Identifying Shared Software Components to Support Malware Forensics

Brian Ruttenberg[1], Craig Miles[2], Lee Kellogg[1], Vivek Notani[2]
Michael Howard[1], Charles LeDoux[2], Arun Lakhotia[2], and Avi Pfeffer[1]

[1] Charles River Analytics
Cambridge, MA, USA
[2] Software Research Lab
University of Louisiana at Lafayette
Lafayette, LA, USA

Abstract. Recent reports from the anti-malware industry indicate similarity between malware code resulting from code reuse can aid in developing a profile of the attackers. We describe a method for identifying shared components in a large corpus of malware, where a component is a collection of code, such as a set of procedures, that implement a unit of functionality. We develop a general architecture for identifying shared components in a corpus using a two-stage clustering technique. While our method is parametrized on any features extracted from a binary, our implementation uses features abstracting the semantics of blocks of instructions. Our system has been found to identify shared components with extremely high accuracy in a rigorous, controlled experiment conducted independently by MITLL. Our technique provides an automated method to find between malware code functional relationships that may be used to establish evolutionary relationships and aid in forensics.

1 Introduction

Malware binaries are rich with information that can aid in developing a profile of the attacker. For instance, a detailed study of Stuxnet and Duqu worms led Kaspersky's researchers to conclude that they were developed using the same attack platform [16]. Similarly, after analyzing several years of malware data, Symantec concluded that the same authors had conducted industrial sector specific attacks [24], such as the defense, automotive, and financial sectors. Using a similar forensics analysis of malware repository, FireEye concluded that "many seemingly unrelated cyber attacks may, in fact, be part of a broader offensive" focused on certain targets [21]. The evidence to support all of these conclusions were found in the code. That there is similarity between malware code follows from the fact that a malware is a complex software developed using software engineering principles that encourage modularity, software reuse, use of program generators, and iterative development [35].

These insights have directed our efforts in a large project MAAGI [26] under the Defense Advanced Research Projects Agency (DARPA) Cyber Genome program, to determine the lineage and purpose of malware and its components. Since malware evolution is often guided by the sharing and adaptation of functional components that perform a desired purpose, the ability to identify shared *components* of malware is central to the problem of determining malware commonalities and lineage. A component can be thought of as a region of binary code that logically implements a "unit" of malicious operation. For example, the code responsible for key-logging would be a component. Components are intent driven, directed at providing a specific malware capability. Sharing of functional components across a malware corpus would thus provide an insight into the functional relationships between the binaries in a corpus and suggest connection between their attackers.

Detecting the existence of such shared components is not a trivial task. The function of a component is the same in each malware sample, but the *instantiation* of the component in each sample may vary. Different authors or different compilers and optimizations may cause variations in the binary code of each component, hindering detection of the shared patterns. In addition, the detection of these components is often unsupervised, that is the number of common components, their size, and their functions may not be known *a priori*.

The key contribution of this paper is an approach for unsupervised identification of shared functional components in a malware corpus. Our approach to this problem is based on an ensemble of techniques from program analysis, functional analysis, and data mining. Given a corpus of (unpacked) malware samples, each binary is reverse engineered and decomposed into a set of smaller functional units, namely procedures. The code in each procedure is then analyzed and converted into a representation of the code's semantics. Our innovation is in the development of a two-stage clustering method. In the first stage similar procedures from across the malware corpus are clustered. The clusters created are then used as features for the second stage clustering where each cluster represents a component. Our two-stage clustering is different from classic multi–stage clustering in which each stage refines the clusters created in the previous stage. In contrast, in our two-stage clustering, the clusters in each stage consists of different elements, representing different groupings.

Under supervision of DARPA, an independent verification and validation (IV&V) of our system was performed by Massachusetts Institute of Technology Lincoln Laboratory (MITLL). Their objective was to measure the effectiveness of the techniques under a variety of obfuscations used in real malware. The team constructed a collection of malware using a very methodical and systematic approach, carefully varying a variety of variables, such as compiler optimization, obfuscations, code evolution, and code reuse. The results of these controlled IV&V indicate that our method is very effective in detecting shared components in malware repositories.

The rest of this paper is organized as follows. Section 2 gives an overview of related works. Section 3 presents our novel approach to component identification and a probabilistic analysis of the method. Section 4 presents an overview of our system and design choices. A controlled experiment for evaluating the performance of the system and the results are presented in Section 5, which is followed by a exploratory study using "in-the-wild" malware in Section 6. Finally, we conclude in Section 7.

2 Related Works

Malware analysis work may be partitioned in three research areas: detection, clustering, and classification. We focus our attention to the latter two, as they are often aimed at supporting triage, and hence can be used for forensics. We direct the reader to other surveys for malware detection [13]. The methods for malware analysis may be classified as: static, dynamic, or hybrid. We use static analysis, since our goal is to find similar code fragments. Thus, we further restrict our focus to static analysis based clustering and classification. A survey of malware clustering and classification using dynamic analysis may be found elsewhere [9].

At the heart of clustering and classification is the problem of computing similarity (or distance) between two programs, which in turn calls for creating some abstraction of the programs. The abstractions commonly used are raw bytes [3,11,31,14,15,32,33,5], opcode and/or mnemonic [22,29,19,37], and abstract instruction [30,18], and instruction coloring [17,4,8]. When creating abstractions researchers also take advantages of the structure of the program as represented by its control flow graph (CFG) and call graph (CG) abstractions [8,7,17,4]. We use the semantic juice abstraction introduced by Lakhotia et al. [18] over a program's CFG. Having abstracted programs, the next issue is method for comparison. It is common to borrow methods from data mining [27], such as, Jaccard Similarity, edit distances, etc. When using graph representation, one may compare the structures using approximate graph isomorphism [7], creating finite sub-graph based features [17], or mapping a graph to a set of strings [4]. Though we do compute CFGs to create program abstractions, we use Jaccard Similarity for computing similarity between two abstractions.

Some researchers have also explored non-data mining methods for comparing malware programs. Gao et al. [10] use symbolic execution and theorem proving to determine when two functions or basic blocks are semantically equivalent. Linger et al. [20] compute the operational semantics of individual functions of a binary. Use of theorem proving to determine equivalence under register renaming is expensive. Semantic juice of Lakhotia et al. [18] allows us to use string comparisons to determine such equivalences.

Whereas prior works have focused on clustering or classifying entire malware, our goal is to find shared components between malware. Though the prior methods that classify or cluster whole programs could be used to cluster procedures, they cannot directly be used for the purposes of identifying shared components. Yavvari et al. [36] have attempted to extract common components in malware using a soft clustering technique. Their focus is in finding component similar to some given component of interest. In contrast, we do not start off knowing what code is interesting and instead search for all shared components.

