides the CVCFS technique that is used to identify functions that were compiled from different versions of the same malware family. This technique uses an SVM model to pre-filter function pairs, and then uses an edit distance technique on the function semantics to filter

[1]The datasets related to this research are available online at http://federation.edu.au/icsl/evolvedsimilarity

and significantly reduce false-positives from the preliminary function pair identification.

Previous research using ad-hoc methods and hard-coded heuristics were used to identify function similarity [17]. CVCFS builds on this research by replacing the heuristics and ad-hoc methods with a Support Vector Machine (SVM) model.

SVM is a machine learning model for binary classification [18]. The SVM algorithm divides an n-dimensional feature space into two classes using a hyperplane. The CVCFS program extracts features from functions in the disassembly of two related programs; an SVM model is used to identify similar function pairs in the two programs.

The Function Similarity Ground Truth (FSGT) dataset contains data defining the pairing of the functions from the two programs being compared. The FSGT dataset contains a name that has been assigned to each function pair and the Relative Virtual Addresses (RVAs) of each function in the pair. The function names are used for researcher convenience and are not used in the similarity algorithms.

A. Function Context

Given a program p containing a set of functions F, the context *c(f)* of a specific function f is function *f*, plus the set of all non-API functions *f'* that can be reached by walking the call graph starting from function *f*.

- **Control Flow Edge** A control flow edge e represents the transfer of control that occurs when function x calls function y. A control flow edge e from function x to function y is represented by e = *(x,y)*.
- **Call Graph** A call graph is represented by the directed graph G = *(V,E)*, where V is the set of functions in a program, and E is the set of control flow edge transitions.
- **Walk** A walk w in G is a finite set of control flow edge transitions that occur as the result of a sequence of function calls from the execution of function *f*.

$$w(f) = \{e_0, e_1, ..., e_{\mathrm{i}}\} \tag{1}$$

- **Path** A path w in G is the set of vertices v traversed due to a walk in a call graph G, this path represents the set of functions *f'* called by the execution of function *f*.

$$w(f) = \{v_0, v_1, ..., v_{\mathrm{i\text{-}1}}\} \tag{2}$$

- **Function Context** The context *c(f)* of function *tf* is the set of all functions *f'* that can be reached by walking all possible paths in G starting from the vertex representing function *f*. Recursion and call graph loops require limiting the walk to paths not previously walked.

$$c(f) = \{f' : f' \ \epsilon \ \forall \ p(f)\} \tag{3}$$

B. Local Features

The CVCFS technique makes use of local features taken only from function *f*, and contextual features taken from the context *c(f)* of the function. The local features consist of the following:

- Set of API calls,
- Set of constants,
- Stack size,
- Function callers count.
- Basic block count,

Set of API calls: The system programming interface for the Windows operating system is provided by Windows Application Programming Interface (API) [19]. This API provides a dynamic function call interface for Windows system services. Windows programs use the portable executable (PE) format. In the case where a call to an API results in calls subsequent API's, only the first API call is recorded. Let *AL(f,p)* be the set of API functions called by function f in program p,

$$AL(f,p) = \{a_0, a_1, \ldots, a_n\}. \tag{4}$$

Set of 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 *CL(f,p)* be the set of constants that are not program or stack addresses contained in function f of program p.

$$CL(f,p) = \{c_0, c_1, ..., c_{\mathrm{m}}\} \tag{5}$$

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

$$SL(f,p) = \{s_0\} \tag{6}$$

Function callers count: Let *FL* be the count of calls made to function f in program p.

$$FL(f,p) = \{f_0\} \tag{7}$$

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 [16]. Let *BL* be the count of basic blocks in function *f*.

$$BL(f,p) = \{b_0\} \tag{8}$$

C. Local Feature Ratios

The CVCFS system calculates feature ratios using 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*, *p2* as would be identified by comparing program *p2*, *p1*. Let *F(p)* be the set of all functions in program p.

$$F(p) = \{f_0, f_1, ..., f_1\} \tag{9}$$

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

$$FP(p1,p2) = F(p1) \times F(p2) \tag{10}$$

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*.

