Binary Function Clustering using Semantic Hashes

Wesley Jin, Sagar Chaki, Cory Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan
Carnegie Mellon University, Pittsburgh, PA, USA

***Abstract*—The ability to identify semantically-related functions, in large collections of binary executables, is important for malware detection. Intuitively, two pieces of code are similar if they have the same effect on a machine's state. Current state-of-the-art tools employ a variety of pairwise comparisons (e.g., template matching using SMT solvers, Value-Set analysis at critical program points, API call matching, etc.) However, these methods are unscalable for clustering large datasets, of size N, since they require $O(N^2)$ comparisons. In this paper, we present an alternative approach based upon "hashing". We propose a scheme that captures the semantics of functions as *semantic hashes*. Our approach treats a function as a set of features, each of which represent the input-output behavior of a basic block. Using a form of locality-sensitive hashing known as *MinHashing*, functions with many common features can be quickly identified, and the complexity of clustering is reduced to $O(N)$. Experiments on functions extracted from the CERT malware catalog indicate that we are able to cluster closely-related code with a low false positive rate.**

***Keywords*-semantic comparison; malware detection; clustering; reverse engineering; binary static analysis;**

I. Introduction

Malware reverse engineers are bombarded with an overwhelming number of unique files each day. For example, the CERT malicious code team at the Software Engineering Institute (SEI) receives nearly a million binaries (i.e., executable files) each month [4], which must be processed, analyzed and indexed. It is estimated that approximately half of these samples are identical to, or variants of, malware that have been previously cataloged [4]. Thus, the ability to automatically detect similarity among binaries enables the faster removal of duplicates, streamlining the malware index. Furthermore, identifying related binaries is an important and manual process. Effective detection of similar code allows improved allocation of the analyst's time and effort.

In recent years, a significant amount of research has been done on comparing the *semantics* of malicious code (e.g., [14], [5], [10], [7], [1]). Despite some variation on the exact definition of semantic similarity, a common idea is that related programs make similar changes to the state of a system (i.e., registers and memory). Previous work has focused on comparing pairs of malware binaries. The set of system-state changes for a binary is extracted and compared with that of another. Unfortunately, in order to categorize a large collection of N malware binaries into distinct families, this pairwise approach would require $O(N^2)$ comparisons.

In this paper, we address this scalability problem by introducing the notion of a *semantic hash* for functions. The goal is to *capture machine-state changes, made at a function-level, in a form that facilitates fast comparison*. This representation reduces the task of comparing a set of binaries to identifying collisions among hashes generated by members of the set. It also enables analysts to search an index of previously cataloged hashes for a specified function.

Detecting similarity between functions achieves two purposes. First, it enables us to focus on a restricted, but still useful, sub-problem of whole program comparison. Functions are the basic semantic and syntactic units of which a binary is composed. Two binaries that share many similar functions are likely to be similar as well. Second, function similarity integrates nicely with existing binary analysis workflows. Reverse engineers typically reconstruct the overall behavior of a binary from that of its constituent functions. Knowing similarity at the function level is therefore useful to judge similarity at the level of binaries.

Specifically, we make the following contributions. First, we design a semantic hash $\mathcal{H}$ for binary functions such that $\mathcal{H}(f_1) = \mathcal{H}(f_2)$ with high probability if f_1 and f_2 have similar input-output behavior over the system state. The overall similarity-detection algorithm then works by: (i) hashing each function; (ii) clustering functions based on their hash (i.e., two functions are in the same cluster iff they have the same semantic hash) and (iii) declaring all functions in the same cluster to be potentially related. This approach is automatable and linear in the number of functions, and therefore scalable to function collections of realistic size.

Second, we evaluate $\mathcal{H}$ over functions derived from the CERT artifact catalog. We empirically demonstrate that $\mathcal{H}$ is effective (i.e., able to detect a large number of duplicates) on current malware samples. In one case, it is able to reduce the sample size by a factor of over 500 compared to a state-of-the-art competing hashing scheme. In addition, our interactions with expert reverse engineers at CERT indicate that this technique is sufficiently accurate for operational use.

