A Malware Instruction Set for Behavior-Based Analysis

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

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

Abstract

We introduce a new representation for monitored behavior of malicious software called *Malware Instruction Set* (MIST). The representation is optimized for effective and efficient analysis of behavior using data mining and machine learning techniques. It can be obtained automatically during analysis of malware with a behavior monitoring tool or by converting existing behavior reports. The representation is not restricted to a particular monitoring tool and thus can also be used as a meta language to unify behavior reports of different sources.

1 Introduction

The field of malicious software (*malware*) is one of the most active and also one of the most challenging areas of computer security. In recent years, we are observing a huge increase in the number of malware samples collected by anti-virus vendors [7, 13]. Therefore, it is mandatory that we develop tools and techniques to analyze malware samples with no or very limited human interaction.

Typically, we distinguish between *static* and *dynamic* malware analysis as well as, in the field of the latter one, also between *code* and *behavior* analysis. Code analysis can be performed in a static way by using a disassembler or decompiler and also dynamically by the usage of a debugger. The main advantage of code analysis is that we can obtain a complete overview of what a given software does. However, code analysis is often obstructed by evasion techniques, such as binary packers, polymorphism and anti-debug techniques [6, 8, 9]. In behavior analysis, the malware is seen as a black box and only its effects to the system, *its behavior*, is analyzed. This can be achieved by several existing monitoring tools. Most of these tools monitor one specific group of operation, e.g. registry or filesystem accesses. But there are also comprehensive analysis suites, which perform an overall monitoring of all of the malware's operations [2, 12, 14]. These suites execute a malware sample in a controlled environment and record all system-level behavior by monitoring the performed system calls. As a result, an analysis report is created summarizing the observed behavior of the sample. In contrast to code analysis, behavior-based dynamic analysis suffers less from evasion and obfuscation techniques, as the code is not examined at all. Of course, there are different techniques, which can be used to prevent or falsify behavior analysis [11].

To handle the increasing amount and diversity of malware, dynamic analysis can be combined with clustering and classification algorithms. A behavior-based clustering of malware helps to find new malware families. Tagging new families at an early stage is essential to effectually fight them. Unlike clustering, the classification of malware does not provide information about new families, but helps to assign unknown malware to known families. Thus, classification of malware filters unknown samples and thereby reduces the costs of analysis. This combination of dynamic analysis and machine learning techniques has been recently studied in different scenarios [e.g., 1, 3, 4, 5, 10].

For effective and efficient analysis, however, algorithms and data representations need to be adjusted to each other, such that discriminative patterns in data are accessible to learning methods. In this article, we introduce a new behavior representation—the *Malware Instruction Set* (MIST)—which is exclusively designed for efficacy of analysis using data mining and machine learning techniques. It can be obtained automatically during analysis of malware with a behavior monitoring analysis tool or by converting existing behavior reports. The representation is not restricted to a particular report layout and thus can also be used as a meta language to unify behavior reports of different sources. Empirically, we demonstrate the accurate representation of behavior realized by MIST, while significantly reducing the size of reports and stored instructions.

This article is organized as follows: the Malware Instruction Set is introduced in Section 2. In Section 3 we present a short empirical evaluation of MIST and demonstrate its capabilities. Section 4 concludes the article.

2 The Malware Instruction Set

The majority of monitoring suites, such as Anubis [2] and CWSandbox [14], employ textual or XML-based formats to store the monitored behavior of malware. While such formats are suitable for a human analyst, they are inappropriate for further automatic analysis. The structured and often aggregated reports hinder application of machine learning. On the one hand, the XML representations are often too rich, providing an over-specific view on behavior which is not appropriate for finding generic behavioral patterns. On the other hand, textual formats are too coarse due to aggregation and simplification, such that involved patterns of behavior are not visible. Moreover, the complexity of textual representations increases the size of reports and thus negatively impacts run-time of analysis.

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 textual and XML-based formats, the monitored behavior of a malware binary is described as a sequence of instructions, where individual execution flows of threads and processes are grouped in a single, sequential 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 different levels of blocks, reflecting behavior with different degree of granularity. 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.

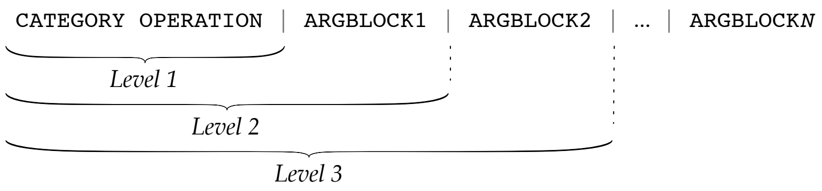

CATEGORY OPERATION | ARGBLOCK1 | ARGBLOCK2 | ... | ARGBLOCKN
Level 1
Level 2
Level 3

Figure 1: Schematic depiction of a MIST instruction. The field CATEGORY encodes the category of system calls where the field OPERATION reflects a particular system call. Arguments are represented as ARGBLOCKs.

