OBA2: An **O**nion Approach to **B**inary Code **A**uthorship **A**ttribution

Saed Alrabaee, Noman Saleem, Stere Preda, Lingyu Wang, Mourad Debbabi

National Cyber Forensics and Training Alliance Canada
Computer Security Laboratory, Concordia University, Montreal,Canada

Abstract

A critical aspect of malware forensics is authorship analysis. The successful outcome of such analysis is usually determined by the reverse engineer's skills and by the volume and complexity of the code under analysis. To assist reverse engineers in such a tedious and error-prone task, it is desirable to develop reliable and automated tools for supporting the practice of malware authorship attribution. In a recent work, machine learning was used to rank and select syntax-based features such as n-grams and flow graphs. The experimental results showed that the top ranked features were unique for each author, which was regarded as an evidence that those features capture the author's programming styles. In this paper, however, we show that the uniqueness of features does not necessarily correspond to authorship. Specifically, our analysis demonstrates that many "unique" features selected using this method are clearly unrelated to the authors' programming styles, for example, unique IDs or random but unique function names generated by the compiler; furthermore, the overall accuracy is generally unsatisfactory. Motivated by this discovery, we propose a layered Onion Approach for Binary Authorship Attribution called OBA2. The novelty of our approach lies in the three complementary layers: preprocessing, syntax-based attribution, and semantic-based attribution. Experiments show that our method produces results that not only are more accurate but have a meaningful connection to the authors' styles.

Keywords:
Authorship Attribution, Reverse Engineering, Binary Program Analysis, Malware Forensics, Digital Forensics

1. Introduction

Malware is an increasing threat to network and distributed systems. To this end, determining the authorship of malware has many practical applications, ranging from post-mortem forensic analysis of malware corpora to online detection of live polymorphic malware. Determining software authorship based on stylistic features is possible because humans are creatures of habit, and habits tend to persist. However, most existing work on software authorship attribution relies on features that are obtained from the program source code [1, 3, 4, 5, 6, 7]. Such methods cannot easily be applied to malware whose source code is not always available.

To the best of our knowledge, the only notable exception is the use of machine learning techniques to correlate syntax-based features with authorship [8], which will be referred to, in this paper, as Identify the Author of Program Binaries (IAPB) approach. The features considered in IAPB are obtained from predefined templates, which include idioms, n-grams, and flow graphs. The features are ranked based on the degree of correlation with authorship during the training phase. The experimental results show that, in most cases, the top-ranked features are unique for each author; this supports the claim that the top-ranked features successfully capture an author's programming style. However, our paper finds that the above conclusion about IAPB may be overly optimistic. Specifically, our analysis demonstrates that many top-ranked features selected using this method, although unique for each author, are in fact clearly unrelated to an author's programming style. For example, they may correspond to a unique identifier or function names containing random (and thus unique) numbers generated by the compiler, which obviously do not reflect programming style. Furthermore, our analysis will show the overall accuracy of IAPB to be less promising.

Motivated by this discovery, the current paper proposes a novel multilayer approach to binary authorship attribution, OBA2, which incorporates three complementary layers, preprocessing, syntax-based attribution, and semantic-based attribution, for improving the accuracy. Moreover, the existing work and our approach are compared using real-world data and similar implementations. The results show that OBA2 generally yields more accurate outcomes, with a more meaningful connection to author style.

1.1. Motivations

IAPB is a pioneering research when it comes to the authorship attribution of binary code. The underlying ideas contributed to our understanding of authorship analysis. Nevertheless, we identify the following gaps:

- The existing IAPB method for binary code authorship attribution does not completely tackle the problem of filtering the features that are clearly unrelated to an author's style, such as compiler-generated functions. This leads us to investigate methods that automatically detect and filter out such features.
- The features obtained with the IAPB method follow generic templates, such as sequences of commands, whose semantics are less known. The selection of features without understanding their underlying semantics may bring misleading results. Therefore, we avoid such generic feature templates in feature selection.
- The IAPB method equates the uniqueness of features to an author's style, which leads to features that are unique but sometimes unrelated to style. To the best of our knowledge, little effort has been made to establish a formal definition of an author's programming style. In this paper, we took the first step toward defining this important concept.

1.2. Contributions

Our paper makes the following contributions.

- We propose OBA2, a layered approach to an effective authorship attribution that combines techniques for filtering out unrelated code – Stuttering Layer (SL), a syntax-based attribution layer – Code Analysis Layer (CAL), and a semantics-based attribution layer – Register Flow Analysis Layer (RFAL).
- For the SL layer, we design a signature-based method capable of automatically detecting software library functions and/or other functions known to be unrelated to an author's style.
- For the CAL layer, we provide a method for building a syntax dictionary to create a profile for each author by establishing mappings between binary code syntax and identified source codes.
- For the RFAL layer, we devise a novel model called the register flow graph that captures semantics-based features representing patterns in the way registers are manipulated.
- We compare the proposed OBA2 approach with IAPB by experimenting with the same real-world data from the Google Code Jam programming competition [10]. The results show that OBA2 has meaningful results with superior accuracy.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 provides a detailed description of the main methodologies. Section 4 evaluates the proposed approach and compares it to existing work. Section 5 gives limitations and future directions, and Section 6 presents the conclusions.