The rest of this paper is structured as follows. In Section II, we survey related work. In Section III, we present preliminary concepts. In Section IV, we state the problem. In Section V, we present our semantic and syntactic features, and the procedures for extracting them from function descriptions. In Section VI, we present our experiments and results. In Section VII, we conclude.

II. Related Work

A number of different approaches have been proposed for determining binary similarity. For example, several researchers have used feature vector comparison to detect code clones [12], and to protect against malware [14].

Gao et al. [7] use a control-flow based analysis that combines graph isomorphism, symbolic simulation, and theorem proving to detect semantic differences in binaries. Leder et al. [10] use value-set analysis to characterize and compare the semantic behavior of programs at various points of interest. Apel et al. [1] propose metrics for malware similarity.

Christodorescu et al. [5] represent malware behavior as a 3-tuple (instructions, variables, constants), called a template. A sequence of instructions s is flagged as potentially malicious if the changes made to memory by s is provably the same as that of a template. This work relies on an SMT-solver to find a bijection between each node of a malware tuple and the one extracted from the target sequence. Thus, it scales poorly to a large number of files and templates.

Leder et al. [10] use value set analysis to approximate the contents of registers and memory at "points of interest" in a program. Two sequences of instructions are deemed equivalent if they produce the same set of values at one of these points. Thus, this technique is able to detect code written with semantically-equivalent, but syntactically different, instructions. By comparing the sets generated by known malware with those of unclassified code, a program is flagged as malicious or not. However, once again, pair-wise comparison of value sets scales poorly.

Zhang et al. [15] use data-flow analysis to resolve library calls and call sequences for comparison. This approach can be viewed as comparing the value of the program counter at various points in the execution of a program. However, sequence comparison suffers from the same pair-wise scalability issue as template matching and set comparison.

Cohen et al. [6] explore the use of cryptographic hashes to study program similarity. This work is closely related to ours, and generates hashes from actual instruction bytes. We extend this work by assigning semantic, as opposed to syntactic, meaning to hashes.

III. Background

We write $F : X \hookrightarrow Y$ to denote a (possibly partial) mapping from X to Y, where $Dom(F)$ is a subset of X that has a valid mapping to Y.

Functions. Intuitively, a function is characterized by: (i) a partial mapping from addresses to instructions, (ii) a specific start address, and (iii) an end address. We assume a totally-ordered set of addresses, $AddressSpace$ (i.e., the set of 32-bit unsigned integers), and a set $Inst$ of instructions, each comprising of an opcode and zero or more operands.

Definition 1: A function is a tuple $(Body, Start, End)$ where: (i) $Body : AddressSpace \hookrightarrow Inst$ is a partial mapping between addresses and instructions, and (ii) $\forall a \in Dom(Body), Start \leq a \leq End$.

We view a function as a sequence of bytes representing each instruction in its body. This is known as the "exact-byte" (or EBYTE) representation, and defined as follows.

EBYTE Representation. Let Str be the set of finite-length byte sequences (or strings), and $\bullet$ be string concatenation. Let $\epsilon : Inst \mapsto Str$ be the mapping from instructions to strings specified in the instruction set. For a function $f = (Body, Start, End)$, 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: $i_k = Body(a_k)$ for $1 \leq k \leq n$. Let $MD5$ be the MD5 hash function. The EBYTE representation and EHASH of f, denoted by $f.EB$ and $f.EHash$ respectively, are:

$$f.EB = \epsilon(i_1) \bullet \cdots \bullet \epsilon(i_n) \qquad f.EHash = MD5(f.EB)$$

Functions compiled from the same source code often have EBYTE representations that differ only in the address values they refer to. This is typically caused by differences in the memory layout during compilation. Such functions therefore have different EHASH values even though they are identical. To overcome this limitation of EBYTES, we define a "position-independent" (or PBYTE) representation.