Figure 1 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. For 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 temporary filenames, 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.

After introducing the main concept of MIST, we describe the concrete representation for several selected system calls. Since MIST features 120 unique calls with their corresponding attributes, we can not describe all of them, but restrict ourselves to some representative examples to explain the overall design philosophy.

2.1 MIST Design

Every MIST report consists of several MIST instructions, which encode individual system calls monitored during run-time of a malware binary. While MIST can be obtained directly during dynamic analysis of a malware binary, we herein focus on translation of XML-based reports generated by the analysis tool CWSandbox [14] to MIST. This conversion is order preserving, i.e., all contained MIST instructions occur in the same order as they were originally monitored and reported by CWSandbox.

As introduced previously, MIST instructions are composed of different fields: a CATEGORY field, an OPERATION field, and several ARGBLOCK fields. The field CATEGORY encodes the global class of the MIST instruction. As shown in Table 1, we distinguish between 20 different categories. Each category groups a set of related operations, e.g., the ‘`winsock_op`’ category contains 13 MIST instructions, including the ‘`create_socket`’, ‘`connect_socket`’, and ‘`send_socket`’ instructions. These instructions encode all system calls which are required to perform *Winsock* based network communication.

	Category	**# syscalls**		**Category**	**# syscalls**
01	Windows COM	4	*0B*	Windows Services	11
02	DLL Handling	3	*0C*	System	2
03	Filesystem	14	*0D*	Systeminfo	7
04	ICMP	1	*0E*	Thread	3
05	Inifile	5	*0F*	User	8
06	Internet Helper	5	*10*	Virtual Memory	5
07	Mutex	2	*11*	Window	5
08	Network	6	*12*	Winsock	13
09	Registry	9	*13*	Protected Storage	9
0A	Process	7	*14*	Windows Hooks	1

Table 1: MIST categories and encoding as well as the number of contained unique operations within each category

The amount and type of `ARGBLOCK`s for each MIST instruction depends on the particular system call. We will give some examples for those later in this text. We implement the concept of using *MIST levels* by dividing the ordered attribute blocks of each MIST operation into several levels, with higher level containing attributes with a higher *variability* and lower levels those which are more constant. The term variability is used with respect to different monitored behavior in multiple executions of one and the same sample, or executions of different variants from the same family. For example, if a monitored application creates a file in the Windows temp directory in one execution, it is highly likely that it will also create such a file in the very same folder in a second execution. In contrast to that, it is also likely that a different file name will be used, since temporary files are normally using random file names. Consequently, in a MIST operation we would encode the fact, that a file is created in level one, the target file path in level two, and the ultimate file name in level three.

The rationale underlying this rearragangement is that if we look at two executions of the similar programs on a lower level, e.g., level one, we observe identical behavior, whereas if we look on a higher level value, we are able to detect fine differences, e.g., in file names. This form of file name decomposition is strictly used in the MIST transformation: each file name is split up into the components `file type`, `file path`, `file name`, and `parameter`. In most cases the `file type` and `file path` are stored on a lower level than the rest, since these are more robust than other parts of file names.

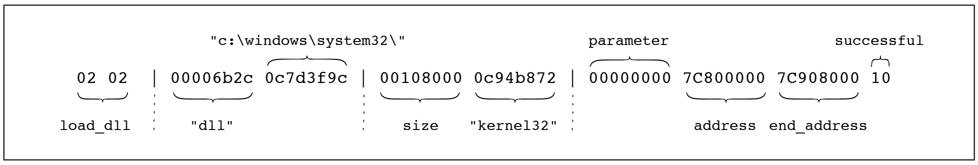
2.2 Examples

load_dll. The ‘load_dll’ system call is executed by every software during process initialization and run-time several times, since under Windows, dynamic-link libraries (DLLs) are used to implement the Windows subsystem and offer an interface to the operating system.