In this work, as has also been done by prior work, we consider unpacking and deobfuscation as independent problems. There has been considerable

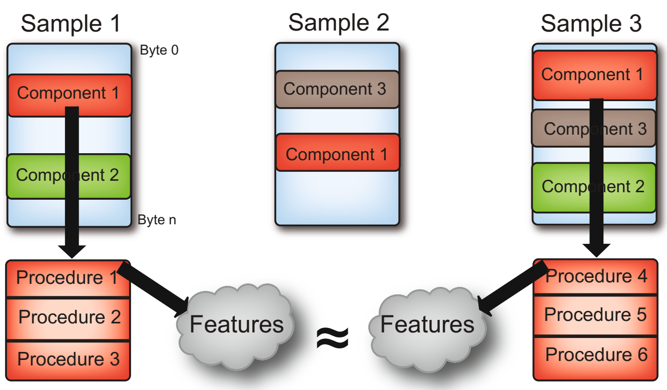


Fig. 1. Component generative process in malware. Instantiations of components in different malware samples should have similar features.

research in developing unpackers, such as, for packers that compress or encrypt binaries [2,4], for packers that use virtualization [28], extracting the unpacker code [6], classifying packed binaries [25]. A comprehensive survey of such works may be found in [4]. We assume that the malware we are analyzing has been unpacked using any of the variety of these methods.

3 Our Approach

We first describe the unsupervised clustering task, as it provides a more general framework for the component identification problem, and can be used with other code analysis techniques, not just the code analysis tool we used.

The basic idea of the unsupervised learning task can be thought of as reverse engineering a malware generative process, as shown in Fig. 1. A malware sample is composed of several shared components that perform malicious operations. Each component in turn is composed of one or more procedures. Likewise, we assume each procedure is represented by a set of features; in our system, features are extracted from the code blocks in the procedure, but abstractly, can be any meaningful features from a procedure.

The main idea behind our method is that features from the procedures should be similar between *instances* of the same component found in different samples. Due to authorship, polymorphism, and compiler variation or optimizations, they may not be exact, however, we expect that two functionally similar procedures instantiated in different samples should be more similar to each other than to a random procedure from the corpus. This generative process provides the foundation for our learning approach to discovery and identification of components.

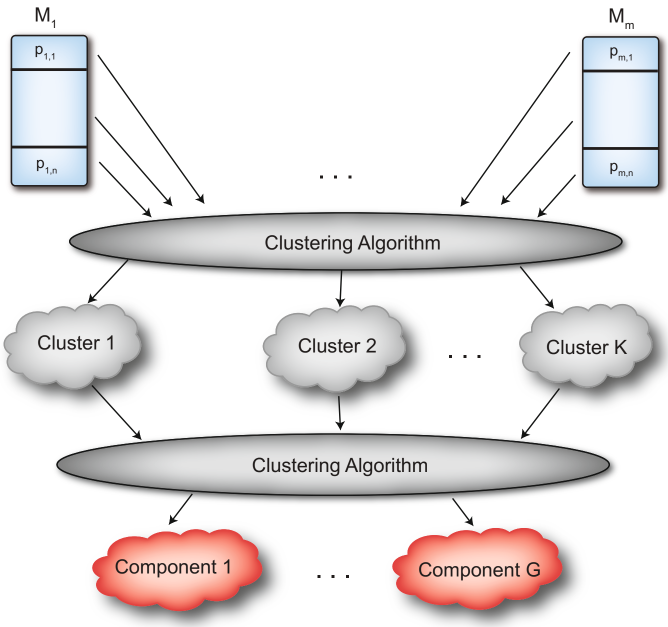


Fig. 2. Two-stage clustering procedure to identify shared components in a malware corpus. Procedures are clustered based on feature similarity, then the centroids are converted to the space of samples and clustered again. The resulting clusters are the shared functional components.

3.1 Basic Algorithm

Building off of the generative process that underlies components, we develop a two-stage clustering method to identify shared components in a set of malware samples, outlined in Fig. 2. For the moment, we assume that each procedure is composed of a set of meaningful features that describe the function of the procedure. Details on the features we employ in our implementation and evaluation can be found in Sections 4.1 and 5.

Given a corpus of malware samples $\mathbb{M} = \{M_1, \ldots, M_{|\mathbb{M}|}\}$, we assume it contains a set of shared functional components $\mathbb{T} = \{T_1, \ldots, T_{|\mathbb{T}|}\}$. However, we are only able to observe $T_{i,j}$, which is the *instantiation* of the i^{th} component in M_j. If the component is not part of the sample M_j, then $T_{i,j}$ is undefined. We also denote $T_{i,*}$ as the set of all instantiations of the i^{th} component in the entire corpus. Note that $T_{i,j}$ may not be an exact code replica of $T_{i,k}$, since the components could have some variation due to authorship and compilation. Each M_j consists of a set of procedures $p_{i,j}$, denoting the i^{th} procedure in the j^{th} sample.

Procedure-Based Clustering. The first stage of clustering is based on the notion that if $T_{i,j} \in M_j$ and $T_{i,k} \in M_k$, then at least one procedure in M_j must have high feature similarity to a procedure in M_k. Since components are shared across a corpus and represent common functions, even among different authors and compilers, it is likely that there is some similarity between the procedures. We first start out with a strong assumption and assert that the components in the corpus satisfy what we term as the *component uniqueness* property.

Definition 1. ***Component Uniqueness****. A component satisfies the component uniqueness property if the following relation holds true for all instantiations of $T_{i,*}$:*

$$\forall\, p_{x,j} \in T_{i,j},\, \exists\, p_{a,k} \in T_{i,k} \mid d(p_{x,j}, p_{a,k}) \ll d(p_{x,j}, p_{*,*}),$$
$$\forall\, p_{*,*} \in T_{*,k},\; T_{i,j}, T_{i,k} \in T_{i,*}$$

where $d(p_{*,*}, p_{*,*})$ is a distance function between the features of two procedures. Informally, this states that *all* procedures in each instantiation of a component are much more similar to a single procedure from the same component in a different sample than to all other procedures.

Given this idea, the first step in our algorithm is to cluster the *entire* set of procedures in a corpus. These clusters represent the common functional procedures found in *all* the samples, and by the component uniqueness property, similar procedures in instantiations of the same component will tend to cluster together. Of course, even with component uniqueness, we cannot guarantee that all like procedures in instantiations of a component will be clustered together; this is partially a function of the clustering algorithm employed. However, as we show in later sections, with appropriately discriminative distance functions and agglomerative clustering techniques, this clustering result is highly likely.

