A Self-Learning AV Scanner

Arun Lakhotia and Andrew Walenstein
The Center for Advanced Computer Studies
University of Louisiana at Lafayette
Lafayette, LA 70506, USA
arun@louisiana.edu, walenste@ieee.org

Abstract

The nonzero "response time" of AV technologies offers a lacuna for hackers to exploit. By the time an AV company responds with a signature to detect a malicious sample, a hacker may release thousands of new variants. We present a self-learning AV scanner that effectively zeroes the response time needed to detect variants. The scanner uses methods from information retrieval research to determine whether a suspected sample is a variant of existing, known variant using an inexact match approach. The system is self-learning in that it is trained initially on known malicious samples in the AV research lab, but in the simplest case the knowledge base is not updated automatically. Utilizing the inexact matching properties of the scanner, this paper introduces several schemes for implementing automatic self-learning on top of the basic Vilo system, both for "in the cloud" learning and by integrating it into existing end-point scanners.

1 Introduction

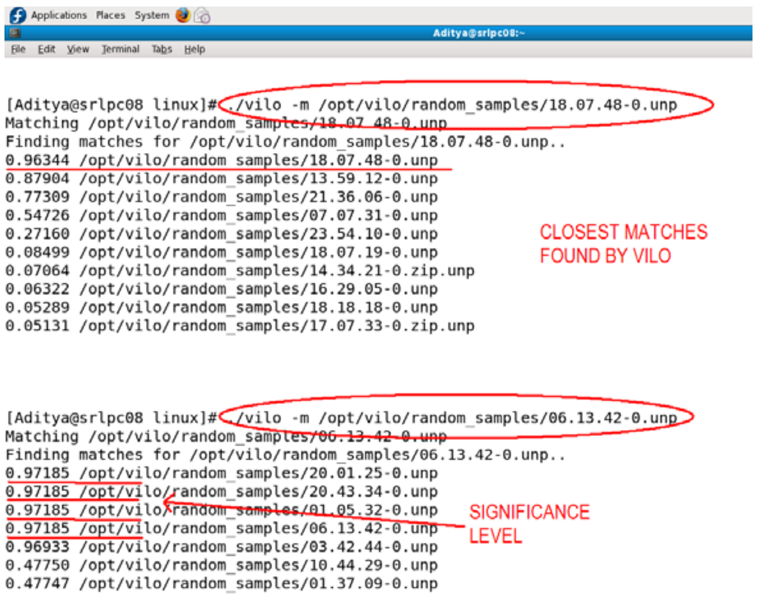
In the "game" between VXers and AVers, the VXers enjoy the first mover advantage. A VXer releases malware into the wild, which, through various means, finds its way to an AVer, who then develops a signature—static or behavioral—for that malware; the signature is then distributed to defend against that malware. In the current state-of-the-art the signatures are developed for a specific (or a collection) of *known* malware. Development of signatures, which happens in an AV lab, consumes a nonzero amount of time—the response time—between the first detection of that malware in the wild and distribution of its signature. The nonzero response time offers a window in which a malware can spread relatively unimpeded because the signature-based technology limits detection to historic malware.

VXers have also found a way to increase the response time, and thereby increase their window of opportunity. On an average an AVer can analyze between 5-10 samples a day. Since AV companies typically have between 5 and 100 AVers, an AV company can manually process between 500 and 20,000 samples a month. As reflected in the exponential growth in unique malware variants reported by most AV companies, VXers perform a denial-of-service (DoS) attack on an AV lab by inundating the wild with thousands of variants of the same malware. With the increased volume of new samples, an AV lab ends up stretching its resources, thus impacting the company's response time.

While signature-based technologies have proven themselves for defending against new and slow-changing malware, a new approach is needed when malware variant production rates increase drastically. Any new approach should enable detection of not just the known samples of a malware but also a reasonable class of variants of the malware as well. It is only by using a known malware to detect a large set of unknown variations of that malware can the AVers defend against the DoS attack of variants.

