Supervised Learning for Provenance-Similarity of Binaries

Sagar Chaki, Cory Cohen, and Arie Gurfinkel
Carnegie Mellon Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA, USA
{chaki,cfc,arie}@sei.cmu.edu

ABSTRACT

Understanding, measuring, and leveraging the similarity of binaries (executable code) is a foundational challenge in software engineering. We present a notion of similarity based on provenance – two binaries are similar if they are compiled from the same (or very similar) source code with the same (or similar) compilers. Empirical evidence suggests that provenance-similarity accounts for a significant portion of variation in existing binaries, particularly in malware. We propose and evaluate the applicability of classification to detect provenance-similarity. We evaluate a variety of classifiers, and different types of attributes and similarity labeling schemes, on two benchmarks derived from open-source software and malware respectively. We present encouraging results indicating that classification is a viable approach for automated provenance-similarity detection, and as an aid for malware analysts in particular.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Algorithms, Measurement, Security

Keywords

Binary Similarity, Classification, Software Provenance

1. INTRODUCTION

Binary similarity is an important area of software engineering research. Techniques for checking binary similarity find applications in areas ranging from code clone detection [18] and software birthmarking [7], to malware analysis [20] and software virology [12]. Several notions of similarity have been proposed and studied in the literature. Our focus is the similarity, which we call *provenance-similarity*, that arises from the fact that many binaries are the result of compiling "similar" source code with "similar" compilers. The term provenance indicates that the similarity originates not only from the source of the binaries, but also from the (compilation) process by which they were derived.

The major application area for provenance-similarity is malware analysis and virology. Empirical evidence [20] suggests that the vast majority of existing malware belong to a relatively small number of "families" of related instances. For example, a 2006 Microsoft report [3] states that *over 97,000* malware variants were detected in the first six months of 2006. At the same time, according to Symantec [19] and Microsoft [3], the typical number of malware families encountered in any six month period is a *few hundred*. In fact, the Microsoft report (see Figure 1 of [20]) further indicates that the 7 and 25 most common families account for over 50% and 75%, respectively, of detected malware instances.

There are several reasons to believe that many malware families consist of provenance-similar instances. Similarity in source code stems from two factors. First, malware authors reuse old programs – it is very unlikely that the 1,131,335 (presumed) malicious executables with unique MD5 checksums collected by CERT in the final quarter of 2010 were all written from scratch in 92 days. Second, many malware families – such as Aliser [1] – generate new instances by changing a data element (e.g., an encryption key, etc.), but not other parts of the source code. In addition, similarity in generative compilers arises because a small number of related compilers are used to produce the vast majority of malware. For example, we scanned a collection of malware available within CMU CERT and found that of all the malware whose generative compilers are known with reasonable certainty, about 60% were compiled with Microsoft Visual C++, about 25% with Delphi, and most of the rest with Microsoft Visual Basic. Such limited variability in compiler and source code suggests that provenance-similarity is a pervasive feature in malware.

Current approaches to checking provenance-similarity of binaries are largely manual (e.g., forensic analysis done by a malware analyst). While good tools are available for binary disassembly and visualization (e.g., IDAPro [14]), most of the technical analysis is still manual, and, therefore, non-scalable. Automated analysis tools, when available, are either not robust (i.e., they have high false positive and false negative rates) or inefficient. Therefore, developing an automated, robust, and efficient approach for checking provenance-similarity is an important and open problem. Addressing this problem is the subject of our paper.

We begin with a definition of provenance-similarity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
KDD'11, August 21–24, 2011, San Diego, California, USA.
Copyright 2011 ACM 978-1-4503-0813-7/11/08 ...$10.00.

Provenance-similarity between two binaries implies similarity in not only the source code from which they were derived, but also in the compilers used to generate them. We define two binaries B_1 and B_2 to be *provenance-similar* if: (i) B_1 and B_2 have been compiled from the same (or very similar) source code; and (ii) B_1 and B_2 have been generated by closely related compilers (or the same compilers with different compilation flags).

Note that our definition excludes the situations where the source code of B_1 and B_2 are very different, such as when B_1 is derived from B_2 via a major patch, or when B_1 and B_2 are radically different implementations of the same functionality, e.g., merge-sort and bubble-sort. We also exclude cases where the compilers involved are very different (e.g., Visual C++ versus GCC). These problems are beyond the scope of this paper.

To understand the challenges in detecting provenance-similarity, consider Microsoft Visual C++. As a security feature to guard against buffer-overflow attacks, the compiler is able to generate extra code that checks for stack integrity at runtime, and aborts if an integrity violation is detected. However, the exact code generated depends on the compiler context, which includes factors such as the compilation flags used (e.g., `/GS` to enable the security feature), the version of the compiler, calling conventions, etc. Consider two binaries, B_1 and B_2, generated from the same source code by Microsoft Visual C++ under different contexts, and therefore containing different code for runtime buffer-overflow checks. A typical problem is to detect the provenance-similarity of B_1 and B_2 in a robust and efficient manner.

Developing a Robust and Efficient Provenance-Similarity Checker. In the general context of binary similarity, the following two main approaches have been proposed:

(a) *Syntactic* approaches compare signatures [8] or feature vectors [18, 21] extracted from the syntactic binary structure. They resemble techniques used by anti-virus tools, and thus are efficient. However, they are not robust, and are foiled by even minor syntactic differences in binaries.

(b) *Semantic* approaches compute and compare "mathematical" descriptions of the semantic meaning of binaries [11, 15]. They are precise, but require considerable manual interaction, and computationally expensive technology for symbolic and mathematical reasoning (e.g., term-rewriting, symbolic simulation, and theorem proving), and thus have limited practicability and scalability.

In this paper, we investigate the use of classification to develop a robust and efficient binary similarity checker (BSC) for provenance-similarity. Classification is particularly suitable for problems where: (i) closed form solutions are hard to develop, and (ii) a solver can be "learned" using a training set composed of positive and negative samples. Both conditions apply to provenance-similarity.

Provenance-Similarity as a Classification Problem. At a high-level, a BSC is a "black box" that accepts a pair of binaries B_1 and B_2 as input, and outputs "yes" if B_1 and B_2 are similar, and "no" otherwise. This is clearly a *binary classification* problem. However, to be effective, a classification-based BSC must be trained properly and use the right classification technology. Moreover, proper training requires good features that are efficiently computable from binaries. We address these challenges, and make the following specific contributions.

Features. First, we develop a suite of "semantic" and "syntactic" attributes (a.k.a. features) (cf. Sec. 4). Semantic features capture the effect of a binary's execution on specific components of the hardware state, viz., registers and memory locations. Syntactic features represent groups of instruction opcodes occurring contiguously in the binary. They are derived from n-grams and n-perms [20]. Second, we extend these features two pairs of binaries to capture the *difference* between two binaries, and not a single binary in isolation.

Benchmark. While malware is a major target domain for provenance-similarity, it is extremely challenging to develop good sample (i.e., training and testing) sets from malware due to the lack of information on source code and generative compilers. Our second contribution is a benchmark (cf. Sec. 5) for provenance-similarity using a benchmark derived from 14 commonly downloaded SourceForge projects, containing more than 21 million LOC.

