CBM: Free, Automatic Malware Analysis Framework Using API Call Sequences

Yong Qiao, Yuexiang Yang, Jie He, Chuan Tang and Zhixue Liu

Abstract Classic static code analysis for malware is ineffective when challenged by diverse variants. As a result, dynamic analysis based on malware behavior is becoming thriving in malware research. Most current dynamic analysis systems are provided as online services for common users. However, it is inconvenient and ineffective to use online services for the analysis of a big malware dataset. In this paper, we propose a framework named CBM enabling tailored construction of an automated system for malware analysis. In CBM, API call sequences are extracted as malware behavior reports by dynamic behavior analysis tool, and then API calls will be transformed to byte-based sequential data for further analysis by a novel malware behavior representation called BBIS. The peculiar characteristic of CBM is that it can be customized freely, contrary to current online systems, which supports local deployment and runs mass malware analysis automatically. Experiments were carried out on a large-scale malware dataset, which have demonstrated that CBM is more efficient in reducing storage size and computation cost while keeping a high precision for malware clustering.

Keywords Automatic malware analysis · Open-source · API-call sequences · Clustering · API-Hook

Y. Qiao (✉) · Y. Yang · J. He · C. Tang
National University of Defense Technology, Changsha 410073, China
e-mail: qiaoyong10@nudt.edu.cn

Y. Yang
e-mail: yyx@nudt.edu.cn

J. He
e-mail: hejie@nudt.edu.cn

C. Tang
e-mail: tangchuan@nudt.edu.cn

Z. Liu
China Navy Equipment Academy, Beijing 100161, China
e-mail: lstprince@163.com

F. Sun et al. (eds.), *Knowledge Engineering and Management*,
Advances in Intelligent Systems and Computing 214,
DOI: 10.1007/978-3-642-37832-4_21,

1 Introduction

Malware, as malicious software, is the basement for attackers to implement intrusions and maintain them. Conventional methods disassemble the malware to carry out detailed analysis on the malware at the assembly code level. There are two prerequisites in such situation: one is that the researchers should have deep technical insights of assembly, the other is that the analysis processes should be efficient enough to cope with the constant renewal and variation of the malware. However, the widespread packing and obfuscation technology utilized by malware make it even more challenging.

Fortunately, dynamic analysis based on behaviors provides a new perspective to analyze malware, different from static code analysis, it runs malware in a controlled environment called *sandbox* and captures the behaviors triggered upon operation systems. With such technique, we can perform the malware analysis automatically at a large scale. Several systems have been implemented, such as *CWSandbox* [1], Anubis [2], Norman,[1] ThreatExpert,[2] et al. Usually, most of them provide free service for the submission and online analysis of malware binaries.

Unfortunately, source codes or packages for local installation are not available for those systems. Moreover, above tools have limitations on the number of submissions and the size of executable applications, which limits their usage in large-scale analysis. Therefore, in this paper we will introduce a locally deployable system for automatic malware analysis. This framework would enable a fully controllable analysis procedure under your control as our framework is based on two open source systems: *Cuckoo sandbox*[3] and *Malheur* [3]. Our contributions can be listed as follows:

- *Reasonable integration solution.* We proposed an automatic malware analysis framework CBM based on the improvement and integration of the existed open source system *Cuckoo sandbox* and *Malheur*. CBM can abstract the analysis reports from *Cuckoo sandbox* and encode the reports to sequential data for *Malheur* to perform clustering and classification analysis.
- *BBIS* (Byte-based Behavior Instruction Set) and CARL (Compression Algorithm of high Repeatability in Logarithmic level). We designed BBIS to transform *Cuckoo sandbox*'s analysis reports and make them recognizable by *Malheur*. In theory, BBIS can maintain a minimum size of reports while keeping the full messages needed. CARL can further compress the reports by means of reducing the high repeatability in API calling while keeping or improving the malware clustering performance.