We propose detecting malware by performing an inexact match on significant portions of the entire malware, instead of using an exact match on an extract (signature), as performed by current AV scanners (Walenstein *et al.*, 2007). The proposed inexact match is based on principles from information retrieval and data mining: a training set of programs is used to automatically create a knowledge base of weighted *features* to match. Once trained, the system can be used for many purposes: it can be used to find matches of provided files, to classify unknown files, and to derive an analysis of similarities and differences between programs.   The learning is automatic in the sense that the only thing that must be specified is the training set.

One question is how best to allow the system to train itself so that it can adapt to the ever-changing defense landscape as new malicious variants are encountered. We use *Vilo*, a system from our lab, to explore the various engineering issues in developing such a scanner. Vilo provides a Google™ like querying capability on a database of binaries, with the twist that the query itself is a binary. Vilo returns a rank-ordered list of binaries in its database that match the query binary, see Figure 1. The system offers a good platform for investigating the issues of performance and accuracy when using inexact match in scanners. Making the technology self-learning add some new issues, such as scalability over a prolonged period of time.



Applications Places System
Aditya@srlpc08:~
File Edit View Terminal Tabs Help
CLOSEST MATCHES FOUND BY VILO
SIGNIFICANCE LEVEL

```
[Aditya@srlpc08 linux]# ./vilo -m /opt/vilo/random_samples/18.07.48-0.unp
Matching /opt/vilo/random_samples/18.07.48-0.unp
Finding matches for /opt/vilo/random_samples/18.07.48-0.unp..
0.96344 /opt/vilo/random_samples/18.07.48-0.unp
0.87904 /opt/vilo/random_samples/13.59.12-0.unp
0.77309 /opt/vilo/random_samples/21.36.06-0.unp
0.54726 /opt/vilo/random_samples/07.07.31-0.unp
0.27160 /opt/vilo/random_samples/23.54.10-0.unp
0.08499 /opt/vilo/random_samples/18.07.19-0.unp
0.07064 /opt/vilo/random_samples/14.34.21-0.zip.unp
0.06322 /opt/vilo/random_samples/16.29.05-0.unp
0.05289 /opt/vilo/random_samples/18.18.18-0.unp
0.05131 /opt/vilo/random_samples/17.07.33-0.zip.unp


[Aditya@srlpc08 linux]# ./vilo -m /opt/vilo/random_samples/06.13.42-0.unp
Matching /opt/vilo/random_samples/06.13.42-0.unp
Finding matches for /opt/vilo/random_samples/06.13.42-0.unp..
0.97185 /opt/vilo/random_samples/20.01.25-0.unp
0.97185 /opt/vilo/random_samples/20.43.34-0.unp
0.97185 /opt/vilo/random_samples/01.05.32-0.unp
0.97185 /opt/vilo/random_samples/06.13.42-0.unp
0.96933 /opt/vilo/random_samples/03.42.44-0.unp
0.47750 /opt/vilo/random_samples/10.44.29-0.unp
0.47747 /opt/vilo/random_samples/01.37.09-0.unp
```

Figure 1 Results of Querying Vilo for Matches

The rest of the paper is organized as follows. Section 2 summarizes the inexact match algorithm of Vilo and presents an overview of its capabilities. Section 3 presents how Vilo may be integrated with an AV scanner, providing it the capability of detecting variants. Section 4 presents explores some alternative architectures for developing a self-learning scanner using Vilo. This section is followed by our conclusions.

2 Using Inexact Match to Detect Variants

Central to constructing a self-learning scanner is the ability to be trained to perform inexact match. This section summarizes the training methods and the inexact match algorithm used by Vilo and also presents its performance results. Further details of the algorithm may be found elsewhere (Walenstein *et al.*, 2007).

2.1 Feature Vectors

Vilo uses a weighted feature matching approach (Karim et al., 2005) for performing inexact match. At an abstract level the comparison method consists of two parts. In the first part, each binary to match is analyzed and represented as a feature vector. In the second part, similarity (or distance) between feature vectors is computed for pairs of binaries. Extracting the feature vector is performed using the following steps:

Step 1: Disassemble binary and extract its sequence of operations.
Step 2: Extract features from the binary.
Step 3: Construct a feature vector, i.e., a histogram.