$$fp = (f1, f2) \tag{11}$$

Local API Ratio: Let ALE_1 and ALE_2 be the sets of API calls extracted from each of the functions in function pair *fp*. Let local API ratio *ARL*, be the ratio of the magnitude of the intersection of ALE_1 and ALE_2 to the larger of the magnitude of ALE_1 and ALE_2.

$$ARL = len(ALE_1 \cap ALE_2)/max(len(ALE_1), len(ALE_2)) \tag{12}$$

Local Constants Ratio: Let CLE_1 and CLE_2 be the sets of constants extracted from each of the functions in function pair *fp*. Thus CL_1 = *CLE(fp[0])* and CL_2 = *CLE(fp[1])*. Let local constants ratio *LCR* be the ratio of the magnitude of the intersection of CLE_1 and CLE_2 to the larger of the magnitude of CLE_1 and CLE_2 .

$$CRL = len(CLE_1 \cap CLE_2)/max(len(CLE_1), len(CLE_2)) \tag{13}$$

Local Stack Ratio: Let SLE_1 and SLE_2 be the stack sizes extracted from each of the functions in function pair *fp*. Let local stack ratio *LSR* be the ratio of the magnitude of the absolute value of the difference between SLE_1 and SLE_2 to the larger of the magnitude of SLE_1 and SLE_2 .

$$SRL = abs(SLE_1 - SLE_2)/max(len(SLE_1), len(SLE_2)) \tag{14}$$

Callers Ratio: Let FCE_1 and FCE_2 be the function callers counts extracted from each of the functions in function pair *fp*. Let *FCD* be the absolute value difference between FCE_1 and FCE_2. Then *FCD* = abs(FCE_1 - FCE_2). Let callers ratio *CR*, be the ratio of *FCD* to the larger of FCE_1 and FCE_2.

$$CR = FCD/max(FCE_1, FCE_2) \tag{15}$$

Blocks Ratio: Let BLE_1 and BCE_2 be the basic block counts extracted from each of the functions in function pair *fp*. Let blocks ratio *BR* be the ratio of BLE_1 and BLE_2.

$$BR = min(BLE_1, BLE_2)/max(BLE_1, BLE_2) \tag{16}$$

D. Contextual Features

The strength of function similarity features can be improved by extracting the feature across all of the functions contained within the context of the function under consideration. The contextual features consist of the following:

- Contextual set of API calls,
- Contextual set of constants,
- Contextual stack size,
- Contextual return count,
- Contextual function calls count.

Contextual set of API calls: Let *AC* be the set of API calls made from the context *c(f)* of function *f*.

Contextual set of constants: Let *CC* be the set of constants from the context *c(f)* of function *f*.

Contextual stack size: Let *SC* be the sum of stack sizes from the context *c(f)* of function *f*.

Contextual return count: Let *RC* be the count of return instructions from the context *c(f)* of function *f*.

Contextual function calls count: Let *CS* be the count of call instructions from the context *c(f)* of function *f*.

E. Contextual Feature Ratios

The contextual feature ratios consist of the following:

Contextual API Ratio: Let AC_1 and AC_2 be the set of API calls made from the context of each of the functions in function pair *fp*. Let the contextual API ratio *ACR*, be the ratio of the magnitude of the intersection of AC_1 and AC_2 to the larger of the magnitude of AC_1 and AC_2.

$$ACR = len(AC_1 \cap AC_2)/max(len(AC_1), len(AC_2)) \tag{17}$$

Contextual Constants Ratio: Let CC_1 and CC_2 be the set of constants from the context of each of the functions in function pair *fp*. Let the contextual constants ratio *CCR* be the ratio of the magnitude of the intersection of CC_1 and CC_2 to the larger of the magnitude of CC_1 and CC_2 .

$$CCR = len(CC_1 \cap CC_2)/max(len(CC_1), len(CC_2)) \tag{18}$$

Contextual Stack Ratio: Let SC_1 and SC_2 be the sum of the stack sizes from the context of each of the functions in function pair *fp*. Let the contextual stack ratio *SCR* be the ratio of the magnitude of the intersection of SC_1 and SC_2 to the larger of the magnitude of SC_1 and SC_2 .