[1] http://www.norman.com/

[2] http://threatexpert.com/

[3] http://www.cuckoobox.org/

- *Evaluation on a large-scale malware data set*. We have achieved bigger then 90% precision of clustering while using less computation time and less storage size.

2 Related Works

One of the first approaches for analysis and detection of malware was introduced by Raymod [4] in 1995. Malware binaries are manually analyzed by extracting static code signs, indicative for malicious activity. Those features are then applied for detection of other malware samples. This approach has been improved further by Christodorescu [5] and Preda [6] et al. and became a semantics-aware analysis method. On the attackers' side, Popov [7], Ferrie [8] et al. proposed obfuscation techniques to thwart static analysis. Although Martignoni [9],Sharif [10] have proposed several systems to generically unpack malware samples, human intervention is still needed.

Dynamic analysis of malware has attracted lots of attention recently. Multiple systems have been proposed, such as *CWSandbox* [1], *Anubis* [2], *BitBlaze* [11]. Those systems can execute malware binaries within an instrumented environment and monitor their behaviors for analysis and development of defense mechanisms. For further analysis, Konard [3] developed *Malheur* to cluster and classify malware by processing the malware behaviors, he employed the CWSandbox for monitoring malware behavios and represented the results in MIST [12] format, by means of n-grams algorithm and several related approaches. Malheur can classify the malware to a predefined set of classes and find novel classes by clustering. Unfortunately *CWSandbox* and MIST are not open-source, so we use *Cuckoo sandbox* and BBIS as the replacement.

Mamoun [13] and Xiaomei [14] adopted API call sequences to reflect malware system behaviors. We adopt the same methodology and it turns out to be feasible and efficient.

3 CBM: Build your Own System for Automatic Malware Analysis

Our system is named CBM since it consists of three major components: *Cuckoo Sandbox*[4], *BBIS* and *Malheur*. The relationship among the three modules is demonstrated in Fig. 1. The workflow is summarized below:

[4] We will use *Cuckoo* as the shorthand of *Cuckoo Sandbox* later in this article.

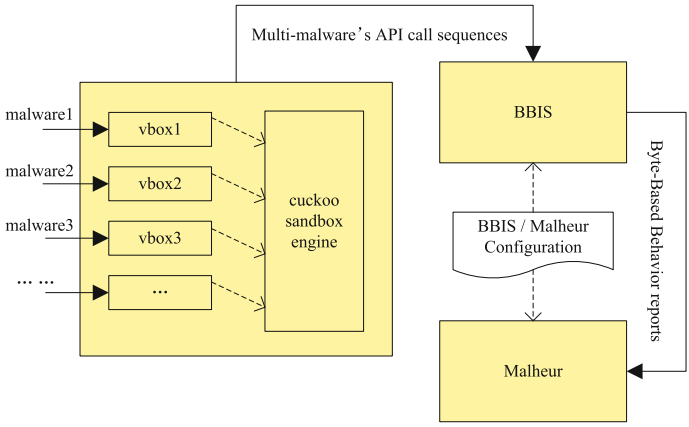

Fig. 1 The framework of CBM

1. CBM first executes and monitors multi malware binaries in Cuckoo simultaneously. Based on the analysis results, CBM extracts the API call sequences as each binary's behavior report.
2. CBM encodes the API call sequences to byte-based behavior reports using BBIS and CARL algorithms.
3. CBM uses Malheur to embed the sequential data in a high-dimensional vector space by n-gram algorithm, and then calculates the similarity of vectors, at last machine learning techniques for clustering and classification are applied on the vectors.

In the following sections, we will discuss each step in detail, including: how to improve Cuckoo's monitoring capabilities by adding API hooks, the design of BBIS and CARL, and the extensions in Malheur.