Figure 2 shows the extraction of operation sequences after disassembling a binary. The binary is disassembled. The sequence of operations, or mnemonic, from the disassembled program is extracted. This sequence of operations, after discarding the operands, is considered for further analysis. Discarding operands leads to some loss of information, though it also helps counter program variations due to register renaming.

	Disassembly	
Binary	**Operation**	**Operands**
`55`	***push***	`ebp`
`b8 11 00 00 00`	***mov***	`$0x11,eax`
`89 e5`	***mov***	`esp,ebp`
`57`	***push***	`edi`
`99`	***cltd***	
`56`	***push***	`esi`
`c7 45 e4 11 00 00 00`	***mov***	`$0x11,0xffffffe4 (ebp)`

Figure 2 Extracting Operations from Disassembly

Figure 3 demonstrates the extraction of features from the sequence of operations. A classical approach of information retrieval uses *n*-grams as features, where an *n*-gram is a sequence of *n* operations appearing in a sequence. Vilo uses *n*-perms instead. An *n*-perm is like an *n*-gram, but where the ordering of the operations is not considered. Figure 3 enumerates this difference using 2-grams and 2-perms. Using 2-grams the sequence of operations in Figure 2 yields five features, where as with 2-perms it has only three features. In a 2-perm the sequence `push-mov` is the same as `mov-push`.

Feaure Type	**Features extracted (for Figure 2)**
2-grams	`push_mov, mov_mov, mov_push, push_cltd, cltd_push`
2-perms	`push_mov, mov_mov, push_cltd`

Figure 3 Extracting Features from Operations

Figure 4 shows the feature vector created from the program sequence of Figure 2. It shows the number of occurrences of each 2-perm feature in the sequence of operations of Figure 2. A zero is shown for the push_pop feature because the program sequence did not have one. If one assigns an order for the different features, this can be written in standard vector notation as [3 1 2 0].

Feature	`push_mov`	`mov_mov`	`push_cltd`	`push_pop`
Number of occurrences	3	1	2	0

Figure 4 Constructing Feature Vector (using 2perms)

2.2 Inexact Comparison and Training

Vilo uses the above approach for creating a knowledge base which is used to perform inexact program comparisons. There are two parts to the training: construction of a knowledge base of feature vectors to match against, and automatically creating a weighting for the features. The former is critical for developing a scanner, as each of the feature vectors in its knowledge base corresponds to a type of "signature" for matching some family of related variants. The latter is instrumental in ensuring accuracy of matching: without an intelligent weighting system, all features found would be treated equally, allowing irrelevant commonalities to taint the results and permitting important commonalities to get lost in the details.

Training involves converting a training set of programs into a knowledge base of feature vectors, and then computing weights for all of the features involved. Vilo uses the so-called TFxIDF approach (Karim et al., 2005), where TF, the Term Frequency is multiplied by IDF, the Inverse Document Frequency. The IDF weights a feature by the inverse of how frequently it appears in the entire collection of binaries. Assume there are 10 programs and [9 8 3 2] represents the number of programs in which a feature appears. Here the first feature appears in 9 out of 10 programs. The corresponding IDF vector will be [$^1/_9$

$^1/_8\,^1/_3\,^1/_2$]. Consider the feature vectors v_1 = [3 4 2 1], v_2 = [4 5 1 0] this collection. Their weighted features will be $w_1 = [^3/_9\,^4/_8\,^2/_3\,^1/_2]$ = [.33 .25 .66 .50] and $w_2 = [^4/_9\,^5/_8\,^1/_3\,^0/_2]$ = [.44 .63 .33 .00].

Vilo uses cosine-similarity function to compute the similarity between two weighted feature vectors. Using the example feature vectors above, the comparison works out as follows:

$$sim(w_1, w_2) = \frac{w_1 \bullet w_2}{|w_1| \times |w_2|} = \frac{.33\times.44+.25\times.63+.66\times.33+.50\times.00}{\sqrt{.33^2+.25^2+.66^2+.50^2}\times\sqrt{.44^2+.63^2+.33^2+.00^2}} = 0.795$$

The expression also shows the computation of similarity for the two weighted vectors presented earlier.