Figure 2 shows the original XML element of the CWSandbox representation and its corresponding MIST representation of one ‘load_dll’ operation. Except for the attribute filename_hash, all attributes are transfused into the MIST representation. We sometimes discard some attributes, if they are not useful for later analysis steps. To achieve a case insensitive transformation, all attribute values are converted to lower case first. Then we apply a fast hash function, such as the standard ELF, and store the results in a lookup table. Finally, the resulting lookup index is used in our MIST representation as a hexadecimal number.

```
<load_dll filename="C:\WINDOWS\system32\kernel32.dll" successful="1"
 address="#7C800000" end_address="#7C908000" size="1081344"
 filename_hash="c88d57cc99f75cd928b47b6e444231f26670138f"/>
```

(a) CWSandbox representation



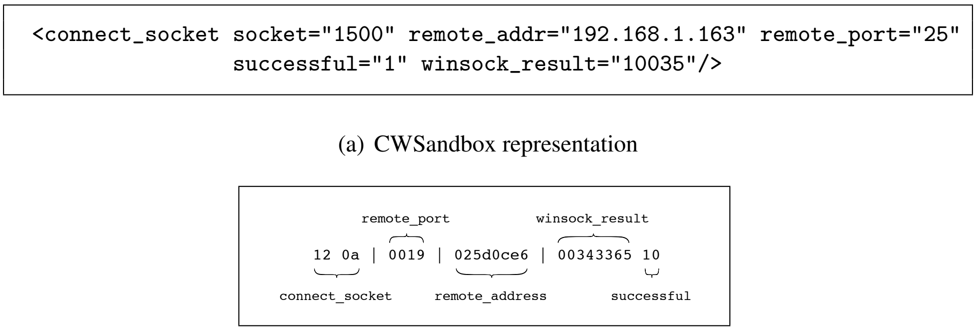
(b) MIST representation

Figure 2: Feature representations of system call ‘load_dll’. The CWSandbox format represents the system call as an attributed XML element, while the malware instruction set (MIST) represents it as a structured string.

For the ‘load_dll’ instruction we order the attributes as follows: the first attributes are the file extension and file path of the library. This information is quite constant and, therefore, is included in the second MIST level. Note that we order all attributes with respect to their *variability*. On level three we store the file size and the file name of the library. Both of these values may differ when two different variants of the same program are considered and, therefore, should only be stored on a lower—hence more detailed—MIST level.

It is evident from this example that the MIST instruction is more suitable for data mining and machine learning techniques than the traditional XML representation. The compact encoding ensures a proper comparability between all ‘load_dll’ instructions and the ordering of all attributes permits the introduction of MIST levels, which finally enhances the quality of analysis.

connect_socket. If a malware binary initiates a TCP network communication via the Winsock library, it has to perform the ‘connect_socket’ system call. Figure 3 shows a monitored system call in CWSandbox and MIST representation. The sandbox records five parameters for this system call, namely the used socket number, the IP address and port of the remote server, the winsock result value and the successful flag, which states if the connection could be established or not.



```
<connect_socket socket="1500" remote_addr="192.168.1.163" remote_port="25"
                successful="1" winsock_result="10035"/>
```

(a) CWSandbox representation

```
12 0a | 0019 | 025d0ce6 | 00343365 10
```

(b) MIST representation

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

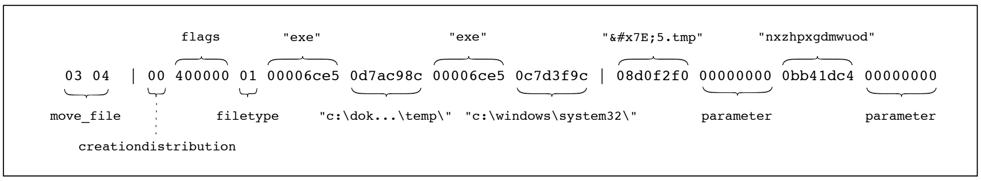
Except for the socket number, all attributes are converted into MIST representation (the socket number is a dynamic value that is created by the operating system and has no further semantic meaning). Furthermore, the attributes are ordered as show in Figure 3(b). Since the IP address of the remote server and the Winsock result value may vary quite often, this information is less interesting for our purpose and, therefore, moved into the third respectively fourth MIST level. Only the remote port remains in the second MIST level of this instruction.

move_file. The third example is one of the most complex conversions. As already shown, we decompose the file names and only use the file paths and extensions on the second MIST level, and delay the ultimate file names and possible parameters to a higher level. For the ‘move_file’ call we have one filename-attribute that specifies the source file, and another one for the destination file. Thus, we have to split and arrange two file names. Figure 4 shows a monitored ‘move_file’ system call in CWSandbox and MIST notation.