$$SCR = len(SC_1 \cap SC_2)/max(len(SC_1), len(SC_2)) \tag{19}$$

Contextual Returns Ratio: Let RC_1 and RC_2 be the count of return instructions from the context of each of the functions in function pair *fp*. Let the contextual returns ratio *RCR*, be the ratio of absolute value difference between RC_1 and RC_2 to the larger of RD_1 and RD_2.

$$RCR = abs(RC_1 - RC_2)/max(len(RC_1), len(RC_2)) \tag{20}$$

Contextual Calls Ratio: Let FC_1 and FC_2 be the count of call instructions from the context of each of the functions in function pair *fp*. Let the callers ratio *FCR*, be the ratio of absolute value difference between FC_1 and FC_2 to the larger of FC_1 and FC_2.

$$FCR = abs(FC_1 - FC_2)/max(FC_1, FC_2) \tag{21}$$

F. Edit Distance Filtering

The function similarity results obtained by the SVM model developed in an earlier section included a large number of false-positive results. Experimentation with the SVM model was performed but the false-positive results remained a problem. Existing research [20], [11] uses machine learning as the first stage pre-filter in identifying similar functions. To overcome the large number of false-positives, a decision was taken to use the SVM model as a pre-filter and to add an edit distance metric using the BinJuice generalised semantics to filter out false-positives.

Although it would be possible to remove the pre-filtering step and to solely use the graph edit distance for the identification of function pairs, this would not be feasible due to

```
Create empty function context
Create empty visited list
From the disassembly of function f
  Add static function calls from f to function context
  For each function in function context
    If function not in visited list
      Add function to visited list
      Recursively get new static functions called by function
      Add new static functions to function context
Add f to function context
```

Fig. 1. Function Context Extraction Algorithm

```
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 individual features for function
    calculate contextual features for function

  for each function in p2
    calculate individual features for function
    calculate contextual features for function

  for each f1 in p1
    for each f2 in p2
      calculate locate feature ratios
      calculate contextual feature ratios
      if f1, f2 in FSGT dataset
        set label = "1"
      else
        set label = "0"

  rva1 = rva(f1)
  rva2 = rva(f2)
  features = map(rva1, rva2, ARL, ARD, FRD,
                 RRD, CR, LCR, CRD, LSR, LSD, LBR)
```

Fig. 2. Feature Extraction Algorithm

the significant execution time of the edit distance calculation. In the research in this paper, the SVM pre-filter ran in approximately two minutes, while the run time for the edit distance filtering could take as long as 12 hours.

BinJuice function semantics contain four levels of abstraction, where the most abstract form of the function semantics is the Generalised Semantics. The Levenshtein edit distance [21] of the Generalised Semantics of each function was calculated. Edit distance increases with function size and cannot be used directly to identify matching function pairs. The edit distance was normalised by dividing the edit distance by the function basic block count to give a Normalised Edit Distance per Basic Block (NEDBB) metric The NEDBB metric was used to identify matching function pairs from the pre-filtered matches.

G. CVCFS Algorithm

The algorithm used to extract the function context of a specified function is shown in Figure 1. The algorithm for feature extraction is shown in Figure 2. The algorithm used to perform edit distance filtering of the pre-filtered function pairs is shown in Figure 3. In the edit distance filtering algorithm, the `filter value` is a hardcoded threshold that is tested against the NEDBB value in order to select the pre-filtered function pairs.

```
let ml_predicted = all fn pairs predicted match
for each fn_pair in ml_predicted
  let sem1 = semantics of fn_pair[1]
  let sem2 = semantics of fn_pair[2]
  let ed = nktk.edit_distance(sem1, sem2)
  let bc1 = blocks count fn_pair[1]
  let ed_per_block = ed / bc1
  if ed_per_block < filter value
    match = True
  else
    match = False
```

Fig. 3. Edit Distance Filtering Algorithm

IV. Empirical Evaluation

The malware samples used in this paper are shown in Table I. The unpacked samples were provided by Cythereal and were disassembled using the Interactive Disassembler (IDA). The linker date in the Portable Executable (PE) header is 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 this time has been modified. The linker dates indicate that sample 2 was produced approximately one month after sample 1, and sample 3 was produced approximately six months after sample 1.