2.3 Scanning

Given the above approach to training and calculating similarity scores, it is possible to define a general approach for creating scanners. In the simplest case, the approach follows three steps:

1. The system is trained with malicious samples, creating a knowledge base.
2. For each unknown sample, a feature vector is extracted.
3. A search is made in the knowledge base for a feature vector with a similarity score greater than some threshold. If one is found, the unknown program is presumed malicious.

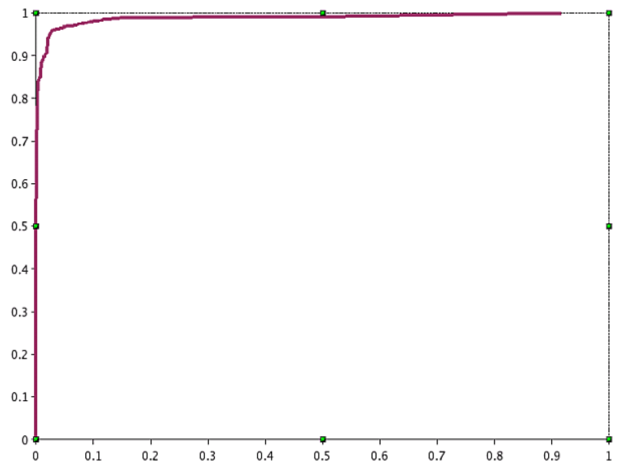


Figure 5 ROC Curve of Vilo

The key idea in the above approach is that variants of the training samples will be compared using the trained, inexact matching process. Since an exact match is not needed, the approach is not easily fooled by making a few relatively minor changes to a malicious program. Depending upon the similarity threshold chosen, the system can be made more or less sensitive to changes. While a great many different variations and improvements on this approach are possible, the above simplified three-step method illustrates the basic idea.

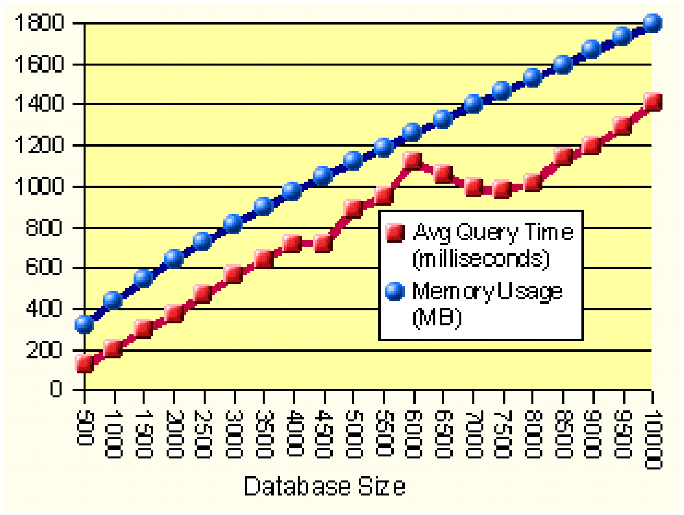


Figure 6 Query Response Time of Vilo vs Database Size

An n-folds cross validation experiment was performed to generate a baseline understanding of the merits of the approach. In the experiment Vilo was trained on 60 randomly selected samples from a group of 1,574 unpacked malware samples. It was then used to scan the remaining malware samples plus just over 20,000 benign programs from Microsoft Vista and XP system installations. This was done 10 times and the number of false positives and true positives for each was averaged. A false positive is recorded if the similarity is above the threshold but the sample is not a variant of the malware. This was then repeated for different threshold parameters.

Figure 5 shows a so-called ROC curve for this test. The ROC curve relates the rate of true positives to false positives as the the threshold parameter is varied. The X-axis shows the rate of false positives (number of false positives divided by the total number of benign samples in the test set) and the Y-axis the true positive rate (number of true positives divided by the total number of malicious samples in the test set). The graph indicates that even when a high similarity threshold is chosen in order to keep false positive rate very low, the system is able to find many variants. Various tweaks to this simplest approach can improve on these results, but it illustrates the capabilities of the basic method.