2. IAPB Approach

In this section, we briefly review the existing Identify the Author of Program Binaries (IAPB) approach [8]. The authors first show that authorship attribution is challenging with only binary code because of the limited understanding of the survival of programming styles and features after compilation. Accordingly, a technique was proposed to automatically detect useful stylistic features of binary code based on predefined feature templates, including idioms, graphlets, supergraphlets, call graphlets, n-grams, and external interactions. Machine learning algorithms are then employed to rank the stylistic significance of those features. Next, classification and clustering techniques are applied to top features for authorship attribution and for clustering binary programs based on authorship. We briefly review the IAPB feature templates and feature ranking in the following, while omitting more details due to space limitations.

2.1. Feature Template

IAPB extracts features using five pre-defined templates as follows:

Idioms: Idioms are short sequences of instructions intended for capturing stylistic characteristics in assembly files obtained from binary code using disassembler tools. The IAPB approach designs a template for idioms with a maximum of three instructions.

Graphlets: In the IAPB approach, a graphlet is a 3-node subgraph of the Control Flow Graph (CFG). To fetch graphlets from a binary file, with each basic block belonging to a function, all 3-node subgraphs of the CFG including this basic block will be computed.

Supergraphlets: Supergraphlets are obtained by collapsing and merging neighbor nodes of the CFG. The node collapsing algorithm operates on the same canonical representation as that for graphlets; two collapsed nodes may lead to a new color, and their edge types may also change.

Libcalls and **Call Graphlets**: Libcalls are function names of imported libraries, e.g., `call ds: printf`. Call graphlets are graphlets containing only nodes with a call instruction.

N-grams: N-grams are a set of byte sequences or instruction level sequences. N-grams are short sequences of bytes of length N.

2.2. Feature Ranking

IAPB first extracts simple, syntax-based features based on the above pre-defined feature templates, without concerning whether the extracted features may indeed be related to author styles. Many of those extracted features may be common to most of the programs, even though those programs are written by different authors. Therefore, in the feature ranking phase, the extracted features are ranked based on a set of binary programs with known authors as the training data. The goal of the feature ranking algorithm is to rank lower those features that are common to most authors, such that they will not be selected later on, while ranking higher those features that are unique for each author, since those are believed to represent the programming style.

3. OBA2 Methodology

In this section, we introduce the OBA2 approach, which consists of three complementary layers as follows.

3.1. Stuttering Layer (SL)

An important initial step in most reverse engineering tasks is to distinguish between user code and library code. This saves considerable time and helps shift the focus to more relevant functions. For example, our results show that the simplest C program, *Hello World!*, contains up to 60 initialization and standard library (shared) functions, even though it has only one main user function. To this end, IDA Pro [9] has an advanced feature for identifying code sequences and standard library functions, which is designed to recognize hex code sequences (e.g., statically linked library code, helper functions, and initialization code) that are generated by common C-family compilers. Figure 1 is a schematic of a proposed methodology that highlights user functions by identifying and eliminating library code.

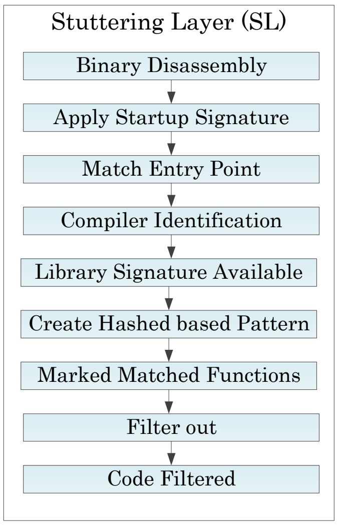

Figure 1: Methodology for identifying and filtering library code

More specifically, the SL process consists of the following four steps.

- The first step is to parse the libraries in order to extract function prototypes, API data structures, and typing information, which are beneficial for recognizing the correct function call conventions and parameters.

- The second step is to create a hash-based pattern for each exported function in the library based on interesting byte sequences, the size of the function, and features from the function implementation such as constant values, strings, imports, and called functions. A double hashing of this information would yield a unique value for the combination of extracted features. To avoid collisions of the resulting signatures, a hierarchical tree-based hashing technique is employed, which takes into account different features from each level. Then, we synthesize the hash values in a way that reduces the probability of collision. This process is applied iteratively to all functions and libraries. The results are saved as signature files.

- The third step involves applying the signatures to the list of function calls in disassembled code, performing signature matching, and marking the library calls. This is accomplished by considering the stack frame information of the function call, including the pushed values, the structure, and the parameters. Next, a hash value is generated for the extracted features, which is compared against the list of existing hashes. If the signature matches any of the known functions, it is tagged with the name of the library.

- Finally, the identified functions are excluded from the function list, and the user code is returned for further analysis.

The outcome of the above process is the generation of function signatures for shared libraries, which will then be used for highlighting user functions to be analyzed in the CAL and RFAL layers.

3.2. Code Analysis Layer (CAL)

The SL layer produces filtered code for further analysis in the next layer, called the Code Analysis Layer (CAL). The intuition behind the CAL layer is the following. Many programming styles may be captured in the syntax of the code. For example, an author may have the habit of using more `for` loops than `while` loops, or `if` conditions than `switch` conditions, etc; an author may also have a particular habit of entering parameters, or a peculiar way in using `scan`, `cin`, `cout`, or `printf`, etc. When such habits become significant enough to distinguish an author among others (i.e., uniqueness) and to correlate different programs written by that author (i.e., similarity), the habits become a style of the author, which is captured by patterns in the syntax of the code. Figure 2 shows the main components of the CAL layer, which is detailed in the following.

