Automatic Analysis of Malware Behavior using Machine Learning

Konrad Rieck[1], Philipp Trinius[2], Carsten Willems[2], and Thorsten Holz[2,3]

[1] Berlin Institute of Technology, Germany
[2] University of Mannheim, Germany
[3] Vienna University of Technology, Austria

Abstract

Malicious software—so called *malware*—poses a major threat to the security of computer systems. The amount and diversity of its variants render classic security defenses ineffective, such that millions of hosts in the Internet are infected with malware in the form of computer viruses, Internet worms and Trojan horses. While obfuscation and polymorphism employed by malware largely impede detection at file level, the dynamic analysis of malware binaries during run-time provides an instrument for characterizing and defending against the threat of malicious software.

In this article, we propose a framework for the automatic analysis of malware behavior using machine learning. The framework allows for automatically identifying novel classes of malware with similar behavior (*clustering*) and assigning unknown malware to these discovered classes (*classification*). Based on both, clustering and classification, we propose an incremental approach for behavior-based analysis, capable of processing the behavior of thousands of malware binaries on a daily basis. The incremental analysis significantly reduces the run-time overhead of current analysis methods, while providing accurate discovery and discrimination of novel malware variants.

1   Introduction

Malicious software, referred to as *malware*, is one of the major threats on the Internet today. A plethora of malicious tools, ranging from classic computer viruses to Internet worms and bot networks, targets computer systems linked to the Internet. Proliferation of this threat is driven by a criminal industry which systematically comprises networked hosts for illegal purposes, such as distribution of spam messages or gathering of confidential data (see Franklin et al., 2007; Holz et al., 2009). Unfortunately, the increasing amount and diversity of malware render classic security techniques, such as anti-virus scanners, ineffective and, as a consequence, millions of hosts in the Internet are currently infected with malicious software (Microsoft, 2009; Symantec, 2009).

To protect from the rapid propagation of malware in the Internet, developers of anti-malware software heavily rely on the automatic analysis of novel variants for designing corresponding defense measures. The automatic analysis of malware, however, is far from a trivial task, as malware writers frequently employ obfuscation techniques, such as binary packers, encryption, or self-modifying code, to obstruct analysis. These techniques are especially effective against static analysis of malicious binaries (e.g., Linn and Debray, 2003; Christodorescu and Jha, 2003; Kruegel et al., 2005; Moser et al., 2007a; Preda et al., 2008). In contrast to static techniques, dynamic analysis of binaries during run-time enables monitoring the behavior of malware, which is difficult to conceal and often indicative for malicious activity. Hence, a substantial amount of research has focused on development of tools for collection and monitoring of malware (e.g., Pouget et al., 2005; Leita et al., 2006; Bächer et al., 2006; Bayer et al., 2006a,b; Willems et al., 2007; Lanzi et al., 2009).

While monitoring binaries during run-time provides means for studying the behavior of malicious software, it is by itself not sufficient to alleviate the threat of malware proliferation. What is needed is the ability to *automatically analyze* the behavior of malware binaries, such that novel strains of development can be efficiently identified and mitigated. Two concepts for such automatic analysis of behavior based on machine learning techniques have been recently proposed: (a) *clustering of behavior*, which aims at discovering novel classes of malware with similar behavior (Bailey et al., 2007; Bayer et al., 2009a) and (b) *classification of behavior*, which enables assigning unknown malware to known classes of behavior (Lee and Mody, 2006; Rieck et al., 2008). Previous work has studied these concepts as competing paradigms, where either one of the two has been applied using different algorithms and representations of behavior.

In this article, we argue that discovery of novel malware classes (clustering) and discrimination between known classes (classification) complement each other and are both indispensable for efficient and effective malware analysis. We thus propose a framework for the automatic analysis of malware behavior jointly using clustering and classification. In particular, our paper makes the following major contributions:

– *Scalable clustering and classification.* We propose a mapping of monitored behavior to a vector space, such that behavioral patterns are efficiently accessible to means of machine learning. To attain scalable computation with thousands of vectors, we introduce an approximation using prototype vectors which is applicable to clustering as well as classification techniques.

– *Incremental analysis of malware behavior*. By combining clustering and classification, we devise an incremental approach to behavior-based analysis capable of processing the behavior of thousands of malware binaries on a daily basis. This incremental analysis significantly reduces the run-time and memory overhead of batch analysis methods, while providing accurate discovery of novel malware.

– *Extensive evaluation with real malware.* In a comparative evaluation, we demonstrate the efficacy of our framework which outperforms state-of-the-art analysis methods. Empirically, the incremental analysis reduces memory requirements by 94% and yields a speed-up factor of 4—ultimately enabling processing 33,000 reports of malware behavior in less than 25 minutes.

Although the proposed analysis framework does not generally eliminate the threat of malicious software, it provides a valuable instrument for timely development of anti-malware products, capable of automatically and efficiently characterizing novel breeds of malware development on a regular basis.

The rest of this article is organized as follows: Our analysis framework for malware behavior is introduced in Section 2 including feature extraction, machine learning techniques, and incremental analysis of behavior. An empirical evaluation of the framework using data from a vendor of anti-malware products is presented Section 3. We discuss related work in Section 4 and conclude this article in Section 5.

2 Automatic Analysis of Malware Behavior

Malicious software is characterized by complex and diverse behavior, ranging from simple modifications of system resources to advanced network activity. Malware variants of the same family, however, share common behavioral patterns, such as the usage of specific mutexes or modifications of particular system files. We aim to exploit these shared patterns for automatic analysis and propose a framework for clustering and classifying malware based on their behavior. A schematic overview of our analysis framework is depicted in Figure 1. Its basic analysis steps are summarized in the following.

1. Our framework proceeds by first executing and monitoring malware binaries in a sandbox environment. Based on the performed operations and actions—in terms of system calls—a sequential report of the monitored behavior is generated for each binary, where system calls and their arguments are stored in a representation specifically tailored to behavior-based analysis (Section 2.1).

2. The sequential reports are then embedded in a high-dimensional vector space, where each dimension is associated with a behavioral pattern, a short sequence of observed instructions. In this vectorial representation, the similarity of behavior can be assessed geometrically, which allows for designing intuitive yet powerful clustering and classification methods (Section 2.2).

3. Machine learning techniques for clustering and classification are then applied to the embedded reports for identifying novel and known classes of malware. Efficient computation of both analysis techniques is realized using prototype vectors which subsume larger groups of reports with similar behavior and thereby provide an effective approximation to exact analysis (Section 2.3).

4. By alternating between clustering and classification steps, the embedded behavior of malware can be analyzed incrementally, for example, on a daily basis. First, behavior matching known malware classes is identified using prototype vectors of previously discovered clusters. Then reports with unidentified behavior are clustered for discovery of novel malware classes (Section 2.4).

In the following sections we discuss these individual steps and the corresponding technical background in more detail—providing examples of monitored behavior, describing its vectorial representation, and explaining the applied clustering and classification methods as well as their incremental application.

3
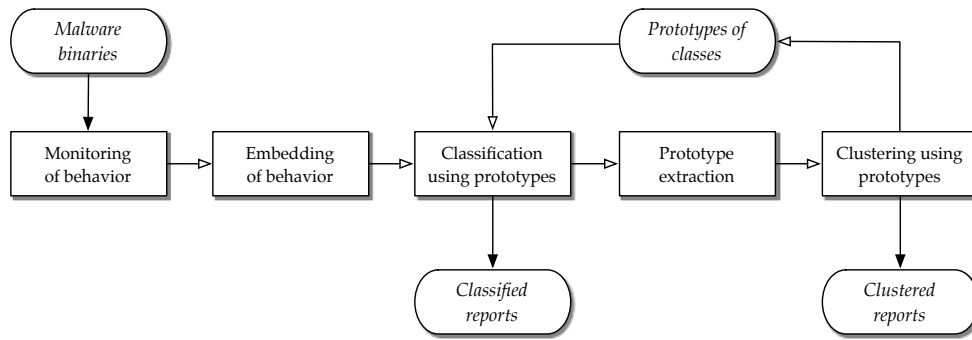
Figure 1: Schematic overview of analysis framework. Incoming reports of malware behavior are classified to known classes or grouped into novel classes of behavior. Prototypes extracted for clustering are fed back to classification for incremental analysis.

2.1 Monitoring of Malware Behavior

A prerequisite for behavior-based analysis is the efficient monitoring of malware behavior as well as a representation of this monitored behavior suitable for accurate analysis. In this section, we present the sandbox technique employed in our framework and describe the underlying representation of behavior denoted as *malware instruction set*.

2.1.1 Malware Sandboxes

For monitoring the behavior of executable binaries, multiple different methods exist from which the majority is based on the interception of system calls (e.g., Bayer et al., 2006a,b; Willems et al., 2007; Dinaburg et al., 2008). In contrast to code analysis, where the binary to be analyzed is disassembled or debugged, the actual code of the file is completely ignored under behavior-based analysis. Instead, the binary is seen as a black box and executed in a controlled environment. This environment is set up in a way, in which all system interaction of the malware is intercepted. By *detouring* system calls, the sandbox can inspect—and optionally modify—all input parameters and return values of system calls during run-time of the malware binary (Hunt and Brubacker, 1999).

This interception can be realized on different levels, ranging from a bird's eye view in out-of-system hypervisor monitoring down to in-process monitoring realized via dynamic code instrumentation or static patching. For our analysis, we employ the monitoring tool *CWSandbox* which intercepts system calls via inline function hooking. The tool overwrites the prologue of each system call with an unconditional jump to a *hook function*. This function first writes the system call and its arguments to a log file and then proceeds to execute the intercepted operation with all this detouring transparent to the caller. A detailed discussion of this technique is provided by Willems et al. (2007).