Appropriate data structures are needed for efficient implementation of feature vectors and computation of similarity. Vilo has been optimized to provide rapid response when querying for matches against a database of thousands of samples. Figure 6 plots the average response time for querying a match for varying size of the training database. Query times are less than second even for databases of over 7,000 training samples (Walenstein et al., 2007) on Dell Precision 650 with a single 2.80 Gz Intel Xeon (dual core) processor and 4 Gb dual-channel ECC DDR RAM.

Recall that each of the feature vectors in the knowledge base is able to match many different variants, so the number of feature vectors needed in a scanner will be related to the number of maware *families* rather than the number of malicious *samples,* and so the required training set is expected to be relatively small and grow slowly. Also, the times are for the full Vilo querying, which is suitable for use in an AV lab but not optimized for use of a scanner. These issues are explored in the following section.

3 Architectures for Self Learning Scanners

Since Vilo is able to detect malicious files it would perhaps be considered ideal if an entire scanner product could be developed using Vilo alone. However, that is not feasible for a variety of reasons. For instance, the Vilo technology is suited for finding variants of a family, but not completely new malware. Further, the definition of what variants are found is also constrained by the match algorithm chosen. In addition, there is the issue of performance: as the size of the training set grows, the straightforward approach outlined above may not scale to the performance requirement of a product.

For these reasons, the most likely method developing a self-learning scanner would be by integrating Vilo to an existing AV scanner. Typical AV scanners are complex compositions several detection methods, such as, generic detection, hashing, entry-point scanning, algorithmic scanning, code emulation, and behavior analysis. A Vilo-based classifier based could be introduced in a scanner as a yet another detection method. Like code emulation, the Vilo-based classifier may be more expensive than other classifiers in the scanner. Hence, the scanner would need to be configured to use such a classifier selectively.

Apart from being selective in uses of the Vilo-based classification, two important concerns need to be addressed. The first is the approach for utilizing the knowledge base in a deployed scanner: the methods most suitable for use in the AV lab for matching samples will not be the same as on a scanner. For example, a deployed scanner may not need to learn in the same way as in the AV research lab. The second is the strategy selected for keeping the knowledge base up-to-date so that it is continually adapted to the changing landscape of malware variations. That is, the issue is how to make the overall system self-learning. In this section we explore architectural strategies of developing a self-learning scanner and the engineering issues involved.

3.1 Strategies for Integrating Vilo into AV Scanners

Two basic strategies for introducing learning capability in a scanner using Vilo:

- Self-learning scanner with local training

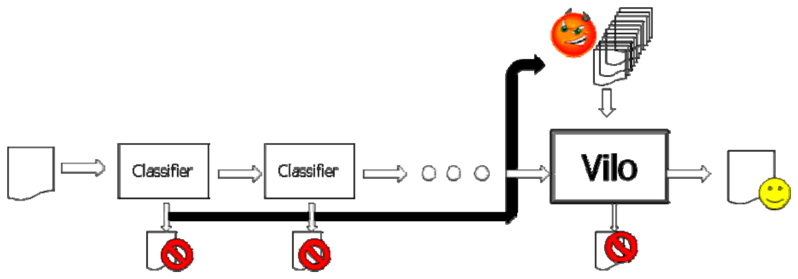


Figure 7 Self-learning Scanner with Local Training

- Self-learning scanner with learning in the cloud

In the first strategy, see Figure 7, the entire training happens in the product on the deployed scanner. A self-learning scanner may be created by introducing a Vilo-like classifier as the last component in the daisy-chain of classifiers. As a sample arrives, it is scanned using the normal classifiers. If the sample is malicious, it is fed to Vilo as a learning sample. If a sample passes the normal classifiers, it is fed to Vilo, which then performs a deep analysis to find any close matches with previously seen samples. If a match is found, Vilo flags the sample as malicious and also uses it to further improve its learning database.