The malware samples were submitted for Cythereal processing. The output from the Cythereal processing is the unpacked malware sample and a dataset containing the disassembly and semantics of the unpacked malware sample [22].

Sample SHA1 Hash	Version	Date
8a7faa25f23a0e72a760075f08d22a91d2c85f57	2.0.8.7	2010-09-14
706bf4dcf01b8eceedf6d05cf5b55a27e4ff8ef0	2.0.8.9	2010-10-15
30c6bb2328299a252436d2a3190f06a6f04f7e3f	2.1.0.1	2011-03-24

TABLE I
Zeus Sample Details

Tool	Version	Count	Version	Count	Match Count
IDA	2.0.8.7	577	2.0.8.9	553	549
Cythereal	2.0.8.7	577	2.0.8.9	553	549
IDA	2.0.8.9	553	2.1.0.1	601	539
Cythereal	2.0.8.9	539	2.1.0.1	601	517

TABLE II
Zeus Function Count and Manual Match Count

The FSGT dataset identifies the function pairs, function names and function RVAs for each malware sample used in this research. Manual analysis using IDA, the unpacked malware samples, and the leaked Zeus source code was performed in order to identify the function pairs for the FSGT dataset. Function identification was performed based on the API calls, constants, and CFG structure. Function names from the Zeus source code provide a convenient identification for researchers, but are not used by the research code. Function RVAs are used to identify functions in the Cythereal semantics and in the research code. The results of the function name labelling are shown in Table II. IDA identified 553 functions in sample 3 while Cythereal MAGIC could only identify 539 functions in sample 3. This may be an artifact of the automatic unpacker.

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

Fig. 4. Function Prologue

A. Features

To calculate the local and contextual constant features, constants were extracted from `mov`, `push`, `add`, `cmp`, and `sub` instructions. 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.

The stack size is taken from the function prologue when it is present; otherwise, it is zero. The stack size *SL* 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.

As this research uses static analysis, the non-API function call counts used in this research are a count of static function calls.

B. SMOTE Oversampling

The function similarity training dataset was imbalanced due to the use of the cartesian product comparison, which resulted 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 performance of the SVM model was improved with the use of Synthetic Minority Oversampling Technique (SMOTE) [23] to rebalance the training dataset.

C. SVM Model Training

In this paper, sample 1 and sample 2 were used for training as these two Zeus samples are similar but exhibit a number of minor differences. Function similarity features for training were calculated using the cartesian product of all functions in samples 1 and 2. These features were labelled as matching or not matching using the FSGT dataset. All possible feature combinations were used to train a series of SVM models in order to identify the best performing feature combinations. As 10 features were used, exhaustive testing required 1023 tests to be performed. The features in each of these tests were assigned a binary identifier, e.g., the first feature is identified as `0000000001`, the second feature was identified as `0000000010`. The use of binary identifiers allowed the numbering of individual tests; these feature identifiers are shown in Table III.

D. Pre-Filtering

The functions in samples 1 and 3 exhibit more differences due to software development than the training dataset. A testing set of function similarity features were created using the cartesian product of all functions in samples 1 and 3. The previously generated SVM models were used to predict the matching function pairs from testing feature set. The results of this prediction were evaluated using the FSGT dataset.

Feat #	Vector	Description
1	0000000001	Basic Block Ratio
2	0000000010	Contextual Stack Ratio
4	0000000100	Local Stack Ratio
8	0000001000	Contextual Constants Ratio
16	0000010000	Local Constants Ratio
32	0000100000	Contextual Callers Ratio
64	0001000000	Contextual Returns Ratio
128	0010000000	Calls Ratio
256	0100000000	Local API Ratio
512	1000000000	Contextual API Ratio

TABLE III
NUMBERING FOR FEATURE COMBINATION TESTS

The performance of each individual feature was assessed by performing function similarity classification using SVM models trained for each individual feature. The results of this evaluation are shown in Table IV. Although the prediction of the SVM model was reasonable, the recall performance was not good resulting in a significant false-positive count. A number of features were tested in an effort to reduce the false-positive count, ultimately it was decided to use the SVM model as a pre-filter of function pairs.