3.1 The Use and Improvement of Cuckoo

Cuckoo can deploy multi-VMs on one single host and run them simultaneously. This guarantees the efficiency of Cuckoo to analyze massive malware binaries quickly. Another advantage is that Cuckoo's new hook engine *cHook.dll*[5] embedded in *cmonitor.dll* has implemented a new technique called *trampoline* which can make the monitoring upon malware more difficult to be discovered. CBM use Cuckoo to obtain the malware binaries's behavior reports composed of API call sequences. A number of API hooks are needed to fetch the individual

[5] http://honeynet.org/node/755

Table 1 The differences between cuckoo and CWSandbox's hooking objects

	Cuckoo	CWSandbox
API Hooks	42	120
Categories	11	14
Winsock API	No	Yes
ICMP	No	Yes
SystemInfo	No	Yes

behaviors. We compared Cuckoo with CWSandbox and found that Cuckoo's monitoring scope is not comprehensive and should be improved.

Improvement in Cuckoo by adding Hooks. *Cmonitior.dll* is a kernel component of Cuckoo, which will be injected into malware's memory space to hook the original API calls for tracking. Herein, we improve Cuckoo's monitoring capabilities by adding API hooks in *cmonitior.dll*. *cmonitior.dll* is written in *CPP* language and compiled by *VisualStudio 2010*, in which 42 windows API functions were hooked and tracked. Thus, we can extract malware behaviors brought about by these API calls. Table. 1 lists the differences between Cuckoo and CWSandbox's hooking number and categories.

The number of hooks in Cuckoo are much less than that in CWSandbox with an approximate number of categories, which means that Cuckoo can hold the basic monitoring ability from 11 categories. However, Table. 1 also shows that Cuckoo does not set hooks in *Winsock API*, *ICMP API* and *SystemInfo API*. Since the malware like *bots*, *worms* et al. always have similar behavior in network communication, therefore, it is not wise to ignore those related API calls. In CBM, we have selected 18 additional API functions in these three aspects to be hooked.

To add hooks on API functions, specific hooking processes in *cmonitor.dll* need to clarified. Figure. 2 demonstrates the differences between regular API calls and API calls hooked by *comnitor.dll*.

We can see from Fig. 2 that *chook.dll* is responsible for the preservation of specific API functions' pointer, the transfer of the calls to new functions, and the return of the original results to invoker. We can add hooks in three steps: (1) Append real API function pointers for applying virtual memory space; (2) append customized functions to call the real API functions inside and record the corresponding message including timestamp, operation parameters, and returning results; (3) install hooks by invoking the output function *Hookattach()* from chook.dll. The space is limited to give adequate coverage of these detailed processes, we will publicize a full version report including the comprehensive solution.

Obtaining API call sequences. In Cuckoo's results folder, each process of malware was assigned a *csv* file that contains the detailed API call messages. However, CBM does not care about intricate things like the arguments, the process ID, the return value, and so on. CBM only extracts the lists of the API calls from all of the *csv* files that correspond to the process and concatenates them to be one sequence. CBM employs three rules to make the concatenation. First, the API call list within the same process are ordered by timestamp, with the earlier one coming

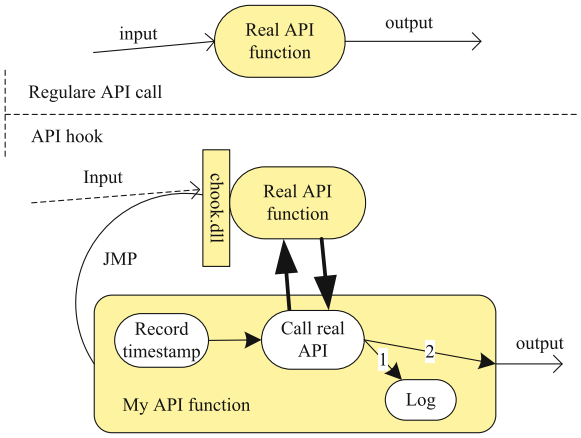

Fig. 2 Hooking processes in cmonitor.dll

first; second, API call lists from different processes will be connected from head to tail in the whole sequence; finally, different lists of multi-process will be ordered by the first timestamp of each API call list, with the earlier one coming first.

