J Comput Virol (2008) 4:335–345
DOI 10.1007/s11416-008-0087-z

ORIGINAL PAPER

A boosting ensemble for the recognition of code sharing in malware

Stanley J. Barr · Samuel J. Cardman · David M. Martin Jr.

Received: 11 November 2007 / Revised: 29 March 2008 / Accepted: 6 April 2008 / Published online: 22 April 2008

Abstract Research and development efforts have recently started to compare malware variants, as it is believed that malware authors are reusing code. A number of these projects have focused on identifying functions through the use of signature-based classifiers. We introduce three new classifiers that characterize a function's use of global data. Experiments on malware show that we can meaningfully correlate functions on the basis of their global data references even when their functions share little code. We also present an algorithm that combines existing classifiers and our new ones into an ensemble for correlating functions in two binary programs. For testing, we developed a model for comparing our work to previous signature based classifiers. We then used that model to show how our new combined ensemble classifier dominates the previously reported classifiers. The resulting ensemble can be used by malware analysts when they are comparing two binaries. This technique will allow them to correlate both functions and global data references between the two and will lead to a quick identification of any sharing that is occurring.

S. J. Barr (✉) · S. J. Cardman
Distributed Systems Center, The MITRE Corporation,
Bedford, MA 01730, USA
e-mail: sbarr@mitre.org

S. J. Cardman
e-mail: sc@mitre.org

D. M. Martin Jr.
Computer Science Department,
University of Massachusetts Lowell,
Lowell, MA 01854, USA
e-mail: dm@cs.uml.edu

1 Introduction

The benefits of reusing a software component are considerable. The cost of the product may decrease because less effort must be expended in development and the reliability of the product may increase because the reused component has already been tested. Thus, it comes as no surprise to learn that writers of malware reuse old code and incorporate each others' designs and implementations into their programs. In this context, there is an urgent need to be able to compare two malware samples and identify commonality.

Researchers can reverse engineer a piece of malicious software to understand its operation [1]. However, reverse engineering is laborious. In cases such as the Bagle virus, new versions keep appearing [2]. A new variant may appear before the analysis of a previous version is complete. Leveraging the analysis of common behavior preserved between generations would be a great step in understanding and combating new malware variants.

Global identifiers such as variable and function labels allow for the correlation of functions between two binaries. But malware is usually presented as a stripped binary, i.e., function and data labels have been removed and no source code or other metadata is available. This makes it more difficult to analyze the binary or compare it to another binary.

Current techniques of comparing two binaries for similarity rely on correlating functions by generating code based signatures [3–10]. Some example features for these code based signatures are comprised of instruction sequences, control flow graphs, function call graphs, and DLL (dynamically linked library) function calls. Thus, for example, binaries can be compared when they reference external functions provided by the operating system through the use of DLLs, because symbol names must be kept when external references are used. A signature can be constructed for a function simply by enumerating the names of externally referenced functions and the number of times each external function is mentioned in the binary code. If two functions in different binaries share such a signature and each signature is unique within its own executable, then this would suggest that the two functions are related.

Now the question naturally arises: can one improve the comparison by including and correlating *global data* references? Roughly speaking, any absolute address named by a binary's machine code is considered a global data reference if it is read from, written to, or stored on the stack. Thus, no distinction is made between scalars, arrays, or pointers.

To answer the question, we have combined several signature based classifiers into an ensemble. This ensemble includes existing classifiers and introduces three new classifiers. The first new classifier, GdrMatcher, associates a piece of global data with the functions that refer to it. The second classifier, AllLocalIds, describes a function by listing its own global data references. The third classifier, AllParentIds, describes a function f by listing the global data references of the callers of f. This new ensemble can be used to correlate both functions and global data references between two binaries. We develop a model to compare the ensemble's performance before and after our new classifiers have been integrated and demonstrate that the new ensemble is an improvement over the former.

We also conduct an experiment to explore the concept of correlating global data references in an attempt to answer two basic questions:

1. Can we use these correlated data references to develop more accurate function correlations?
2. Can we correlate global data references among entire binaries known to be related and avoid spurious correlations between unrelated binaries?

We conducted our experiments with malware for the following reasons.

– There are large data sets of hand-classified malware variants that we can compare with our algorithm's output.
– When malware first appears, it must be analyzed in order to understand the full extent of its behavior. However, reliable detection and identification even of bounded length mutating viruses is NP-complete [11]. Alternate approaches to malware identification are therefore quite appropriate.
– Malware authors modify existing malware to improve functionality and evade detection by antivirus products. These modifications are accomplished both by modifying source code and directly manipulating existing binaries. While it is suspected that malware authors share code, no one has an accurate assessment of the extent or scope of this practice.
– In our sample set we found the average malware program had 134 global data references, suggesting that at least this subset of malware is amenable to correlation of global data references.

Combined with the threats posed by malware, these reasons make it a reasonable data set for analysis, comparison, and exploration.

As previously stated, our current data set has on average 134 data references per executable. As malware programs grow in complexity and size we believe more references will appear in binaries. Provided that each data reference has a unique usage pattern (see signature (10)), we currently believe any number of them could be correlated between two related samples.

It should be noted although we are analyzing malware, we are not attempting to create an antivirus tool for detecting malware variants. Antivirus tools may be able to detect that two programs are variants even when we cannot identify any shared functions: their goal is to use any distinctive portion of the code or data to construct a signature for reliable detection. We instead seek to correlate information in order to support the reverse engineering process, and allow researchers to find commonalities between binaries.