Empirical Evaluation. Finally, we do a systematic evaluation of classification for provenance-similarity over both open-source and malware benchmarks. Our main results are: (i) we compare a suite of classifiers implemented in WEKA [22] on the open-source benchmark, and find RandomForest [4] to be the most effective; (ii) for the malware benchmark, we develop a suite of labeling schemes that do not require source-level information; we show very good classification using these labels and RandomForest; we argue that this implies applicability of classification for provenance-similarity in domains, such as malware, without reliable source-level information; (iii) we compare semantic and syntactic attributes, and show that 1-grams are almost as effective as semantic features, while being much less expensive to compute. Our experiments and results are presented in Sec. 6.

Function Similarity. We focus on similarity between binary "functions" instead of complete binaries. Intuitively, a function is a binary fragment that results from the compilation of a procedure or method at the source code level. We choose functions because: (i) they are the smallest externally identifiable units of behavior generated by compilers; (ii) similarity at the function level is desirable for further analysis; indeed, malware of similar provenance, will rarely be identical everywhere, but instead share important functions; thus, malware analysts look for common functions to judge if two binaries are related; also, information about function similarity are useful for clustering and as building blocks for measuring similarity at the binary level; and (iii) we were able to use extensive function-based malware datasets from CERT for our experimentation.

The rest of this paper is structured as follows. In Sec. 2, we survey related work. Sec. 3, we formalize our notion of functions. In Sec. 4, we present our semantic and syntactic features, and the procedures for extracting them from function descriptions. In Sec. 5, we present a benchmark derived from open-source software, and our methodology for deriving testing and training sets from the benchmark. In Sec. 6, we present our experiments and results. In Sec. 7, we conclude.

2. RELATED WORK

A number of different approaches have been proposed for binary similarity. For example, several researchers have used feature vector comparison to detect code clones [18], and to protect against malware [21]. Gao et al. [11] use

a control-flow based analysis that combines graph isomorphism, symbolic simulation, and theorem proving to detect semantic differences in binaries. Apel et al. [2] propose metrics for measuring similarity of malware. Dullien et al. [9] and Flake [10] investigate approaches to compare binaries at the level of graphs derived from their syntax. None of these approaches use supervised learning. Also, they require ultimately the selection of a threshold to decide similarity. In contrast, we use supervised learning to automatically compute a threshold that is appropriate for the target binaries.

Rosenblum et al. [17] propose the use of machine learning to extract *compiler provenance* from binaries. The goal of compiler provenance is to correctly identify the compiler that was used to generate a target binary. Rosenblum et al. use a specific type of machine learning technology, called the linear chain conditional random field. They also rely on syntactic features, known as idiom-features. The research presented in this paper is different in terms of both the problem – we consider provenance similarity between two binaries – and the solution – we explore a wide variety of classification techniques, and semantic features, to discover the most effective solution.

Hu et al. [13] present an approach to index malware based on the syntactic call-graph structure of programs. Our approach is complementary since it focuses on features extracted from semantic behavior of functions. Ye et al. [23] use clustering to categorize malware. In contrast, we focus on using classification.

Cohen et al. [8] explore the use of cryptographic hashes to study program similarity. A possible basis for semantic function signatures are "mathematical functions" extracted [15] from binaries to describe their behavior. In contrast to these signature-based approaches, our approach is based on supervised learning.

A number of researchers have developed techniques, such as alias analysis [5] and interface identification [6], for binaries. These techniques are crucial for identifying functions and their boundaries, a problem that is orthogonal to our goal of developing effective techniques for binary similarity.

3. FUNCTIONS

In this section, we present the notion of functions we use for this research. We write $Dom(X)$ to denote the domain of a mapping X. We write $X \hookrightarrow Y$ to denote a (possibly partial) mapping from X to Y with a finite domain. Intuitively, a function is a partial mapping from addresses to instructions, together with a starting address. We assume a totally ordered set $Addr$ of addresses, viz. the set of 32-bit unsigned integers, and a set $Inst$ of instructions, each comprising of an opcode, and one or more operands.

Definition 1. *A function is a pair $(Body, Start)$ where: (i) the partial mapping $Body : Addr \hookrightarrow Inst$ is its body, and (ii) $Start \in Dom(Body)$ is its starting address.*

In general, a function body is non-contiguous, and comprises of a set of instruction blocks that are dispersed throughout the binary. This is true of code generated by state-of-the-art compilers and obfuscators which relocate and reorganize code fragments in non-trivial ways. In practice, however, many functions are *contiguous*, i.e., their bodies consist of a single sequence of instructions spanning a contiguous range of addresses. For such functions, we now present a more concrete definition.

Address a	Instruction $i = Body(a)$	$\epsilon(i)$	$\pi(i)$
100026E3	push dword [ebp+0x8]	FF7508	FF7508
100026E6	lea eax,[ebp-0x208]	8D85F8FDFFFF	8D85F8FDFFFF
100026EC	push eax	50	50
100026ED	lea eax,[ebp-0x308]	8D85F8FCFFFF	8D85F8FCFFFF
100026F3	push dword 0x10006470	6870640010	6800000000
100026F8	push eax	50	50
100026F9	call [0x1000509c]	FF159C500010	FF1500000000
100026FF	add esp,0x10	83C410	83C410
10002702	mov esi,0x100078d2	BED2780010	BE00000000
10002707	push esi	56	56
10002708	call [0x10005064]	FF1564500010	FF1500000000
1000270E	mov edi,[0x1000501c]	8B3D1C500010	8B3D00000000
10002714	test eax,eax	85C0	85C0
10002716	jz 0xa	7408	7408

Figure 1: A basic block of an example function, with ϵ and π mapping.

Representing Contiguous Functions as Byte-Sequences. Let Str be the set of finite-length byte sequences (or strings), and $\bullet$ be concatenation. Let $\epsilon : Inst \mapsto Str$ and $\pi : Inst \mapsto Str$ be mappings such that: **(INJ)** ϵ is injective, and **(PIC)** $\pi(i_1) = \pi(i_2)$ iff i_1 and i_2 differ only in their address operands. Intuitively, $\epsilon(i)$ and $\pi(i)$ are the exact and position-independent representations of i, respectively. For a contiguous function $f = (Start, Body)$, let $A = \langle a_1, \ldots, a_n \rangle$ be the sequence of addresses obtained by sorting $Dom(Body)$ in increasing order, and $I = \langle i_1, \ldots, i_n \rangle$ be the sequence of instructions such that: $\forall 1 \leq k \leq n \cdot i_k = Body(a_k)$. Then, the *exact*, $f.EB$, and *position-independent*, $f.PB$, representations of f are:

$$f.EB = \epsilon(i_1) \bullet \cdots \bullet \epsilon(i_n) \quad f.PB = \pi(i_1) \bullet \cdots \bullet \pi(i_n)\,.$$