2.1.2 The Malware Instruction Set

The predominant format for representation of monitored behavior are textual and XML-based reports, for example, as generated by the malware sandboxes Anubis (Bayer et al., 2006b) and CWSandbox (Willems et al., 2007). While such formats are suitable for a human

analyst or computation of general statistics, they are inappropriate for automatic analysis of malware behavior. The structured and often aggregated textual reports hinder application of machine learning methods, as the true sequences of observed behavioral patterns are not directly accessible. Moreover, the complexity of textual representations increases the size of reports and thus negatively impacts run-time of analysis algorithms.

To address this problem and optimize processing of reports, we propose a special representation of behavior denoted as *malware instruction set* (*MIST*) inspired from instruction sets used in processor design. In contrast to regular formats, the monitored behavior of a malware binary is described as a sequence of instructions, where individual execution flows of threads and processes are sequentially appended to a single report. Each instruction in this format encodes one monitored system call and its arguments using short numeric identifiers, such as '03 05' for the system call 'move_file'. The system call arguments are arranged in blocks at different levels, reflecting behavior with different degree of specificity. We denote these levels as *MIST levels*. Moreover, variable-length arguments, such as file and mutex names, are represented by index numbers, where a global mapping table is used to translate between the original contents and the index numbers.
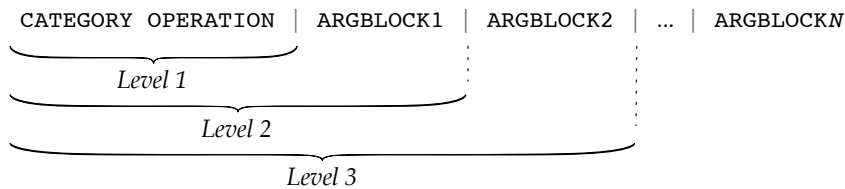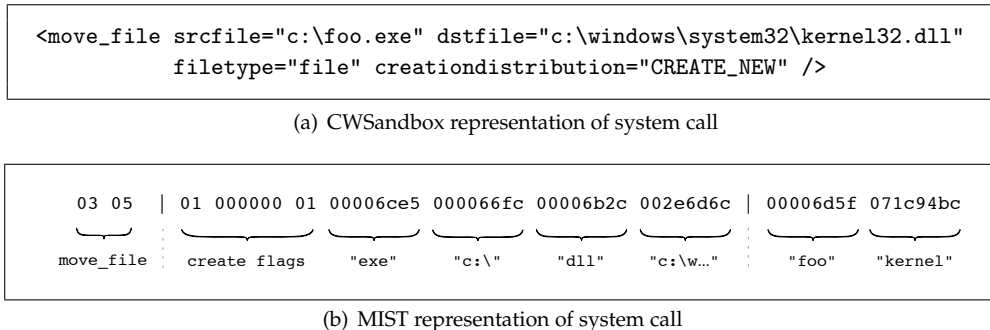

Figure 2: Schematic overview of a MIST instruction. The field `CATEGORY` encodes the category of system calls where the field `OPERATION` reflects a particular system call. Arguments are represent as blocks by `ARGBLOCKN`.

Figure 2 shows the basic structure of a *MIST instruction*. The first level of the instructions corresponds to the category and name of a monitored system call. As an example, '03 05' corresponds to the category 'filesystem' (03) and the system call 'move_file' (05). The following levels of the instruction contain different blocks of arguments, where the specificity of the blocks increases from left to right. The main idea underlying this rearrangement is to move "noisy" elements, such as process and thread identifiers, to the end of an instruction, whereas stable and discriminative patterns, such as directory and mutex names, are kept at the beginning. Thus, the granularity of behavior-based analysis can be adapted by considering instructions only up to a certain level. As a result, malware sharing similar behavior may be even discovered if minor parts of the instructions differ, for instance, if randomized file names are used. A detailed description of MIST and the arrangement of argument blocks is provided by Trinius et al. (2010).

As an example, Figure 3 compares the original XML representation of CWSandbox and the novel MIST format. The operation move_file is presented with respective arguments. Although the formats strongly differ in visual appearance, they carry the same information. The order of arguments in the MIST instruction, however, is rearranged, where the path names and file extensions are covered in level 2 and the base names of the files in

level 3. Moreover, the MIST instruction is significantly shorter than the XML representation due to the application of numeric identifiers and index numbers.

```
<move_file srcfile="c:\foo.exe" dstfile="c:\windows\system32\kernel32.dll"
           filetype="file" creationdistribution="CREATE_NEW" />
```

(a) CWSandbox representation of system call

```
   03 05   |  01 000000 01  00006ce5 000066fc 00006b2c 002e6d6c  |  00006d5f 071c94bc
   ‿‿‿        ‿‿‿‿‿‿‿‿ ‿‿‿‿   ‿‿‿‿‿   ‿‿‿‿‿   ‿‿‿‿‿   ‿‿‿‿‿      ‿‿‿‿‿   ‿‿‿‿‿
 move_file     create flags     "exe"    "c:\"    "dll"    "c:\w…"        "foo"    "kernel"
```

(b) MIST representation of system call

Figure 3: Feature representations of a system call (Windows API call). The CWSandbox format represents the system call as an attributed XML element, while the malware instruction set (MIST) represents it as a structured instruction.

2.2   Embedding of Malware Behavior

The proposed feature representation enables an expressive characterization of behavior, where each execution of a binary is represented as a sequential report of MIST instructions. Typical behavioral patterns of malware, such as changing registry keys or modifying system files, are reflected in particular subsequences in these reports. Yet, this representation is still not suitable for application of efficient analysis techniques, as these usually operate on vectors of real numbers. To tackle this issue, we introduce a technique for embedding behavior reports in a vector space which is inspired by concepts from natural language processing and host-based intrusion detection (see Salton et al., 1975; Damashek, 1995; Forrest et al., 1996; Lee et al., 1997).

2.2.1   Embedding using Instruction Q-grams

In an abstract view, a *report x* of malware behavior corresponds to a simple sequence of instructions. To characterize the contents of this sequence, we move a fixed-length window over the report, where we consider a subsequence of length q at each position. The resulting "snippets" of instructions, referred to as *instruction q-grams*, reflect short behavioral patterns and thus implicitly capture some of the underlying program semantic. For constructing an embedding of reports using instruction q-grams, we consider the set $\mathcal{S}$ of all possible q-grams, defined as follows

$$\mathcal{S} = \{(a_1, \ldots, a_q) \mid a_i \in \mathcal{A} \text{ with } 1 \leq i \leq q\}, \tag{1}$$

where $\mathcal{A}$ denotes the set of all possible instructions. Note that depending on the considered MIST level, the granularity of $\mathcal{A}$ and $\mathcal{S}$ may range from plain system calls (level $= 1$) to full instructions covering different blocks of system call arguments (level > 1).

Using the set $\mathcal{S}$, a report x of malware behavior can be embedded in an $|\mathcal{S}|$-dimensional vector space, where each dimension is associated with one instruction q-gram and thus a short behavioral pattern. The corresponding *embedding function* φ resembles an indicator for the presence of instruction q-grams and can be formally defined as follows

$$\varphi(x) = (\varphi_s(x))_{s \in \mathcal{S}} \text{ with } \varphi_s(x) = \begin{cases} 1 & \text{if report } x \text{ contains } q\text{-grams } s, \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

As an example, let us consider the artificial report $x =$ '`1|A 2|A 1|A 2|A`' containing only two simplified instructions $\mathcal{A} = \{$`1|A, 2|A`$\}$. If we consider instruction q-grams with $q = 2$ for characterizing the contents of x, the vector $\varphi(x)$ looks as follows

$$\varphi(\text{`1|A 2|A 1|A 2|A'}) \longmapsto \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \begin{matrix} \text{`1|A 1|A'} \\ \text{`1|A 2|A'} \\ \text{`2|A 1|A'} \\ \text{`2|A 2|A'.} \end{matrix} \quad (3)$$

In contrast to this simple example, the vector space induced by real instructions exhibits a huge dimension. For example, for 2-grams with MIST level 2, a set of 1,000 reports easily exceeds over $100,000$ unique q-grams and hence is embedded in a vector space with over $100,000$ dimensions. At the first glance, computing and comparing vectors in such high-dimensional spaces seems infeasible. The number of instruction q-grams contained in a single report, however, is linear in its length. That is, a report x containing m instructions comprises at most $(m - q)$ different q-grams. Consequently, only $(m - q)$ dimensions are non-zero in the feature vector $\varphi(x)$—irrespective of the actual dimension of the vector space. This sparsity of $\varphi(x)$ can be exploited to derive linear-time methods for extraction and comparison of embedded reports, which ultimately enables efficient analysis of behavior as demonstrated in Section 3. A detailed discussion of linear-time methods for analysis of embedded sequences is provided by Rieck and Laskov (2008).

The number of non-zero dimensions in the vector space also depends on other factors, such as the redundancy of behavior, the considered alphabet, or the length of reports. In practice, the length of reports dominates these factors and introduces an implicit bias, rendering comparison of small and large reports problematic. To compensate this bias, we introduce a *normalized embedding function*

$$\hat{\varphi}(x) = \frac{\varphi(x)}{||\varphi(x)||} \quad (4)$$

that scales each vector $\varphi(x)$ such that its vector norm equals one. As a result of this normalization, a q-gram counts more in a report that has fewer distinct q-grams. That is, changing a constant amount of instructions in a report containing repetitive behavior has more impact on the embedded vector than in a report comprising several different behavioral patterns. This type of normalization is widely used in the domain of information retrieval for comparing text documents, where it is usually applied as part of the cosine similarity measure (see van Rijsbergen, 1979).