PBYTE Representation. Let $\pi : Inst \mapsto Str$ be the mapping from instructions to byte sequences such that $\pi(i)$ is obtained by replacing all address values in $\epsilon(i)$ with zeroes. Then the PBYTE representation and PHASH of f, denoted by $f.PB$ and $f.PHash$ respectively, are:

$$f.PB = \pi(i_1) \bullet \cdots \bullet \pi(i_n) \qquad f.PHash = MD5(f.PB)$$

Example 1: Figure 1 shows a basic block of a function f with starting address `80483BC`, $Body$, ϵ and π. Zeroed-out and zero addresses are shown in red and blue, respectively. $f.EB$ and $f.PB$ are, respectively, the concatenation of the elements of the third and fourth columns.

Address a	Instruction $i = Body(a)$	$\epsilon(i)$	$\pi(i)$
80483BC	push %ebp	55	55
80483BD	mov %esp,%ebp	89E5	89E5
80483BF	sub $0x14,%esp	83EC14	83EC14
80483C2	mov 0x10(%ebp),%eax	8B4510	8B4510
80483C5	mov %eax,(%esp)	890424	890424
80483C8	call 80483b4	E8E7FFFFFF	E800000000
80483CD	mov %eax,-0x4(%ebp)	8945FC	8945FC
80483D0	cmpl $0x0,0x8(%ebp)	837D0800	837D0800
80483D4	je 80483e4	740E	740E
80483D6	mov 0xc(%ebp),%eax	8B450C	8B450C
80483D9	mov %eax,(%esp)	890424	890424
80483DC	call 80483b4	E8D3FFFFFF	E800000000
80483E1	mov %eax,-0x4(%ebp)	8945FC	8945FC
80483E4	mov -0x4(%ebp),%eax	8B45FC	8B45FC
80483E7	leave	C9	C9
80483E8	ret	C3	C3

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

PHASH improves over EHASH by eliminating duplication due to memory layout differences. However, it fails

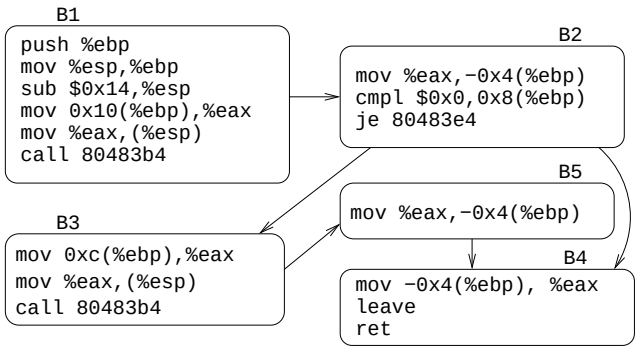

Figure 2. An example CFG. B1 is the entry node and B4 is the exit node.

against two other types of duplication: (i) instruction substitution – where an instruction sequence is replaced with a semantically equivalent one, e.g., to obfuscate malware; (ii) instruction sequence reordering – due to compiler optimizations. In the next section, we present our semantic hash $\mathcal{H}$ designed to eliminate both these duplications.

MinHashing. MinHashing – developed by Andrei Broder [2] – is a form of locality-sensitive hashing that permits fast set comparison. The MinHash algorithm produces the same hash for a pair of sets – A and B – with probability $JI(A, B)$, where $JI(A, B)$ is the Jaccard Index [8] of A and B is defined as:

$$JI(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

We use $MinHash(A)$ to denote the MinHash of set A. The $MinHash$ algorithm uses k independent hash functions. Unless otherwise mentioned, our implementation of $MinHash$ uses five hash functions, i.e., $k = 5$.

IV. Problem Statement

Given a large set of binary functions D, out goal is to generate hashes for each function, such that collisions are likely to occur for similar functions. More precisely, we want to construct a clustering function $h : D \to (N)$, such that for two functions $f, g \in D, h(f) = h(g)$ implies that there is a high probability that f is similar to g.