Example 1. *Figure 1 shows a basic block of a function f with starting address* 100025E2*, Body, ϵ and π. Note that while the function starts at* 100025E2*, the basic block displayed in the figure starts at* 100026E3*. There are additional addresses in the function both before and after the displayed basic block. The mapping $\epsilon(i)$ follows the Intel IA-32 instruction definitions, and, thus, satisfies* **(INJ)**. *The mapping $\pi(i)$ is obtained by replacing the bytes in $\epsilon(i)$ which represent address operands with zeros, and, thus, satisfies* **(PIC)**. *For instructions i at addresses* 100026F3*,* 100026F9*,* 10002702*,* 10002708*, and* 1000270E*, $\epsilon(i) \neq \pi(i)$ because the operand bytes corresponding to an address have been replaced with zeros in $\pi(i)$. Instructions at addresses* 100026F9*,* 10002708*, and* 1000270E *have operands which are easily identified as memory dereferences by the type of the operand, while the operands for the instructions at addresses at* 100026F3 *and* 10002702*, can only be identified as addresses by the observation that the values of these operands fall inside the defined range of addresses for the program. This heuristic approach is occasionally flawed (e.g., in invalid truncated programs), but we don't believe this detail is relevant to our results. For all remaining instructions, $\epsilon(i) = \pi(i)$ since i does not have any address operands. Finally, $f.EB$ and $f.PB$ are, respectively, the concatenation of the elements of the third and fourth columns.*

Our approach assumes a reliable way to extract functions from executables. Identifying function boundaries in binaries is a complex and open problem. This is especially true for obfuscated (i.e., packed and/or encrypted) malware. These issues are beyond the scope of this paper.

4. ATTRIBUTES

Recall that our BSC classifies function pairs into two groups – similar (the "yes" group), and dissimilar (the "no" group). We now present our approach to extract attribute vectors from a function pair (f_1, f_2). Since we are interested in the degree of similarity between f_1 and f_2, each attribute captures the difference between f_1 and f_2 along some dimension. More specifically, the attribute vector is constructed as follows:

1. We apply a function-level attribute extraction scheme $\mathcal{A}$ to extract two attribute vectors: $\vec{v_1} = \mathcal{A}(f_1)$ and $\vec{v_2} = \mathcal{A}(f_2)$. We assume that all attributes extracted by $\mathcal{A}$ are numeric.

2. We construct the attribute vector for (f_1, f_2), denoted by $\mathcal{A}^{\Delta}(f_1, f_2)$, by taking the pointwise difference of $\vec{v_1}$ and $\vec{v_2}$. That is:

$$\forall 1 \leq i \leq |\vec{v_1}| \cdot \mathcal{A}^{\Delta}(f_1, f_2)[i] = \vec{v_1}[i] - \vec{v_2}[i], .$$

   Note that $\mathcal{A}^{\Delta}$ is asymmetric, i.e., in general, $\mathcal{A}^{\Delta}(f_1, f_2) \neq \mathcal{A}^{\Delta}(f_2, f_1)$. It is possible to make $\mathcal{A}^{\Delta}$ symmetric by ordering each function pair such that $f_1.EB < f_2.EB$.

We now present two function-level attribute extraction schemes, which we use for our experiments:

1. A *semantic* approach based on the effect of executing the function on a hardware state.

2. A *syntactic* approach based on n-grams and n-perms [21] – this approach represents existing work on attribute vector extraction from binaries, and serves as a point of comparison. In Sec. 6, we show empirically that classification with semantic attributes outperforms classification with syntactic attributes.

4.1 Semantic Attributes

The semantic attribute vector of a function f, denoted by $SemAtt(f)$, is a vector of values of semantic attributes. $SemAtt(f)$ represents the effect of executing f on a state of the hardware. Moreover, semantic attributes are defined so that if $SemAtt(f_1)$ and $SemAtt(f_2)$ are similar, then the executions of f_1 and f_2 have similar effects. We assume that every function f has a control flow graph $f.CFG$, and that each execution of f follows a specific path in $f.CFG$. The process of extracting a semantic attribute vector from a path in $f.CFG$ involves simulation, sign abstraction, and counting abstraction. We now present this process in detail.

4.1.1 Extracting Attribute Vectors from Paths

Let $\mathsf{SymConst} = \{v_0, v_1, \dots\}$ be a denumerable set of symbolic constants. Let $\pm$ denote the set of signs $\{+, -\}$ and $\mathbb{N} = \{1, 2, \dots\}$ be the set of numerals representing natural numbers. Symbolic expressions, denoted by SymExp, are linear combinations of symbolic constants (with unit or zero coefficient) and integers. Formally, their syntax has the following BNF form:

$$\mathsf{SymExp} := \pm\mathsf{SymConst} \mid \pm\mathbb{N} \mid \pm\,\mathsf{SymConst} \pm \mathbb{N}$$

Symbolic State. Let $\mathsf{Reg} = \{r_0, r_1, \dots\}$ be a denumerable set of virtual registers, and $\mathsf{Mem} = \{m_0, m_1, \dots\}$ be a denumerable set of memory locations. A symbolic state is a pair (R, M) where: (i) $R : \mathsf{Reg} \hookrightarrow \mathsf{SymExp}$ maps registers to the symbolic values of their contents; and (ii) $M : \mathsf{Mem} \hookrightarrow \mathsf{SymExp} \times \mathsf{SymExp}$ maps memory locations to the symbolic values of their addresses and contents.

Sign Abstraction. The sign abstraction of a symbolic expression e, denoted by $\mathsf{Sign}(e)$, is an element of Σ, where $\Sigma = (\pm \cup \{0\}) \times (\pm \cup \{0\}) \setminus \{(0, 0)\}$. Intuitively, $\mathsf{Sign}(e)$ is obtained by removing all symbolic constants and numerals from e, and replacing any missing component with 0. Formally, it is defined as follows:

$$\mathsf{Sign}(q\ v) = (q, 0)\ \ \mathsf{Sign}(q\ n) = (0, q)\ \ \mathsf{Sign}(q\ v\ q'\ n) = (q, q')$$

where $q, q' \in \pm$, $v \in \mathsf{SymConst}$, and $n \in \mathbb{N}$. Note that $\{(0, 0)\}$ is not a valid sign abstraction, and $|\Sigma| = 8$. Sign abstraction extends to pairs of symbolic expressions in the natural manner: $\mathsf{Sign}(x, y) = (\mathsf{Sign}(x), \mathsf{Sign}(y))$.

Sign-Abstract State. Applying sign abstraction to a symbolic state $s = (R, M)$ results in the sign-abstract state $\mathsf{Sign}(s)$. Formally, $\mathsf{Sign}(s)$, is a pair of functions $(R^{\#}, M^{\#})$ such that for $f = R, M$, the following holds:

$$Dom(f) = Dom(f^{\#}) \bigwedge \forall x \in Dom(f) \cdot f^{\#}(x) = \mathsf{Sign}(f(x))$$

Counting Abstraction. Let $f : X \hookrightarrow \Sigma$ be any mapping. The counting abstraction of f, denoted by $\mathsf{Count}(f)$ is the vector of numbers defined as follows. First we order the elements of $\pm \cup \{0\}$ strictly as: $- < 0 < +$. This induces the following strict ordering on Σ:

$$(x, y) < (x', y') \iff (x' < x') \vee (x = x' \wedge y < y')$$

Let $\vec{\Sigma}$ be the vector of elements of Σ induced by the above strict ordering on Σ. Thus, $\vec{\Sigma}[1] = (-, -)$ and $\vec{\Sigma}[8] = (+, +)$. Then, $\mathsf{Count}(f)$ is the vector such that:

$$\mathsf{Count}(f)[i] = |\{x \in X \mid f(x) = \vec{\Sigma}[i]\}|$$

In other words, the i-th element of $\mathsf{Count}(f)$ is the number of elements of $Dom(f)$ that are mapped by f to the i-th element of $\vec{\Sigma}$. Note that $\mathsf{Count}(f)$ always has 8 elements.