2.2.2 Comparing Embedded Reports

The embedding of reports in vector spaces enables expressing the similarity of behavior *geometrically*, which allows for designing intuitive yet powerful analysis techniques. To assess the geometric relations between embedded reports, we define a distance d by

$$d(x,z) = ||\hat{\varphi}(x) - \hat{\varphi}(z)|| = \sqrt{\sum_{s \in \mathcal{S}} (\hat{\varphi}_s(x) - \hat{\varphi}_s(z))^2} \qquad (5)$$

which compares the behavior of the embedded reports x and z, and corresponds to the Euclidean distance in $\mathbb{R}^{|\mathcal{S}|}$. The values of d range from $d(x,z) = 0$ for identical behavior to $d(x,z) = \sqrt{2}$ for maximally deviating reports due to the normalization.

Access to the geometry of the induced vector space enables grouping and discriminating embedded reports effectively by means of machine learning. Malware variants originating from the same class share several instruction q-grams in their behavior and thus lie close to each other, whereas reports from different families yield large distances and are scattered in the vector space. In comparison to related approaches using distances (e.g., Lee and Mody, 2006; Bailey et al., 2007), the proposed embedding gives rise to an *explicit vector representation*, where the contribution of each q-gram can be traced back to individual behavioral patterns for explaining the decisions made by analysis methods.
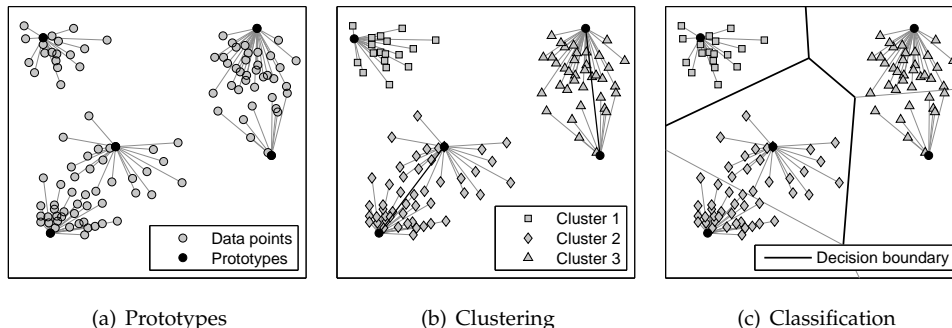
2.3 Clustering and Classification

Based on the embedding of reports in a vector space, we apply techniques of machine learning for the analysis of behavior. In particular, we study two learning concepts for analysis: *Clustering of behavior*, which enables identifying novel classes of malware with similar behavior and *classification of behavior*, which allows to assign malware to known classes of behavior. To keep abreast of the increasing amount of malware in the wild, clustering and classification methods are required to process thousands of reports on a daily basis. Unfortunately, most learning methods scale super-linear in the number of input data and thus are not directly applicable for malware analysis.

To address this problem, we propose an approximation for clustering and classification techniques inspired by the work of Bayer et al. (2009a). A set of malware binaries often contains similar variants of the same family which exhibit almost identical behavioral patterns. As a consequence, the embedded reports form dense clouds in the vector space. We exploit this dense representation by subsuming groups of similar behavior using *prototypes*—reports being typical for a group of homogeneous behavior. By restricting the computation of learning methods to prototypes and later propagating results to all embedded data, we are able to accelerate clustering as well as classification techniques. The extracted prototypes correspond to regular reports and thus can be easily inspected by a human analyst, whereas the approximation of locality sensitive hashing employed by Bayer et al. (2009a) is opaque, providing almost no insights into groups of behavior.

Figure 4(a) illustrates the concept of prototypes for representing groups of similar vectors on an artificial data set. The figure shows roughly 100 vectors arranged in three clusters which are effectively represented using six prototypes (indicated by black dots). Note that the prototypes do not necessary coincident with clusters. For example, the clusters in the lower and right part of Figure 4(a) are each represented by two prototype vectors. As

demonstrated in Section 3, prototypes allow for run-time improvements over exact methods while inducing a minimal approximation error.

(a) Prototypes   (b) Clustering   (c) Classification

Figure 4: Behavior analysis using prototypes: (a) prototypes of data, (b) clustering using prototypes, and (c) classification using prototypes. Black lines in Figure 4(b) indicate prototypes joined by linkage clustering. Black lines in Figure 4(c) represent the class decision boundary.

2.3.1 Prototype Extraction

Extracting a small yet representative set of prototypes from a data set is not a trivial task. Most approaches for prototype extraction rest on clustering (Bezdek and Kuncheva, 2001) or super-linear computations (Harmeling et al., 2006), and thus are inappropriate as basis for efficient approximation. Even worse, the task of finding an optimal set of prototypes can be shown to be NP-hard (Garey and Johnson, 1979). Fortunately, we can adapt a linear-time algorithm by Gonzàlez (1985) which provably determines a set of prototypes only twice as large as the optimal solution. The algorithm is sketched in Algorithm 1.

Algorithm 1 Prototype extraction

1: $prototypes \leftarrow \emptyset$
2: $distance[x] \leftarrow \infty$ for all $x \in reports$
3: **while** $\max(distance) > d_p$ **do**
4:     choose z such that $distance[z] = \max(distance)$
5:     **for** $x \in reports$ and $x \neq z$ **do**
6:         **if** $distance[x] > ||\hat{\varphi}(x) - \hat{\varphi}(z)||$ **then**
7:             $distance[x] \leftarrow ||\hat{\varphi}(x) - \hat{\varphi}(z)||$
8:     add z to $prototypes$

The algorithms proceeds by iteratively selecting prototypes from a set of reports. The first prototype is either fixed or chosen at random. During each run, the distance from the current set of prototypes to the remaining embedded reports is computed (line 5–7). The farthest report is chosen as new prototype, such that the data set is iteratively covered by a web of prototypes (line 4). This procedure is repeated until the distance from each vector to its nearest prototype is below the parameter d_p (line 3).

9

The run-time complexity of this algorithm is $O(kn)$ where n is the number of reports and k the number of prototypes. Given that d_p is reasonably chosen, the algorithm is linear in the number of reports, as k solely depends on the distribution of data. If no such d_p can be determined though, the overall iterations can still be restricted by a fixed limit k.

2.3.2 Clustering using Prototypes

Clustering refers to a basic technique of machine learning which aims at partitioning a given data set into meaningful groups, so called *clusters*. The partitioning is determined, such that objects within one cluster are similar to each other, whereas objects in different clusters are dissimilar. Clustering enables discovery of structure in unknown data and thus has been employed in a large variety of applications (see Anderberg, 1973).

Clustering for analysis of malware behavior has been proposed by Bailey et al. (2007) and later refined by Bayer et al. (2009a). We pursue this line of research and study the standard technique of *hierarchical clustering* (Duda et al., 2001) for determining groups of malware with similar behavior. In contrast to previous work, we base our analysis on the concept of prototypes. That is, first clusters of prototypes are determined and then propagated to the original data. An example for such clustering is illustrated in Figure 4(b), where black lines indicate prototypes grouped into clusters. A corresponding clustering algorithm is presented in Algorithm 2.

Algorithm 2 Clustering using prototypes

1: **for** $z, z' \in prototypes$ **do**
2:     $distance[z, z'] \leftarrow ||\hat{\varphi}(z) - \hat{\varphi}(z')||$
3: **while** $\min(distance) < d_c$ **do**
4:     merge clusters z, z' with minimum $distance[z, z']$
5:     update $distance$ using complete linkage
6: **for** $x \in reports$ **do**
7:     $z \leftarrow$ nearest prototype to x
8:     assign x to cluster of z
9: reject clusters with less than m members

Starting with each prototype being an individual cluster, the algorithm proceeds by iteratively determining and merging the nearest pair of clusters (line 4). This procedure is terminated if the distance between the closest clusters is larger than the parameter d_c. To compute distances between clusters, the algorithm considers the maximum distance of their individual members—a standard technique of hierarchical clustering referred to as *complete linkage* (see Anderberg, 1973; Duda et al., 2001). Once a clustering has been determined on the prototypes, it is propagated to the original reports (line 6–8). Moreover, clusters with fewer than m members are rejected and kept for later incremental analysis, as discussed in Section 2.4.

The algorithm has a run-time complexity of $O(k^2 \log k + n)$, where n is the number of reports and k the number of prototypes. In comparison to exact hierarchical clustering with a run-time of $O(n^2 \log n)$, the approximation provides a speed-up factor of $\sqrt{n/k}$.

2.3.3 Classification using Prototypes

We next consider classification, which allows to learn a discrimination among different classes of objects. Classification methods require a learning phase prior to application, where a model for discrimination is inferred from a data set of *labeled* objects. This model can then be applied for predicting class labels on unseen data. As many real-world application fall into this concept of learning, a large body of research exists on designing and applying classification methods (e.g., Mitchell, 1997; Duda et al., 2001; Müller et al., 2001; Shawe-Taylor and Cristianini, 2004)

The application of classification for the analysis of malware behavior has been studied by Lee and Mody (2006) and Rieck et al. (2008). In both approaches, the behavior of unknown malware is classified to known classes of behavior, where the initial training data is labeled using anti-virus scanners. Unfortunately, most anti-virus products suffer from inconsistent and incomplete labels (see Bailey et al., 2007) and do not provide sufficiently accurate labels for training. As a remedy, we employ the malware classes discovered by clustering as labels for training and thereby learn a discrimination between known clusters of malware behavior. As these clusters are represented by prototypes in our framework, we again make use of approximation to accelerate learning. As an example, Figure 4(c) shows the decision boundary for classification determined using prototypes of clusters. A corresponding classification algorithm is presented in Algorithm 3.