To evaluate our solution, we need a way to determine whether two functions are semantically similar, as for example, obtained from the same source code by different compilers, or by small changes. Because similarity is hard to assess objectively, we seek the help of a malware analyst to validate the similarity declared by our approach. A successful outcome is one in which the two are in agreement.

Assumptions and Limitations. Fundamentally, our system relies on the accurate representation of a function's control flow graph and basic blocks. As others [3][5][13] have noted, malware authors often employ obfuscation techniques – e.g., insertion of extraneous jumps, dead-code insertion, self-modifying code – that complicate the process of meeting these dependencies. While deobfuscation/normalization (perhaps using the techniques presented by these same researchers) is an important step that must be performed prior to hash generation, it is a concern that is orthogonal to the work presented in this paper.

V. Approach

Let $f = (Start, Body)$ be a function. The control flow graph (CFG) of f, denoted by $CFG(f)$, is a directed graph whose nodes (also known as basic blocks or BBs) are labeled with a sequence of instructions, and whose edges correspond to possible flow of control. The key idea is that any execution of f follows a path in $CFG(f)$. Figure 2 shows the CFG of the function shown in Figure 1.

A. SemanticHash

Recall that $MinHash$ is useful for estimating the "distance" between *sets*. Thus, in order to use $MinHash$ for comparing functions, we first reduce each function to a set of features. More specifically, we use the input-output behavior of BBs as a feature. Our choice of BBs is motivated by two facts: (i) BB boundaries provide a natural way of dividing a function by its control flow; and (ii) the input-output behavior of a BB (in terms of registers and memory) depends solely on the instructions within the BB. The overall semantic hash – $\mathcal{H}(f)$ – of f is then computed as follows:

1) Disassemble f and construct $CFG(f)$.
2) Let $BB(f)$ be the set of basic blocks of $CFG(f)$. Let Φ be a hash from basic blocks to strings, and let $\Phi(f) = \{\Phi(b) \mid b \in BB(f)\}$. Then, the function hash $\mathcal{H}(f)$ is defined as:

$$\mathcal{H}(f) = MinHash(\Phi(f))$$

We refer to a hash from basic blocks to strings as a BBHash. We write $\mathcal{H}[\Phi]$ to denote the semantic function hash parameterized by the BBHash Φ. Note that the probability that $\mathcal{H}[\Phi](f_1) = \mathcal{H}[\Phi](f_2)$ is equal to the Jaccard distance [8] between $\Phi(f_1)$ and $\Phi(f_2)$. Suppose that $\Phi(b)$ has the following property: **(PROP)** if b_1 and b_2 are input-output (IO) equivalent, then $\Phi(b_1) = \Phi(b_2)$ with high probability. Then, two functions that share many IO equivalent basic blocks will also yield the same semantic hash (i.e., will be clustered together) with high probability. We now define three BBHashes that satisfy **PROP**. We begin with a set of concepts that capture the IO behavior of basic blocks.

B. Input-Output Behavior of Basic Blocks

We consider the function's execution state to consist of values assigned to a set of registers R, and memory M. Our model of the IO behavior of a basic block b consists of four components: (i) the effect of executing b on registers; (ii) the effect of executing b on memory; (iii) the arguments passed to any function called at the end of b; (iv) the condition

on which any jump instruction at the end of b depends. We describe each component separately.

Effect on Registers. For a register r, let $regval(b, r)$ be a logical formula that expresses the value of r after executing the instructions of b in terms of the initial values of R and M. For example, let b be `B1` from Figure 2. Then:

$$regval(b, esp) \triangleq esp - 24 \quad regval(b, ebp) \triangleq esp - 4$$
$$regval(b, eax) \triangleq M[esp + 12]$$

Note that we use variable $M[a]$ to denote the initial value of the memory cell at address a.