Again, the hash values are not converted into the MIST representation, because the malware binary may change, e.g., in discharge of the used packer. Therefore, only the ultimate file names and the parameters are those converted attributes which are not contained in the second MIST level, see Figure 4. In contrast to the file names and all prior discussed attributes, the values of the filetype, the desiredaccess, and the flags attributes are not hashed while converting into MIST representation, but transferred di-

```
<move_file filetype="file" srcfile="C:\DOKU...\Temp\&#x7E;5.tmp.exe"
srcfile_hash="hash_error"
dstfile="C:\WINDOWS\system32\nxzhpxgdmwuod.exe"
dstfile_hash="hash_error" desiredaccess="FILE_ANY_ACCESS"
flags="MOVEFILE_REPLACE_EXISTING"/>
```

(a) CWSandbox representation



(b) MIST representation

Figure 4: Feature representations of system call `'move_file'`. The CWSandbox format represents the system call as an attributed XML element, while the malware instruction set (MIST) represents it as a structured string.

rectly into the MIST instruction. Since all possible values are known in advance, we use a fix mapping between the attribute values and the MIST values. Table 2 shows a fragment of the mapping table of the flags attribute. There are 26 predefined values which are freely combinable.

Value	MIST bit vectors
FILE_ATTRIBUTE_ARCHIVE	00000000000000000000000001
FILE_ATTRIBUTE_COMPRESSED	00000000000000000000000010
FILE_ATTRIBUTE_HIDDEN	00000000000000000000000100
FILE_ATTRIBUTE_NORMAL	00000000000000000000001000
FILE_ATTRIBUTE_OFFLINE	00000000000000000000010000
...	...
MOVEFILE_WRITE_THROUGH	00100000000000000000000000
MOVEFILE_CREATE_HARDLINK	01000000000000000000000000
MOVEFILE_FAIL_IF_NOT_TRACKABLE	10000000000000000000000000

Table 2: Mapping between CWSandbox and MIST representation for possible values of the flags attribute.

For attributes like `flags` or `desiredaccess`, we sum up the corresponding MIST bit vectors. The result is then interpreted as numeric value and transformed into hexadecimal encoding. This approach is robust against permutations between the single values and allows re-translating of the MIST encoding.

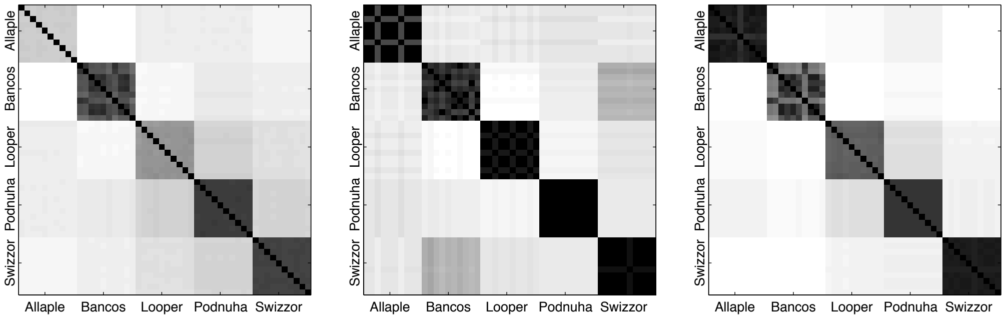
3 Empirical Evaluation

We now proceed to present an empirical evaluation of the MIST representation. For this evaluation, we consider a small sample of 500 malware binaries which have been obtained from the CWSandbox web site available at `http://cwsandbox.org`. The malware binaries have been collected over a period of more than two years from a variety of sources, such as honeypots, spam traps, anti-malware vendors, and security researchers. From the overall database, we select a subset of 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 [1], we expect the selection using different scanners to be reasonable consistent and accurate. The labeled malware binaries are then executed and monitored using CWSandbox, resulting in a total of 500 behavior reports of 5 common malware classes.

For each report we consider four different representations of behavior: First, the original XML format as generated by CWSandbox, second an extended version of the XML format proposed by Rieck et al. [10] and, third and forth, the MIST representation of behavior for level 1 and level 2.

3.1 Behavior Representation

In this first experiment we compare the utility of MIST for representing malware behavior in a concise form. In particular, we apply the technique for embedding textual data of malware reports to a vector space introduced by Rieck et al. [10]. That is, each report is represented by a vector, such that the similarity of behavior can be assessed in terms of geometric distances.



(a) XML

(b) MIST level 1

(c) MIST level 2

Figure 5: Comparison of feature representations for malware behavior. Distance matrices for the behavior of five malware families are shown, each represented by ten reports. Dark shading indicates small distances and light shading large distances.