Algorithm 3 Classification using prototypes

1: **for** $x \in$ *reports* **do**
2:      $z \leftarrow$ nearest prototype to x
3:      **if** $||\hat{\varphi}(z) - \hat{\varphi}(x)|| > d_r$ **then**
4:          reject x as unknown class
5:      **else**
6:          assign x to cluster of z

For each report x, the algorithm determines the nearest prototype of the clusters in the training data (line 1–2). If the nearest prototype is within the radius d_r, the report is assigned to the respective cluster, whereas otherwise it is rejected and hold back for later incremental analysis (line 4–6). This procedure is referred to as *nearest prototype classification* and resembles an efficient alternative to costly k-nearest neighbor methods (see Bezdek and Kuncheva, 2001; Duda et al., 2001)

The naive run-time of the algorithm is $O(kn)$, as for each of the n reports the nearest prototype needs to be determined from the k prototypes in the training data. By maintaining the prototypes in specialized tree structures, the run-time complexity can be reduced to $O(n \log k)$ in the worst-case (Omohundro, 1989; Beygelzimer et al., 2006). Nevertheless, for our analysis framework, we favor the naive implementation over specialized algorithms, as it can be effectively parallelized and hence provides better run-time performance on common multi-core systems.

2.4 Incremental Analysis

Based on a joint formulation of clustering and classification, we devise an incremental approach to analysis of malware behavior. While previous work has been restricted to batch analysis, we propose to process the incoming reports of malware behavior in small chunks, for example on a daily basis. To realize an incremental analysis, we need to keep track of intermediate results, such as clusters determined during previous runs of the algorithm. Fortunately, the concept of prototypes enables us to store discovered clusters in a concise representation and, moreover, provides a significant speed-up if used for classification. Hence, we construct our incremental analysis around prototype-based clustering and classification as illustrated in Algorithm 4.

Algorithm 4 Incremental Analysis

1: *rejected* ← ∅, *prototypes* ← ∅
2: **for** *reports* ← data source ∪ *rejected* **do**
3:    classify *reports* to known clusters using *prototypes*   ▷ see Algorithm 3
4:    extract prototypes from remaining *reports*   ▷ see Algorithm 1
5:    cluster remaining *reports* using prototypes   ▷ see Algorithm 2
6:    *prototypes* ← *prototypes* ∪ prototypes of new clusters
7:    *rejected* ← rejected reports from clustering

The reports to be analyzed are received from a data source such as a set of honeypots or a database of collected malware in regular intervals. In the first processing phase, the incoming reports are classified using prototypes of known clusters (line 3). Thereby, variants of known malware are efficiently identified and filtered from further analysis. In the following phase, prototypes are extracted from the remaining reports and subsequently used for clustering of behavior (line 4–5). The prototypes of the new clusters are stored along with the original set of prototypes, such that they can be applied in a next run for classification. This procedure—alternating between classification and clustering—is repeated incrementally, where the amount of unknown malware is continuously reduced and the prevalent classes of malware are automatically discovered.

The number of reports available during one incremental run, however, may be insufficient for determining all clusters of malware behavior. For example, infrequent malware variants may only be represented by few samples in the embedding space. To compensate for this lack of information, we reject clusters with fewer than m members and feed the corresponding reports back to the data source. Consequently, infrequent malware is agglomerated until sufficient data is available for clustering. As demonstrated in Section 3.4, this procedure ensures an accurate discovery of malware classes, even if not all relevant information is available during the first incremental runs.

The run-time of the incremental algorithm is $O(nm + k^2 \log k)$ for a chunk of n reports, where m is the number of prototypes stored from previous runs and k the number of prototypes extracted in the current run. Though the run-time complexity is quadratic in k, the number of extracted prototypes during each run remains constant for chunks of equal size and distribution. Thus, the complexity of incremental analysis is determined by m, the number of prototypes for known malware classes, similar to the linear complexity of signature matching in anti-malware products.

3 Experiments & Application

We proceed to an empirical evaluation of the proposed analysis framework for malware behavior. First, we evaluate and calibrate the individual components of the framework on a reference data set (Section 3.2). We then compare the prototype-based clustering and classification to state-of-the-art analysis methods (Section 3.3). Finally, we study the efficacy and run-time performance of our framework in a real application with malware obtained from a vendor of anti-malware products (Section 3.4).

3.1 Evaluation Data

For our experiments we consider two data sets of malware behavior: A *reference data set* containing known classes of malware which is used to evaluate and calibrate our framework and an *application data set* which comprises unknown malware obtained from the security center of an anti-malware vendor.

Reference data set. The reference data set is extracted from a large database of malware binaries maintained at the CWSandbox web site[1]. The malware binaries have been collected over a period of three years from a variety of sources, such as honeypots, spam traps, anti-malware vendors and security researchers. From the overall database, we select binaries which have been assigned to a known class of malware by the majority of six independent anti-virus products. Although anti-virus labels suffer from inconsistency, we expect the selection using different scanners to be reasonable consistent and accurate. To compensate for the skewed distribution of classes, we discard classes with less than 20 samples and restrict the maximum contribution of each class to 300 binaries. The selected malware binaries are then executed and monitored using CWSandbox, resulting in a total of 3,133 behavior reports in MIST format (Table 1).

	Malware class	#		Malware class	#
a	ADULTBROWSER	262	m	PORNDIALER	98
b	ALLAPLE*	300	n	RBOT	101
c	BANCOS	48	o	ROTATOR*	300
d	CASINO	140	p	SALITY	85
e	DORFDO	65	q	SPYGAMES	139
f	EJIK	168	r	SWIZZOR	78
g	FLYSTUDIO	33	s	VAPSUP	45
h	LDPINCH	43	t	VIKINGDLL	158
i	LOOPER	209	u	VIKINGDZ	68
j	MAGICCASINO	174	v	VIRUT	202
k	PODNUHA*	300	w	WOIKOINER	50
l	POSION	26	x	ZHELATIN	41

Table 1: Reference data set of 24 malware classes. The data set contains 3,133 reports of malware behavior. Frequent malware has been restricted to 300 samples (indicated by *).

---

[1] CWSandbox—Behavior-based Malware Analysis, http://www.mwanalysis.org

Application data set. While the reference data set covers known malware for evaluation and calibration, the application data set specifically contains unknown malware. The data set consists of seven chunks of malware binaries obtained from the anti-malware vendor *Sunbelt Software*. The binaries correspond to malware collected during seven consecutive days in August 2009 and originate from a variety of sources, such as deployed anti-malware scanners, exchange with other vendors or malware honeypots. Sunbelt Software uses these very samples to create and update signatures for their *VIPRE* anti-malware product as well as for their security data feed *ThreatTrack*. Similar to the reference data set, the malware binaries in each chunk are executed and monitored using CWSandbox, resulting in a total of 33,698 behavior reports in MIST format. Statistics for the data set and the characteristics of the contained behavior reports are provided in Table 2.

Data set description			
Collection period			August 1–7, 2009
Collection location			Sunbelt Software
Data set size (kilobytes)			21,808,644
Number of reports			33,698
Data set statistics	*min.*	*avg.*	*max.*
Reports per day	3,760	4,814	6,746
Instructions per report	15	11,921	103,039
Size per report (kilobytes)	1	647	5,783

Table 2: Application data set of 33,698 reports. The data set has been collected at the anti-malware vendor Sunbelt Software during August 1–7, 2009.

3.2 Evaluation of Components

As the first experiment, we evaluate and calibrate the components of our framework, namely the prototype extraction, the clustering, and the classification. To assess the performance of these components, we employ the evaluation metrics of *precision* and *recall*. The precision P reflects how well individual clusters agree with malware classes and the recall R measures to which extent classes are scattered across clusters. Formally, we define precision and recall for a set of clusters C and a set of malware classes Y as

$$P = \frac{1}{n} \sum_{c \in C} \#_c \quad \text{and} \quad R = \frac{1}{n} \sum_{y \in Y} \#_y \tag{6}$$

where $\#_c$ is the largest number of reports in cluster c sharing the same class and $\#_y$ the largest number of reports labeled y within one cluster. If each report is represented as a single cluster, we obtain $P = 1$ with low recall, whereas if all reports fall into the same cluster, we have $R = 1$ with low precision. Consequently, we seek an analysis setup which maximizes precision and recall. To this end, we consider an aggregated performance score for our evaluation, denoted as *F-measure*,

$$F = \frac{2 \cdot P \cdot R}{P + R} \tag{7}$$

14

which combines precision and recall (van Rijsbergen, 1979). A perfect discovery of classes yields $F = 1$, whereas either a low precision or recall result in a low F-measure.