These discovered clusters, however, are not necessarily the common components in the corpus. Components can be composed of multiple procedures, which may exhibit little similarity to each other (uniqueness does not say anything about the similarity between procedures in the same component). In Fig. 1, for example, Component 1 contains three procedures in sample 1 and sample 3. After clustering, three clusters are likely formed, each with one procedure from sample 1 and 3. This behavior is often found in malware: A component may be composed of a procedure to open a registry file and another one to compute a new registry key. Such overall functionality is part of the same component, yet the procedures could be vastly dissimilar based on the extracted features. However, based on component uniqueness, procedures that are part of the same shared component should appear in the same *subset* of malware samples.

Sample-Based Clustering. Next, we perform a second step of clustering on the results from the first stage, but first convert the clusters from the space of procedure similarity to what we denote as sample similarity. Let C_i represent a procedure cluster, where $p_{x_1,y_1}, \ldots, p_{x_k,y_k} \in C_i$ are the procedures from the corpus that were placed into the cluster. We then represent C_i by a vector $\boldsymbol{S_i}$, where

$$S_i[j] = \begin{cases} 1 & \text{if } \exists\, p_{x_k,y_k} \in C_i \mid y_k = j \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

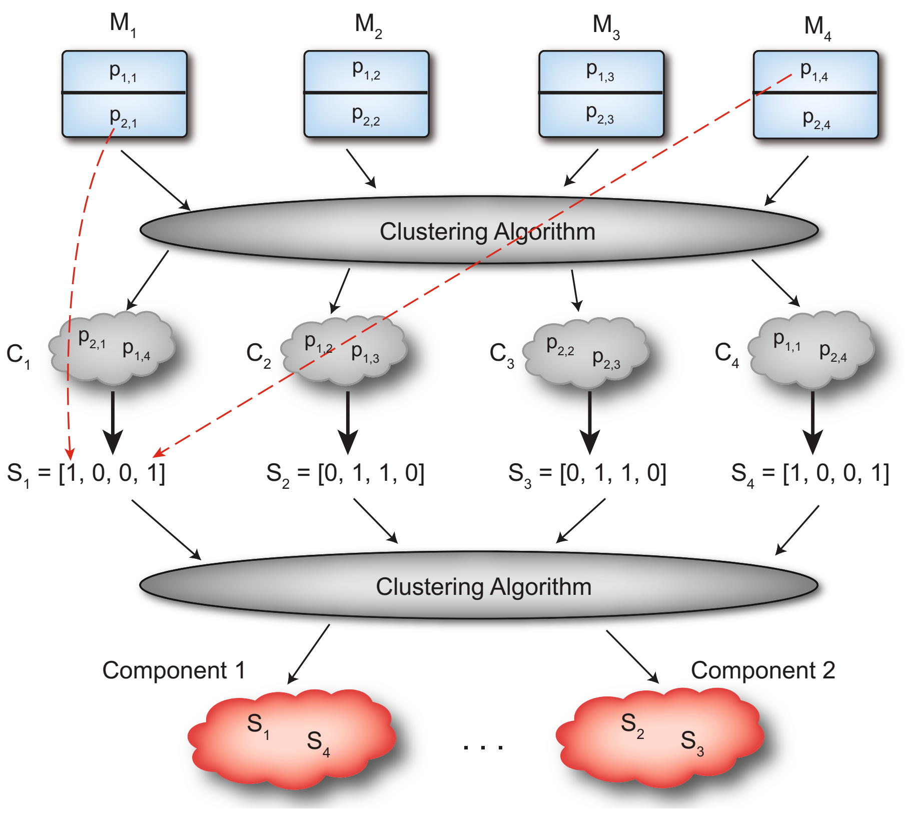


Fig. 3. Conversion of procedure clusters into a vector space representing the presence of a procedure from each sample in the cluster, and then the subsequent clustering of the vector space representations. The clusters of vector space representations are the shared components.

That is, $S_i[j]$ represents the presence of a procedure from M_j in the cluster. In this manner, each procedure cluster is converted into a point in an $|\mathbb{M}|$-dimensional space, where $|\mathbb{M}|$ is the number of malware samples in the corpus. Consider the example shown in Fig. 3. The procedures in the corpus have been clustered into four unique procedure clusters. Cluster C_1 contains procedures $p_{2,1}$ from M_1, and $p_{1,4}$ from sample M_4. Hence, we convert this cluster into the point $\boldsymbol{S_1} = [1, 0, 0, 1]$, and convert the other clusters into the vector space representation as well.

This conversion now allows us to group together clusters that appear in the same *subset* of malware samples. Using component uniqueness and a reasonable clustering algorithm in the first step, it is likely that a $p_{x,j} \in T_{i,j}$ has been placed in a cluster C_v with other like procedures from $T_{i,*}$. Similarly, it is also likely that a different procedure in the same component instance, $p_{y,j}$, is found in cluster C_w with other procedures from $T_{i,*}$. Since C_v and C_w both contain procedures from $T_{i,*}$, then they will contain procedures from the same set of samples, and therefore their vector representations $\boldsymbol{S_v}$ and $\boldsymbol{S_w}$ will be very similar. We can state this intuition more formally as

$$d(\boldsymbol{S_v}, \boldsymbol{S_w}) \approx 0 \Rightarrow p_{x,j}, p_{y,k} \in T_{i,*} \,\forall\, p_{x,j}, p_{y,k} \in \{C_v, C_w\}$$

Based on these intuitions, we then cluster the newly generated $\boldsymbol{S_i}$ together to yield our components. Looking again at Fig. 3, we can see that when cluster C_1 and C_4 are converted to S_1 and S_4, they contain the same set of samples. These two procedure groups therefore constitute a single component instantiated in two samples, and would be combined into a single cluster in the second clustering step, as shown.

Analysis. Provided the component uniqueness property holds for all components in the data set, then the algorithm is very likely to discover all shared components in a malware corpus. However, if two components in the corpus are found in the exact same subset of malware samples, then they become indistinguishable in the second stage of clustering; the algorithm would incorrectly merge them both into a single cluster. Therefore, if each component is found in the corpus according to some prescribed distribution, we can compute the probability that two components are found in the same subset of malware.

Let $\mathcal{T}_i$ be a random variable denoting the set of malware samples that contain the i^{th} component. If $\mathcal{T}_i$ is distributed according to some distribution function, then for some $t = \{M_x, \ldots, M_y\} \subseteq \mathbb{M}$, we denote the probability of the component being found in exactly the set t as $Pr(\mathcal{T}_i = t)$. Assuming uniqueness holds, we can now determine the probability that a component is detected in the corpus.