The paper is structured as follows. Section 2 describes previous efforts from which we have drawn components. Section 3 discusses desirable properties for our classifiers. Section 4 describes our algorithm. In Sect. 5, we describe our experiments. Section 6 offers our conclusions and some possible future directions. More details can be found in an extended version of this paper [12].

2 Related work

There are a number of techniques that have been shown to help identify software variants. We have drawn heavily from these works.

– *Hashes*: Cryptographic hashes have been used to identify software clones in source and in binaries [4], although they are overly sensitive to minor variations in the input.
– *Sequence analysis*: Sequence analysis algorithms [9,13] have proven successful in the analysis of related binaries. In fact, most antivirus tools are based on sequence matching. These techniques usually do not require disassembling binary code. This is appealing, since malicious coders can employ a number of techniques to thwart disassembly [14]. However, they are susceptible to certain types of obfuscation such as the substitution of semantically equivalent instruction sequences.

In a recent paper [10], the author demonstrated some interesting results based on Boolean function modeling. The patterns he used to detect malware using this new model allow for efficient detection. Also, the patterns are robust to black-box analysis, making it difficult for malware authors to determine what portions of the executable comprise the pattern being checked.
The author of [15] takes a completely different approach for sequence analysis. This work uses the notion of *normalized compression distance*, which is derived from Kolmogorov complexity [16]. The idea put forth is that if two strings are similar then the concatenation and subsequent compression of the strings will result in a length which is not that much larger than the length of compressing the larger of the two by itself. And conversely, the same operation on two strings which are not similar would result in an output length significantly larger than the result of compressing either individually. The difference of the resulting length's is considered a distance between the two strings. The file contents of malware samples are used as input strings, and the resulting distances between the samples are used to cluster them.
– *Graph matching*: We use the techniques given in [3], which allow for the construction of hashes from control flow graphs. Other research has offered novel techniques for identifying variants of binaries by mapping program control flow features into directed graphs and then using graph theoretic techniques to determine the similarity of the original code [5,8].
– *Function signatures*: The authors in [5,6] describe using structural components of functions to create signatures. Some of the structural components which have been used to compare functions in different executables are: the strings referenced, system calls made, and assembly language opcodes used.

This research has focused on enhancing these approaches to increase their detection of software variants. When we have the ability to correlate functions between two binaries, it is useful to have a metric that gives a numeric value to the amount of code shared. We have incorporated the similarity metric developed in [6] as our measure.

3 Desirable properties for binary code classifiers

In order to reliably detect variant binaries, it is important to understand how changes can manifest themselves in the binaries. The work done in [5,3] provides more information on variability in compiled code bases. Simply put, traditional compilers follow a deterministic process during code generation. Therefore, variants occur because a different compiler is used, the source code is modified, or modifications are made directly to the binary. Changing optimization levels to trade computation time for program size can significantly change the generated code. Different compilers or compiler versions may generate different binaries for the same code. While in general authors can make an infinite variety of modifications, they often follow the ancient adage "If it ain't broke, don't fix it!" By only altering code whose behavior they seek to change, they may leave enough information intact to identify and correlate functions.

In this research we restrict ourselves to the following types of variants: instances where functions or blocks have been reordered, code where the compiler settings may have introduced subtle changes, and cases where changes have left the structure of the code relatively similar. These restrictions allow for variability in the structure or functionality when comparing functions. They also require some measurable feature be preserved. This measurable feature becomes the basis for the claim that two pieces of code being compared are variants. To identify these variations our ensemble will be insensitive to the following categories of code modifications, as referenced in [5]:

– Function reordering
– Alternate register allocation schemes
– Locations and offset changes
– Branch inversion and block re-ordering
– Simple insertion, deletion, transposition, and substitution of instructions

Instead of trying to accomplish the above objectives with a single complex classifier, we use a number of simple classifiers. Each one focuses on identifying functions based on different characteristics. The goal is to construct an ensemble with classification ability better than the sum of each of the classifiers working independently. Works such as [17,18] have laid foundations showing that one can combine multiple weak clustering and classification algorithms to produce a single strong ensemble.

4 Boosting algorithm

4.1 High-level description

Our ensemble takes two binaries as input. Processing starts by extracting features from both binaries. These features will form the basis of our signature construction. Features are provided primarily by parsing the C-like output of the REC Reverse Engineering Compiler [19]. Each feature will be introduced and developed as needed.

Once the features have been extracted the ensemble starts an iterative matching process, during which it accumulates Φ, a common set of equivalent functions, and Γ, a common set of equivalent global data references. Both sets are initialized to the empty set and share the form $\{(a_1, b_1), (a_2, b_2), \ldots\}$. In Φ each a_i and b_i represent function addresses in binaries A and B respectively. Similarly, in Γ, a_i and b_i represent global data reference pairs.

Pairs are matched and added to these sets using one of our signature algorithms as follows. First, only currently unlabeled objects are considered during each iteration. We then focus on those objects that have a unique signature in the first binary and search for that signature to be present uniquely in the second binary. If found, the two items are paired and labeled, the pair is inserted into the appropriate Φ or Γ set, and the two items are removed from the pool of unlabeled objects. At the beginning of the next iteration, we reevaluate the signatures previously ignored due to their non-uniqueness: a signature that was previously non-unique in its binary may later become unique again if all but one of its objects in the same executable have since been successfully matched between the executables. The algorithm terminates after a cycle in which no new pairs are matched.