For any pair $x = (x_1, x_2)$, let us write $\mathsf{Fst}(x)$ and $\mathsf{Snd}(x)$ to mean x_1 and x_2 respectively. Let $f : X \mapsto \Sigma \times \Sigma$ be any mapping. Then the two functions $\mathsf{Fst}(f), \mathsf{Snd}(f)$ are defined as follows:

$$\forall x \in X \cdot \mathsf{Fst}(f)(x) = \mathsf{Fst}(f(x)) \wedge \mathsf{Snd}(f)(x) = \mathsf{Snd}(f(x))$$

We write $\vec{x} \bullet \vec{y}$ to denote the concatenation of vectors $\vec{x}$ and $\vec{y}$. The counting abstraction of a sign abstract state $s^{\#} = (R^{\#}, M^{\#})$, denoted by $\mathsf{Count}(s^{\#})$, is the vector of numbers defined as follows:

$$\mathsf{Count}(s^{\#}) = \mathsf{Count}(R^{\#}) \bullet \mathsf{Count}(\mathsf{Fst}(M^{\#})) \bullet \mathsf{Count}(\mathsf{Snd}(M^{\#}))$$

In other words, we apply counting abstraction to the registers, memory addresses, and memory data, and concatenate the resulting vectors. Note that $\mathsf{Count}(f^{\#})$ always has 24 elements. Intuitively, each index of $\mathsf{Count}(f^{\#})$ corresponds to a different a semantic attribute.

Post-Condition. A path p in $f.CFG$ is a sequence of assembly instructions. The post-condition of p, denoted by $\mathsf{Post}(p)$, is a symbolic state representing the hardware configuration after the execution of p. The exact value of $\mathsf{Post}(p)$ depends on the semantics of the instructions in p. In our experiments, we rely on the ROSE [16] infrastructure to compute $\mathsf{Post}(p)$.

EXAMPLE 2. *Consider the path p with three instructions* `inc ax`, `inc ax`, *and* `mov [bx] ax`. *Thus, p increments register* `ax` *twice, and then copies the contents of* `ax` *into the memory location whose addresses is stored in* `bx`. *Then,* $\mathsf{Post}(p) = (R, M)$ *where:*

$$R(r_0) = +v_0 + 2 \qquad R(r_1) = +v_1$$
$$M(m_0) = (+v_1, +v_0 + 2)$$

Note that r_0 and r_1 represent registers `ax` *and* `bx`, *respectively. Also, v_0 and v_1 are the initial values of* `ax` *and* `bx`, *respectively. Finally, m_0 is the memory location whose address is stored in* `bx`. *Also,* $\mathsf{Sign}(\mathsf{Post}(p)) = (R^{\#}, M^{\#})$ *where:*

$$R^{\#}(r_0) = (+,+) \qquad R^{\#}(r_1) = (+,0)$$
$$M^{\#}(m_0) = ((+,0),(+,+))$$

Recall that $\vec{\Sigma}[7] = (+,0)$ and $\vec{\Sigma}[8] = (+,+)$. Therefore, $\mathsf{Count}(\mathsf{Sign}(\mathsf{Post}(p))) = \vec{x} \bullet \vec{y} \bullet \vec{z}$, *where:*

$$\vec{x}[7] = \vec{x}[8] = \vec{y}[7] = \vec{z}[8] = 1 \ ,$$

and all other elements of $\vec{x}$, $\vec{y}$, and $\vec{z}$ are zero.

Extracting Attribute Vectors from Paths. The semantic attribute vector of a path p, denoted by $SemAtt(p)$, is defined as follows:

$$SemAtt(p) = \mathsf{Count}(\mathsf{Sign}(\mathsf{Post}(p)))$$

4.1.2 Extracting Attribute Vectors from Functions

Let f be a function represented by the byte sequence $f.EB$. The attribute vector $SemAtt(f)$ is constructed from $f.EB$ as follows:

1. $f.EB$ is parsed and the control flow graph $f.CFG$ is constructed. In our implementation, we use the ROSE [16] infrastructure to perform this step.
2. In general, $f.CFG$ has cycles, and therefore infinitely many paths. We use a bounded-depth-first-search traversal to extract $\#p$ paths of depth at most $\#d$ from $f.CFG$. The search is randomized, i.e., successors for traversal are picked randomly. For our experiments, we use $\#p = 50$ and $\#d = 50$. Note that since the number of possible paths is exponential in $\#d$, we limit $\#p$ as well. Let P be the set of paths extracted.
3. From each $p \in P$, we construct $SemAtt(p)$ as defined in the previous section. Note, $|SemAtt(p)| = 24$.
4. Let $\uplus$ be the element-wise addition of two vectors, i.e.,

$$\forall i \cdot (\vec{x} \uplus \vec{y})[i] = \vec{x}[i] + \vec{y}[i]$$

Then, we compute $SemAtt(f)$ as follows:

$$SemAtt(f) = \biguplus_{p \in P} SemAtt(p)$$

Increasing $\#p$ or $\#d$ also increases the time to extract semantic attributes. Empirically, we found $\#p = 50$ and $\#d = 50$ to be a good tradeoff. For example, to extract semantic attributes from a random sample of 541 functions in our benchmark using $\#p = 50$ and $\#d = 50$, the time required is 95 mins. When $\#d$ is increased to 100, the time is 165 mins, a 74% jump. However, the number of attributes for which values differ between the two cases is only about 1.2% of the total number of attributes.

EXAMPLE 3. *Let f be a function such that $f.CFG$ has two paths p_1 and p_2. Let p_1 be the same as p from Example 2. Therefore,* $SemAtt(p_1) = \mathsf{Count}(\mathsf{Sign}(\mathsf{Post}(p_1))) = \vec{x} \bullet \vec{y} \bullet \vec{z}$, *where:*

$$\vec{x}[7] = \vec{x}[8] = \vec{y}[7] = \vec{z}[8] = 1 \ ,$$

and all other elements of $\vec{x}$, $\vec{y}$, and $\vec{z}$ are zero.