The feature combinations which provided the best performance are shown in Table V. The function pair prediction results in this research vary from run to run due to the stochastic nature of machine learning.

The F-measure (F1) [24] defined in equation 22 is used to assess the precision and recall of results. From the F-measure, the best performing feature combination is test 611, that makes use of the Contextual API ratio, the Contextual Returns Ratio, the Contextual Callers Ratio, the Contextual Stack Ratio and the Basic Blocks Ratio, as shown in Table V.

$$F1 = 2 * (Precision * Recall) / (Precision + Recall) \quad (22)$$

Test	Vector	TP	FP	FN	Pr	Rc	F1
1	00000000001	500	64486	17	0.01	0.97	0.02
2	00000000010	280	19528	237	0.01	0.54	0.03
4	00000000100	265	20460	252	0.01	0.51	0.02
8	00000001000	479	10338	38	0.04	0.93	0.08
16	0000010000	460	18767	57	0.02	0.89	0.05
32	0000100000	378	69778	139	0.01	0.73	0.01
64	0001000000	490	36333	27	0.01	0.95	0.03
128	0010000000	444	65050	73	0.01	0.86	0.01
256	0100000000	279	746	238	0.27	0.54	0.36
512	1000000000	408	6565	109	0.06	0.79	0.11

TABLE IV
INDIVIDUAL FEATURE PERFORMANCE

E. Feature Performance

The relative performance of individual features can be assessed from Table IV or it can be assessed from a count of

Test	Vector	TP	FP	FN	Pr	Rc	F1
39	0000100111	510	66327	7	0.01	0.99	0.02
423	0110100111	509	50222	13	0.01	0.98	0.02
475	0111011011	511	25759	6	0.02	0.99	0.04
513	1000000001	510	21120	7	0.02	0.99	0.05
577	1001000001	509	41955	8	0.01	0.98	0.02
579	1001000011	509	21816	8	0.02	0.98	0.04
611	1001100011	509	12661	8	0.04	0.98	0.07
614	1001100110	509	42280	8	0.01	0.98	0.02
643	1010000011	511	42743	6	0.01	0.99	0.02
711	1011000111	509	26980	8	0.02	0.99	0.04
742	1011100110	511	54320	6	0.01	0.99	0.02
769	1100000001	512	69733	5	0.01	0.99	0.01
771	1100000011	511	34024	6	0.01	0.99	0.03
775	1100000111	512	63062	5	0.01	0.99	0.02
838	1101000110	509	32581	8	0.02	0.98	0.03
865	1101100001	511	31977	6	0.02	0.99	0.03
962	1111000010	509	45490	8	0.01	0.98	0.02

TABLE V
HIGHEST PERFORMING FEATURE COMBINATIONS

those features present in the best performing feature combinations shown in Table V. Table VI summarises the performance of each of the features, the "Ind Rank" column ranks individual feature performance based on the F-Score with higher numbers indicating better performance. From this ranking, the Local API Ratio outranked the Contextual API Ratio, the Contextual Constants Ratio outranked the Local Constants Ratio, and the Contextual Stack Ratio outranked the Local Stack Ratio.

Feature performance was also assessed by observing the number of times the feature was present in the test runs of the Highest Performing Feature combinations in Table V. This is shown in Table VI as "TFP Count" (Top Performing Feature Count), this value is used to create a rank as shown in the "TPF Rank" column. In this ranking, the Contextual API Ratio outranks the Local API Ratio, and the Contextual Stack Ratio outranks the Local Stack Ratio, the Contextual Returns Ratio performed well with a ranking of 5. The features based on constants did not rank highly. The Basic Blocks Ratio ranked equally with the Contextual Stack Ratio.

Feature	Ind Rank	TPF Count	TPF Rank
Basic Blocks Ratio	2	13	6
Contextual Stack Ratio	3	13	6
Local Stack Ratio	2	7	3
Contextual Constants Ratio	5	1	1
Local Constants Ratio	4	1	1
Contextual Callers Ratio	1	6	2
Contextual Returns Ratio	3	10	5
Calls Ratio	1	6	2
Local API Ratio	7	8	4
Contextual API Ratio	6	14	7

TABLE VI
RANKING OF FEATURE PERFORMANCE

F. Edit Distance Filtering

An edit distance metric was used to filter the pre-filtered function pair predictions to reduce the false-positive count from the SVM pre-filtering.