4.2 Signatures

Our ensemble classifier uses two types of signatures. Static signatures depend only on the extracted features. Dynamic signatures make use of matchings accumulated in Φ and Γ in addition to extracted features. This improves the quality of subsequent signatures generated, although at the cost of computing some signatures on the fly.

The following two lists describe our signature generation algorithms. Each algorithm will be depicted as

$$\mathit{Name} : (arg_1[, arg_2, ..., arg_n]) \rightarrow \mathit{Signature}. \tag{1}$$

Here *Name* is a short descriptive identifier. The first argument arg_1 will represent both the binary and either some extracted feature of a function or the location of a global data reference in that binary. Any other required features will be passed in arg_2 through arg_n. Each algorithm will yield *Signature*, a value which can be compared with others (generated by the same algorithm) in linear time based on the length of the signatures.

4.2.1 Statically constructed signatures

– *Named symbol look up*: Executables generally use DLLs to invoke external functions by name, whether they are implemented in the operating system or in other application libraries. Compilers support this by generating wrapper stubs in the caller's executable. Each stub is a simple wrapper function used to call the desired target through an import table that is populated at runtime. Our first signature labels each such stub address with the name (as a string) of the function it is dynamically linked against. Since not all functions are wrapper stubs, this signature may be only partially defined. The partial signature

$$\mathrm{DllName} : (\mathit{fn}) \rightarrow \mathit{RealName} \tag{2}$$

takes *fn*, the location of a wrapper function, and yields a text string *RealName* that denotes the name of the imported function.

– *MD5 hash of binary function*: The signature

$$\mathrm{MD5}_{\mathrm{bin}} : (\mathit{fn}) \rightarrow \text{128-bit hash} \tag{3}$$

takes as input a binary representation of the function *fn* and yields its 128-bit hash. MD5 was chosen due to its low probability of collisions.[1] It exhibits very high precision in determining exact matches, so there is almost no chance of getting a false positive. However, a single bit change causes an unrecoverable mismatch. Therefore, we would like a more resilient approach that gives roughly the same precision without being as sensitive to a single bit change.

– *MD5 hash of opcodes from a function*: As suggested in [4], we relax the MD5 hash to apply to the string form of the opcodes that comprise a function. The signature

$$\mathrm{MD5}_{\mathrm{opcode}} : (\mathit{fn}) \rightarrow \text{128-bit hash} \tag{4}$$

takes as input the sequence of opcodes that comprise the function *fn* (e.g., "xor", "inc", "mov", etc.) and yields its 128-bit hash. This feature is invariant to differences in register assignment, immediate values, and other operands but still represents the sequence of instructions in *fn*. It would seem that the opcode signature alone should be sufficient to differentiate functions. However, during our experiments we found many counterexamples to this notion. It appeared that many functions were constructed from template code, perhaps as a consequence of macro expansion. Sometimes, the only differentiator was a simple constant value, or a reference to a specific global memory location.

– *Control flow graph*: Control flow graphs allow us to compare the control structure of two functions. Each function can be depicted as a directed graph whose nodes represent the basic blocks of contiguous instructions where control only passes directly from one instruction to the next, and edges represent the flow of control connecting the blocks of contiguous instructions. This allows us to compare functions without being concerned about their constituent instructions.

Since detecting graph isomorphism is GI-complete [20], we relax the signature to make comparisons feasible. The

[1] Note that the specific MD5 collisions demonstrated in 2004 do not appear to directly affect the probability of collisions on random inputs.

signature represents the graph in a way that captures some of the edges in the graph, but also allow for linear time complexity for comparison. This representation is safe: it captures features necessary for detecting isomorphism, although it does not capture all such features.
Our specific approach follows that of [3]. Kruegal et. al. constructed adjacency matrices using all subsets of nodes for a given fixed size k, and the edges that connected those nodes in the function's control flow graph. For our research we arbitrarily set $k = 4$. The individual rows of each 4-by-4 adjacency matrix were then concatenated together to form a 16 bit value, each of which is labeled h_i in the following signature.[2] The list of 16 bit values constructed for each function is sorted once during the feature extraction phase. To compare two functions during a run, we simply compare the sorted lists of values for equality. This test can be done in time linear in the number of features.
The signature

$$\text{Cfg} : (fn) \rightarrow \{h_1, h_2, \ldots\} \tag{5}$$

requires *fn* be the representation of the control flow graph for the given function and yields a set of features $\{h_1, h_2, \ldots\}$ representing that control flow graph.

– *Parameter count*: We also count the number of parameters passed at entry and total number of unique global variables referenced by the function. The signature

$$\text{Params} : (fn) \rightarrow (p, g_f, g_d) \tag{6}$$

yields a tuple for function *fn* where p is the number of parameters that *fn* receives, g_f is the number of pointers to recognized functions that appear in *fn*, and g_d counts how many other global data references appear in *fn*. Each of these values tends to be small, so the Params signature is fairly coarse.

4.2.2 Dynamically constructed signatures

The preceding signatures do not depend on the greater context in which the functions appear. We call them *static*, since they only need to be computed once. The remaining signatures depend both on the function itself and knowledge deduced about the overall executable. Since this knowledge grows as the algorithm runs, we call these signatures *dynamic*.