Theorem 1. *The probability that the i^{th} component is detected in a set of malware samples is*

$$\sum_{t \in all\ subsets\ of \mathbb{M}} Pr(\mathcal{T}_k \neq t, \ldots, \mathcal{T}_i = t, \ldots, \mathcal{T}_k \neq t)$$

Proof. If $\mathcal{T}_i = t_j$ for some $t_j \subseteq \mathbb{M}$, the component will be detected if no other component in the corpus is found in the exact same subset. That is, $\mathcal{T}_k \neq t_j$ for all other components in the corpus. Assuming nothing about component or sample independence, the probability of no other component sharing t_j is the joint distribution $Pr(\mathcal{T}_k \neq t_j, \ldots, \mathcal{T}_i = t_j, \ldots, \mathcal{T}_k \neq t_j)$. Summing over all possible subsets of $\mathbb{M}$ then yields Thm 1.

Thm. 1 assumes nothing about component and sample independence. However, if we do assume that components are independent of each other and a component T_i appears independently in each sample with probability p_i, then $\mathcal{T}_i$ is distributed according to a binomial distribution. As such, we can compute a lower bound for the probability of detection by ignoring equality between distribution sets as

$$\begin{aligned}
&Pr(\text{Detection of } T_i)\\
&= \sum_{t \in \text{all subsets of} \mathbb{M}} Pr(\mathcal{T}_k \neq t, \ldots, \mathcal{T}_i = t, \ldots, \mathcal{T}_k \neq t)\\
&= \sum_{t \in \text{all subsets of} \mathbb{M}} Pr(\mathcal{T}_i = t) \prod_{k \neq i} (1 - Pr(\mathcal{T}_k = t))\\
&\geq \sum_{x=0}^{|\mathbb{M}|} Pr(|\mathcal{T}_i| = x) \prod_{k \neq i} (1 - Pr(|\mathcal{T}_k| = x))\\
&= \sum_{x=0}^{|\mathbb{M}|} Bin(x, |\mathbb{M}|, p_i) \prod_{k \neq i} (1 - Bin(x, |\mathbb{M}|, p_k))
\end{aligned}$$

where $Bin(\cdot)$ is the binomial probability distribution function. This lower bound can provide us with reasonable estimates on the probability of detection. For instance, even in a small data set of 20 samples with two components that both have a 20% chance of appearing in any sample, the probability of detection is at least 0.85.

Based on component uniqueness, the basic algorithm can easily locate the shared components in a set of malware samples. However, in practice, component uniqueness rarely holds in a malware corpus. That is, it is likely that some procedures in different components are quite similar. This situation can be quite problematic for the basic algorithm. In the next section, we relax the component uniqueness assumption and detail a more sophisticated algorithm intended to identify components.

3.2 Assumption Relaxation

When component uniqueness does not hold, the basic algorithm may not correctly identify components. Consider the example shown in Fig. 4. There are two components in four samples, each composed of two procedures. Assuming component uniqueness does not hold, then the second procedure in each sample could show high similarity to each other (it is possible they perform some basic function to set up malicious behavior). After the first step, $p_{2,1}$, $p_{2,2}$, $p_{2,3}$, and $p_{2,4}$ are placed in the same cluster; this results in creation of $\boldsymbol{S_2}$ that does not resemble any other cluster vectors. Hence, any clustering of $\boldsymbol{S_2}$ with $\boldsymbol{S_1}$ or $\boldsymbol{S_3}$ will result in a misidentification of the procedures in each component.

To remediate this error, we utilize an algorithm that "splits" clusters discovered in the first step of the algorithm before the second stage of clustering is performed. This requires that we relax the component uniqueness assumption in Def. 1 to what we term as *procedure uniqueness.*

Definition 2. ***Procedure Uniqueness.*** *A component satisfies the procedure uniqueness property if the following relation holds true for all instantiations of*

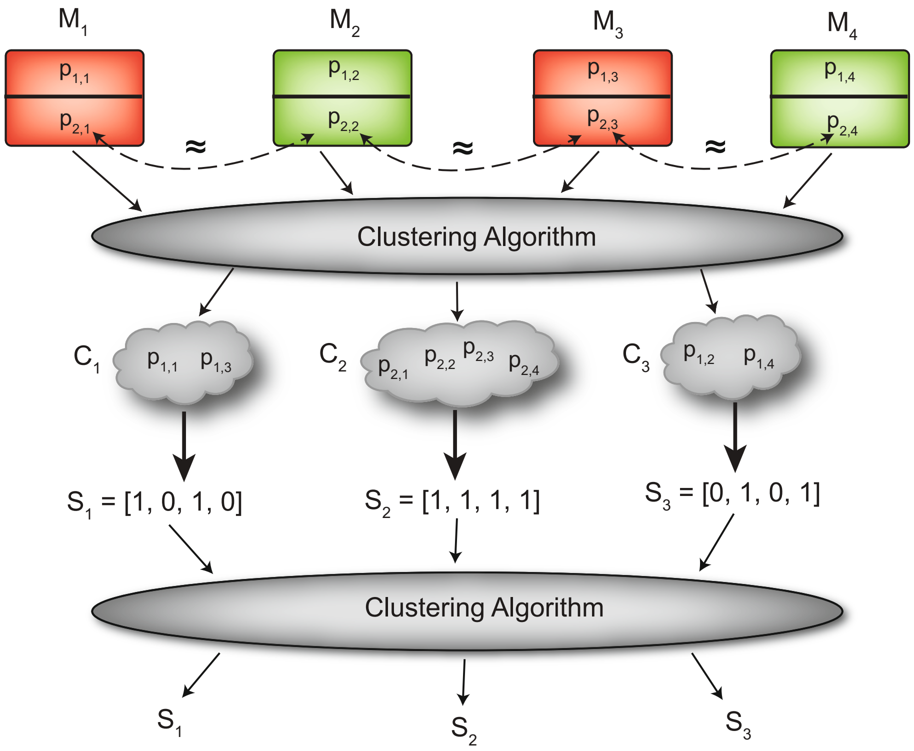


Fig. 4. Basic algorithm without the component uniqueness property. There are two components in the set, yet the end result clustering produces three, of which one is a combination of two components.