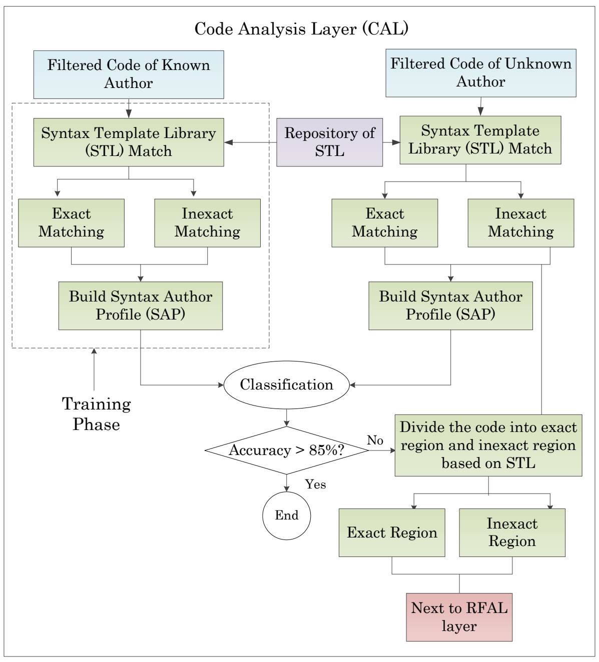
Code Analysis Layer (CAL)
Filtered Code of Known Author
Filtered Code of Unknown Author
Syntax Template Library (STL) Match
Repository of STL
Syntax Template Library (STL) Match
Exact Matching
Inexact Matching
Exact Matching
Inexact Matching
Build Syntax Author Profile (SAP)
Build Syntax Author Profile (SAP)
Training Phase
Classification
Accuracy > 85%?
No
Divide the code into exact region and inexact region based on STL
Yes
End
Exact Region
Inexact Region
Next to RFAL layer

Figure 2: Code Analysis Layer (CAL)

Source Code	Assembly
`array`	`xor-push-mov-push-lea-call`
`pointer`	n/a
`reference sign`	`lea eax, * - mov *, eax`
`c-libraries`	n/a
`cin`	`lea-push-mov-call-cmp-call`
`if`	`mov-cmp-jnz-xor-jmp`
`if/else`	`mov-cmp-jnz-mov-jmp`
`for loop`	`cmp-jg-xor-jmp`
`while`	`mov-cmp-jge-mov-add-mov-jmp-xor`
`cout`	`push-mov-cmp-call-cmp-call`
Integers, Doubles, Floats, and Chars	n/a
`printf`	`push-call-add-cmp-call`
`scan`	`lea-push-push-call-add-cmp-call`
`if/else if`	`mov-cmp-jl-mov-add-mov-jmp`
`cerr`	`push-call-add-mov-call-cmp-call`
`memset`	`lea-push-call-add-xor-push-mov-push-lea-call`

Table 1: Part of the STL repository

STL Repository

The main challenge of the CAL layer is to reconstruct the source code-level syntax from binary code. To that end, we have created an *Syntax Template Library (STL)* that matches C++ vocabularies to assembly instructions. An excerpt is shown in Table 1, where *n/a* indicates that no corresponding assembly pattern is available. Note that the matching will not always be exact, so, later in this section, we will explain how to use STL to detect both exact and inexact matching.

Syntax Signature Verification

To use syntax as a tool for capturing the programming styles of authors, we introduce a set of syntax signature templates with which we will perform syntax signature verification. These are designed to capture an author's choices of programming options (e.g., `if` or `switch`), which, along with their relative frequency of appearances, can reflect different levels of author style and expertise. By considering the use of containers in programming languages and their associated templates, we develop STL and extend it by introducing templates to capture choices of algorithms, the use of API and I/O devices, the type of encryption algorithms, OpenSSL, socket type, and data structure. The availability of STL enables the architecture to verify syntax signatures over user-related code based on functions in assembly language in order to flag author-related code that may indicate an author's style.

Exact Matching

We deal with the filtered code as a collection of basic blocks, where a basic block is a maximal sequence of instructions that executes, without branching, from beginning to end. Each basic block has a single entry point (the first instruction in the block) and a single exit point (the last instruction in the block). By checking all code blocks in the filtered code, we extract code blocks that exactly match the predefined syntax signature templates based on STL. The resulting code blocks reflect the use of a particular syntax in user-generated code and also the frequency with which it occurs. The process of exact matching compares assembly code instructions. Two code blocks are considered an exact match only if all statements in the two blocks are identical. One way to identify exact matches is to compare each code block in the user-generated code with all the signature templates, but this would be computationally prohibitive, with a complexity of $O(n^2)$, where n is the total number of code blocks. Alternatively, as shown in Algorithm 1, we employ an indirect and more efficient approach based on hashing: two code blocks are an exact match if they have the same hash value. In the algorithm, the first loop calculates the hash values of the code blocks and stores the results; the second loop counts the occurrences in STL; and if the count is not less than two, the corresponding code blocks are an exact match and will be stored in the third loop.