Let p_2 have 4 instructions `mov [ax] bx`, `dec ax`, `mov bx 10`, `mov [ax] bx`. *Then,* $\mathsf{Post}(p_2) = (R, M)$ *where:*

$$R(r_0) = +v_0 - 1 \qquad R(r_1) = +10$$
$$M(m_0) = (+v_0, +v_1) \qquad M(m_1) = (+v_0 - 1, +10)$$

Note that r_0 and r_1 represent registers `ax` *and* `bx`, *respectively. Also, v_0 and v_1 are the initial values of* `ax` *and* `bx`, *respectively. The addresses of memory locations m_0 and m_1 are stored in* `ax` *initially and finally, respectively. Also,* $\mathsf{Sign}(\mathsf{Post}(p_2)) = (R^{\#}, M^{\#})$ *where:*

$$R^{\#}(r_0) = (+,-) \qquad R^{\#}(r_1) = (0,+)$$
$$M^{\#}(m_0) = ((+,0),(+,0)) \qquad M^{\#}(m_1) = ((+,-),(0,+))$$

Recall that $\vec{\Sigma}[5] = (0,+)$ and $\vec{\Sigma}[6] = (+,-)$. Therefore, $SemAtt(p_2) = \mathsf{Count}(\mathsf{Sign}(\mathsf{Post}(p_2))) = \vec{x} \bullet \vec{y} \bullet \vec{z}$, *where:*

$$\vec{x}[5] = \vec{x}[6] = \vec{y}[6] = \vec{y}[7] = \vec{z}[5] = \vec{z}[7] = 1 \ ,$$

and all other elements of $\vec{x}$, $\vec{y}$, and $\vec{z}$ are zero. Finally, $SemAtt(f) = SemAtt(p_1) \uplus SemAtt(p_2) = \vec{x} \bullet \vec{y} \bullet \vec{z}$, *where:*

$$\vec{x}[5] = \vec{x}[6] = \vec{x}[7] = \vec{x}[8] = 1 \qquad \vec{y}[6] = 1$$
$$\vec{z}[5] = \vec{z}[7] = \vec{z}[8] = 1 \qquad \vec{y}[7] = 2$$

and all other elements of $\vec{x}$, $\vec{y}$, and $\vec{z}$ are zero.

Implementation Details. For our empirical evaluation, we generalize semantic attributes as follows. First, we split registers into three sub-categories: general-purpose, segment, and flag – based on their use by the compiler. We apply sign and counting abstractions separately to each category. Since now there are 3 types of registers in addition to memory addresses and data, $|SemAtt(p)| = |\Sigma| \times (3+2) = 40$ for any path p. Second, we consider three ways of combining the elements of $SemAtt(p)$ to obtain an element of $SemAtt(f)$ – sum (as described above), minimum, and maximum. We also consider splitting up a path at call-sites, and replacing a counting abstraction with a 0-1 abstraction (i.e., replacing all counts greater than 0 with 1). In all, we have 36 possible combinations, leading to $40 \times 36 = 1440$ attributes.

4.2 Syntactic Attributes

We present two flavors of syntactic attributes, based on n-grams and n-perms [20], respectively. Let Mnem be the sequence of x86 instruction mnemonics, and Mnem^n be the set of sequences of n elements drawn from Mnem. Let $\vec{\mathsf{Mnem}}^n$ be the vector of elements of Mnem^n ordered lexicographically. For any positive value of n, the n-gram vector of f, denoted by $NGram_n(f)$, is computed as follows:

1. Parse $f.EB$ to construct of sequence of assembly instructions $I = \langle i_1, \ldots, i_k \rangle$. Extract the mnemonic from each instruction to obtain a sequence of mnemonics $M = \langle m_1, \ldots, m_k \rangle$. In our implementation, this step is performed using ROSE.

2. Then, $NGram_n(f)$ is the vector of $|\vec{\mathsf{Mnem}}^n|$ numbers, such that for any $1 \leq i \leq |\vec{\mathsf{Mnem}}^n|$, $NGram_n(f)[i]$ equals the number of times $\vec{\mathsf{Mnem}}^n[i]$ occurs as a subsequence of M.

EXAMPLE 4. *Let* $\mathsf{Mnem} = \{\mathtt{inc}, \mathtt{dec}, \mathtt{mov}\}$. *Then,* $\mathsf{Mnem}^2 = \{\mathtt{inc}\cdot\mathtt{inc}, \ldots, \mathtt{mov}\cdot\mathtt{mov}\}$. *Let f be a function such that $f.EB$ leads to the following sequence of mnemonics:* $M = \langle \mathtt{mov}, \mathtt{inc}, \mathtt{inc}, \mathtt{mov}, \mathtt{dec}, \mathtt{mov}\rangle$. *For any $x \in \mathsf{Mnem}^2$, let us write $NGram_2(f)[x]$ to mean $NGram_2(f)[i]$ such that $\vec{\mathsf{Mnem}}^2[i] = x$. Then,*

$$NGram_2(f)[\mathtt{mov}\cdot\mathtt{inc}] = NGram_2(f)[\mathtt{inc}\cdot\mathtt{inc}] = 1$$
$$NGram_2(f)[\mathtt{inc}\cdot\mathtt{mov}] = NGram_2(f)[\mathtt{mov}\cdot\mathtt{dec}] = 1$$
$$NGram_2(f)[\mathtt{dec}\cdot\mathtt{mov}] = 1$$

and all other elements of $NGram_2(f)$ are 0.

For any positive value n, the n-perm vector of f is denoted by $NPerm_n(f)$. n-perm vectors are similar to n-gram vectors, except that ordering is ignored.

EXAMPLE 5. *For the* Mnem *and f in Example 4, we have,*

$$NPerm_2(f)[\mathtt{inc}\cdot\mathtt{mov}] = NPerm_2(f)[\mathtt{dec}\cdot\mathtt{mov}] = 2$$
$$NPerm_2(f)[\mathtt{inc}\cdot\mathtt{inc}] = 1$$

and all other elements of $NPerm_2(f)$ are 0.

Detailed evaluation of our attributes is presented in Sec. 6.2.

5. TRAINING AND TESTING SETS

Since our training and testing sets are constructed using the same procedure, we refer to them simply as sample sets. A (N^+, N^-)-sample set – consisting of N^+ "yes" samples and N^- "no" samples – is constructed as follows:

1. First, we construct a benchmark $\mathcal{B}$ comprising of sets of provenance-similar functions.
2. Next, using $\mathcal{B}$, we create N^+ and N^- random samples belonging to "yes" and "no" classes respectively. The final result is the union of all the samples obtained.

We now describe these two steps in more detail.

5.1 Benchmark Construction

Our benchmark $\mathcal{B}$ consists of a set of clusters, where each cluster is a set of provenance-similar and contiguous functions. $\mathcal{B}$ was constructed as follows:

1. We manually selected a set of 14 open-source software packages (containing collectively over 21 million LOC) available at SourceForge (`http://www.sf.net`) that are among the most downloaded, written in C/C++, and executable on Windows. Each software yielded a set of clusters by the following steps.
2. The software was compiled with Microsoft Visual Studio 2003 .NET, 2005, and 2008 on Windows XP SP3.
3. The binaries were processed with IDAPro 5.6 [14] together with custom Python extensions to extract a set of functions. We write $f.Name$ to mean the compiler-generated name of a function f. We use this name to relate functions across the different compilers, as described in the next step.

Table 1: Benchmark summary: KLoC = Kilo Lines of Code; BSz = size of binaries in MB; #Cl = no. of equivalence classes; Avg, StD, Med = mean, standard deviation and median of equivalence class sizes.