3.2 The Design of BBIS and CARL

BBIS is designed to transfer API call sequences to byte-based sequential data recognizable by Malheur in order to do clustering and classification.

Currently, Malheur only supports MIST [12] and byte-based sequential data. MIST can transfer the Malware original behavior reports to multi-level instructions, reflecting behaviors with different degree of granularity. For example, the category and operation are classified into the level 1 in MIST, followed by the arguments. Those considerations are good at recoding the whole messages of behaviors. However, testing results of Malheur showed that MIST with more levels consistently got worse results than with only one level in clustering and classification computations. Therefore, in CBM we only extract the API names and in BBIS we do not set multi-levels. We only need to build a suitable mapping table to change the API call sequence to a byte-based sequence. Byte-based here means each byte or fixed length of bytes will reflect one feature in the sequence. For instance, *abc* can reflect three API calls. In CBM, we utilize the following rules to build the mapping table.

Mapping rules of BBIS. There are two parameters that should be set up first. One is *IsVisible*, which indicates whether visible characters are used or not in the mapping table. The other *UpFeatures* is the upper limit of amount of unique features in all of the sequences. For example, Cuckoo has hooked 42 API functions by default, so the *UpFeatures* here is 42, CBM has improved the number of API hooks to 60, so the *UpFeatuers* will be 60 in CBM. Once the two parameters are specified, BBIS can build the mapping table in following way:

Table 2 Part of BBIS mapping table

API Function	BBIS-1	MIST	BBIS-2
LoadLibraryA	a	02 02 | ...	06
CreateFileW	b	03 01 | ...	08
RegQueryValueExW	p	09 05 | ...	2D
bind	#	12 06 | ...	68

(1) If *IsVisible* is TRUE, visible characters will be chosen to build the table. In the ASCII coding table, one byte can represent 94 different visible characters (from 33 to 126 in decimal system). When $UpFeatures <= 94$, we can select characters randomly with the number of *UpFeatures* to map the features in sequences. If $UpFeatures > 94$, BBIS need more bytes to map one feature, the number of bytes needed can be calculated by Eq. (1):

$$94^{n-1} < UpFeatures \leq 94^{n} \tag{1}$$

(2) If *IsVisible* is FALSE, It is not necessary to choose the visible characters to map the features. In such case, one byte can represent $2^8 = 256$ features. Consequently, in the map table, we can use the hexadecimal characters (from 00 to FF) to map the features and write the hexadecimal sequence to report file in binary mode. If $UpFeatures > 256$, BBIS also needs more bytes to map one feature by calculating Eq. (2):

$$256^{n-1} < UpFeatures \leq 256^{n} \tag{2}$$

Since in CBM, the $UpFeatures = 60$, which is less then 94, we can select the visible mode to build the mapping table. For example, the left column of Table. 2 demonstrates the part of mapping table we used in CBM. BBIS is a common way to represent features, not only designed for CBM. For example, MIST has 120 features, since $120 > 94$, thus we can use 2 bytes to represent MIST code, just like the right column of Table. 2 which uses the 2 hexadecimal characters to map the MIST code, it is interesting that the 2 hexadecimal characters can also be written to file in binary mode within one byte size if *isVisible* is set to *FALSE*.

CARL - Compression Algorithm of high Repeatability in Logarithmic level. During the process of dynamic analysis of malware, one notable thing is that some API functions can be continuously invoked thousands of times in a single analysis. Usually, this can be attributed to mistakes in coding or unsuitable execution environment for the malware. The API functions like *ReadFileW*, *VirtualAllocEx*, *RegOpenKeyW* are found many times with such situations. However, if parts of malware in the family have high repeatability problem, the similarity between malware will be greatly affected and much more redundant computation might be introduced. Therefore, we proposes an algorithm called CARL in CBM to reduce the repeatability.