Inexact Matching

We also extract from the user-generated code the code blocks that do not exactly match the predefined syntax signature templates in the `STL` but exhibit similar patterns. The presence of inexact matching code blocks reflects the probable use of a particular syntax in user-generated code

Algorithm 1: Exact Syntax Signature Detection

```
input : $CB$: set of all code blocks
        $STL$: set of all syntax templates library
        in repository
        $H$: an empty Hash Table
        $EM_i$: Set of empty code blocks
        $v$: Hash value
output: EM: Set of code blocks exact match
begin
    for each $cb$ in $CB$ do
        $H \leftarrow Hash\,(cb)$;
    for each $stl$ in $STL$ do
        $H \leftarrow Hash\,(stl)$;
    for each $v_i \in values\,(H)$ do
        if $|v| \geq 2$ then
            $EM_i \leftarrow$ all $cb$ in $v_i$;
    for $i = 0$ to $length\,(EM)$ do
        for $j = 0$ to $length\,(EM_i)$ do
            $EM \leftarrow cb_j$ and $cb_{j+1}$;
```

and the number of times it occurs. Inexact matching of code blocks is detected as follows. For each code block, some syntax templates are extracted from the `STL` to form a template vector denoted by v. Two code blocks `rx` and `ry` are considered to match inexactly if the similarity between their template vectors, denoted by `sim (vx; vy)`, is within a user-specified minimum similarity threshold `minS`. The template vector is constructed as a combination of the five types of templates in the `STL`: the first type represents the container, i.e., each container is a template; the second represents the choice of algorithm in user-related code; the third represents the socket APIs; the fourth represents the use of I/O devices; and the fifth represents the encryption. To detect inexact matches, Algorithm 2 first maps template vectors to a user-specified vector length, and then maps them to a hash table. Algorithm 2 detects inexact matches by iterating through five steps:

1. Computing the median value: The first step is to compute the median of the number of template occurrences for all the code blocks. Since the number of template occurrences may be distributed over a wide range, using the median value, which is resistant to outliers, helps to separate the templates which occur more frequently.
2. Filtering templates: The second step is to filter the templates based on the median value 0 to find those which occur most often. Templates that appear only a few times within all the extracted code blocks are unimportant for the purpose of comparing code blocks. The strategy is to restrict attention to the templates that are the most significant ones for inexact matching of code blocks.

Algorithm 2: Inexact Syntax Signature Detection

```
input : CB: set of all code blocks
        T: Set of syntax templates
        $V_i$: template vector for each code
        block
        H: an empty Hash Table
        U: a user-specified length for
        Combination and Binary array
        Combination: a $U$ length array
        Binary: a $U$ length array
        $minS$ threshold
output: IM: Set of inexact match blocks
begin
    for each $t$ in $T$ do ;                      /* Step1 */
        $M \leftarrow$ Compute Median;
    for each $m$ in $M$ do ;                      /* Step2 */
        if $m = 0$ then
            remove $m$ from M;
    for $i = 0$ to $length\,(M)$- $U$ do ;        /* Step3
    */
        for $j = 0$ to 5 do
            $Combination_i\,[j] \leftarrow M\,[j]$;
    for each $cb$ in $CB$ do                      /* Step4 */
        for each $Combination$ do
            for $k = 0$ to $U$ do
                if $V\,[k] \geq Combination\,[k]$ then
                    $Binary\,[k] \leftarrow 0$;
                else
                    $Binary\,[k] \leftarrow 1$;
            Compute decimal number of Binary;
            $H_{Combination} \leftarrow$ decimal number;
    for each $cb$ in $CB$ do                      /* Step5 */
        for each $h$ in $H$ do
            $CB\prime \leftarrow$ find other code blocks in the
            same bucket;
        for each $cb\prime$ in $CB\prime$ do
            if $cb$ and $cb\prime$ occurred more than the
            $minS$ threshold then
                $EM \leftarrow cb$ and $cb\prime$;
```

3. First-level mapping: The third step is to map all templates to a set of user-specified length vectors. These new vectors group together templates having the same user-specified length; the comparison of code blocks will be based on these groups.
4. Generating binary vectors: For each code block, we

generate a binary vector by comparing the value of a template vector with the corresponding value in the median vector. If the template value is larger than the corresponding median, we insert 1 into the binary vector; otherwise, 0. The size of the binary vector is the same as that of the user-specified length vector in the previous step.

5. Hashing binary vectors (second-level mapping): Given that the binary vectors have the same size k, there are 2^k possible combinations. In this step, each code block is hashed to a bucket based on its binary vector. The inexact code blocks are identified by keeping track of the frequency of code block occurrences in the hash tables. The code blocks having more than `minS` matches are considered inexact matches.

Code Region Detection

We identify regions in the assembly code files that match the exact or inexact code blocks in the user-generated code. These regions are then divided into the regions that exactly or inexactly match the actual regions in the assembly code files.

Development of Syntax Author Profiles

For each author, we build a syntax author profile (SAP) that includes the type of semantics, the style of creating and calling functions, and the complexity of the programs. The resulting profile may be used in various authorship attribution tasks (e.g., classification or clustering). We show a sample of a syntax profile in Table 2, which is a score-boarding profile where the last column shows the number of occurrences.