Software	KLoC	BSz	#Cl	Avg	StD	Med
`7zip`	153	2.4	2992	4.4	9.4	3
`cppunit`	36	26.4	616	15.2	128.4	3
`flac`	78	5.1	269	4.4	8.4	3
`net-snmp`	395	5.9	1507	4.6	62.4	3
`notecase`	48	0.8	228	4.7	7.8	3
`NppExec`	35	3.8	888	4.1	6.6	3
`ogl`	285	3.0	1205	2.6	17.7	2
`poco`	268	4.8	3745	5.1	61.5	3
`speed`	154	0.3	376	2.9	3.9	3
`tcl`	251	8.9	872	2.9	1.0	3
`tightvnc`	138	3.7	1455	5.4	48.9	3
`tinyxml`	7	1.4	639	4.2	18.9	3
`ultravnc`	183	6.0	1414	7.4	33.9	2
`wincvs`	154	27.3	1646	9.8	63.8	3

4. Recall that $f.PB$ is the position-independent byte-representation of a function f. The functions were clustered into equivalence classes induced by the reflexive-transitive closure of the following relation $\mathcal{R}$:

$$f_1 \mathcal{R} f_2 \iff f_1.Name = f_2.Name \vee f_1.PB = f_2.PB$$

Thus, we assume that if $f_1 \mathcal{R} f_2$, then f_1 and f_2 are provenance-similar. Alternatively, we assume that two provenance-dissimilar functions have different names *and* position-independent byte representations.

5. An equivalence class c was discarded if it satisfied the following condition:

$$|c| = 1 \bigvee \mathrm{Max}_{f\in c}|f.EB| < 50$$

This eliminates equivalence classes that are: (i) singleton – since, as we see later, singletons are not useful in generating "yes" samples; and (ii) have no functions with bodies larger than 50 bytes – since we believe that classification is not applicable to detect similarity between very small functions. The cutoff of 50 was chosen empirically.

6. Each remaining equivalence class yields a cluster.

Table 1 summarizes our benchmark $\mathcal{B}$. A median of (mostly) 3 indicates that the most common case is, as expected, one function for each of the 3 compilers. The number of equivalence classes, averages and standard deviations vary widely, indicating that we have a good mix of clusters in terms of their sizes.

5.2 Sample Set Construction

We now present our approach to construct a (N^+, N^-)-sample set from $\mathcal{B}$. When constructing (either "yes" or "no") samples, we ignore function pairs (f_1, f_2) such that $f_1.PB = f_2.PB$[1], since, by assumption, such function pairs

[1]Without this restriction, the overall trend in our results is similar, but with even higher F-measures.

are already provenance-similar, and hence need not be classified. In the following, Ψ is a scheme to extract feature vectors from function pairs, as described in Sec. 4.

First, to construct a "yes" sample, we use procedure **PickYes**, consisting of the following steps: (i) Randomly pick a cluster $c \in \mathcal{B}$. Recall that c contains at least two functions. (ii) Randomly pick two distinct functions $f_1, f_2 \in c$ s.t. $f_1.PB \neq f_2.PB$. (iii) Output $\Psi(f_1, f_2)$ labeled with "yes". Next, to construct a "no" sample, we use procedure **PickNo**, consisting of the following steps: (i) Randomly pick two distinct clusters $c_1, c_2 \in \mathcal{B}$. (ii) Randomly pick two functions $f_1 \in c_1$ and $f_2 \in c_2$. (iii) Output $\Psi(f_1, f_2)$ labeled with "no". Finally, to construct the desired (N^+, N^-)-sample set, we repeat **PickYes** N^+ times, **PickNo** N^- times, and collect together the resulting samples.

The benchmark $\mathcal{B}$ consists of over six million functions, divided into 17,852 clusters. Therefore, we have over 36×10^{12} possible samples available for training and testing. In our experiments, we use sample sets consisting of an equal proportion of "yes" and "no" samples. Thus, we say "(n_1, n_2)-classification" to mean a classification with a $(\frac{n_1}{2}, \frac{n_1}{2})$-training set, and a $(\frac{n_2}{2}, \frac{n_2}{2})$-testing set.

Since our testing and training sets are randomly created, we repeat our experiments several times, and use the averages of the measurements for our conclusions. Specifically, for any metric M, we say "M of $(n_1 \diamond k_1, n_2 \diamond k_2)$-classification" to mean the average of the values of M for $k_1 \times k_2$ classifications, done by: (i) selecting k_1 random training sets of size n_1, and for each training set, (ii) learning a classifier and testing it with k_2 random testing sets of size n_2. We use three main metrics for our evaluation – the F-measure, training time, and testing time.

6. EXPERIMENTAL RESULTS

We performed three types of experiments: learner selection, feature selection, and to judge the applicability of our approach to detect similarity in malware. All experiments were performed on a quad-core 2.8 GHz machine. Each session (training or testing) was run with a time limit of 1800s and a memory limit of 700MB. Our benchmark and tools are available at `http://www.contrib.andrew.cmu.edu/user/schaki/binsim`.

6.1 Learner Selection

These experiments were done with the semantic attributes – i.e., with $\Psi = SemAtt^{\Delta}$ – on our open-source benchmark $\mathcal{B}$. We first evaluated 24 classifiers implemented in the WEKA [22] tool (version 3.6.2). For each classifier, we used the default WEKA configuration, and measured F for a $(10000 \diamond 5, 20000 \diamond 5)$-classification. We observed a clear separation between the classifiers: seven are effective – having F-measures ≥ 0.875; the rest are ineffective – having F-measures around ≈ 0.5 (like random guessing). The results for the 7 effective schemes, in order of decreasing F-measure, are summarized in Table 2.

Comparing Effective Schemes. Next, we compared the performance of the 7 effective schemes. The IBk and J48graft schemes are eliminated immediately. They have worse performance – lower F-measure, and higher training and testing times – compared to RandomForest. We evaluated each of the remaining 5 schemes using a series of $(N \diamond 5, 2N \diamond 5)$-classifications, where the value of N was varied from 10000 to 30000 in increments of 2000. The results

Table 2: Comparison of various classifiers on a $(10000 \diamond 5, 20000 \diamond 5)$-classification. F-measure = avg/stdev; Train = training time (avg/stdev in secs); Test = testing time (avg/stdev in secs).

Classifier	F-measure	*Train*	*Test*
RandomForest	0.928/0.002	19.41/0.23	4.56/0.075
J48graft	0.909/0.002	97.96/6.28	6.52/0.301
J48	0.900/0.003	93.74/4.95	3.34/0.056
Ridor	0.895/0.005	266.7/27.1	2.79/0.045
REPTree	0.887/0.003	22.15/0.64	3.08/0.044
RandomTree	0.876/0.003	5.72/0.13	3.89/0.123
IBk	0.861/0.003	164.6/2.97	459.0/26.6

are summarized in Figure 2. All classifiers show increasing F-measures with increasing N. However, RandomForest is the clear winner. All classifiers also require more testing and training time with increasing N. In particular, Ridor's training is most expensive, and it times out for training sets of size 24,000 and up. Overall, we conclude that RandomForest is the most effective classifier.

RandomForest. Next, we evaluated RandomForest by varying its three parameters – **(P1)** number of trees, **(P2)** number of attributes, and **(P3)** tree depth.

First, we performed $(N \diamond 5, 2N \diamond 5)$-classifications with increasing **P1**, and with N ranging from 5 to 150. Figure 3-(top) summarizes the F-measures we observed. The F-measure improves rapidly with increasing **P1** up to $N = 20$, then improves very slowly till $N = 40$. For $N > 40$, the improvement tapers off, and is difficult to ascertain.