Definition 1 REPEAT. For a sequence *Seq* and a subsequence *SSeq*, if *SSeq* is composed of consistent characters and the length of it is more than one, we define the length as the REPEAT of the *SSeq*.

Definition 2 SN and NL. For a sequence *Seq* and a subsequence *SSeq*, *SN* is a start number to run the CARL process, *NL* is the new length of *SSeq*. If the *REPEAT* of *SSeq* is bigger then *SN*, the length of *SSeq* will be reduced to *NL* by Eq. (3).

The calculation of CARL is illustrated in the following Equation:

$$NL = SN + \beta \cdot Round(log_{\gamma}(REPEAT - SN)) \tag{3}$$

In Eq. (3), β and γ are used as the adjustment coefficients. Usually, γ is set as 2, β is set as 1. For example, if $S = \{aQcQQQQQQQQQQQQddhhhdddd\}$, we can get sub-sequences:$SSeq1 = \{QQQQQQQQQQQQ\}$, $SSeq2 = \{dd\}$, $SSeq3 = \{hhh\}$, $SSeq4 = \{dddd\}$, which REPEAT are bigger then 1. If we define $SN = 5$ here, the only subsequence that needs to be reduced is $SSeq1$, we can see the REPEAT of $Seq1$ is 13, so we bring the parameters to Eq. (3) and get the new length of $Seq1$ as: $NL\_SSeq1 = 5 + Round(log_2(13 - 5)) = 5 + 3 = 8$. In this case, the length of this subsequence has changed from 13 to 8, which is not an obvious reduction. However, if the REPEAT is a big number like 1000, the new length of the subsequence will be $NL\_SSeq = 5 + Round(log_2(1000 - 5)) = 5 + 10 = 15$ with a huge amount of reduction.

3.3 The Use and Improvement of Malheur

CBM uses Malheur to perform clustering and classification analysis. Malheur can embed the byte-based sequential data in a high-dimensional vector space using n-gram algorithm, and then extract the prototypes for clustering and classification. Konrad et al. [3] has introduced brief details of the algorithms related to Malheur. Here we focus on the modification of Malheur. CBM has made two changes in Malheur. First, to accommodate the features of multi-byte, CBM improves the function in Malheur to parse the contents from sequential data. Second, in order to use historical data, CBM adds database function to Malheur for analysis upon global malware reports. Once the database system is set up, Malheur can analyze single behavior based on historic data. Limited to the space, we will talk about this part in detail in our public report.

4 Empirical Evaluation

We carried out a systematic evaluation of CBM. For this evaluation, we consider a total number of 3131 malware binaries obtained from the website http://pi1.informatik.uni-mannheim.de/Malheur/. The malware binaries have been collected over a period of three years via various sources, including honeynets, spamtraps, anti-malware vendors and security researchers. Those binaries have also been assigned a known class of malware by the majority of six-independent anti-virus products.

In the following experiments, the 3131 malware binaries are executed and monitored using Cuckoo, CuckooEx[6] individually. We compare CBM to the state-of-the-art analysis framework CMM(*CWSandbox/MIST/Malheur*) throughout our experiments.

In the following experiments, we will evaluate clustering results of CBM. To assess the performance of clustering, we employ the evaluation metrics of *precision* and *recall* [3]. The precision P reflects how well individual clusters agree with malware classes and the *recall* R measures to what 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:

$$P = \frac{1}{n}\sum_{c \in C} \#c \qquad and \qquad R = \frac{1}{n}\sum_{y \in Y} \#y \tag{4}$$

where $\#c$ is the largest number of reports in cluster c sharing the same class and $\#y$ the largest number of reports labelled y within one cluster. Consequently, the goal is to seek an analysis setup which maximizes precision and recall. An aggregated performance score is adopted for our evaluation, denoted as *F-measure*, which combines precision and recall. A perfect discovery of classes yields $F = 1$, while either a low precision or recall results in a lower *F-measure*.