$T_{i,*}$:

$$\exists\ \mathcal{W} = \{p_{x,j} \in T_{i,j}, \ldots, p_{y,k} \in T_{i,k}\} \mid \\ \forall\ p_{x,j}, p_{y,k} \in \mathcal{W},\ d(p_{x,j}, p_{y,k}) \ll d(p_{x,j}, p_{*,*})\ \forall\, p_{*,*} \notin \mathcal{W}$$

This relaxation states that for a component to satisfy procedure uniqueness, only *one* procedure in an instantiation of a component must exhibit high similarity to a procedure in another instantiation (as opposed to component uniqueness where all procedures must satisfy this property). Furthermore, this similarity is transitive; if two procedures exhibit high similarity in two instantiations, both also exhibit high similarity to the same procedure in a third. For brevity, we assume there is only one procedure in each component satisfying this condition. We denote by $p'_{i,j}$ the procedure used to satisfy Def. 2 in each $T_{i,j}$.

The intuition is that after the first stage of clustering, there are clusters $C'_1 \ldots C'_{|\mathbb{T}|}$ that each contain the set of procedures $p'_{i,*}$. That is, from procedure uniqueness and a reasonable clustering algorithm, it is highly likely that we get $|\mathbb{T}|$ clusters, one for each component in the data set.

Using Thm. 1 and procedure uniqueness, we can now state the following corollary.

Corollary 1. *Let $\boldsymbol{S'_1} \dots \boldsymbol{S'_{|\mathbb{T}|}}$ be the conversion of each C'_i into sample space according to Eq. 1. Then $Pr(\boldsymbol{S'_i} \neq \boldsymbol{S'_j}) \forall j \neq i$ is defined according to Theorem 1*

This corollary is extremely important: It states that when each cluster from the first step is transformed into sample space, there exist (with high probability) $|\mathbb{T}|$ unique vectors. We will use these unique vectors to further refine the first stage clustering. We first state that each $\boldsymbol{S'_i}$ is *atomic*, where a vector $\boldsymbol{S_i}$ in set of vectors $\mathbb{S}$ is atomic if

$$\boldsymbol{S_i} \not\supseteq \boldsymbol{S_j} \;\forall\; \boldsymbol{S_j} \in \mathbb{S} \tag{2}$$

In other words, an atomic vector is not a super-set of any other vectors in a set. Atomic vectors can be used to "split" clusters discovered from the first stage of the algorithm. Each cluster is converted into $\boldsymbol{S_i}$, its sample space representation. Then, we determine the set of atomic vectors in the resulting set, which we assume corresponds to $S'_1, \dots, S'_{|\mathbb{T}|}$. Finally, for each non-atomic vector, we split the cluster it represents into k new clusters, one for each atomic vector that is a subset of the non-atomic vector. For example, let us assume that vector $\boldsymbol{S_i} \supset \{\boldsymbol{S'_1}, \boldsymbol{S'_2}\}$. We split vector $\boldsymbol{S_i}$ into two new vectors $\boldsymbol{S_{i_1}}$ and $\boldsymbol{S_{i_2}}$ where $\boldsymbol{S_{i_1}} = \boldsymbol{S'_1}$ and

$$C_{i_1} = \{p_{*,j} \mid S'_1[j] = 1,\, j = 1 \dots |\mathbb{M}|\} \tag{3}$$

That is, the cluster C_i is broken into two clusters, containing the procedures found in the samples labeled by $\boldsymbol{S'_1}$ and $\boldsymbol{S'_2}$, respectively. These two new clusters are then converted into the sample space vectors $\boldsymbol{S_{i_1}}$ and $\boldsymbol{S_{i_2}}$. The purpose of this splitting is to decompose clusters composed of procedures from multiple components into their atomic patterns, where each component is represented by an atomic vector.

Again consider the example shown in Fig. 4. $\boldsymbol{S_1}$ and $\boldsymbol{S_3}$ are atomic since they are not super-sets of any other vector. Since $\boldsymbol{S_2}$ is a super-set of the two atomic vectors, it is broken into two clusters, where C_{2_1} contains $p_{2,1}$ and $p_{2,3}$, and C_{2_2} contains $p_{2,2}$ and $p_{2,4}$. We then proceed with the second stage of clustering as previously described; in this case, C_1 and $C_{2,1}$ will be clustered together to form a component.

We now formulate the probability of component detection using the splitting method as

Theorem 2. *Assuming procedure uniqueness, the probability of correct component identification is defined according to Thm. 1*

Proof. Let $\{\boldsymbol{S_j}, \dots, \boldsymbol{S_k}\} \cup \{\boldsymbol{S'_1}, \dots, \boldsymbol{S'_{|\mathbb{T}|}}\}$ be the union of all non-atomic and atomic vectors after the first stage of clustering. After splitting the non-atomic vectors, we get $\{\boldsymbol{S_j}, \dots, \boldsymbol{S_k}\} = \{\boldsymbol{S_{j_1}}, \dots, \boldsymbol{S_{k_n}}\}$, where each $\boldsymbol{S_{k_n}}$ is equal to an atomic vector (from Eq. 3). Hence, $\{\boldsymbol{S_{1_1}}, \dots, \boldsymbol{S_{k_j}}\} \cup \{\boldsymbol{S'_1}, \dots, \boldsymbol{S'_{|\mathbb{T}|}}\} = \{\boldsymbol{S'_1}, \dots, \boldsymbol{S'_{|\mathbb{T}|}}\}$. Since $\{\boldsymbol{S'_1}, \dots, \boldsymbol{S'_{|\mathbb{T}|}}\}$ are atomic, then we know that $S'_i \neq S'_j$, and therefore $\mathcal{T}_i \neq \mathcal{T}_j$, which is the same probability as expressed in Thm. 1.

Using splitting, we can identify components with the exact same probability as assuming component uniqueness. While procedure uniqueness is a weak assumption, in reality, the set of atomic vectors discovered after the first stage of clustering may not correspond exactly to the number of components in the data. First, if some component in the corpus is found in a super-set of samples of another component's distribution, then the number of atomic vectors will be less than the number of components. In such an instance, a correctly identified procedure cluster may be broken apart and incorrectly merged with another component's procedures in the second clustering step. Our method is also susceptible to random procedure noise found in each sample. Noisy procedures that appear at random may be clustered together and be converted to a vector that is a subset of a real atomic vector $\boldsymbol{S}'_i$. Similar to a component being a super-set of another component, in this case a correct procedure cluster may be split apart. However, we have found that limiting the number of times a non-atomic vector is split and ensuring that each atomic vector has a minimum magnitude greatly reduces the chance that noisy procedures impact the method.

4 System Implementation

Our component identification system is intended to discover common functional sections of binary code shared across a corpus of malware. The number, size, function and distribution of these components is generally unknown, hence our system architecture reflects a combination of unsupervised learning methods coupled with semantic generalization of binary code. The system uses two main components:

1. BinJuice: To extract the procedures and generate a suitable *Juice* features.
2. Clustering Engine: To perform the actual clustering based on the features.