Effect on Memory. Let $a_1, \ldots, a_j$ be the addresses of the memory cells modified by executing the instructions of b. Then $memval(b)$ is the set of pairs $\{(a_1, v_1), \ldots, (a_j, v_j)\}$ where v_i is a formula – in terms of the initial values of R and M – denoting the final value of the memory cell at address a_i after the execution of b. For example, let b be `B1` from Figure 2. Then:

$$mem(b) \triangleq \quad \{(esp - 24, M[esp + 12])\}$$

Function Call. If the last instruction of b is a function call with k arguments, then let $p_1(b), \ldots, p_k(b)$ be formulas that represent the values of the parameters passed to the called function in terms of the initial values of R and M. For example, let b be `B3` from Figure 2. The function called at the end has a single argument, which is stored in memory by the instruction at `80483D9`. Thus, $p_1(b) \triangleq M[ebp+12]$.

Branch Condition. If the last instruction of b is a jump, then let $brcond(b)$ be the formula denoting the condition on which the branch instructions target depends. If the branch is unconditional, then $brcond(b) = true$. For example, let b be `B2` from Figure 2. The target of the jump instruction at the end depends on the result of the comparison at address `80483D0`. Therefore, $brcond(b) \triangleq M[ebp + 8] = 0$.

C. Sampling-Based Basic Block Hash

The hash $Samp(b)$ is derived from the output of b on a pseudo-random set of inputs. The key idea is that if b_1 and b_2 are IO equivalent, then they produce the same output on equal inputs, which entails **PROP**. Specifically, for any formula ϕ over the initial values of R and M, let $var(\phi)$ be the set of variables appearing in ϕ. An assignment a of ϕ is a mapping from $var(\phi)$ to concrete values, and $\phi[a]$ is (a string representation of) the evaluation of ϕ under a.

Computing Pseudo-Random Assignments. Let ϕ be a formula over n variables, i.e., $var(\phi) = v_1, \ldots, v_n$. Let x be a $n \times m$ array of concrete values. Each column of x yields an assignment of ϕ by mapping v_i to the concrete value in the i-th row of the column. Thus, x yields m assignments of ϕ. Let this set of assignments be denoted by $\alpha(x)$. We construct a set of assignments $A(\phi)$ as follows. Create a $n \times m$ seed array X of concrete values (for our experiments, we used $m = 48$). Let $\Pi(X)$ be the set of all arrays obtained by permuting the rows of X. Then $A(\phi)$ is defined as:

$$A(\phi) = \bigcup_{x \in \Pi(X)} \alpha(x)$$

Note that $A(\phi)$ contains up to $m \times n!$ assignments.

Computing Sample-Based Hash. To compute $Samp(b)$, we compute the output of b using the set of assignments computed above as inputs, and hash the result. For a set of strings W, let $\diamond(W)$ be the concatenation of the lexicographic ordering of W. Formally, the sample-based hash of a formula ϕ is defined as:

$$h(\phi) = MD5(\diamond(\bigcup_{a \in A(\phi)} \phi[a]))$$

In other words, we evaluate ϕ under each assignment in $A(\phi)$, sort and concatenate the results, and take the $MD5$ hash of the final string. Finally, to define $Samp(b)$, recall the formulas $regval(b, r)$, $memval(b)$, $p_i(b)$, and $brcond(b)$ defined in Section V-B. Also recall that $memval(b)$ is a set $\{(a_1, v_1), \ldots, (a_j, v_j)\}$. Then, $Samp(b)$ is defined as:

$$MD5(h(\bigwedge_{r \in R} regval(b, r)) \bullet h(v_1) \bullet \ldots \bullet h(v_j) \bullet$$
$$h(p_1(b)) \bullet \ldots \bullet h(p_k(b)) \bullet h(brcond(b)))$$