As the first component of our framework, we evaluate the performance of prototype extraction. That is, we consider the precision of extracted prototypes on the reference data set in relation to the attained reduction of reports (*compression ratio*). The recall is irrelevant in this setting, as individual prototypes are not required to represent complete malware classes. To obtain different sets of prototypes we vary the distance parameter d_p in Algorithm 1. Moreover, we also evaluate different settings for embedding reports, where we choose the length q of instruction q-grams from the set $\{1, 2, 3, 4\}$.



| | (a) MIST level 1 | (b) MIST level 2 |
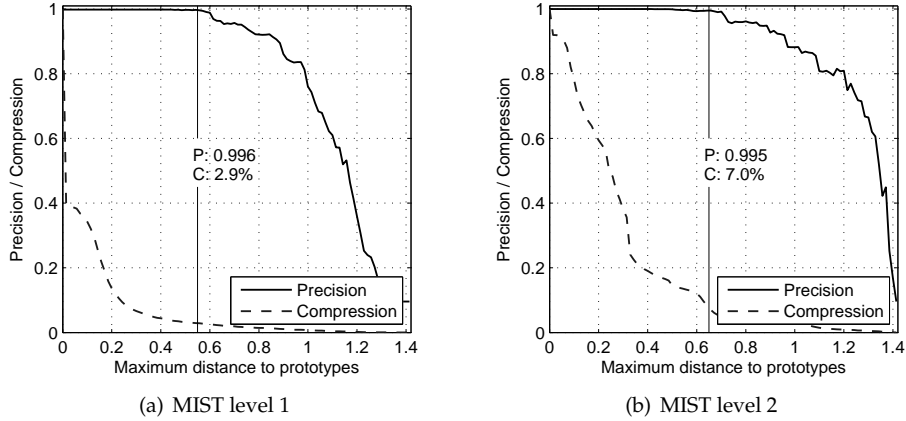
Figure 5: Prototype extraction on reference data. The precision and compression are shown for varying prototype threshold d_p (Maximum distance to prototypes).

Figure 5 shows results for the evaluation of prototype extraction for MIST level 1 and 2 averaged over 10 runs, where the initial prototype is selected randomly. The results are presented for a q-gram length of $q = 2$ which provided the best balance between precision and compression. For the optimal setup, the prototypes yield a precision of 0.99 while compressing the corpus of reports to 2.9% and 7.0%, respectively. The average standard deviation between the experimental runs is below 1.5% and the reported precision can be considered statistical significant. Hence, we calibrate the distance parameter d_p according to this setup and fix $q = 2$ in the following experiments.

We proceed to evaluate the performance of the proposed clustering using prototypes. For this experiment, we compute the F-measure for the prototype-based clustering and an implementation of regular hierarchical clustering on the reference data set. We vary the distance parameter d_c from 0 to $\sqrt{2}$ of Algorithm 2 to obtain clusterings with different granularity and number of clusters.

Results for the evaluation of clustering are presented in Figure 6 for MIST level 1 and 2. The performance is again averaged over 10 runs with random initialization of prototype extraction. The accuracy of both methods varies with the distance parameter d_c where prototype-based and regular clustering perform almost identical. The F-measure increases with the parameter d_c as long as similar reports are correctly grouped into clusters and finally reaches a plateau between 0.6 and 1.1. Both techniques attain an F-measure of
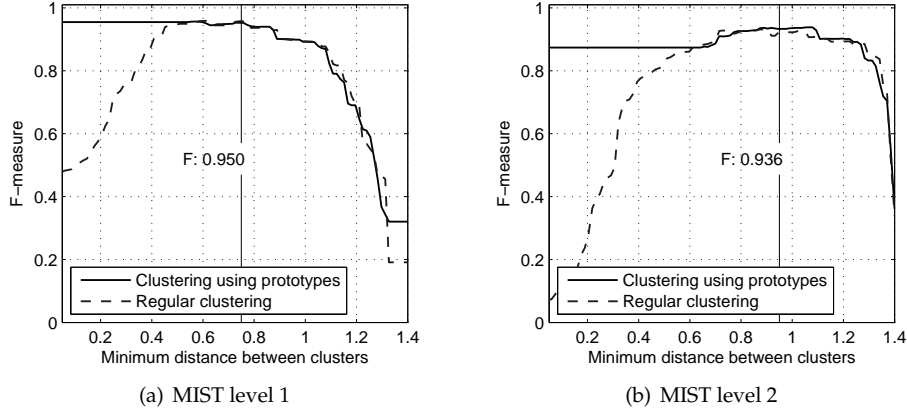
Figure 6: Clustering performance on reference data. The performance (*F*-measure) for clustering using prototypes and regular clustering is presented for varying distance d_c (Minimum distance between clusters).

0.93 and 0.95, respectively, corresponding to a near perfect discovery of the 24 malware classes in the reference data set. For the following experiments, we thus fix the parameter d_c accordingly. Note that, although both methods perform similarly, the prototype-based clustering requires less than a tenth of the original reports for discovery of clusters.

In the third evaluation, we assess the performance of classification. As classification is a supervised learning task involving a training phase, we randomly split the reference data into a training and a testing partition over 10 runs and average results. Moreover, we consider half of the malware classes as *unknown* in each run and do not provide instances of these classes for training. Consequently, we consider two different *F*-measures:

F_k : The measure F_k is used for evaluating the classification of known classes. It is computed as the regular *F*-measure in Equation (7) except that precision and recall are only determined on the set of known classes.

F_u : The measure F_u is used for evaluating the rejection of unknown classes. It is calculated as in Equation (7) on all data instances; however, only two different decisions are considered for determining precision and recall: rejection and no rejection.

Figure 7 shows results of this evaluation for MIST level 1 and 2. The *F*-measure for classification of known classes increases with the distance parameter d_r, while the *F*-measure for rejection of unknown classes decreases. The larger we choose the distance d_r, that is, the region around the prototypes, the better we can describe the known classes of malware. However, if the parameter d_r is too large, unknown malware is incorrectly assigned to prototypes of other malware classes. In the optimal setting an *F*-measure between 0.96 and 0.99 is obtained for both types of malware classes, resulting in a classification setup suitable for discrimination of known classes as well as rejection of unknown variants for subsequent clustering.

16

| (a) MIST with level 1 | (b) MIST with level 2 |
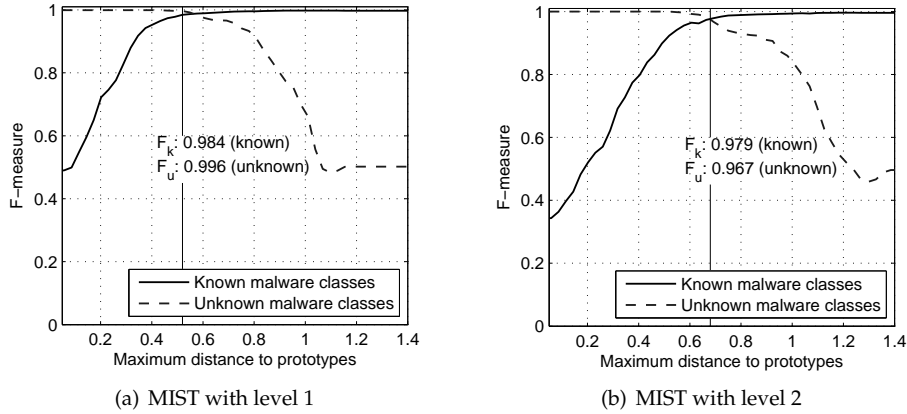
Figure 7: Classification performance on reference data. The performance (*F*-measure) for classification of known malware classes and rejection of unknown malware classes is presented for varying rejection threshold d_r (Maximum distance to prototypes).

3.3 Comparative Evaluation with State-of-the-Art

After evaluating and calibrating the individual components of our framework, we present a comparative evaluation with state-of-the-art methods for analysis of malware behavior. For clustering of behavior, we consider the method proposed by Bayer et al. (2009a) which uses behavioral features from the Anubis sandbox for computing an approximate clustering using locality sensitive hash (LSH). For classification, we apply the method proposed by Rieck et al. (2008) which learns a discrimination using support vector machines (SVM) with XML features of monitored behavior. We follow the calibration procedure detailed in the previous section for both analysis concepts and select optimal parameters for each method accordingly.

Clustering methods	*F*-measure	
Clustering using prototypes with MIST level 1	0.950	
Clustering using prototypes with MIST level 2	0.936	
Clustering using LSH with Anubis features (Bayer et al., 2009a)	0.881	
Classification methods	F_k	F_u
Classification using prototypes with MIST level 1	0.981	0.997
Classification using prototypes with MIST level 2	0.972	0.964
Classification using SVM with XML features (Rieck et al., 2008)	0.807	0.548

Table 3: Comparison of analysis methods on reference data set. The performance of analysis (*F*-measure) is presented for clustering and classification. For classification the *F*-measure is given individually for known malware classes (F_k) and unknown malware classes (F_u).

Result for the comparative evaluation are presented in Table 3 using the *F*-measure as performance criteria. In both analysis settings the components of our framework outperform the related methods. The prototype-based clustering attains an *F*-measure of 0.95 for MIST level 1 and 0.936 for level 2, while the method of Bayer et al. (2009a) reaches only

0.881. Similarly, our classification yields an *F*-measure of over 0.96, whereas the method of Rieck et al. (2008) achieves 0.807 for classification of known malware and less than 0.55 for classification of unknown malware. This superior analysis performance of our framework stems from the geometric formulation of clustering and classification, which directly exploits the behavioral patterns provided by the sequential MIST representation.

It is necessary to note that the clustering of Bayer et al. (2009a) has been performed separately by courtesy of Vienna University of Technology. In particular, the malware binaries have been executed six weeks after the generation of the respective MIST reports and thus the monitored behavior is not strictly identical. As a consequence, differences in performance may also result from the offset between monitoring dates. Still, the prototype-based clustering yields improved *F*-measures for five of the 24 malware classes which are unlikely to be all induced by the recording period.