The malware samples input to the system are assumed to have been unpacked [4]. We use IDA Pro to disassemble each malware binary, decompose it into its procedures, and construct its CFG. The collection of procedures, with each procedure made of blocks, are used as features to an unsupervised learning algorithm.

4.1 BinJuice

We use Lakhotia et al.'s BinJuice system [18] to translate the code of each block (or a procedure) into four types of features: `code`, `semantics`, `gen_semantics`, and `gen_code`. The `code` feature is simply the disassembled code. The `semantics` feature gives the operational semantics of the block, computed using symbolic interpretation and algebraic simplification. It describes the cumulative effect of the instructions in the block on specific registers and memory locations. In contrast, `gen_semantics`, which Lakhotia et al. also term as "juice", abstracts away from the semantics the specific registers and memory locations, and makes the semantics a function of logic variables. This abstraction has the benefit that two code segments that are equivalent, except for the choice of registers and addresses of variables, have juice that is identical, modulo the choice of logic variables. Lakhotia et al. describe an encoding of juice that enables constant time test of equivalence of two juice terms. The `gen_code` feature is analogous to `gen_semantics` in that it is created by abstracting away the registers and constants of the corresponding `code`.

We thus have four feature representations for each procedure: `code`, `semantics`, `gen_semantics`, and `gen_code`. Since each of the features are strings, they may be represented using a fixed size hash, such as md5. For each representation, a procedure is thus a set of hashes, thus, ignoring the ordering of blocks. We measure similarity between a pair of procedures using the Jaccard index [34] of their sets of features.

4.2 Clustering Engine

For the first stage of clustering, we choose to use a data driven clustering method. Even if we know the number of shared components in a corpus, it is far less likely that we will know how many procedures are in each component. Thus, it makes sense to use a method that does not rely on prior knowledge of the number of procedure clusters.

We use Louvain clustering for the procedure clustering step [1]. Louvain clustering is a greedy agglomerative clustering method, originally formulated as a graph clustering algorithm that uses modularity optimization [23]. We view procedures as nodes in a graph and the weights of the edges between the nodes as the Jaccard index between the procedure features. Modularity optimization attempts to maximize the modularity of a graph, which is defined as groups of procedures that have higher intra–group similarity and lower inter–group similarity than would be expected at random. Louvain clustering iteratively combines nodes together that increases the overall modularity of the graph until no more increase in modularity can be attained.

For the second stage, we experimented with two different clustering methods: Louvain and K–means. These methods represent two modalities of clustering, and various scenarios may need different methods of extracting components. For instance, in situations where we know a reasonable upper bound on the number of components in a corpus, we wanted to determine if traditional iterative clustering methods (i.e., K–means) could outperform a data driven approach. In the second step of clustering, the L_2 distance between vectors was used for K–means, and since Louvain clustering operates on similarity (as opposed to distance), an inverted and scaled version of the L_2 distance was employed for the second stage Louvain clustering.

5 Experimental Evaluation

It is quite a challenge to perform scientifically valid controlled experiments that would estimate the performance of a malware analaysis system in the real-world. The challenges are

1. Obtaining malware samples with known ground truth such that the correctness of the results produced by the system can be verified and,
2. Having a collection of samples that represents the distribution of malware in the wild.

While there are malware repositories that contain data that can be used for evaluating malware detectors and classifiers, there are no such repositories that contain validated information about components within a malware. That is, for each malware we do not know the exact virtual memory addresses of each byte that is part of a particular component.

To address this pitfall, our sponsor, DARPA, recruited MITLL to create a collection of malware binaries with byte-labelled components, that is, for each component they know the exact virtual memory addresses of each byte that is part of the component in every malware. Our system was subject to a controlled experiment using this dataset for independent verification and validation (IV&V).

In the section below, we discuss this controlled experiment. We first present the data-set used for IV&V, followed by the quality metrics used to analyze the test results, then present the results, followed by a performance and scalability analysis of our system.

5.1 Data Sets

The malware used for IV&V was based on actual malware source code that performs a variety of functions (e.g., key logging, clip board stealing, etc). The source code was acquired by DARPA, combined into different executables, and compiled using various flags into Windows 32-bit binaries. There are three data sets associated with this data, TC1, TC2 and TC3. TC1 contains 50 samples of malware and eight components. TC2 contains same eight components, but added compiler variations (e.g., optimizations on or off) to produce a data set of 250 malware samples. Finally, TC3 contained 27 total components over 500 malware samples, where 250 of the malware samples are the same ones from TC2.

Note that in all tests, the algorithms do not have prior knowledge of the number of components in the data set. For the K-means tests, we set a reasonable upper bound on the estimated number of components. For IV&V we used $K = 50$.

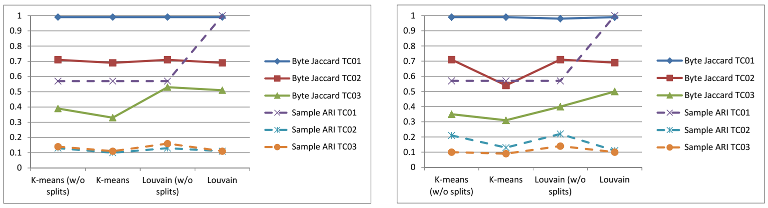
5.2 Quality Metrics

The quality metrics employed are motivated by MITLL's testing methodology. The ultimate goal of DARPA is to identify sections of binary code that are shared among malware. Since the ground truth of each data set can provide the byte level virtual addresses of each component in the malware, the most accurate method to measure the quality of component identification is using a byte-level Jaccard index. To do so, however, requires that our algorithm labels identified components using the same label set as the ground truth. Therefore, *after* we have identified the components in the malware using our algorithm, we are provided with the virtual address byte labels of the $|\mathbb{T}|$ components in $|\mathbb{T}|$ different samples (the byte locations on the rest of the samples are not provided; those are only used during evaluation by the sponsoring agency). We then create a mapping from our discovered components to the revealed components by greedily assigning the best match of our components to the revealed ones, where multiple discovered components can be assigned to a single revealed component. Finally, we compute the Jaccard index between the bytes labeled by our component identification with the ground truth identified byte labels.

We also used an additional metrics based on the Adjusted Rand Index [12]. The ARI is a method to compare two clusterings of a data set as compared to a random clustering. Values closer to one indicate that the two clusterings tend to group procedures in a similar manner. ARI values of zero correspond to random guessing. After the mapping is complete, each malware sample is labeled with a binary vector where the i^{th} bit indicates that the sample contains component i. A vector is created for each sample, and we treat it as the output of a clustering algorithm. We create a set of vectors for the discovered component labeling and the ground truth labeling, and compare them using the ARI. Note that in general, the Jaccard index is a much more accurate assessment of the quality of component identification, as with the ARI metrics we can still receive a perfect 1.0 score even if we don't match exactly on the byte labels, since falsely identified components are not penalized.