Example 2: Consider the basic block b consisting of two instructions: `sub ecx,edx` and `add eax,ecx`. Let $\phi = regval(b, eax) = eax + ecx - edx$. Then $var(\phi) = \{eax, ecx, edx\}$. Thus, an assignment of ϕ is a mapping from $var(\phi)$ to concrete values. Specifically, let $\alpha(v_1, v_2, v_3)$ be the assignment $[eax \mapsto v_1, ecx \mapsto v_2, edx \mapsto v_3]$. Assume that $m = 3$ and the initial 3×3 seed matrix, X, is

$$\begin{pmatrix} 8 & 20 & 3 \\ 73 & 15 & 54 \\ 46 & 23 & 26 \end{pmatrix}$$

Therefore, $A(\phi)$ contains all possible assignments to v_1, v_2, v_3 from X and matrices formed by permuting the rows of X. In other words:

$$A(\phi) = \{\alpha(8, 73, 46), \alpha(20, 15, 23), ..., \alpha(73, 8, 46), ...\}$$

Then, $h(\phi)$ is computed by: (i) evaluating ϕ on each assignment in $A(\phi)$, (ii) concatenating the results in ascending order, and (iii) computing the MD5 of the final value.

D. Summary-Based Basic Block Hash

The hash $Summ(b)$ is essentially the $MD5$ hash of (the string representation of) a logical formula that represents the IO behavior of b. Recall the formulas $regval(b, r)$, $memval(b)$, $p_i(b)$, and $brcond(b)$ defined in Section V-B. Let $reg(b)$ be the formula:

$$\bigwedge_{r \in R} r' = regval(b, r)$$

Note that we use r' to denote the final value of register r.

Recall that $memval(b)$ is a set $\{(a_1, v_1), \ldots, (a_j, v_j)\}$. Let $mem(b)$ be the formula:

$$\bigwedge_{(a,v) \in memval(b)} M'[a] = v$$

Note that we use $M'[a]$ to denote the final value of the memory cell at address a.

Let $called, a_1, \ldots, a_k$ be distinguished variables such that $called$ is Boolean, and a_i has the same type as $p_i(b)$. Then, $call(b)$ is the formula defined as follows:

1) if the last instruction of b is a function call, then:

$$call(b) = called \wedge (a_1 = p_i(b)) \wedge \cdots \wedge (a_k = p_k(b))$$

2) if the last instruction of b is not a function call, then:

$$call(b) = \neg called$$

Finally, the summary-based hash of b, denoted by $Summ(b)$, is defined as follows:

$$Summ(b) = MD5(reg(b) \wedge mem(b) \wedge call(b) \wedge brcond(b))$$

E. Combined Basic Block Hash

Both $Samp$ and $Summ$ can produce identical hashes for basic blocks that are not IO equivalent. $Samp$ collisions occur because the set of inputs we use for sampling does not cover the entire input space, and in particular, does not include any input for which the two basic blocks produce distinct outputs. In contrast, $Summ$ collisions occur because of imprecisions in computing $reg(b)$, $mem(b)$, $call(b)$, and $brcond(b)$. Since the two hashes produce collisions for different reasons, we propose a third BBHash, called $Comb$, that combines $Samp$ and $Summ$ as follows:

$$Comb(f) = MD5(Samp(f) \bullet Summ(f))$$

By definition, $Comb$ detects no more duplicate functions than either $Samp$ or $Summ$. Nevertheless, our experiments indicate that it is effective at detecting function similarity.

VI. Experimental Evaluation

We have implemented our semantic hashes on top of the ROSE [11] infrastructure. To evaluate the effectiveness of the hashes, we used them to cluster a series of 16 benchmarks derived from the CERT Artifact Catalog. Each benchmark consists of a set of functions with distinct $PHashes$. Therefore, a $PHash$-based clustering would not identify any duplication in them. The first four benchmarks were designed to contain a lot of duplication, i.e., only a few clusters of similar functions. For example, Memcpy1 and Memcpy2 each consist of functions that were identified by IDAPRO to be implementations of the `memcpy` library routine. These benchmarks were subsequently inspected by a malware analyst to confirm that all members of each set were in fact the same function.