Next, we performed $(N \diamond 5, 2N \diamond 5)$-classifications with increasing **P2**, and with different N. Figure 3-(bottom) summarizes the F-measures we observed. As expected, the F-measure improves with increasing **P2**, but the gains taper off after about 12.

Finally, we performed $(N \diamond 5, 2N \diamond 5)$-classifications with increasing **P3** – starting at 0 (which indicates unlimited depth), and moving up to 80 in increments of 10 – and with different N. We saw that **P3** ≥ 20 is required for an F-measure within 0.1 of the F-measure with unlimited **P3**. However, at **P3** $= 20$, the training and testing times are similar to those with unlimited **P3**. Therefore, unlimited **P3** is the optimal choice.

In all cases, training and testing times increase monotonically with the values of **P1**, **P2**, **P3**, and N.

6.2 Feature Selection

Next, we compared $SemAtt^{\Delta}$, ${NGram_n}^{\Delta}$ and ${NPerm_n}^{\Delta}$ for $n = \{1, 2\}$ using our open-source benchmark $\mathcal{B}$. We evaluated their performances using a series of $(N \diamond 5, 2N \diamond 5)$-classifications, where the value of N was varied from 10000 to 30000 in increments of 2000. Table 3 shows our results. We see that ${NGram_1}^{\Delta} (= {NPerm_1}^{\Delta})$ is superior to $SemAtt^{\Delta}$ in terms of training and testing times, and almost as good in terms of F-measure. Feature extraction with ${NGram_1}^{\Delta}$ is also less expensive than $SemAtt^{\Delta}$ (e.g., about one day vs. a week for our open-source benchmark). A combination of $SemAtt^{\Delta}$ and ${NGram_1}^{\Delta}$ improves negligibly over $SemAtt^{\Delta}$ alone.

As expected, classification with ${NGram_2}^{\Delta}$ and ${NPerm_2}^{\Delta}$ is slower than with ${NGram_1}^{\Delta}$ due to a greater number of attributes. However, surprisingly, it is

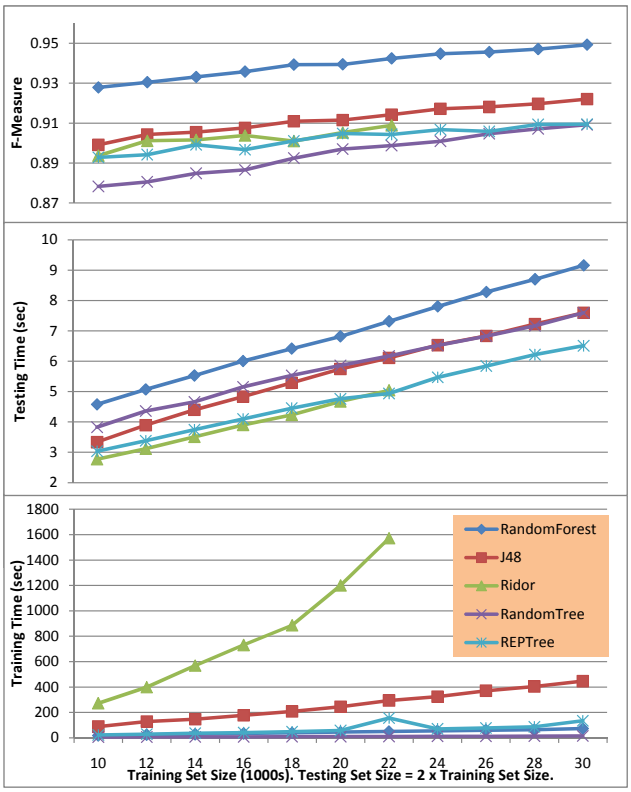

Figure 2: Comparison of effective classifiers; legend at the bottom applies to all charts.

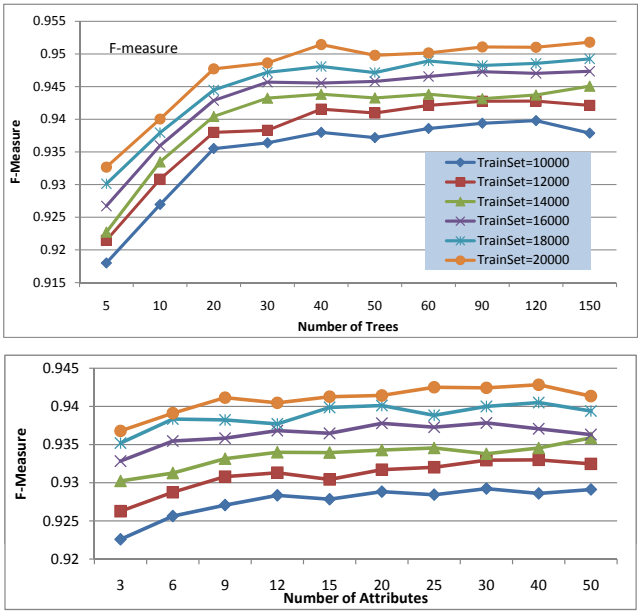

Figure 3: Performance of RandomForest with different tree numbers (top) and attribute numbers (bottom); legend at the top applies to both charts.

also worse in terms of F-measure. This indicates that counts of pairs of adjacent instruction mnemonics characterize a function less well than the counts of single mnemonics. We believe that one reason for this is instruction reordering due to compiler optimizations and changes in source code, the two sources of provenance similarity. The same set of instructions, upon reordering, will lead to different $NGram_2{}^{\Delta}$ and $NPerm_2{}^{\Delta}$, but the same $NGram_1{}^{\Delta}$. Overall, we believe $NGram_1{}^{\Delta}$ is the best choice if time is limited, and $SemAtt^{\Delta}$ is best otherwise.

Recall that $\mathcal{A}^{\Delta}$ computes the element-wise difference of function-level attribute vectors. We also evaluated $\mathcal{A}^{\Delta}$ against another approach – denoted by $\mathcal{A}^{\bullet}$ – that simply concatenates the two function-level vectors. Specifically,

$$\mathcal{A}^{\bullet}(f_1, f_2) = \mathcal{A}(f_1) \bullet \mathcal{A}(f_2) \qquad [\bullet \text{ denotes concatenation}]$$

Note that $\mathcal{A}^{\bullet}(f_1, f_2)$ has more attributes than $\mathcal{A}^{\Delta}(f_1, f_2)$, and captures information about the difference between f_1 and f_2 less directly. Empirically, we found $SemAtt^{\Delta}$ to be superior to $SemAtt^{\bullet}$, as shown in Table 3.

6.3 Applicability to Malware

To judge the applicability of classification to detect similarity in malware, we experimented with a benchmark $\mathcal{M}$ derived from 12 malware families from the CERT database. To ensure that we only consider true families, we picked a dozen candidates from cases on which a suite of anti-virus tools all agree in their family identification. The details of the families are presented in Table 4.

The main problem with malware is designing a good labeling function for training provenance-similarity. Since source-level information is rarely available, it is difficult to automatically decide if two functions are similar (or not) with high confidence. While the labeling problem for malware is beyond the scope of our paper, as a starting point, we experimented with a labeling function – denoted L^{PIC} – based on position-independent (PIC) byte representations. Specifically, L^{PIC} labels an attribute vector derived from a function pair (f_1, f_2) "yes" iff f_1 and f_2 have the same PIC byte representation. We don't claim that L^{PIC} is the best possible labeling function for malware similarity. However, we believe that it is a good approximation to the extent that if classification with L^{PIC} is not accurate, it will likely be inaccurate with a better labeling function as well.