3.3. Register Flow Analysis Layer (RFAL)

In this section, we introduce a novel model for binary code representation, called the Register Flow Graph (RFG), which can be used to capture programming styles based on an important semantics of the code, i.e., how registers are manipulated.

3.3.1. Register Flow Graph

This section describes the construction and use of the register flow graph. The flow and dependencies between the registers are important semantic aspects about the behavior of a program, which might indicate the author's skills or habits. Regardless of the number or degree of complexity of functions, following registers are often accessed: `ebp`, `esp`, `esi`, `edx`, `eax`, and `ecx`. Therefore, the steps involved in constructing the RFG for these registers are the following:

- Counting the number of compare instructions,
- Checking the registers for each compare instruction,
- Checking the flow of each register from the beginning until the compare is reached,
- Classifying the register changes according to the classes given in Table 3.

STL				
Category	Class	Exact	Inexact	No
Container	Pair	x		2
	Vector			
	List		x	1
	Queue			
	Stack	x		1
	Hash sets			
	Valarray		x	2
Algorithm	Sort			
	Depth-first search		x	1
	Dijkstra's algorithm			
	Minimum spanning			
	Breadth-first search		x	1
Socket	Data transfer mech			
	Options management		x	2
	Network addressing	x		3
	Connection setup	x		3
I/O	stdin	x		3
	printf	x		4
	scanf	x		2
	puts	x		1
	gets			
	getche		x	1
Encryption	TEA			
	RC4		x	2
	AES			
	MD5			
	RSA			

Table 2: Part of score-boarding profile

Class	**Arithmetic**	**Logical**	**Generic**	**Stack**
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	1	1	0	0
6	1	0	1	0
7	1	0	0	1
8	1	1	1	0
9	1	1	0	1
10	1	0	1	0
11	1	1	1	1
12	0	1	1	0
13	0	1	0	1
14	0	0	1	1
15	0	1	1	1

Table 3: Classes of register access

We classify the assembly instructions into four families: stack, arithmetic, logical operation, and generic operation families, which makes a total of 15 classes, as shown in

Table 3). The four families are explained as follows:

- Arithmetic: this class contains the following; `add`, `sub`, `mul`, `div`, `imul`, `idiv`, etc.
- Logical: this class contains the following; `or`, `and`, `xor`, `test`, `shl`.
- Generic: this class contains the following; `mov`, `lea`, `call`, `jmp`, `jle`, etc.
- Stack: this class contains `push` and `pop`.

3.3.2. Extraction of RFG

We employ an example to show how to extract the RFG. The first step is to highlight the `cmp` instructions in the `main` function. For illustration purpose, we base our discussion upon a randomly selected author from the Google jam code set [10] and by selecting part of the `main` function as follows:

```
(00411670) main
   push    ebp
   mov     ebp, esp
   sub     esp, 0D8h
   push    ebx
   push    esi
   push    edi
   lea     edi, [ebp+var_D8]
   mov     ecx, 36h
   mov     eax,0CCCCCh
     rep stosd
   mov     esi, esp
   push    offset ?tt@3HA ;
   push    offset Format   ;
   call    ds:__imp__scanf
   add     esp, 8
   cmp     esi, esp
   call    j___RTC_CheckEsp
   mov     esi, esp
   push    offset ?s@@3A ;
   call    ds:__imp__gets
   add     esp, 4
   cmp     esi, esp
```

We note that the above code fragment contains two compare instructions, so we can construct two RFGs. We filter the instructions and keep only those related to the registers in the compare instructions. The first graph is constructed as follows:

- Step 1: We keep all the instructions until the first compare.

```
(00411670) main
push    ebp
mov     ebp, esp
sub     esp, 0D8h
push    ebx
push    esi
push    edi
lea     edi, [ebp+var_D8]
mov     ecx, 36h
mov     eax,0CCCCCh
 rep stosd
mov     esi, esp
push    offset ?tt@3HA ;
push    offset Format   ;
call    ds:__imp__scanf
add     esp, 8
cmp     esi, esp
```

- Step 2: We filter these instructions by eliminating the instructions that are not related to the registers of the compare instructions, so the sequences will be:

```
1.  sub     esp, 0D8h
2.  push    esi
3.  mov     esi, esp
4.  add     esp, 8
5.  cmp     esi, esp
```

- Step 3: We construct the state graph from the previous sequences as in Fig 3(a). Nodes 1 and 4 represent arithmetic operations; referring to Table 3, the vector [1 0 0 0] is of class 1, so we replace the two nodes by one node. Nodes 2 and node 3 represent a stack operation and a normal operation, respectively; referring to Table 3, the vector [0 0 1 1] is of class 14.