Example	$\#Funcs$	$\#Samp$	$\#Summ$	$\#Comb$
FormCreate	2048	2.24	2.49	2.49
Memcpy1	10000	0.17	.19	.19
Memcpy2	4479	0.44	.51	.51
memmove	355	6.47	8.16	8.16
Random1	4615	86.04	96.74	97.52
Random2	9390	83.02	96.80	97.48
DistName	193	89.11	93.26	93.78
DialogFunc	944	93.43	98.83	99.15
EnumFunc	961	95.31	98.95	99.27
TimerFunc	517	85.88	98.64	99.41
Start	2568	69.82	84.81	88.31
DllEntryPoint	696	46.55	62.78	67.67
DriverEntry	915	68.52	88.63	92.89
onexit	1000	4.4	14.60	14.70
abort	96	64.58	81.25	82.29
filebuf	890	57.19	97.41	98.31

Table I
NUMBER OF CLUSTERS FOR DIFFERENT SEMANTIC HASHES, EXPRESSED AS A PERCENTAGE OF THE NUMBER OF CLUSTERS OBTAINED VIA $PHash$.

The next six benchmarks were designed to contain few duplications, i.e., mostly of functions that are not similar. For example DistName consists of functions that were each identified by IDAPRO to be implementations of a different library routine.

The last six benchmarks were constructed by: (i) randomly selecting six function names; and (ii) collecting functions corresponding to one name to produce one benchmark. These benchmarks are therefore expected to contain varying levels of duplication.

We evaluated each of our semantic hashes – $\mathcal{H}[Samp]$, $\mathcal{H}[Summ]$, and $\mathcal{H}[Comb]$ – on each example E, as follows:

1) Cluster the functions using $\mathcal{H}[Samp](f)$ as the cluster id of f. Let the number of clusters be x_1. Define $\#Samp = \frac{x_1}{|E|} \times 100$. A smaller $\#Samp$ means a more similar functions according to $\mathcal{H}[Samp]$.
2) Repeat Step 1 using $\mathcal{H}[Summ](f)$ instead of $\mathcal{H}[Samp](f)$. Denote the result by $\#Summ$.
3) Repeat Step 1 using $\mathcal{H}[Comb](f)$ instead of $\mathcal{H}[Samp](f)$. Denote the result by $\#Comb$.

Table I summarizes our results. Each row shows $\#Samp$, $\#Summ$, and $\#Comb$ obtained for a specific example. As expected, the number of clusters is small for the examples designed to contain lots of duplication, while it is large for the examples designed to contain little duplication. This indicates that our semantic hashes identify substantial real duplication over and above $PHash$, without incurring too much false duplication. For the randomly selected benchmarks, the degree of detected duplication span a wide range.

For Memcpy1, our hashes reduce the number of distinct functions by a factor of 500. Fig. II shows the difference between two implementations of memcpy that produced the same $\mathcal{H}$ but different $PHashes$. Note that the first uses an add instruction, while the second uses an inc.

Also, as expected, $\#Comb$ is greater than either $\#Samp$ or $\#Summ$. However, the difference is small, indicating that duplication among functions discovered by $\mathcal{H}[Samp]$ and $\mathcal{H}[Summ]$ overlap significantly. In addition, $\#Summ$ is, in general, greater than $\#Samp$. This indicates that $\mathcal{H}[Summ]$ detects less duplication (real and false) compared to $\mathcal{H}[Samp]$. Interestingly, the difference between $\#Summ$ and $\#Samp$ is small for the benchmarks designed to have little or lots of duplication. However, it is significant for the randomly constructed benchmarks. This may mean that $\mathcal{H}[Summ]$ is more precise that $\mathcal{H}[Samp]$, especially for function collections where similarity detection is non-trivial.