The results in Table VII show the performance variation due to the use of different values of the Edit Metric Filter (EMF). This shows that the best performance is obtained using an EMF value of 7.

Test	EDF	OP	TP	FP	FN	Pr	Rc	F1
611		"ML"	508	22248	9	0.02	0.98	0.04
611	5	"EM"	423	172	94	0.71	0.82	0.76
611	6	"EM"	438	178	79	0.71	0.85	0.77
611	7	"EM"	441	184	76	0.71	0.85	0.77
611	8	"EM"	444	214	73	0.67	0.86	0.76
611	9	"EM"	452	243	65	0.65	0.87	0.75
611	10	"EM"	458	279	59	0.62	0.89	0.73
611	11	"EM"	464	325	53	0.59	0.90	0.71
611	12	"EM"	469	395	48	0.54	0.91	0.68

TABLE VII
EDIT METRIC FILTER PERFORMANCE

In the results shown in Table VIII, an edit distance filter value of 7 was used to reject false-positive predictions. The "ML" identifier in the operation ("OP") column denotes the results from the SVM pre-filtering stage, the "EM" identifier indicates the final results obtained using the edit distance filtering. Referring to Table VIII, the highest F-measure (F1) value occurs with Test 614 with a value of 0.77, and 441 function pairs correctly identified. This corresponds to a function pair identification accuracy of 85 percent.

Test	Op	TP	FP	FN	Pr	Rc	F1
39	ML	505	41721	12	0.01	0.98	0.02
39	EM	443	231	74	0.66	0.86	0.74
423	ML	487	42284	30	0.01	0.94	0.02
423	EM	422	164	95	0.72	0.82	0.77
475	ML	496	7353	21	0.06	0.96	0.12
475	EM	429	167	88	0.72	0.83	0.77
513	ML	510	46724	7	0.01	0.99	0.02
513	EM	442	233	75	0.65	0.85	0.74
577	ML	504	14513	13	0.03	0.97	0.06
577	EM	440	183	77	0.71	0.85	0.77
579	ML	509	11928	8	0.04	0.98	0.08
579	EM	441	182	76	0.71	0.85	0.77
611	ML	509	24339	8	0.02	0.98	0.04
611	EM	441	185	76	0.70	0.85	0.77
614	ML	508	33414	9	0.01	0.98	0.03
614	EM	441	175	76	0.72	0.85	0.78
643	ML	423	6536	94	0.06	0.82	0.11
643	EM	360	146	157	0.71	0.70	0.70
711	ML	497	18675	20	0.03	0.96	0.05
711	EM	430	200	87	0.68	0.87	0.76
742	ML	475	15504	42	0.03	0.92	0.06
742	EM	413	154	104	0.73	0.80	0.76
769	ML	412	7199	105	0.05	0.80	0.10
769	EM	345	157	172	0.69	0.67	0.68
771	ML	428	11205	89	0.04	0.83	0.07
771	EM	360	156	157	0.70	0.70	0.70
775	ML	508	29646	9	0.02	0.98	0.03
775	EM	442	229	75	0.66	0.85	0.74
838	ML	507	34360	10	0.01	0.98	0.04
838	EM	440	192	77	0.70	0.85	0.77
865	ML	511	27297	6	0.02	0.99	0.04
865	EM	441	186	76	0.70	0.85	0.77
962	ML	456	16292	61	0.03	0.88	0.05
962	EM	413	173	104	0.70	0.80	0.75

TABLE VIII
RESULTS FOLLOWING EDIT METRIC FILTERING

G. Future Work

Work presented in this paper can be extended as follows:

- Extend the concept of function context and its use in program analysis,
- Investigate the performance variation in existing function similarity programs over increasing evolutionary distance,
- Generalize the techniques used in this paper for identifying function similarity,
- Automatically create the FSGT dataset,
- Research function similarity in programs that have been subject to significant software development.

V. Conclusion

The research in this paper introduces the concept of contextual features and provides methods for their calculation. A combination of individual and contextual features are used in the CVCFS technique for the identification of similar functions in two related programs. This paper demonstrates that contextual features provide improved performance compared to the corresponding features extracted from individual functions.