We created sample sets using our benchmark $\mathcal{M}$ and L^{PIC}, and performed a series of $(N \diamond 5, 2N \diamond 5)$-classifications, where the value of N was varied from 10000 to 30000 in increments of 2000. We repeated our experiments with

Table 4: Malware families in our benchmark; #Ehashes and #Phashes = no. of unique exact and position-independent function byte representations.

Name	#Files	#Funcs	#Ehashes	#Phashes
F1	9	6631	1889	487
F2	265	80831	15512	1766
F3	308	9360	2363	1070
F4	7	2410	1053	704
F5	10	1090	112	112
F6	38	5159	3053	346
F7	22	10887	4384	1790
F8	27	10815	431	395
F9	18	1259	1055	581
F10	9	792	180	62
F11	24	49318	19241	1251
F12	6	2400	416	102

Table 3: Comparison of $SemAtt^{\Delta}$, ${NGram_n}^{\Delta}$, and $SemAtt^{\bullet}$; $SA + NG_1 = SemAtt + NGram_1$; N = size of training set; Tr = Avg. training time (s); Te = Avg. testing time (s).

N	$SemAtt^{\Delta}$			${NGram_1}^{\Delta}$			$(SA+NG_1)^{\Delta}$			${NGram_2}^{\Delta}$			${NPerm_2}^{\Delta}$			$SemAtt^{\bullet}$		
	F	Tr	Te	F	Tr	Te	F	Tr	Te	F	Tr	Te	F	Tr	Te	F	Tr	Te
10K	0.929	19	4.6	0.919	17	3.3	0.930	22	4.8	0.884	90.9	5.5	0.880	83	5.1	0.905	39	9.1
12K	0.930	25	5.1	0.924	22	3.5	0.933	29	5.3	0.890	118	6.1	0.889	98	5.6	0.909	52	10.5
14K	0.933	28	5.5	0.928	25	3.8	0.935	33	5.8	0.894	145	6.8	0.894	120	6.2	0.914	58	11.9
16K	0.936	34	6.0	0.931	28	4.1	0.939	40	6.3	0.900	155	7.2	0.899	125	6.7	0.918	70	13.2
18K	0.939	40	6.4	0.935	33	4.4	0.943	46	6.8	0.904	185	7.8	0.904	158	7.2	0.922	82	14.6
20K	0.939	45	6.8	0.937	37	4.6	0.944	56	7.3	0.908	211	8.4	0.907	171	7.8	0.926	94	16.0
22K	0.942	50	7.3	0.938	40	4.8	0.947	60	7.8	0.913	233	9.4	0.912	190	8.3	0.926	99	17.5
24K	0.945	55	7.8	0.942	46	5.0	0.947	67	8.3	0.915	261	9.8	0.914	224	9.2	0.928	112	18.7
26K	0.946	60	8.3	0.944	49	5.2	0.949	73	8.8	0.918	292	10.0	0.918	236	9.4	0.932	125	20.0
28K	0.947	63	8.7	0.945	51	5.4	0.950	84	9.4	0.922	310	10.6	0.920	257	9.6	0.935	138	21.6
30K	0.949	73	9.2	0.948	58	5.7	0.952	89	9.8	0.924	362	11.3	0.924	285	10.1	0.936	146	23.0

$SemAtt^{\Delta}$ and ${NGram_1}^{\Delta}$. In all cases, we got a high F-measure (0.998), indicating that classification is a promising approach for provenance-similarity in malware.

7. CONCLUSION

In this paper, we explore the use of classification to detect provenance-similarity of binaries. Specifically, we use classification to predict if a pair of functions is "similar" or "dissimilar". Our classification uses features that capture the difference between the target function pair. We evaluate our approach on a benchmark derived from open-source software, with encouraging results. Preliminary experiments also indicate that classification is a promising approach to detect malware similarity. A major challenge with malware is the difficulty of obtaining sufficiently many samples with reliable similarity labeling. We believe that semi-supervised learning is a promising avenue in addressing this issue.

8. REFERENCES

[1] Aliser worm. `http://www.sophos.com/security/analyses/viruses-and-spyware/w32aliserdam.html`.

[2] M. Apel, C. Bockermann, and M. Meier. Measuring similarity of malware behavior. In *Proc. of LCN*, 2009.

[3] M. Braverman, J. Williams, and Z. Mador. Microsoft security intelligence report: January–June 2006, 2006. `http://microsoft.com/downloads/details.aspx?FamilyId=1C443104-5B3F-4C3A-868E-36A553FE2A02`.

[4] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[5] D. Brumley and J. Newsome. Alias analysis for assembly. Technical report CMU-CS-06-180[R], Carnegie Mellon University, Pittsburgh, 2006.

[6] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. Technical report UCB/EECS-2009-133, University of California, Berkeley, Berkeley, CA, October 2009.

[7] S. Choi, H. Park, H. il Lim, and T. Han. A Static Birthmark of Binary Executables Based on API Call Structure. In *Proc. of ASIAN*, 2007.

[8] C. Cohen and J. Havrilla. Function Hashing for Malicious Code Analysis, 2009. `www.cert.org/research/2009research-report.pdf`.

[9] T. Dullien and R. Rolles. Graph-based comparison of Executable Objects. In *Proc. of SSTIC*, 2005.

[10] H. Flake. Structural Comparison of Executable Objects. In *Proc. of DMIVA*, 2004.

[11] D. Gao, M. K. Reiter, and D. X. Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Proc. of ICICS*, 2008.

[12] M. Hayes, A. Walenstein, and A. Lakhotia. Evaluation of malware phylogeny modelling systems using automated variant generation. *Journal in Computer Virology (JCV)*, 5(4):335–343, November 2009.

[13] X. Hu, T. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proc. of CCS*, 2009.

[14] IDA Pro. `http://www.hex-rays.com/idapro`.

[15] R. Linger, S. Prowell, and K. Sayre. Computing the behavior of malicious code with function extraction technology. In *Proc. of CSIIRW*, 2009.

[16] ROSE. `http://rosecompiler.org`.

[17] N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. In *Proc. of PASTE*, 2010.

[18] A. Sæbjørnsen, J. Willcock, T. Panas, D. J. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proc. of ISSTA*, 2009.

[19] Symantec. Symantec internet security threat report: Trends for January 06–June 06, 2006. `http://www.symantec.com/enterprise/threatreport/index.jsp`.

[20] A. Walenstein and A. Lakhotia. The Software Similarity Problem in Malware Analysis. In *Duplication, Redundancy, and Similarity in Software*, volume 06301 of *Dagstuhl Seminar Proceedings*, 2007.

[21] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia. Exploiting Similarity Between Variants to Defeat Malware: "Vilo" Method for Comparing and Searching Binary Programs. In *Proc. of BLACKHAT DC*, 2007.

[22] WEKA website. `http://www.cs.waikato.ac.nz/ml/weka`.

[23] Y. Ye, T. Li, Y. Chen, and Q. Jiang. Automatic malware categorization using cluster ensemble. In *Proc. of KDD*, 2010.