3.4 An Application Scenario

Thus far our framework has been evaluated using the reference data set of known malware samples. In practice, however, it is the discovery of unknown malware classes that renders behavior-based analysis superior to related methods. Hence, we now turn to an application scenario involving unknown malware. We apply the incremental analysis method presented in Section 2.4 to the application data set, where we process one of the seven daily chunks of malware per iteration. This setting reflects a typical scenario at an anti-malware vendor: Incoming malware is analyzed for signature updates on a daily basis.

We fix the parameters of prototype extraction, clustering, and classification according to the previous evaluation on the reference data set. The rejection threshold m in Algorithm 2 is set to 10, that is, clusters with less than 10 members are initially rejected and kept back for following iterations. Moreover, we restrict our analysis to MIST level 2, as this granularity provides adequate accuracy while being resistant against simple evasion attacks. Additionally, we employ a regular hierarchical clustering on the application data set for comparison with the incremental analysis.

(a) Comparison with regular clustering   (b) Clustering-classification ratio
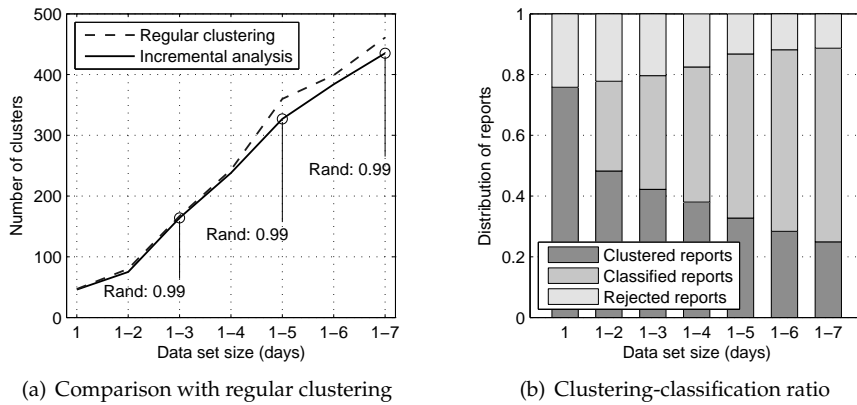
Figure 8: Evaluation of clustering on day 1–7 of application data set. Figure 8(a) compares regular clustering and incremental analysis in terms of cluster number and Rand index. Figure 8(b) shows the clustering-classification ratio for the incremental analysis.

18

Figure 8 shows results for behavior-based analysis of the application data set, where a comparison between incremental analysis and regular batch clustering is presented in Figure 8(a). The differences between both approaches is marginal. Even when processing all seven days of the application data set, both methods nearly determine the same number of clusters. This result is confirmed when looking at the *Rand index* additionally provided in Figure 8(a). The Rand index is a standard measure for comparison of clusterings, which is 1 for identical and 0 for maximally different clusterings (Rand, 1971). For all iterations of the incremental analysis, the Rand index is 0.99 between the clusterings and indicates an almost perfect match of both analysis methods.

During incremental analysis each behavior report is either *classified*, *clustered*, or *rejected*. Figure 8(b) shows the cumulative ratio of classified, clustered, and rejected reports for each of the seven days in the application data set. The number of rejected reports continuously decreases as more and more data is available for determining stable clusters. Furthermore, the number of classified reports significantly increases after the initial iteration, such that less data needs to be clustered and thus crucial run-time is saved.

These results demonstrate the advantage of combining clustering and classification in an incremental analysis: While the discovered clusters are almost identical to a regular batch approach, the processing of reports is shifted from costly clustering to more efficient classification. Before presenting the gained run-time improvements in more detail, we provide a discussion of the discovered malware classes from this experiment.

3.4.1 Evaluation of Malware Clusters

Results and statistics for the discovered clusters of malware are presented in Figure 9(a). A total of 434 clusters is determined from the 33,698 malware binaries of the application data set. The distribution of the cluster size decays exponentially, where the largest clusters comprise up to 2,500 reports and on average a cluster contains 69 reports. Overall, the reports are effectively represented using 1,805 prototypes, such that on average a cluster is associated with only 4 prototypes, resulting in a compression ratio of 5.4%.

Figure 9(b) shows the ten largest clusters discovered by incremental analysis. As a guideline for presentation of these clusters, we display the most frequent anti-virus label within each cluster generated using *Kaspersky Anti-Virus*. As the corresponding malware binaries have been unknown at the time of acquisition, we generate these labels with a delay of 8 weeks to allow for updates of anti-virus signatures.

For each of the ten clusters, one anti-virus label is prevalent, which indicates the discovery of consistent groups of behavior. For example, cluster 3 is mainly associated with the file infector VIRUT and comprises typical behavior of this malware class, such as IRC bot functionality on the channel '&virtu'. Cluster 8 entirely corresponds to the polymorphic worm ALLAPLE and is characterized by typical behavioral patterns, such as the creation of the file 'urdvxc.exe' and excessive ping scans. As a rather unexpected result, cluster 4 is labeled NF (nothing found) and indeed contains benign behavior. Nevertheless, all reports of this cluster correspond to self-extracting archives originating from tools like WinZip and WinRAR, which also demonstrates the ability of our analysis framework to discover classes of behavior—independent of malicious activity.

Apparently, there seems to be an inconsistency in the analysis, as the malware classes BASUN and SWIZZOR are spread among different clusters. This seeming lack of accurate

Analysis statistics	#
Analyzed reports	30,089 (89.3%)
→ by classification	22,472
→ by clustering	7,617
Rejected reports	3,609 (10.7%)
Number of clusters	434
Number of prototypes	1,805 (5.4%)
Cluster size	69 ± 217
Prototypes per cluster	4 ± 4

(a) Statistics of incremental analysis
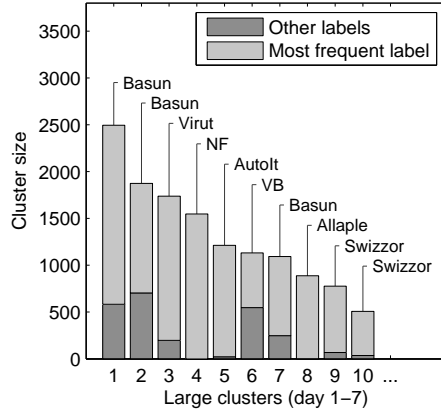
(b) Labeling of large clusters

Figure 9: Incremental analysis on day 1–7 of application data set. The labels in Figure 9(b) have been generated 8 weeks after acquisition of application data using *Kaspersky Anti-Virus*.

separation, however, manifests the main advantage of behavior-based analysis over regular anti-virus tools: malware is analyzed with respect to its *behavior* irrespective of similarities in file contents. For example, all binaries labeled BASUN share the same master host 'all-internal.info', but once contacting this host exhibit diverging behavior, which ranges from failed communication (cluster 2) to retrieving Windows updates (cluster 1), or connecting to malicious Web sites (cluster 7). Similarly, the binaries of SWIZZOR first spawn a process of the Internet Explorer, but then proceed differently. In case of cluster 9 various binary files are downloaded and executed, while for cluster 10, data is transferred to remote hosts via HTTP POST requests.

As a consequence, the clusters reflect different behavioral realizations of the malware classes which are difficult to discriminate from static context alone and may only be discovered by means of behavior-based analysis. The ability of modern malware to conduct different activity based on context, remote control and downloaded payloads renders the analysis of malware behavior vitally important for effectively updating signatures and crafting defense mechanisms.
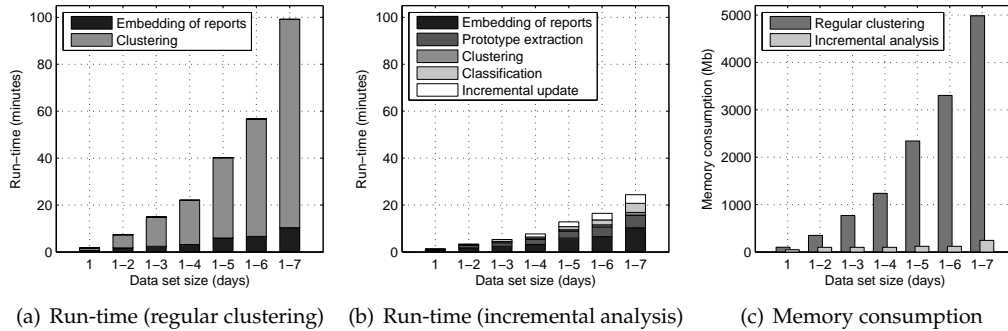
3.4.2 Run-time and Memory Requirements

As the last issue of the practical application of our framework, we consider its run-time performance and memory requirements. First, we evaluate individual run-times for embedding reports and computing distances in the vector space induced by q-grams. Then we assess the run-time and memory requirements for incremental analysis of the full application data set in comparison to regular batch analysis. All experiments are conducted on a quad-core AMD Opteron processor with 2.4 GHz. Although the implementation of our framework provides support for parallelization, we restrict all computations to one core of the processor. Moreover, we omit measurements for generation of MIST reports, as this format is naturally obtained during monitoring of malware binaries and thus induces an insignificant overhead.