RB1	**INS1**	**RB2**	**INS2**
f3a5	rep movsd	f3a5	rep movsd
ff24957 c8b4100	jmp [edx*4+ 0x418b7c]	ff2495d 8b84400	jmp [edx*4+ 0x44b8d8]
90	nop	90	nop
23d1	and edx,ecx	23d1	and edx,ecx
8a06	mov al,[esi]	8a06	mov al,[esi]
8807	mov [edi],al	8807	mov [edi],al
83c601	add esi,0x1	46	inc esi
c1e902	shr ecx,0x2	c1e902	shr ecx,0x2
83c701	add edi,0x1	47	inc edi
83f908	cmp ecx,0x8	83f908	cmp ecx,0x8
7288	jc 0xffffff8a	728c	jc 0xffffff8e
f3a5	rep movsd	f3a5	rep movsd

Table II
PORTIONS OF TWO IMPLEMENTATIONS OF MEMCPY() WITH MATCHING SEMANTIC HASHES. **RB1** = MEMCP1 RAW BYTES; **INS1** = MEMCPY1 INSTRUCTIONS; **RB2** = MEMCP2 RAW BYTES; **INS2** = MEMCPY2 INSTRUCTIONS.

VII. CONCLUSION

We present an approach for representing the semantics of a machine-code function as a hash. This enables quick identification and clustering of binary code that exhibit a high-degree of similarity. A function is first represented as a set of basic block hashes, which are then combined using a form of Locality-Sensitive Hashing, called MinHashing. We present two basic block hashing algorithms. The first produces hashes from the outputs generated by the same formulas, when fed a reproducible sequence of input values. The second produces hashes directly by hashing the simplified, symbolic formulas of memory and registers contents at the end of blocks. The proposed algorithms have been implemented and tested on real-world functions extracted from the CERT artifact catalog. Our experimental results indicate that we are able to assign functions that share a high-degree of similarity the same hash value.

In this paper, we used basic blocks as the unit of division for generating features for functions. However, to improve the robustness of our system against obfuscations (i.e., basic block splitting using unconditional jumps and dead code), we are currently exploring other possibilities. For example, one could generate semantic hashes from slices [9] of a function, which are sets of instructions that affect the value of a variable at a particular point. One advantage of this approach is that junk instructions will be excluded from the slices of the malware's actual code.

Acknowledgment. Copyright 2012 Carnegie Mellon University and IEEE. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. CarnegieMellon®, CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM-0000063.

REFERENCES

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

[2] A. Broder. On the resemblance and containment of documents. In *Proc. of Compr. and Compl. of Seq.*, 1997.

[3] D. Bruschi. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In *Lecture Notes in Computer Science*, volume 4064, pages 129–143. Springer-Verlag, 2006.

[4] Incoming malware statistics 5/21/2012. Personal communication.

[5] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-Aware Malware Detection. In *Proc. of IEEE Symposium on Security and Privacy*, 2005.

[6] C. Cohen and J. Havrilla. Function Hashing for Malicious Code Analysis. Technical report, SEI, Pittsburgh, PA, USA, 2009. www.cert.org/research/2009research-report.pdf.

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

[8] P. Jaccard. Nouvelles recherches sur la distribution florale. *Bull. Soc. Vaudoise Sci. Nat.*, 44:223–270, 1908.

[9] A. Kiss, J. Jsz, G. Lehotai, and T. Gyimthy. Interprocedural static slicing of binary executables. In *Proc. of SCAM*, 2003.

[10] F. Leder. Classification and detection of metamorphic malware using value set analysis. In *Malicious and Unwanted Software (MALWARE), 4th International Conference on Malware*, pages 39–46, 2009.

[11] ROSE website. http://www.rosecompiler.org.

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

[13] A. Walenstein, R. Mathur, M. Chouchane, and A. Lakhotia. Normalizing metamorphic malware using term rewriting. In *Proc. of SCAM*, 2006.

[14] 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.

[15] Q. Zhang. MetaAware: Identifying Metamorphic Malware. In *Proc. of ACSAC*, 2007.