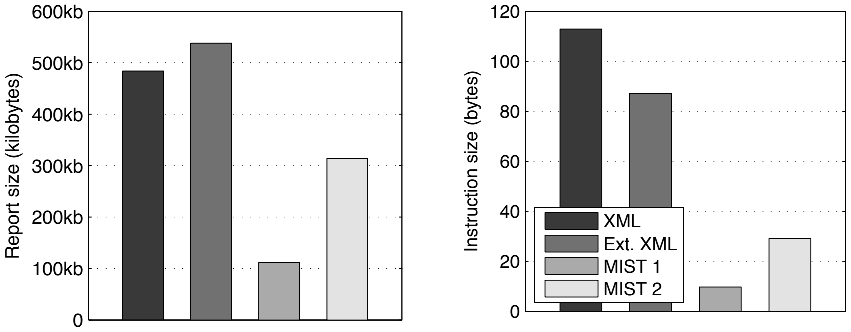
Figure 5 shows results for the representation using different formats. Distance matrices are shown for three behavior representations, where dark color indicates low distances and light color high distances. The extended XML format is omitted, as it only marginally differs from the regular XML representation. The representation provided by XML data is not sufficient to reflect all presented classes of malware. For the two malware families ALLAPLE and LOOPER the distance matrix for the XML data shows very light colored sections. Thus, the distance between the individual behavior reports is huge which precludes a good clustering or classification based on XML data. A threshold to classify all ALLAPLE members correctly would be too low for other malware families, for example, PODNUHA and SWIZZOR would also be classified as ALLAPLE.

The distance matrices for the MIST encoded data show a much darker coloration along the diagonal. The members of the individual malware families are much closer to each other, resulting in almost black colored sections for four malware families using MIST level 1 representation. Overall the distance matrix for MIST level 1 shows the closest match within the malware families, but in exchange also shows matches among different malware families, e.g., BANCOS and SWIZZOR. For MIST level 2 representation, we reduce the noise between the families at the cost of a sligthly less accurate separation between classes. This reduction of noise results from the more detailed information contained in MIST level2.

This experiment demonstrates the good representation of behavior realized by MIST. Although we have only studied five classes of malware, it is evident that MIST is more suitable for behavior-based analysis than XML, as samples of the same malware classes exhibiting similar behavior are close to each other in the vector space.

3.2 Data Reduction

In the second experiment we compare the size of instructions and reports between XML-based formats and MIST. Figure 6 shows a comparison for the four considered formats, namely the original XML format of CWSandbox, an extended representation used by Rieck et al. [10], and MIST level 1 and 2.



(a) Average size of behavior reports

(b) Average size of instructions

Figure 6: Comparison of feature representations for malware behavior in terms of size.

The MIST representation significantly reduces both the length of the reports, as well as, the average length of instructions. While reports in XML format on average comprise more than 450 kilobytes, for MIST level 1 only 100 kilobytes and for MIST level 2 300 kilobytes are necessary to store the same behavior – though with a different granularity. Similarly, for the size of instructions MIST provides a more concise representation, where on average instruction requires less than 40 bytes. By contrast, for both XML formats the required size per instruction is twice as large, yielding over 80 bytes. Particularly with regard to data mining and machine learning methods this data reduction is quite important, since it dramatically reduces the run-time of analysis on large datasets.

This experiment demonstrates the advantages of representing behavior using the proposed MIST representation. The behavior is represented in a way which allows far better to discriminate classes, but at the same time storage size is significantly reduced, which ultimately provides the basis for effective and efficient further analysis of monitored malware behavior.

4 Conclusions

The *Malware Instruction Set* (MIST) is a meta language for monitored behavior, which can be used to make the results of different behavior monitoring systems more comparable. In addition, the MIST representation is optimized for analysis of software behavior using data mining and machine learning techniques. We restrict all instructions to significant attributes only and rearrange these attributes to achieve a well sorted ordering which allows the introduction of accuracy levels. Furthermore, we reduce the size of the reports by encoding each instruction and attribute with the help of hash tables. In summary, using the MIST representation as input enhances both the run-time and the results of data mining and machine learning analysis.

Acknowledgements

This work has been accomplished in cooperation with the German Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik (BSI)).

References

[1] 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, 2007.

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

[3] 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)*, 2009.

[4] J. Kolter and M. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 2006.

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

[6] 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)*, 2003.

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

[8] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of IEEE Symposium on Security and Privacy*, 2007.

[9] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *Proceedings of the 16th USENIX Security Symposium*, pages 275–290, 2007.

[10] 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, 2008.

[11] M. Sharif, A. Lanzi, G. Jonathon, and W. Lee. Impeding malware analysis under conditional code obfuscation. In *Proceedings of Symposium on Network and Distributed System Security (NDSS)*, 2008.

[12] 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 4th International Conference on Information Systems Security*, Hyderabad, India, Dec. 2008.

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

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