– *Function call vector matchers*: The function call vectors we extract allow us to create a signature for each function *fn* depending only on the list of functions directly invoked by *fn*. This signature is invariant under control flow changes and other types of code modifications. We limit this signature to enumerating the number of times an unlabeled function makes calls to functions that are already known to exist in both binaries.
The signature

$$\text{CallRefs} : (fn, \Phi) \rightarrow \{(n_1, x_1), \ldots\} \tag{7}$$

takes as input *fn*'s entry from the call vector enumerating all of the functions that *fn* invokes along with the number of times they are invoked. The result is a set of tuples of the form (n_i, x_i), with each tuple indicating that *fn* calls x_i in n_i different places, and each $x_i \in \Phi$. The latter condition means that this signature only considers those functions called by *fn* that have already been matched. We emphasize that this tuple results from a static analysis of *fn*: the number n_i indicates the number of different instructions in *fn* that call x_i, rather than the number of times x_i is called at run time. A simpler form of this signature was used in [6], wherein each n_i is a Boolean value rather than a call count.
Likewise, the signature

$$\text{ParentRefs} : (fn, \Phi) \rightarrow \{(n_1, x_1), ...\} \tag{8}$$

takes as input *fn*'s entry from the call vector enumerating all of the functions that invoke *fn* and the number of times *fn* is invoked by that function. The result is a set of tuples of the form (n_i, x_i), with each tuple indicating that x_i calls *fn* in n_i different places, and each $x_i \in \Phi$.

– *Duplicate signature combiner*: We define a compound signature

$$\text{JoinSig} : (fn, D) \rightarrow \{(t_1, v_1), (t_2, v_2), \ldots\} \tag{9}$$

where D is a set of tuples (f_i, t_i, v_i) in which function f_i has a signature of type t_i with value v_i. The output is those pairs (t_i, v_i) of signature types and values that describe *fn*. In our algorithm, the set D will be populated with a list of signatures that are duplicates: they are *not* unique in their functions' own executables. By combining duplicate signatures of different types, we may be able to construct a unique signature. For example consider a program A, where $DA = \{(f_1, \text{Params}, (1, 1, 0)), (f_2, \text{Params}, (1, 1, 0)), (f_2, \text{Cfg}, (x, y, z)), (f_3, \text{Cfg},$

[2] It is important to note that many 4-by-4 adjacency matrices are isomorphic to one another. As we were constructing a 16 bit value for comparison we needed to make sure all isomorphic adjacency matrices generated the same 16 bit value. As part of our experimental setup we constructed a 4-by-4 adjacency matrix for each possible 16 bit value. All isomorphic adjacency matrices were grouped together and for each group a representative 16 bit value was chosen to be returned.

$(x, y, z))\}$. Note that none of the signature values in DA is unique.
Then $\text{JoinSig}(f_1, DA) = \{(\text{Params}, (1, 1, 0))\}$. However, this consists of a single signature type, and our algorithm will separately use the Params (1, 1, 0) signature to attempt to match functions in A and B. Therefore, the JoinSig does not contribute any new information in the attempt to match f_1 to a function in B. Similarly, f_3 does not benefit from JoinSig in this case. However, $\text{JoinSig}(f_2, DA) = \{(\text{Params}, (1, 1, 0)), (\text{Cfg}, (x, y, z))\}$. Neither f_1 nor f_3 share this JoinSig; it is therefore unique and may be used to search for a matching function in B.

– *Global data reference matcher*: This new signature makes use of global data references contained in the binary program. The signature

$$\text{GdrMatcher} : (gdr, \Phi) \rightarrow \{x_1, x_2, ...\} \tag{10}$$

takes a global data reference gdr and Φ, and yields a set of functions where each $x_i \in \Phi$ contains a reference to gdr. If the signature value is unique within a program, then its global data reference gdr is uniquely shared among the named subgroup of functions. Also, this subgroup of functions is known to exist in both binaries, because each $x_i \in \Phi$ represents a matched function. If some global data reference from the other binary shares this unique signature, then our algorithm will equate them. When two variables are equated during the matching process, we insert the pair into a common set of shared variables Γ, just as functions are placed into Φ.

– *All labeled locations matchers*: These two signatures for matching functions make use of knowledge contained in both Φ and Γ. The signature

$$\text{AllLocalIds} : (fn, \Phi, \Gamma) \rightarrow (\{x_1, ...\}, \{y_1, ...\}) \tag{11}$$

takes as input a function fn, Φ, and Γ and yields a pair of sets. The $x_i \in \Phi$ are the previously matched functions that fn calls, and the $y_i \in \Gamma$ are the previously matched global data references that fn refers to.
Likewise, the signature

$$\text{AllParentIds} : (fn, \Phi, \Gamma) \rightarrow (\{x_1, ...\}, \{y_1, ...\}) \tag{12}$$

takes as input a function fn, Φ, and Γ and yields pair of sets. The $x_i \in \Phi$ are the previously matched functions that call fn, and the $y_i \in \Gamma$ are the previously matched global data references that are referred to by some x_i.

4.3 Ensemble classifier algorithm

The ensemble classifier algorithm maintains two sets for each binary; a set of unlabeled functions and a set of unlabeled global data references. The function StaticSig(S, ItemSet) takes a signature-generating function S to invoke and a set ItemSet of unlabeled items of the type S takes as input. It yields a pair of sets (Uniqs, Dups) where Uniqs = $\{(x_1, \text{Sig}_1), (x_2, \text{Sig}_2), ...\}$ with each item $x_i \in$ ItemSet, $S(x_i) = \text{Sig}_i$, and no Sig_i is repeated in the list. The set Dups = $\{(\text{Sig}_1, x_1, x_2, ...), ...\}$ where each signature Sig_i associated with at least two items from ItemSet. Every item in ItemSet appears either in Uniqs or Dups.