(a) Embedding of reports   (b) Distance computation
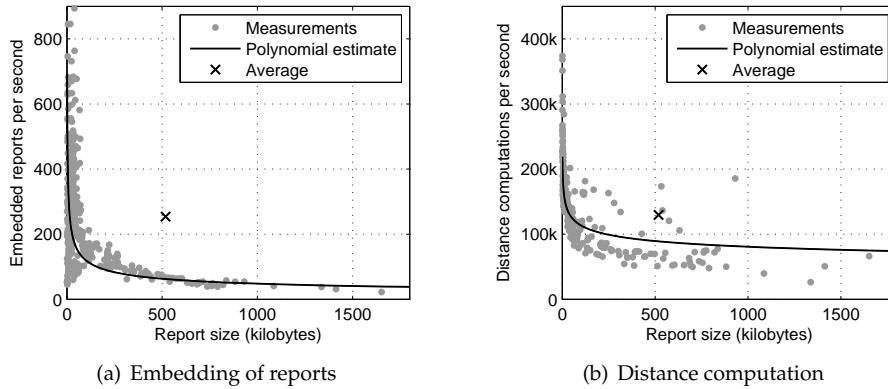
Figure 10: Run-time performance for embedding of reports and distance computation. The run-time is measured using 1,000 reports from the application data set with MIST level 2.

Figure 10(a) shows the run-time performance for embedding a sample of 1,000 reports. Depending on the report size, between 100 to 800 reports can be processed per second where on average a rate of 254 reports per second is achieved. This performance allows to process around 15,000 reports per minute and is clearly sufficient for large-scale analysis of malware behavior in practice. The run-time performance for distance computation between embedded reports is shown in Figure 10(b). On average 129,000 distances can be computed per second, which, as an example, allows to compare a report of malware behavior against 1,000,000 prototypes of clusters in less than 8 seconds. Moreover, a single distance computation takes around 0.0077 milliseconds and is significantly below the performance numbers reported by Bayer et al. (2009a).

Besides run-times of individual components, we also consider the total run-time and memory requirements for incremental analysis. Figure 10 shows the cumulative run-time for processing the individual days of the application data set, where results for regular clustering are presented in Figure 11(a) and for the incremental processing in Figure 11(b). The regular clustering exhibits a steep quadratic growths, resulting in a run-time of 100 minutes for processing the seven days. By contrast, the incremental analysis allows to compute clusters for this data within 25 minutes, thereby providing a speed-up factor of 4. The memory consumption of both approaches is compared in Figure 11(c). The regular clustering requires about 5 Gigabytes of memory for computing the clustering, while the incremental approach allocates less than 300 Megabytes during analysis.

| (a) Run-time (regular clustering) | (b) Run-time (incremental analysis) | (c) Memory consumption |

Figure 11: Run-time and memory requirements on day 1–7 of application data set. The run-times and memory are presented for regular clustering and incremental analysis.

These results demonstrate the advantages of the incremental analysis. By processing the behavior reports in chunks, the run-time as well as memory requirements can be significantly reduced. This renders long-term application of behavior-based analysis feasible, for instance, as part of daily operation at a vendor of anti-malware products.

4   Related Work

Malicious software in its many forms poses a severe threat to the security of the Internet. Therefore, this area has received a lot of attention in the research community, and many different concepts and techniques for malware analysis have been proposed. Our contribution is related to several of these approaches, as we discuss in the following.

Static analysis of malware.   One of the first approaches for analysis and detection of malicious software has been introduced by Lo et al. (1995). Malware binaries are manually analyzed, such that *tell-tale signs*, indicative for malicious activity, can be extracted and later applied for detection of other malware samples. This approach has been developed further by Christodorescu and Jha (2003), who propose an architecture for detection of certain malicious patterns that are resilient to common obfuscation techniques. Finally, semantics-aware analysis of malware binaries has been devised by Christodorescu et al. (2005) and later on extended by Preda et al. (2007, 2008).

Still, static analysis of malware binaries is largely obstructed by obfuscation techniques which are frequently employed by attackers to thwart analysis (e.g., Linn and Debray, 2003; Szor, 2005; Popov et al., 2007; Ferrie, 2008, 2009). Although simple obfuscation may be compensated to a certain degree, the generic analysis of obfuscated code can be shown to be NP-hard (Moser et al., 2007a). Consequently, we apply dynamic analysis in our framework, as behavior is more difficult to conceal than its underlying code.

Dynamic analysis of malware.   To hinder static analysis of binaries the majority of current malware makes use of binary packers and cryptors. Recently, several systems have been proposed to generically unpack malware samples (e.g., Royal et al., 2006; Martignoni

et al., 2007; Dinaburg et al., 2008; Sharif et al., 2009). The common idea of all these systems is to execute malware binaries and decide during run-time, when the samples are un-packed, that is, all packing steps have successfully finished. The unpacked malware samples can be analyzed easier, as most obfuscation is removed. While these approaches share concepts of dynamic and static analysis, we base our work entirely on dynamic analysis of malware behavior, as this setting is more resilient to code obfuscation and independent of involved unpacking solutions.

Dynamic analysis of malware has received a lot of attention in the research community. Analysis systems such as CWSandbox (Willems et al., 2007), Anubis (Bayer et al., 2006a,b), BitBlaze (Song et al., 2008), Norman[2] or ThreatExpert[3] execute malware samples within an instrumented environment and monitor their behavior for analysis and development of defense mechanisms. In our framework, we employ the analysis tool CWSandbox for monitoring malware behavior, yet our approach is agnostic to the underlying sandbox, as long as a conversion to the proposed MIST representation is feasible. The analysis output of our framework provides the basis for timely defense against novel strains of malware and can be coupled with several related approaches. For example, samples of discovered malware classes can be directly used for observing current trends in malware development (Bayer et al., 2009b), constructing efficient detection patterns (Kolbitsch et al., 2009) and designing automata of malicious behavior (Jacob et al., 2009).

The main limitation of dynamic analysis is that typically only a single execution path is examined. This may lead to an incomplete picture of malware activity, as particular operations might only take place under specific conditions. To overcome this limitation, Moser et al. (2007b) propose the technique of *multi-path execution*. The behavior of malware samples is iteratively monitored by triggering and exploring different execution paths during run-time. Although we have not made use of this information, multiple traces of behavior can be incorporated into our approach by extending analysis from sequences to trees and graphs. As means for efficiently learning over trees have been very recently proposed (Rieck et al., 2010), we consider this extension a vital direction of future work.

Machine learning for malware analysis. Machine learning for classification of malware binaries has been first studied by Schultz et al. (2001) and Kolter and Maloof (2006). Both approaches employ string features of binary executables for training learning algorithms and discriminating malicious and benign files. An extension of this work to unpacked malware binaries has been recently devised by Perdisci et al. (2008). Furthermore, Stolfo et al. (2007) propose to analyze file content for detection of malware samples embedded within files using similar techniques. The first application of classification for malware behavior has been introduced by Lee and Mody (2006) and Rieck et al. (2008). We extend this work by proposing a prototype-based classification method, which significantly advances classification accuracy as well as efficiency.

Another line of research has focused on clustering of malware behavior for discovery of novel malware and reduction of manual analysis effort. The first clustering system for observed behavior has been introduced by Bailey et al. (2007) and later on extended by Bayer et al. (2009a) to be scalable, where the system devised by Bayer et al. (2009a)

---

[2]Norman Sandbox Center: http://sandbox.norman.no
[3]ThreatExpert – Automated Threat Analysis: http://www.threatexpert.com/

provides an excellent run-time performance in practice. However, both approaches require a single batch of malware samples and thus are limited in the overall capacity. We extend this work and propose an incremental analysis which by combination of clustering and classification enables iterative analysis of malware behavior in chunks of samples.

Evasion and mimicry. We propose a framework for the analysis of *malicious* behavior, hence we operate in an adversarial setting. An attacker may attempt to evade and obstruct behavior-based analysis in different ways. Common evasion techniques include mimicry attacks (Wagner and Soto, 2002; Kruegel et al., 2005; Fogla et al., 2006) and attacks against the analysis system (Tan et al., 2002; Chen et al., 2008). The input for our algorithm is based on dynamic analysis with CWSandbox. The tool is resilient to common obfuscation techniques designed to thwart static analysis. Thus, we assume that CWSandbox can observe the execution of a sample within the analysis environment. Note that specific attacks against this environment are still possible, but out of scope for this article.

The embedding using q-grams in our approach rests on work on host-based intrusion detection (e.g., Forrest et al., 1996; Lee et al., 1997; Hofmeyr et al., 1998) which is known to suffer from evasion and mimicry attacks (e.g., Wagner and Soto, 2002; Tan and Maxion, 2002; Tan et al., 2002). However, our framework differs from previous work in that the granularity of q-grams depends on the considered MIST level. For levels larger than 1, the resulting instruction q-grams comprise system calls as well as their arguments, thus reaching beyond plain system call identifiers. This inclusion of arguments hardens our approach against evasion techniques proposed for host-based intrusion detection. While an attacker may still aim at masquerading as a certain malware class, he is required to include valid system call arguments and thus triggers true malicious behavior.

Hence, the most promising evasion attack against our framework corresponds to triggering fake and real malicious behavior during individual executions, similar to evasion techniques against signature generation (Perdisci et al., 2006; Venkataraman et al., 2008). While this attack would be effective to certain degree, it can be alleviated by means of multi-path execution, which ultimately exposes all behavior of a malware binary to our analysis framework and rules out possible masquerading.

5 Conclusions

Malicious software is one of the major threats in the Internet today. Many problems of computer security, such as denial-of-service attacks, identity theft, or distribution of spam and phishing contents, are rooted in the proliferation of malware. Several techniques for automated analysis of malware have been developed in the last few years, ranging from static code inspection to dynamic analysis of malware behavior. While static analysis suffers from obfuscation and evasion attacks, dynamic analysis alone requires a considerable amount of manual inspection for crafting detection patterns from the growing amount and diversity of malware variants.