The technique for the generation of contextual features involves summing features extracted from the context of each function. This technique of strengthening features by summing over the function context is generally applicable to a wide range of machine learning function similarity research.

The CVCFS function similarity technique is presented using a case study; however, this technique is generally applicable to identifying similar functions in both malware and benign programs. A comparison of the effectiveness of the local features and the contextual features was performed using a ranking based on feature performance and of the frequency of features in the highest performing feature combinations. This ranking shows that contextual features outperform local features in four out of six cases.

An edit distance technique was used to filter the false-positives from the machine learning results and a maximum accuracy of 85 percent identification of true function pairs was achieved with an F-measure value of 0.77.

Acknowledgement

The authors would like to thank Cythereal[2] for providing access to Cythereal MAGIC [22] and to the malware dataset used in this research. This research was funded in part through the Internet Commerce Security Laboratory (ICSL), a joint venture between Westpac, IBM and Federation University Australia. Paul Black is supported by an Australian Government Research Training Program (RTP) Fee-Offset Scholarship through Federation University Australia.

References

[1] S. Alrabaee, M. Debbabi, and L. Wang, "On the feasibility of binary authorship characterization," *Digital Investigation*, vol. 28, pp. S3–S11, 2019.

[2] A. Walenstein and A. Lakhotia, "A transformation-based model of malware derivation," in *2012 7th International Conference on Malicious and Unwanted Software*. IEEE, 2012, pp. 17–25.

[3] F. C. C. Osorio, H. Qiu, and A. Arrott, "Segmented sandboxing-a novel approach to malware polymorphism detection," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2015, pp. 59–68.

[4] I. U. Haq, S. Chica, J. Caballero, and S. Jha, "Malware lineage in the wild," *arXiv preprint arXiv:1710.05202*, 2017.

[5] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[6] T. Kim, Y. R. Lee, B. Kang, and E. G. Im, "Binary executable file similarity calculation using function matching," *The Journal of Supercomputing*, vol. 75, no. 2, pp. 607–622, 2019.

[7] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 2013, p. 5.

[8] B. H. Ng and A. Prakash, "Expose: Discovering potential binary code re-use," in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. IEEE, 2013, pp. 492–501.

[9] A. Lakhotia and P. Black, "Mining malware secrets," in *Malicious and Unwanted Software (MALWARE), 2017 12th International Conference on*. IEEE, 2017, pp. 11–18.

[10] C. Miles, A. Lakhotia, C. LeDoux, A. Newsom, and V. Notani, "Virusbattle: State-of-the-art malware analysis for better cyber threat intelligence," in *Resilient Control Systems (ISRCS), 2014 7th International Symposium on*. IEEE, 2014, pp. 1–6.

[11] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 363–376.

[12] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 309–329.

[13] S. Alam, S. Yildirim, M. Hassan, and I. Sogukpinar, "Mining dominance tree of api calls for detecting android malware," in *2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*. IEEE, 2018, pp. 1–4.

[14] G. Salton and C. Buckley, "Term weighting approaches in automatic text retrieval," Cornell University, Tech. Rep., 1987.

[15] Z. Yun-tao, G. Ling, and W. Yong-cheng, "An improved tf-idf approach for text classification," *Journal of Zhejiang University-Science A*, vol. 6, no. 1, pp. 49–55, 2005.

[16] M. Lam, R. Sethi, J. Ullman, and A. Aho, *Compilers: Principles, techniques, and tools*. Pearson Education, 2006.

[17] P. Black, I. Gondal, P. Vamplew, and A. Lakhotia, "Evolved similarity techniques in malware analysis," in *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)*. IEEE, 2019, pp. 404–410.

[18] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[19] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows internals*. Pearson Education, 2012.

[20] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *NDSS*, 2016.

[21] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[22] Cythereal Inc, "Cythereal magic," 2018. [Online]. Available: https://www.cythereal.com

[23] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[24] C. J. Van Rijsbergen, *Information retrieval*. Citeseer, 1979.

[2]Cythereal has licensed VirusBattle from the University of Louisiana at Lafayette