The function DynamicSig(S,A_f,Φ, Γ) works the same way as StaticSig except that it takes as arguments Φ and Γ, which as stated earlier are the sets of functions and globals, respectively, that have been deemed equivalent between the two programs. Dynamic signatures use information previously accumulated in Φ and Γ. Therefore, if Φ and Γ are empty then no new signatures will be generated by DynamicSig.

The function EquateEquivs(Uniqs_A, Uniqs_B, ItemSet_A, ItemSet_B, LabeledSet) processes Uniqs sets. Any Sig_i found in both Uniqs_A and Uniqs_B describes an item that is considered equivalent in the two binaries. The items paired with Sig_i in their corresponding Uniqs set are removed from their

Algorithm 1 Ensemble classifier

```
Require: $A_f$ and $B_f$ to be unlabeled function locations.
  $A_g$ and $B_g$ to be unlabeled global data references locations.
  $\Phi = \{\}$
  $\Gamma = \{\}$
  $(U_A, DA_{dlls}) = \text{StaticSig}(\text{DllName}, A_f)$
  $(U_B, DB_{dlls}) = \text{StaticSig}(\text{DllName}, B_f)$
  $\text{EquateEquivs}(U_A, U_B, A_f, B_f, \Phi)$
  repeat
    for $S \in \{\text{MD5}_{\text{bin}}, \text{MD5}_{\text{opcode}}, \text{Cfg}, \text{Params}\}$ do
      $(U_A, DA_S) = \text{StaticSig}(S, A_f)$
      $(U_B, DB_S) = \text{StaticSig}(S, B_f)$
      $\text{EquateEquivs}(U_A, U_B, A_f, B_f, \Phi)$
    end for
    for
    $S \in \{\text{CallRefs}, \text{ParentRefs}, \text{AllLocalIds}, \text{AllParentIds}\}$
     do
      $(U_A, DA_S) = \text{DynamicSig}(S, A_f, \Phi, \Gamma)$
      $(U_B, DB_S) = \text{DynamicSig}(S, B_f, \Phi, \Gamma)$
      $\text{EquateEquivs}(U_A, U_B, A_f, B_f, \Phi)$
    end for
    $DA = \bigcup DA_S$
    $DB = \bigcup DB_S$
    $(U_A, U_B) = \text{JoinSigsOfDups}(DA, DB)$
    $\text{EquateEquivs}(U_A, U_B, A_f, B_f, \Phi)$
    $U_A = \text{DynamicSig}(\text{GdrMatcher}, A_g, \Phi, \Gamma)$
    $U_B = \text{DynamicSig}(\text{GdrMatcher}, B_g, \Phi, \Gamma)$
    $\text{EquateEquivs}(U_A, U_B, A_g, B_g, \Gamma)$
  until $\Phi$ stops growing
  return $\Phi$,$\Gamma$
```

respective ItemSets and are paired together and appended to LabeledSet: specifically, function pairs are appended to Φ and global data references to Γ.

The algorithm starts by using DllName (see (2)) to identify common imported functions. As all imported functions are guaranteed to be unique this is done only once. It then enters a repeat/until loop which iterates until no new function pairs are appended to Φ. At the start of each iteration the algorithm executes each of the static signature classifiers to equate functions. Although the static signatures themselves do not change, their classification as unique or duplicate may change, because the sets A_f and B_f shrink as functions are equated. The algorithm then equates functions using our dynamic signatures.

After all function based signatures are used individually we call JoinSigsOfDups(DA, DB), where DA and DB are the sets of all the Dups sets generated by all the individual signature algorithms. This function uses the JoinSig (see (9)) and the supplied Dups sets to identify those JoinSigs that are unique among the JoinSigs in their own binary. These unique signatures are then used to match functions, if possible.

Finally, after all signatures primarily based on functions have been used, we use that information in GdrMatcher to correlate global data references into Γ. After all global data references have been correlated this iteration is complete. If Φ had no new functions appended during this iteration the loop is terminated.

Each iteration removes at least one function from A_f and B_f. Therefore, the maximum number of iterations for our ensemble is bounded by the number of the unlabeled functions in the input programs.

4.4 Measure of similarity

Upon termination the ensemble has computed a set of function matches Φ and data reference matches Γ for the given pair of binaries. To represent the correlation of the binaries we use the measure of similarity between two programs developed in [6]. In computing this measure let programs A and B have a total of i and j compiled functions respectively and let $n = |\Phi|$, the number of functions deemed equivalent between the two programs. The fraction n/i is in the interval of [0, 1] and relates what percentage of the functions in program A are in the common set. Similarly, n/j relates information pertaining to program B. The metric

$$Sim_{\Phi} = \frac{n^2}{ij} \tag{13}$$

will also be in the interval [0, 1] and will be used to relate the overall percentage of functions in the common set.

In a corresponding way we let $n = |\Gamma|$ and i, j represent the global data references in the two programs. The same computation with n, i, j will produce Sim_{Γ}. We use this to relate the percentage of global data references in the common set.

5 Experiment

5.1 Data set

We acquired the malware data set from [21] to conduct our experiment. There were 4200 programs that were unpackable with RAR [22], UPX [23], or that required no unpacking and appeared to be non-obfuscated. For the experiment we selected the 638 programs that seemed UPX-unpackable and unpacked them for analysis. The name-based taxonomy included with these binaries indicated that some were already considered variants of each other while others were not known to be related.

5.2 Performance model

To show that our new signatures (10), (11), and (12) offer improvement, we need an improvement measure other than a statistical test for accuracy. This is simply because no such objective accuracy test exists.