5.3 Results

We ran all of our tests using the two clustering algorithms (Louvain and K-means), and additionally tested each method with and without splits to determine how much the relaxation of component uniqueness helps the results. Note that no parameters (besides K) were needed for evaluation; we utilize a completely data driven and unsupervised approach.



(a) gen_code (b) gen_semantics

Fig. 5. Byte Jaccard and sample ARI comparisons of the different methods on the IV&V data-set using three BinJuice features

The results of the component identification on the IV&V data-set are shown in Fig. 5, where each metric is shown for all three data sets (TC1, TC2 and TC3). The `code` and `semantics` feature, as expected, produced inferior results as compared to `gen_code` and `gen_semantics` features during initial testing. Hence subsequent testing on those feature was discontinued.

In general, all four methods have fairly low ARIs regardless of the BinJuice feature. This indicates that in our component identifications the false positives are distributed across the malware collection, as opposed to concentrated in a few samples. Furthermore, as indicated by the Jaccard index results, the misclassification rate at the byte level is not too high. The data also shows that the Louvain method outperforms K-means on all BinJuice features, though in some cases the splitting does not help significantly. The `gen_code` and `gen_semantics` features of BinJuice also provide the best abstraction of the code for component identification. Note that the difference between Louvain with and without splitting is mainly in the sample ARI. Since Louvain without splitting is not able to break clusters up, it mistakenly identifies non-component code in the malware as part of a real component; hence, it believes that samples contain many more components than they actually do. These results demonstrate the robustness of the Louvain method and the strength of the BinJuice generated features. The data also shows that relaxing the component uniqueness property can improve the results in real malware.

5.4 Performance and Scalability

In Fig. 6 we show the component identification time on the IV&V data-set using the Louvain method (the results are nearly identical using K-means). As the number of components in the data set increases, so does the time to identify the components. This is due to the fact that our algorithm clusters procedures, so as the number of components increases, so does the number of procedures. Not surprisingly, as the number of samples in the data set is increased, the time to identify components also increases. The first clustering stage in the algorithm must compute a distance matrix between all procedures in the data set, which increases with the number of samples.

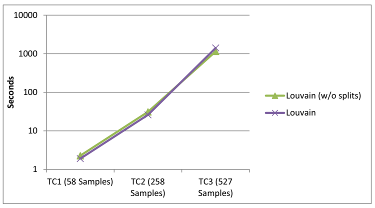
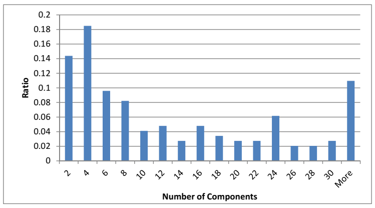
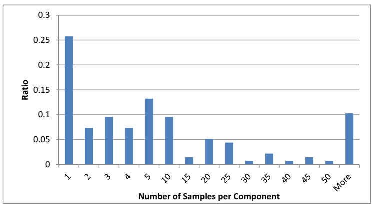


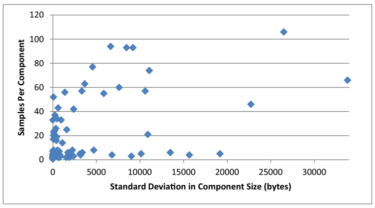
Fig. 6. Time to identify components on the IV&V data-set. The results using K-means are not shown as they are nearly identical to the Louvain results.



(a) Components per sample



(b) Samples per component



(c) Component size compared to samples per component

Fig. 7. Histograms of the number of components found in each sample and the number of samples per identified component for the wild malware. In addition, we also show the variation in component size as a function of the number of samples containing each component.

While there are many possible avenues of making the component identification process more efficient, any clustering algorithm must ultimately have access to the distance between any two arbitrary procedures in the corpus. Thus, even with scalability enhancements, we do not foresee component identification being performed on large, arbitrary malware corpora. Rather, we envision this task will be performed on specific malware families or specialized corpora of moderate size.

6 Study with Wild Malware

We also performed component identification on a small data set of wild malware consisting of 136 wild malware samples provided by DARPA. No other filtering or selection was performed on these samples. We identified a total of 135 unique components in the data set. The similarity between the number of samples (136) and the number of unique components (135) is coincidental, as evident from the following discussion.

On an average 13 components were identified per malware sample. Fig. 7(a) shows the histogram of the number of components discovered per sample. As evident from the graph, the distribution is not uniform. Most malware samples have few components, though some can have a very large number of shared components. In addition, we also show the number of samples per identified component in Fig. 7(b). As can be seen, most components are only found in a few samples. For example, 25% of components are only found in a single sample, and thus would most likely not be of interest to a malware analyst (as components must be shared among malware samples in our definition).

In general, many of the identified components are similar in size (bytes), as shown in Fig. 7(c). In the figure, we plot the variance of the size of the instantiations in each of the 135 components against the number of samples that contain the component. As can be seen, many of the samples have low variance in their component size, indicating that it is likely that many of the components are representing the same shared function (components with large variation in observed instantiation size are likely false positive components). In addition, many of these low variance components are non-singleton components, meaning that the component has been observed in many malware samples. While further investigation is needed to determine the exact function and purpose of these components, these results do indicate that our method is capable of extracting shared components in a corpus of wild malware.

7 Conclusions

We have described a method for identifying functional components that are shared across a corpus of malware. We utilize an innovative two-step clustering procedure to group together similar procedures into shared components, even when there are similar pieces of code found in each component. Using features constructed from abstracted semantics of basic blocks of a binary, we demonstrate that our method can identify shared components in a malware corpus with high accuracy down to the byte level. As malware becomes more prevalent and sophisticated, determining the commonalities between disparate pieces of malware will be key in thwarting attacks or tracking their perpetrators. We plan to continue working on enhancing our algorithm for component identification, and apply it towards our larger goal of understanding the lineage and evolution of malware.