The second strategy, see Figure 8, is a variation of the first. Vilo is still introduced as the last component in the daisy chain, and is used to classify only those samples that are considered clean by other methods. However, in this strategy the Vilo system is split in two parts: Vilo Learner and Vilo Classifier. The former resides in the cloud, meaning at an AV laboratory. Alternatively, it could also be located on a machine within an organization or at the ISP-level. The latter is integrated in the AV product, and resides on a customer's platform. Vilo Learner receives new samples and generates the weighted feature vectors. These vectors are passed on to Vilo Classifier. There is also feedback loop from the Vilo Classifier to the Vilo Learner. New variants detected by Vilo Classifier are sent to Vilo Learner in the cloud for training. Vilo Learner may also receive new samples directly from the AV laboratory, obviating the need to send to it malware detected by other classifiers.

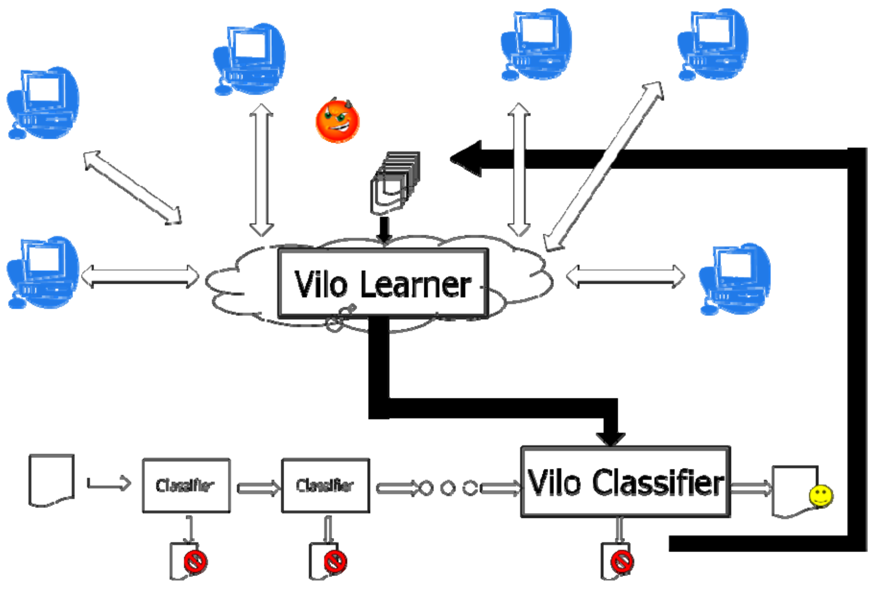


Figure 8 Self-learning Scanner with Training in the Cloud

Both strategies have their pros and cons. The local-training strategy helps address one weakness of the current AV infrastructure, a weakness we have not seen discussed. All installations of an AV product are homogeneous, since they have essentially the same signatures, modulo last update. Hence, if a VXer can defeat an installation of a scanner with the most recent update, she can be sure to defeat all installs. However, if the local training strategy is used, individual installations of an AV scanner will become heterogeneous, since each installation will be trained using only those malware samples that are actually detected by the scanner. Heterogeneity is usually good for security. A con is that one installation does not benefit by learning from other installations. So if variants arriving at an installation are several generations apart, they may not be detected.

The most significant pro of the training-in-the-cloud strategy is that the entire AV infrastructure would learn rapidly. If a variant is detected by one installation, its detection procedure will be communicated to all other installations. The con is the additional communication overhead and the potential of DoS on the cloud by releasing lots of variants to different machines.

The in-the-cloud strategy may also provide a tighter integration with the AV analysis process of an AV Lab. In an earlier paper (Venable et al., 2007) we describe how Vilo may be used in the AV Lab for triage, analysis, and signature generation. In such usage, Vilo introduces learning capability in the AV analysis process, thus reducing the response time in generating signatures for a new sample.