In this article, we introduce a framework to overcome this deficiency and enhance the current state-of-the-art. Our main contribution is a learning-based framework for the automatic analysis of malware behavior. To apply this framework in practice, it suffices to collect a large number of malware samples and monitor their behavior using a sandbox

environment. By embedding the observed behavior in a vector space, we are able to apply learning algorithms, such as clustering and classification, for the analysis of malware behavior. Both techniques are important for an automated processing of malware samples and we show in several experiments that our techniques significantly improve previous work in this area. For example, the concept of prototypes allows for efficient clustering and classification, while also enabling a security researcher to focus manual analysis on prototypes instead of all malware samples. Moreover, we introduce a technique to perform behavior-based analysis in an incremental way that avoids run-time and memory overhead inherent to previous approaches.

The proposed analysis framework enables automatic and efficient analysis of malware behavior, which provides the basis for timely defense against malware development. In combination with recent techniques for construction of detection patterns and heuristics, learning-based analysis of malware significantly strengthens security in the arms race with developers of malicious software.

Acknowledgements

Data Sets and Software

To foster research in the area of malware clustering and classification, and to enable a comparison of different approaches, we make the reference and application data sets available to other researchers at `http://pi1.informatik.uni-mannheim.de/malheur`. Moreover, we provide an open-source implementation of our analysis framework called *Malheur* at `http://www.mlsec.org/malheur`.

References

M. Anderberg. *Cluster Analysis for Applications*. Academic Press, Inc., New York, NY, USA, 1973.

P. Bächer, M. Kötter, T. Holz, F. Freiling, and M. Dornseif. The nepenthes platform: An efficient approach to collect malware. In *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 165–184, Hamburg, Germany, 2006. Springer.

M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 178–197, Queensland, Australia, 2007. Springer.

U. Bayer, C. Krügel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *Proceedings of Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, Hamburg, Germany, April 2006a.

U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006b.

U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proceedings of Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2009a.

U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. Insights into current malware behavior. In *Proc. of USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Boston, MA, USA, 2009b. USENIX Association.

A. Beygelzimer, K. S., and J. Langford. Cover trees for nearest neighbor. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 97–104, Pittsburgh, PA, USA, 2006. Omnipress.

J. Bezdek and L. Kuncheva. Nearest prototype classifier designs: An experimental study. *International Journal of Intelligent Systems*, 16:1445–1473, 2001.

X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of Conference on Dependable Systems and Networks (DSN)*, Anchorage, AK, USA, 2008.

M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of USENIX Security Symposium*, pages 12–22, Washington, DC, USA, 2003. USENIX Association.

M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 32–46, Oakland, CA, USA, 2005. IEEE CS Press.

M. Damashek. Gauging similarity with *n*-grams: Language-independent categorization of text. *Science*, 267(5199):843–848, 1995.

A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of Conference on Computer and Communications Security (CCS)*, pages 51–62, Alexandria, VA, USA, 2008. ACM Press.

R. Duda, P.E.Hart, and D.G.Stork. *Pattern classification*. John Wiley & Sons, second edition, 2001.

P. Ferrie. Anti-unpacker tricks 2 part one, Dezember 2008.

P. Ferrie. Anti-unpacker tricks 2 part seven, June 2009.

P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic Blending Attacks. In *Proceedings of USENIX Security Symposium*, pages 241–256, Vancouver, BC, Canada, 2006. USENIX Association.

S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, USA, 1996. IEEE CS Press.

J. Franklin, V. Paxson, A. Perrig, and S. Savage. An inquiry into the nature and causes of the wealth of internet miscreants. In *Proceedings of Conference on Computer and Communications Security (CCS)*, pages 375–388, Alexandria, VA, USA, 2007. ACM Press.

M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Co., 1979.

T. Gonzàlez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science 38*, pages 293–306, 1985.

S. Harmeling, G. Dornhege, D. Tax, F. C. Meinecke, and K.-R. Müller. From outliers to prototypes: ordering data. *Neurocomputing*, 69(13–15):1608–1618, 2006.

S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

T. Holz, M. Engelberth, and F. Freiling. Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones. In *Proceedings of European Symposium on Research in Computer Security (ESORICS)*, Saint Malo, France, 2009. Springer.

G. C. Hunt and D. Brubacker. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, WA, USA, 1999. USENIX Association.

G. Jacob, H. Debar, and E. Filiol. Malware behavioral detection by attribute-automata using abstraction from platform and language. In *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 81–100, Saint Malo, France, 2009. Springer.

C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of USENIX Security Symposium*, Montreal, QC, Canada, 2009. USENIX Association.

J. Kolter and M. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 8(Dec):2755–2790, 2006.

C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of USENIX Security Symposium*, Baltimore, MD, USA, 2005. USENIX Association.

A. Lanzi, M. Sharif, and W. Lee. K-Tracer: A system for extracting kernel malware behavior. In *Proceedings of Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2009.

T. Lee and J. J. Mody. Behavioral classification. In *Proceedings of Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, Hamburg, Germany, April 2006.

W. Lee, S. Stolfo, and P. Chan. Learning patterns from unix process execution traces for intrusion detection. In *Proceedings of AAAI Workshop on Fraud Detection and Risk Management*, pages 50–56, Providence, RI, USA, 1997.

C. Leita, M. Dacier, and F. Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID)*, Hamburg, Germany, 2006. Springer.

C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of Conference on Computer and Communications Security (CCS)*, Washington, DC, USA, 2003. ACM Press.

R. W. Lo, K. N. Levitt, and R. A. Olsson. MCF: a malicious code filter. *Computers & Security*, 14(6):541–566, 1995.

L. Martignoni, M. Christodeorescu, and S. Jha. OmniUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of Anual Computer Security Application Conference (ACSAC)*, pages 431–441, Miami Beach, FL, USA, 2007. ACM Press.

Microsoft. Microsoft security intelligence report (SIR). Volume 7 (January – June 2009), Microsoft Corporation, 2009.

T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proceedings of Anual Computer Security Application Conference (ACSAC)*, Miami Beach, FL, USA, 2007a. ACM Press.

A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2007b. IEEE CS Press.

K.-R. Müller, S. Mika, G. Rätsch, K. Tsuda, and B. Schölkopf. An introduction to kernel-based learning algorithms. *IEEE Neural Networks*, 12(2):181–201, 2001.

S. Omohundro. Five balltree construction algorithms. Technical Report TR-89-063, International Computer Science Institute (ICSI), 1989.

R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 17–31, Oakland, CA, USA, 2006. IEEE CS Press.

R. Perdisci, A. Lanzi, and W. Lee. McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Proceedings of Anual Computer Security Application Conference (ACSAC)*, pages 301–310, Anaheim, CA, USA, 2008. ACM Press.

I. V. Popov, S. K. Debray, and G. R. Andrews. Binary Obfuscation Using Signals. In *Proceedings of USENIX Security Symposium*, Boston, MA, USA, 2007. USENIX Association.

F. Pouget, M. Dacier, and V. H. Pham. Leurre.com: on the advantages of deploying a large scale distributed honeypot platform. In *ECCE'05, E-Crime and Computer Conference, 29-30th March 2005, Monaco*, Mar 2005.

M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *Proceedings of 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Nice, France, 2007. ACM Press.

M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst.*, 30(5), 2008.

W. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66:846–850, 1971.

K. Rieck and P. Laskov. Linear-time computation of similarity measures for sequential data. *Journal of Machine Learning Research*, 9(Jan):23–48, 2008.

K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *Proceedings of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 108–125, Paris, France, 2008. Springer.

K. Rieck, T. Krueger, U. Brefeld, and K.-R. Müller. Approximate tree kernels. *Journal of Machine Learning Research*, 11(Feb):555–580, 2010.

P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of Anual Computer Security Application Conference (ACSAC)*, pages 289–300, Miami Beach, FL, USA, 2006. ACM Press.

G. Salton, A. Wong, and C. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2001. IEEE CS Press.

M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2009. IEEE CS Press.

J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the International Conference on Information Systems Security (ICISS)*, Hyderabad, India, 2008. Springer.

S. J. Stolfo, K. Wang, and W.-J. Li. *Towards Stealthy Malware Detection*, volume 27 of *Advances in Information Security*, pages 231–249. Springer, 2007.

Symantec. Internet security threat report. Volume XIV (January – December 2008), Symantec Corporation, 2009.

P. Szor. *The art of computer virus research and defense*. Symantec Press, 2005.

K. Tan and R. Maxion. "Why 6?" Defining the operational limits of stide, an anomaly-based intrusion detector. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 188–201, Oakland, CA, USA, 2002. IEEE CS Press.

K. Tan, K. Killourhy, and R. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 54–73, Zurich, Switzerland, 2002. Springer.

P. Trinius, C. Willems, T. Holz, and K. Rieck. A malware instruction set for behavior-based analysis. In *Proceedings of 5th GI Conference "Sicherheit, Schutz und Zuverlässigkeit"*, Berlin, Germany, 2010. German Informatics Society.

C. van Rijsbergen. *Information Retrieval*. Butterworths, 1979.

S. Venkataraman, A. Blum, and D. Song. Limits of learning-based signature generation with adversaries. In *Proceedings of Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2008.

D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *Proceedings of Conference on Computer and Communications Security (CCS)*, pages 255–264, Washington, DC, USA, 2002. ACM Press.

C. Willems, T. Holz, and F. Freiling. CWSandbox: Towards automated dynamic binary analysis. *IEEE Security and Privacy*, 5(2), March 2007.