The data set does contain labels indicating that certain binaries are considered variants of each other. However, malware classification and taxonomy construction is an open problem [24]. We cannot assume that the labeling in our input data set is complete or even consistent. There is evidence that malware authors reuse code, so if we discover more matches among code than are reported in the provided labels, we should not necessarily conclude that our matches are incorrect. And although malware authors may rewrite code merely in order to disguise it, they may also rewrite code in order to change its functionality. In that situation, decreasing the computed numerical correlation between two generations of the same program is appropriate. Therefore, there is currently no objective way to prove that one classifier ensemble is more accurate than another simply because it produces more (or fewer) matches. More generally, there are no universally accepted quantifiable software engineering metrics for measuring changes in binary code. Thus, developing a statistical test for improvement of accuracy is not practical when comparing classifiers in sets of compiled programs.

Instead of making a case for the improvement of accuracy, we demonstrate that by including our new signatures we have created a new classifier that should be used in place of another one. If our new classifier produces what we think is a rational behavior at least as often as the original version, following [25] we will say that the new classifier is *dominant* for this classification task.

Given the following criteria, we claim our new ensemble dominates the original. Let s_1 and s_2 be programs, LabeledVariants be a function that determines whether they are labeled as variants in the input data set, and Matches(*classifier*) be the number of functions matched by *classifier* when comparing s_1 and s_2. Here *classifier* is either C_1, the old ensemble of signatures (2–9), or C_2, which also includes our new signatures (10–12). The function VarsFound returns true when C_2 finds s_1 and s_2 share at least one variable. In (14) we simply accumulate votes for which classifier does a better job based on the available information. The classifier that accumulates the higher total of votes will be considered the dominant classifier.

$$dom(s1, s2) = \begin{cases} C_1 & \text{if not LabeledVariants}(s_1, s_2) \text{ and} \\ & \quad \text{Matches}(C_1) < \text{Matches}(C_2) \\ C_1 & \text{if LabeledVariants}(s_1,s_2) \text{ and} \\ & \quad \text{Matches}(C_1) > \text{Matches}(C_2) \\ C_1 & \text{if not LabeledVariants}(s_1, s_2) \text{ and} \\ & \quad \text{Matches}(C_1) = \text{Matches}(C_2) \\ & \quad \text{and VarsFound}(s_1, s_2) \\ C_1 & \text{if LabeledVariants}(s_1, s_2) \text{ and} \\ & \quad \text{Matches}(C_1) = \text{Matches}(C_2) \\ & \quad \text{and not VarsFound}(s_1, s_2) \\ C_2 & \text{otherwise} \end{cases} \tag{14}$$

For the remainder of the paper we will say the classifier without our new signatures is the *original ensemble classifier*, and the *new ensemble classifier* contains our new signatures.

5.3 Similarity results

Analysis of the 638 programs resulted in 203203 pair wise program comparisons, each of which resulted in a vote for a classifier as defined in (14). This number includes 800 program pairs labeled as variants in the input data set and the remaining 202403 not designated variants. This labeling was done by the maintainers of the malware repository [21]. The experiment resulted in a total of 149590 (74%) votes in favor of our new classifier. Note that the overwhelming majority of pairs (99.6%) are not labeled as variants. When comparing such pairs, equation (14) results in a vote cast for the *old* classifier if the new classifier matches more functions than the old one does. Thus this result shows that on 74% of all pairs, the new classifier is not finding any more function matches than the old one did.

Of those pairs that *were* labeled variants in the input, there were 610 (76%) votes for the new classifier. This means that the new classifier was either able to match more functions or it matched the same number and was also able to match a global variable.

Together, these results indicate that the new classifier is dominant for the classification task on this data set.

We also examined the individual changes in matchings that come from incorporating global data references.

Figure 1 depicts the operation of the original ensemble in relation to the new ensemble. Each point in the graph represents comparisons of two programs that are labeled as variants in our data set. The x-coordinate is the percentage Sim_Φ for the original ensemble. The y-coordinate is the percentage Sim_Φ for our new ensemble.

In the 800 pairings of programs labeled variants, sixty-eight of them resulted in a higher Sim_Φ value and nine pairings resulted in a lower value. The decrease in Sim_Φ in these nine pairings was due to the new classifier matching twelve fewer function pairs and constructing one different function pairing. We manually reviewed these twelve function pairs. In all but one case our manual review agreed there was reason to remove each of the matchings; in the manual review of the one different pairing, we agreed with the new ensemble.

We expected that programs that had been previously labeled as variants of each other would have generally higher degrees of similarity than unrelated ones. We believed this would allow for a number of variables identified in one version to be transferred to another. We took every pair of programs labeled variants and plotted Sim_Φ as a percent versus the percentage of variables named in Fig. 2. As programs share more functions in common, we are able to detect more shared data references.