3.2 Underlying Engineering Issues

Irrespective of the strategy used for developing the self-learning scanner, some engineering issues will need to be addressed to beyond those discussed earlier. It will be naïve to continue to feed new variants to the learning system. The memory required to manage the database of feature vector increases, albeit linearly, with the number of variants, as does the processing time required to answer a query. Moreover, it is simply not helpful to add every single variant to the knowledge base simply because the variants are highly redundant and the technology involves inexact matching.   For instance if two variants would create the same match capabilities if they are added to the knowledge base it is only necessary to add one of them.

In order to make a system that can run continuously for a long time, and one that adapts to the changing landscape of malicious variants, it is necessary that training be done in such a way that maximal matching capabilities are derived from a given, limited set of memory and computing resources. This can be done in various ways:

- Forgetting variants that are old.
- Keeping a small representative set of variants of a family, instead of keeping all variants.
- Being selective about which features to use, instead of using all features.

By using techniques that achieve one or more of the above, the space requirement for the self-learning scanner can be controlled, thus ensuring that the system can function continuously for a long time.

4 Conclusions

A self-learning scanner, used as a component of existing AV scanners, provides a powerful and immediate way to close the lacuna of the current signature-based technology. In the “co-evolutionary” game between Avers and Vxers, the technology significantly raises the threshold of effort needed for the Vxers to penetrate the improved defenses. The self-learning scanner improves upon existing technologies in many ways. The scanner's detection ability continues to strengthen as it sees more variants. Thus, a hacker's attempt to inundate the scanner with new variants—a strategy used to defeat current scanners—only works to her detriment. Learning-based scanning also addresses another weakness of current AV systems—homogeneity. Hackers defeat current AV systems by installing a local copy of a scanner and creating a variant that it does not detect. Since these scanners are homogeneous, modulo there signature updates, the variant will most likely not be detected by other installations of the scanner. On the other hand because our scanners may learn independently, the technology automatically leads to heterogeneity. Since no two scanners may have reached the same state of learning, it does not help a hacker to attack a local copy of the scanner. The host-based scanners may naturally be connected in a network topology to share samples, thus creating a community of scanners. The community may be local-area based, protecting a single organization, or wide-area based, protecting a large collection of organizations. The learning component of the system may be moved to a machine in the cloud, thus providing a larger base of training samples while also taking advantage of significantly more computing resources.

5 References

Md. Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida (2005). Malware Phylogeny Generation Using Permuatations of Code. *Journal of Computer Virology*, 1 (1), pp. 13-23.

Rachit Mathur and Aditya Kapoor (2007). Exploring the Evolutionary Patterns of Tibs-packed Executables. *Virus Bulletin*, December 2007.

Michael Venable, Andrew Walenstein, Matthew Hayes, Christopher Thompson, and Arun Lakhotia (2007). VILO: A Shield in the Malware Variation Battle. *Virus Bulletin*, July, 2007, pp.5-10.

Andrew Walenstein, Michael Venable, Matthew Hayes, Christopher Thompson, and Arun Lakhotia (2007). Exploiting Similarity Between Variants to Defeat Malware. *Proceedings of BlackHat Briefings DC 2007*, Washington, DC. February 28-March 1, 2007.

6 Biography

Dr. Arun Lakhotia is a Professor of Computer Science at the Center for Advanced Computer Studies in the University of Louisiana at Lafayette. He is also Director of Software Research Lab, which specializes in malware analysis. His research has led to new ways to counter metamorphic viruses, to de-obfuscate code, and to detect new viruses by comparing their code with previously known viruses. Dr. Lakhotia teaches a course on malware analysis and has given tutorial on the subject in IEEE Working Conference in Reverse Engineering. He is recipient of the 2004 Louisiana Governor's University Technology Leader of the Year award. He received his Ph.D. from Case Western Reserve University in 1990.

Dr. Andrew Walenstein is an Assistant Professor of Computer Science at the University of Louisiana at Lafayette and a co-investigator in the Software Research Laboratory. He is currently studying methods for malware analysis, and brings in experience from the area of reverse engineering and human-computer interaction. He is the key designer and architect of Vilo, a Malware Defense Portal. He received his Ph.D. from Simon Fraser University in 2002.