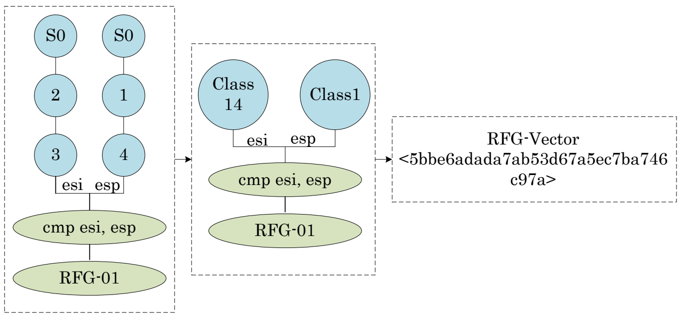

Figure 3: (a) The states of registers (`esi` and `esp`) (b) The classes of nodes in RFG-01 (c) The RFG hash vector

- Step 4: We reshape Figure 3(a) for the sake of simplicity, so the new shape is as shown in Figure 3(b).
- Step 5: We construct the RFG vector by calculating the hash value of the compare operands and their corresponding color class. For example, we have `cmp esi, esp` in Figure 3, therefore we hash this instruction for the the *esi*, *class*14, *esp*, and *class*1 classes and the hash value will be `<5bbe6adada7ab53d67a5ec7ba746c97a>` as shown in Figure 3(c).

3.3.3. Applying the RFG to Author Attribution

As an abstract intermediary representation between the ASM and the source code, the `RFG` model may be closely related to programming styles since it captures details of control statements, variable instantiations, parameter passing, etc. We now explain how to apply this model to defining signatures of author styles.

The aforementioned informal graphical descriptions provide a foundation for the concepts needed to identify stylistic signatures based on the RFG for a known author with a set of programs $P = \{p_1, \ldots, p_n\}$. We consider the set of registers $REG = \{esi, esp, \ldots\}$ supplied with a function $FReg \in REG \rightarrow TREG$, where $TREG$ represents a register type set, i.e., $TREG = \{pointer, stack, arithmetic, general, \ldots\}$.

For example, $FReg[\{esp\}] = \{stack\}$ means that *esp* is of the stack pointer type; $FReg[\{eax\}] = \{arithmetic\}$ means that *eax* is often used in arithmetic operations. We also assume that there is a set of assembly instruction classes; in Table 3, $IC = \{arithmetic, logical, normal, stack\}$ (instruction class). Accordingly, our algorithm uses the variable relations $ClrsReg \in REG \leftrightarrow Clrs$, where$Clrs$ is a set of register colors in $\mathbb{P}(IC)$, thus linking a register, and by extending a register type, with a set of assembly instruction types.

We formulate the problem of style signature identification as the problem of constructing a *valued class system* based on a dissimilarity between two *compare* (e.g., `cmp`) type instructions in the assembly code:

- Let A be an author with a set P of programs;
- For each program p, and for each *interesting* function in p, construct a *valued class system* based on the assembly compare type instructions with a specific dissimilarity function;
- Considering all authors, compute the mutual information between these classes and select, for each author, a set of representative classes. This step may be achieved with manual intervention if the selected classes for a given author are too numerous.

$FReg$ and IC are the input to our algorithm, $ClrsReg$ and $CMP \in \mathbb{P}(ClrsReg \times ClrsReg)$ – the set of the compare type instructions colors – being obtained after parsing the program. A set K in $\mathbb{P}(CMP)$ is defined as class system if the following properties are verified [2]:

C_1: $CMP \in K$,

C_2 : $\forall\, c \in CMP$, $\{c\} \in K$,

C_3 : $\forall\, A, B \in K$ with $A \cap B \neq \emptyset$, then $A \cap B \in K$.

All elements of K are called *classes*; the singletons $\{c\}$ and CMP are trivial classes. The set K is a *valued class system* if there is a function f so that for each $c \in CMP$, $f(\{c\}) = 0$ and $\forall\, A, B \in K$ with $A \subsetneq B \Rightarrow f(A) < f(B)$. In order to compute the pair (K,f) given the CMP set with the elements ordered on the assembly address offsets, we introduce the following dissimilarity function d between two assembly compare type instructions. For any consecutive, i.e., based on their offset, c_1 and $c_2 \in CMP$,

$$d(c_1, c_2) = \begin{cases} 0 & \text{if } c_1 \text{ and } c_2 \text{ involve the same types} \\ & \text{of registers and colors;} \\ 1 & \text{if } c_1 \text{ and } c_2 \text{ involve the same registers} \\ & \text{(or types) but different colors;} \\ 2 & \text{if } c_1 \text{ and } c_2 \text{ involve one common register} \\ & \text{(or type) with the same color.} \end{cases}$$

Else, c_2 forms only a singleton in our class system. First, K is initialized to the set of CMP. Whenever two compare (e.g., `cmp`) instructions, c_1 and c_2, are dissimilar (i.e., the above definition applies), a subset $\{c_1, c_2\}$ forms a class which is added to K with $f(c_1, c_2) = d(c_1, c_2)$. If three consecutive compare instructions, c_1, c_2, c_3, have common (types of) registers (i.e., $d(c_1, c_2)$ and $d(c_2, c_3)$ can be assessed), the algorithm constructs an extra class $\{c_1, c_2, c_3\}$ added to K with $f(\{c_1, c_2, c_3\}) = \max\,(d(c_1, c_2), d(c_1, c_3)) + 1$. Designed for pseudo-hierarchies, the most appropriate graphical representation of our system (K,f) is the *pyramid*. For instance, Fig. 4 depicts seven compare type instructions ($\{c_1, \ldots, c_7\}$) having led to a class system with eight trivial classes ($\{c_1, \ldots, c_7\}$, $\{c_1\}, \ldots, \{c_7\}$) and seven non trivial classes: $\{c_1, c_2\}$, $\{c_2, c_3\}$, $\{c_1, c_2, c_3\}$, $\{c_3, c_4\}$, $\{c_2, c_3, c_4\}$, $\{c_1, c_2, c_3, c_4\}$, $\{c_5, c_6\}$. In this example, $f(c_1, c_2) = 1$, meaning that c_1 and c_2 involve the same (type of) registers and colors.