Acknowledgments. This work was supported by DARPA under US Air Force contract FA8750-10-C-0171, with thanks to Mr. Timothy Fraser. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

1. Blondel, V.D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment 2008(10), P10008 (2008)
2. Böhne, L.: Pandora's bochs: Automated malware unpacking. Master's thesis, University of Mannheim (2008)
3. Caillat, B., Desnos, A., Erra, R.: Binthavro: Towards a useful and fast tool for goodware and malware analysis. In: Proceedings of the 9th European Conference on Information Warfare and Security: University of Macedonia and Strategy International Thessaloniki, Greece, July 1-2, p. 405. Academic Conferences Limited (2010)

4. Cesare, S., Xiang, Y., Zhou, W.: Malwise–an effective and efficient classification system for packed and polymorphic malware. IEEE Transcation on Computers 62, 1193–1206 (2013)
5. Cohen, C., Havrilla, J.S.: Function hashing for malicious code analysis. In: CERT Research Annual Report 2009, pp. 26–29. Software Engineering Institute, Carnegie Mellon University (2010)
6. Debray, S., Patel, J.: Reverse engineering self-modifying code: Unpacker extraction. In: 2010 17th Working Conference on Reverse Engineering (WCRE), pp. 131–140 (2010)
7. Dullien, T., Carrera, E., Eppler, S.-M., Porst, S.: Automated attacker correlation for malicious code. Technical report, DTIC Document (2010)
8. Dullien, T., Rolles, R.: Graph-based comparison of executable objects (english version). SSTIC 5, 1–3 (2005)
9. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Computing Surveys (CSUR) 44(2), 6 (2012)
10. Gao, D., Reiter, M.K., Song, D.: Binhunt: Automatically finding semantic differences in binary programs. In: Chen, L., Ryan, M.D., Wang, G. (eds.) ICICS 2008. LNCS, vol. 5308, pp. 238–255. Springer, Heidelberg (2008)
11. Hemel, A., Kalleberg, K.T., Vermaas, R., Dolstra, E.: Finding software license violations through binary code clone detection. In: Proceedings of the 8th Working Conference on Mining Software Repositories, pp. 63–72. ACM (2011)
12. Hubert, L., Arabie, P.: Comparing partitions. Journal of Classification 2(1), 193–218 (1985)
13. Idika, N., Mathur, A.P.: A survey of malware detection techniques. Technical report, Department of Computer Science, Purdue University (2007)
14. Jang, J., Brumley, D., Venkataraman, S.: BitShred: feature hashing malware for scalable triage and semantic analysis. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, pp. 309–320. ACM, New York (2011)
15. Jang, J., Woo, M., Brumley, D.: Towards automatic software lineage inference. In: Proceedings of the 22nd USENIX Conference on Security, pp. 81–96. USENIX Association (2013)
16. Kaspersky Lab. Resource 207: Kaspersky Lab Research proves that Stuxnet and Flame developers are connected (2012) (last accessed: September 13, 2012)
17. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 207–226. Springer, Heidelberg (2006)
18. Lakhotia, A., Dalla Preda, M., Giacobazzi, R.: Fast location of similar code fragments using semantic 'juice'. In: SIGPLAN Program Protection and Reverse Engineering Workshop, p. 5. ACM (2013)
19. Lakhotia, A., Walenstein, A., Miles, C., Singh, A.: Vilo: a rapid learning nearest-neighbor classifier for malware triage. Journal of Computer Virology and Hacking Techniques, 1–15 (2013)
20. Linger, R., Daly, T., Pleszkoch, M.: Function extraction (FX) research for computation of software behavior: 2010 development and application of semantic reduction theorems for behavior analysis. Technical Report CMU/SEI-2011-TR-009, Carnegie Mellon University, Software Engineering Institute (February 2011)
21. Moran, N., Bennett, J.T.: Supply chain analysis: From quartermaster to sunshop. Technical report, FireEye Labs (November 2013)

22. Moskovitch, R., Feher, C., Tzachar, N., Berger, E., Gitelman, M., Dolev, S., Elovici, Y.: Unknown malcode detection using OPCODE representation. In: Ortiz-Arroyo, D., Larsen, H.L., Zeng, D.D., Hicks, D., Wagner, G. (eds.) EuroIsI 2008. LNCS, vol. 5376, pp. 204–215. Springer, Heidelberg (2008)
23. Newman, M.E.: Modularity and community structure in networks. Proceedings of the National Academy of Sciences 103(23), 8577–8582 (2006)
24. O'Gorman, G., McDonald, G.: The Elderwood Project (August 2012)
25. Perdisci, R., Lanzi, A., Lee, W.: Classification of packed executables for accurate computer virus detection. Pattern Recognition Letters 29(14), 1941–1946 (2008)
26. Pfeffer, A., Call, C., Chamberlain, J., Kellogg, L., Ouellette, J., Patten, T., Zacharias, G., Lakhotia, A., Golconda, S., Bay, J., et al.: Malware analysis and attribution using genetic information. In: 2012 7th International Conference on Malicious and Unwanted Software (MALWARE), pp. 39–45. IEEE (2012)
27. Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets. Cambridge University Press (2012)
28. Rolles, R.: Unpacking virtualization obfuscators. In: Proceedings of the 3rd USENIX Conference on Offensive Technologies, p. 1. USENIX Association (2009)
29. Runwal, N., Low, R.M., Stamp, M.: Opcode graph similarity and metamorphic detection. Journal in Computer Virology 8(1-2), 37–52 (2012)
30. Sæbjørnsen, A., Willcock, J., Panas, T., Quinlan, D., Su, Z.: Detecting code clones in binary executables. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, pp. 117–128. ACM (2009)
31. Schultz, M.G., Eskin, E., Zadok, F., Stolfo, S.J.: Data mining methods for detection of new malicious executables. In: Proceedings. 2001 IEEE Symposium on Security and Privacy, SP 2001, pp. 38–49 (2001)
32. Shabtai, A., Menahem, E., Elovici, Y.: F-sign: Automatic, function-based signature generation for malware. IEEE Transactions on Systems, Man, and Cybernetics, Part C 41(4), 494–508 (2011)
33. Tahan, G., Rokach, L., Shahar, Y.: Mal-id: Automatic malware detection using common segment analysis and meta-features. The Journal of Machine Learning Research 98888, 949–979 (2012)
34. Theodoridis, S., Koutroumbas, K.: Pattern Recognition. Elsevier Science (2008)
35. Walenstein, A., Lakhotia, A.: A transformation-based model of malware derivation. In: Malicious and Unwanted Software (MALWARE), pp. 17–25. IEEE (2012)
36. Yavvari, C., Tokhtabayev, A., Rangwala, H., Stavrou, A.: Malware characterization using behavioral components. In: Kotenko, I., Skormin, V. (eds.) MMM-ACNS 2012. LNCS, vol. 7531, pp. 226–239. Springer, Heidelberg (2012)
37. Zhou, W., Zhou, Y., Grace, M., Jiang, X., Zou, S.: Fast, scalable detection of piggybacked mobile applications. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy, pp. 185–196. ACM (2013)