In general, we are able to correlate about 9% global data references as computed by Sim_Γ when two programs labeled variants share 50% code as computed by Sim_Φ. The question arises: what is the quality of the matches made by our ensemble? We picked global data reference matches generated by our ensemble at random and reviewed the code manually. Program pairings that shared less than 25% code had many variable matches in which our manual review disagreed with our ensemble. But when programs exhibit function similarity

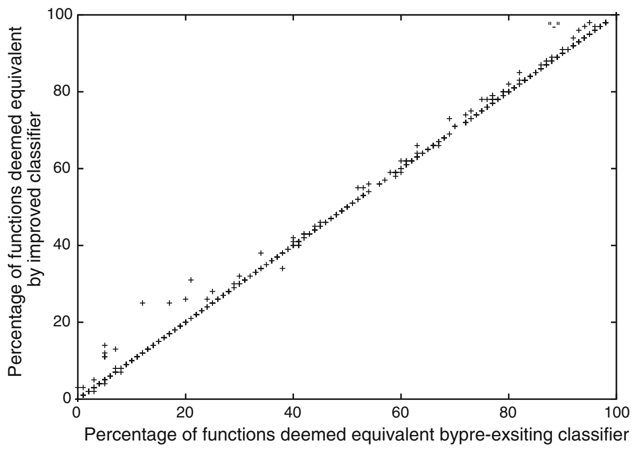

Fig. 1 Function matches in the original ensemble compared to the new ensemble on binaries that are labeled variants

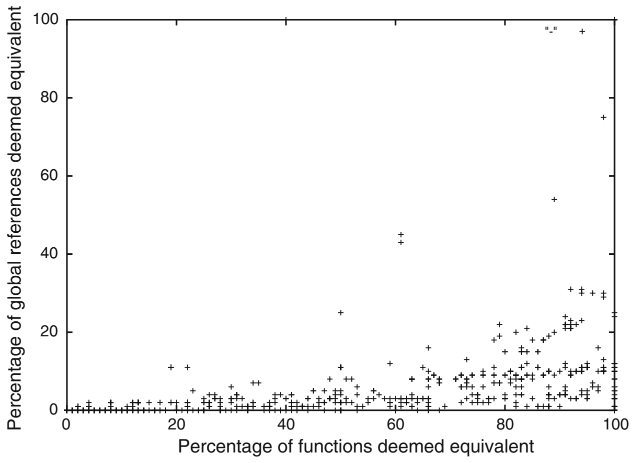

Fig. 2 Named functions versus named global data references

metrics above 50%, the variables matched generally seemed to have similar usage roles in the two executables.

Thus it appears that global variable matching is better used for associating variables in executables that are suspected to be related, and boosting confidence in that assessment, than as a primary means for determining whether two executables are in fact variants of each other. We revisit this question in Sect. 5.4.

We next compare a subset of our results with those generated by Carrera et al. [6]. Carrera had access to a different subset of samples than we did. Carrera split his results into two tables, one for variants A, B, C, D, E, and F with another for variants H, I, J, L, and M. For space reasons we show only the latter in Table 1, results for the former are similar. Each filled entry represents the $Sim_\Phi \times 100$ value generated by (C)arrera on the left, our (O)riginal ensemble in the middle, and our (N)ew ensemble on the right. This table shows that our results are generally consistent with Carrera's.

The most interesting result was the discovery of programs, which while unrelated by the taxonomy, still seemed to share a large number of functions and global data references. No analysis has been conducted yet to determine the nature of the units of code shared among these executables. However, both the percentage and sheer number of matched functions between the executables strongly suggests that the VxHeavens names for these programs do not adequately reflect their shared ancestry. Furthermore, the matched data references provide important points of reference for an analyst trying to discern the shared behaviors of the matched programs.

Table 1 Carrera's and our results for Email-Worm.Win32.Mimail samples H, I, J, L, and M

	I			J			L			M		
	C	O	N	C	O	N	C	O	N	C	O	N
H	82	84	84	83	84	84	91	94	94	89	96	96
I				95	98	98	82	79	81	80	85	85
J							83	79	81	81	85	85
L										90	98	98

Table 2 Top five programs thought unrelated by name with percentage of the functions and global data references shared

		Fns		Data	
Program A	Program B	#	%	#	%
Trojan-Spy.Banker.di	Trojan-Spy.Coiboa.b	520	89	28	19
Trojan.ZomJoiner.01.a	Trojan-PSW.Zombie.12	493	88	82	17
Backdoor.Fluxay.0473	HackTool.SmbCrack.4	560	74	78	8
Backdoor.DarkSky.b	Trojan.ZomJoiner.01.a	479	76	56	7
Backdoor.DarkSky.b	Trojan-PSW.M2.147	490	71	81	14

Table 3 Statistics generated from votes cast

Total pairs	203203
Number of variants	800
Precision	1%
Recall	77%
Accuracy	73%
False negatives	22%
False positives	26%

Table 2 shows our top five novel discoveries of this code sharing. Columns one and two are the names of the programs, columns three and four show the number and percentage of functions shared according to Sim_Φ, and finally columns five and six show the number and percentage of the total data references shared as according to Sim_Γ.

5.4 Using global data references as a discriminator

We also considered using global data references to predict whether two binaries are variants. For this experiment, we regarded the input labels of executables as ground truth and attempted to determine whether two executables are variants of each other. If we detected at least one shared global data reference, then we reported that the two programs are variants; otherwise we reported that they were unrelated.

From the votes tabulated we constructed five standard statistical measures, shown in Table 3. Precision indicates what percentage of examples correctly classified as variants were predicted to be variants. Recall shows what percentage of actual variants are correctly classified. Accuracy shows the percentage of correct classification decisions. False positives show the percentage of negative examples incorrectly classified as positives. False negatives show the percentage of positive examples incorrectly classified as negatives.

The good recall and accuracy suggests that our new ensemble makes reasonable assessments about whether two programs may be variants. More analysis needs to be done to understand the nature of the false negatives; it may simply be that those variants do not share any correlatable data references. The low precision and high false positive percentages are not surprising: we found evidence of code sharing among programs not designated as variants. We believe that the results generated in Table 2 are not spurious, and instead reveal commonality not designated by the current taxonomy.