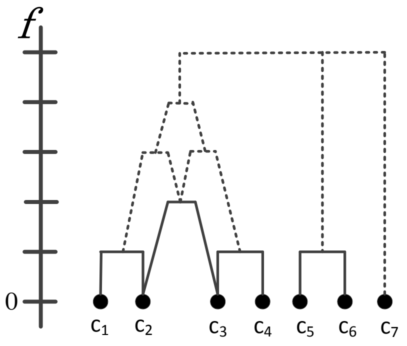

Figure 4: Dissimilarity scale

If ever c_i has a frequency $freq > 1$, it affects the pyramid by stretching the hierarchy in which it belongs with $(1/freq)$ up. Therefore, (K,f) is automatically obtained for each function and program belonging to a given author whose `RFG` style is identified as the common elements in the corresponding (K,f)s which do not overlap those belonging to other authors. Mutual information definition helps automatically identify these classes.

4. Evaluation

4.1. Google Code Jam

Google Code Jam [10] is an international programming competition hosted and administered by Google. The competition consists of a set of algorithmic problems over multiple rounds that must be solved in a fixed amount of time.

Competitors may use any programming language and development environment to obtain their solutions. Each round of the competition involves writing a program to solve a small number (usually 3 to 6) of problems. We use the 2009 and 2010 contest data in our evaluation as the one utilized by Rosenblum et al. [8].

		Code Property		
Features	**#**	**Inst.**	**Control flow**	**Ext.**
N-grams	591,698	*		
Idioms	75,277	*		
Graphlets	75,132	*	*	
Supergraphlets	14,945	*	*	
Callgraphlets	2,098		*	*
Library calls	169			*
Inexact STL)	1733	*		
Exact STL	675	*		
RFG	269	*	*	*

Table 4: The number of features for the Google Code Jam corpus

Although Code Jam solutions are not restricted to any particular programming language, we restrict our attentions to those solutions that were written in C or C++ as utilized by Rosenblum et al. In the Google Code Jam competition, each contestant submits two solutions (long and short test cases). We remove the second solution in order to be identical with the data sets used by Rosenblum et al. Table 4 summarizes binary code feature templates and the number of each instantiated one in a typical corpora for Google Code Jam.

4.2. Results and Analysis of OBA2

In this subsection, we compare OBA2 with the existing IAPB approach in terms of accuracy and false positive rate. The false positive rate comparison is shown in Figure 5. From the results, it is clear that OBA2 leads to more accurate results than IAPB. The reason is three-fold. First, the SL layer in the former eliminates style-unrelated functions, such as compiler-generated functions, run time functions, and security functions, so the false positive rate of OBA2 is reduced, while IAPB does not take into account such functions, Second, OBA2 is designed to perform on a large database of assembly code (ASM) files to automatically detect all syntax templates and updates the authors' profiles accordingly, Third, OBA2 evaluates the feasibility of detecting exact syntax templates with different levels of normalization. Figure 7 shows the impact of not using the SL layer in OBA2. We notice that the false positive rate will increase accordingly due to the mixture of style-unrelated code and style-related code. Finally, we compare in finer details the relative contribution of different features of OBA2 with those of the IAPB method in Figure 6

We can observe that the semantics-based RFG layer performs the best while the syntax-based layer (CAL) follows,

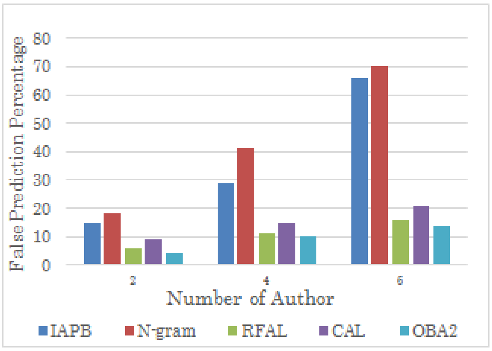

Figure 5: False positive rate comparison

both of which are superior in contrast to various features of the IAPB method. Also, we can see that the accuracy of IAPB decreases quickly when the number of authors is 5 or more, with the results below 30%. The reason is that, with more candidate authors present, the results of IAPB become less reliable since the amount of style-unrelated functions increase (e.g., it could be due to two authors using the same compiler, or even the same OS environment).

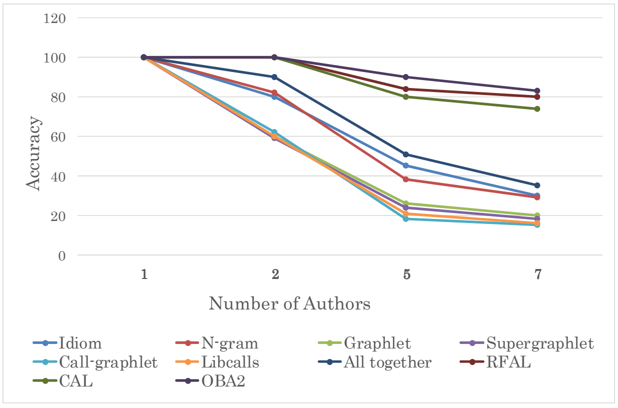

Figure 6: Comparison between different features

5. Limitations and Future Direction

The previous section shows evidences that the proposed OBA2 approach yields more accurate results than the existing IAPB method. Nonetheless, the OBA2 approach still has many limitations, including the following:

- Like most existing work, OBA2 assumes the binary code is already de-obfuscated. In practice, deobfuscation of malware can be demanding. How to conduct authorship attribution directly over obfuscated code is an important but very challenging issue.
- Again, like existing work, OBA2 requires training data with known authorship in order to collect sufficient features before the method can be applied to new code.

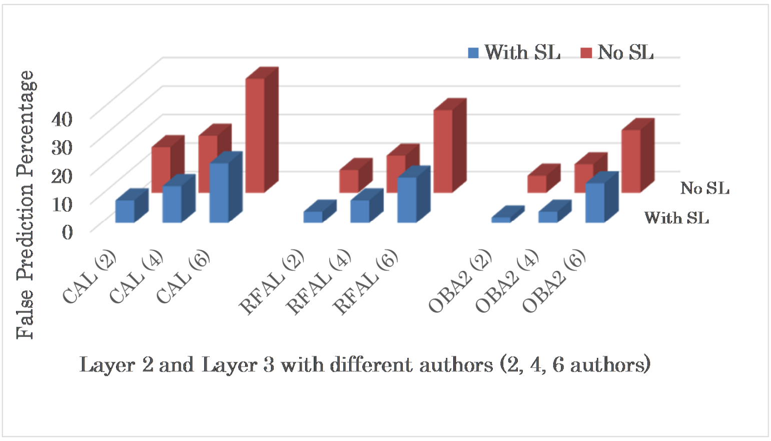

Figure 7: False positive rate comparison without the stuttering layer (SL)

- As the experimental results show, the accuracy will decrease as the number of candidate authors increases (in which regard OBA2 performs better than IAPB).
- We have not investigated the impact of different compilers in this work.
- We have not evaluated the proposed method over other programming languages.

Besides addressing the above issues, our future work will include extending the approach to characterizing, instead of identifying, the malware author and its origin, by determining information like the level of programming skills, the geographic location of the author or organization, the type of targets of the malware, the language, etc. Another possible extension would be to give OBA2 the capability of verifying its output using formal models such as STL or LTL.

6. Conclusion

The reverse engineering of malware binaries has been an important but challenging issue. In particular, authorship attribution for malware binary code has received only limited attention compared to source code authorship attribution, since most stylistic features will not survive the compilation. In this paper, we have presented a multi-layer approach to improving the accuracy of existing binary code author attribution method. We have implemented and compared the proposed method to existing research, and experiments have shown our method to yield superior results.

Acknowledgments

The authors thank the anonymous reviewers for their valuable comments. The authors are thankful to Philippe Charland from Defence Research and Development Canada for early discussions on this topic. Initial versions of this work have been funded through a research contract with DRDC Canada. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

[1] I. Bai, Y. Yang, S. Mu, and Y. Ma, *Malware Detection through Mining Symbol Table of Linux Executables*, In Knowledge and Information Systems, Springer, 2013.

[2] F. Brucker, *Modèles de classification en classes empiétantes.* Ph.D. Thesis, Dép. IASC de l'École Nationale Supérieure des Télécommunications de Bretagne, France, July 2001.

[3] H. Ding and M. H. Samadzadeh, *Extraction of Java program fingerprints for software authorship identification*,Journal of Systems and Software,Elsevier, Volume 72, Issue 1, June 2004, Pages 49-57

[4] G. Frantzeskou, S. Gritzalis, and S. G. MacDonell, *Source code authorship analysis for supporting the cybercrime investigation process*, In the First International Conference on E-business and Telecommunication Networks, Kluwer Academic Publishers: Setubal, 2004; Pages 85-92.

[5] I. Krsul, and E. H. Spafford, *Authorship analysis: identifying the author of a program*, Journal of Computers and Security, Elsevier, Volume 16, Issue 3, 1997, Pages 233-257.

[6] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, *Polymorphic worm detection using structural information of executables*, In the Proceedings of the International Conference on Rapid Advances in Intrusion Detection, (RAID 2005), Springer, Pages 207-226.

[7] S. G. MacDonell, A. R. Gray, G. MacLennan, and P.J. Sallis, *Software forensics for discriminating between program authors using case-based reasoning, feed-forward neural networks and multiple discriminant analysis*, In Proceedings of the 6th International Conference on Neural Information, 1999, vol. 1, Dunedin, New Zealand, Pages 66-71.

[8] N. E. Rosenblum, Z. Xiaojin, and P. Miller, *Who wrote this code? Identifying the authors of program binaries*, 16th European Symposium on Research in Computer Security, Lecture Notes in Computer Science, Springer, Volume 6879, 2011, Pages 172-189.

[9] IDA Pro multi-processor disassembler and debugger. Available from: `https://www.hex-rays.com/products/ida/`. Accessed on: Feb 24th 2014.

[10] The Google Code Jam. Available from: Available from: `http://code.google.com/codejam/`. Accessed on: Feb 24th 2014.