6 Conclusion and future work

We have shown that several weak classifiers can be combined to build a stronger ensemble classifier for use in reverse engineering. The ensemble we presented can be used to correlate global data references between binaries. Among binaries known to be related, we can correlate roughly 9% of the total data references using Sim_Γ as our measure.

Thus, we believe we can answer yes to both questions posed in the introduction: correlating data references does lead to better function correlations, and correlated data references do usefully predict whether two binaries are related.

There is no standard quantitative measure for comparing binaries. As stated earlier, we incorporated the similarity metric used by [6] to produce an overall correlation metric between two programs. While the quantitative metrics of Sim_Γ and Sim_Φ do not adequately describe the correlations found, they at least allow us a standard measure for comparison. A robust quantitative metric for comparing binaries would allow researchers to compare results in a standardized way. We believe quantitative change metrics is an important area for future investigation.

We stated that when executables shared 50% of a common code base most data references matched were almost always corroborated by the manual review whereas with fewer than 25% sharing the matches were often error prone. We plan to conduct a more thorough review to plot the decline in the quality of matches with the decrease in the shared code base. Such a curve would potentially allow us to know how much confidence can be placed in the matches generated.

The modular construction of this ensemble classifier allows for new signatures to be added quickly. We plan on extending our data reference signatures. We plan to correlate data references based on how they are accessed in functions. And we plan on incorporating the roles of variables [26] into our framework as fine grain signatures on global data references. Lastly, except for the MD5 checksum signatures, this type of analysis is independent of microprocessor, so another line of analysis would be to analyze variants compiled for different microprocessors.

Acknowledgments We thank Dr. Richard Greene and Dr. Desiree Beck of The MITRE Corporation, and Dr. Karen Daniels of The University of Massachusetts at Lowell for taking an interest in this research and providing valuable insights. We also thank Stacey Arnold as well as the anonymous reviewers for their time and thoughtful suggestions.

References

1. Rozinov, K.: Reverse code engineering: an in-depth analysis of the Bagle virus. In: Systems, Man and Cybernetics (SMC) Information Assurance Workshop, 2005. Proceedings from the Sixth Annual IEEE, pp. 380–387 (2005)
2. Gordon, J.: Lessons from virus developers: The beagle worm history through April **24** (2004)
3. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic Worm Detection Using Structural Information of Executables. In: Rapid Advances in Intrusion Detection (RAID), pp. 207–226 (2005)
4. Schulman, A.: Finding binary clones with opstrings and function digests. Dr. Dobbs **374, 375, 376**, 69–73, 56–61, 64–70 (2005)
5. Dullien, T., Rolles, R.: Graph-based comparison of executable objects. In: Proceedings of the Symposium sur la Sécurité des Technologies de lìnformation et des Communications (SSTIC), pp. 421–433 (2005). http://www.sstic.org/
6. Carrera, E., Erdélyi, G.: Digital genome mapping—advanced binary malware analysis. Virus Bulletin Conference, pp. 187–197 (2004)
7. Flake, H.: Structural comparison of executable objects. In: Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment, pp. 161–174 (2004)
8. Sabin, T.: Comparing binaries using bindview. Technical report, Sabre (2004)
9. Karim, E., Walenstein, A., Lakhotia, A., Parida, L.: Malware phylogeny generation using permutations of code. J. Comput. Virol. **1**(1–2), 13–23 (2005)
10. Filiol, E.: Malware pattern scanning schemes secure against blackbox analysis. J. Comput. Virol. **2**(1), 35–50 (2006) EICAR 2006 Special Issue
11. Spinellis, D.: Reliable identification of bounded length viruses is NP-complete. IEEE Transactions on Information Theory, pp. 280–284 (2003)
12. Matching global data references in related executables (2007)
13. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically generating signatures for polymorphic worms. In: SP'05: Proceedings of the 2005 IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE Computer Society, pp. 226–241 (2005)
14. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM Conference on Computer and Communication Security, pp. 290–299 (2003)
15. Wehner, S.: Analyzing worms and network traffic using compression. J. Comput. Secur. **15**(3), 303–320 (2007)
16. Li, M., Vitányi, P.: An Introduction to Kolmogorov Complexity and Its Applications. Springer, Berlin (1997)
17. Freund, Y., Schapire, R.E.: Experiments with a new boosting algorithm. In: International Conference on Machine Learning, pp. 148–156 (1996)
18. Topchy, A., Jain, A.K., Punch, W.: Clustering ensembles: Models of consensus and weak partitions. IEEE Trans. Pattern Anal. Mach. Intell. **27**(12), 1866–1881 (2005)
19. Capriono, G.: REC Reverse Engineering Compiler, Version 1.6 (2000)
20. Toran, J.: On the hardness of graph isomorphism. SIAM J. Comput. **33**(5), 1093–1108 (2004)

21. Vx heavens website (2006)
22. Labs, R.: Rar Compression Homepage (2006)
23. UPX: Upx Homepage (2007)
24. Filiol, E., Helenius, M., Zanero, S.: Open problems in computer virology. J. Comput. Virol. **1**(3–4), 55–66 (2006)
25. Provost, F., Fawcett, T., Kohavi, R.: The case against accuracy estimation for comparing induction algorithms. In: Proceedings of the Fifteenth International Conference on Machine Learning, pp. 445–453 (1998)
26. Kuittinen, M., Sajaniemi, J.: Teaching roles of variables in elementary programming courses. In: ITiCSE'04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education, New York, NY, USA, pp. 57–61. ACM Press, New York (2004)