$$F = \frac{2 \cdot P \cdot R}{P + R} \tag{5}$$

Experiment 1: Comparisons of CBM and CMM

Results for the evaluation of clustering are presented in Fig. 3, with CBM using Cuckoo yields an best F-measure 88.4% corresponding to a discovery of 28 known malware classes in the malware data set while using CuckooEx yields a best F-measure 90.9% corresponding to a discovery of 25 known malware classes very close to the real 24 classes. Compared with CBM, CMM performs better with a best F-measure 95.0% corresponding to a discovery of 24 malware classes. But we find that CBM using CuckooEx can achieve a better clustering performance even

[6] In the following experiments,we use *CuckooEx* to represent the improved *Cuckoo*.

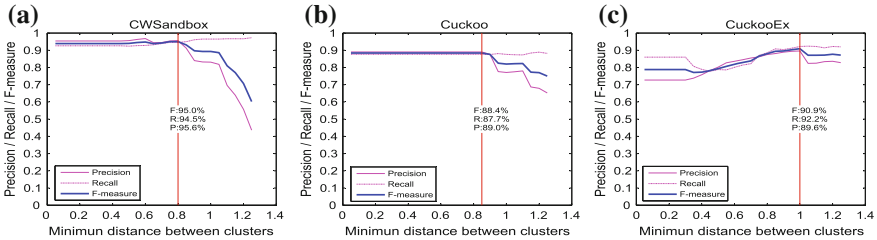

Fig. 3 Clustering result

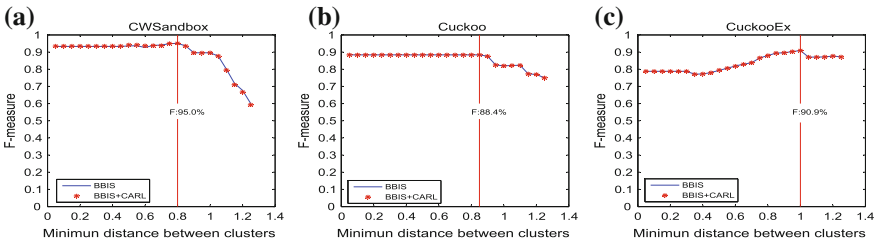

Fig. 4 Comparisons of BBIS and (BBIS + CARL) for clustering

the minimum distance between clusters is bigger than 1. Overall, CBM can achieve a competitive performance compared with CMM and it is more robust.

Experiment 2: Comparisons of BBIS and (BBIS + CARL) for clustering

In this experiment we evaluate the performance of clustering by using BBIS with or without CARL algorithm. Figure. 4 shows that all three systems have achieved nearly the same results by using CARL to compress the behavior reports. It seems that CARL is an effective way to reduce the computation cost by reducing the size of reports while keeping the performance of clustering.

Experiment 3: Time consumption

From the experiments, CBM uses 8.9 s per clustering on average while 11.6 s for CMM, which demonstrates that CBM is more efficient then CMM at time consumption.

Experiment 4: The storage size of Reports

From the statistic of the storage size of reports from above experiments, we get that BBIS can approximately reduce the MIST reports size to a 15% proportion and CARL algorithm can further reduce the size by half, which demonstrates that BBIS can reduce the size of behavior reports to a large extent, which is a huge advantage over MIST in terms of storage and computation. Moreover, CARL is useful to further reduce the size while keeping the inherent features of reports as testified by the clustering results illustrating in experiments 2.

5 Conclusion

In this article, we have introduced a framework named CBM, which utilizes and connects the open-source software tools to construct an automated malware analysis system. In CBM, we use Cuckoo to extract the API call sequences obtained from monitoring malware dynamic execution and create a new representation method of malware behaviors called BBIS to convert the API calls to byte-based sequential data. We have introduced how to improve Cuckoo's monitoring capability by appending new API hooks. The CARL algorithm proposed for reducing the high repeatability in API call sequences can effectively reduce the computation cost without a significant loss in performance. Serious experiments demonstrate that our framework is a competitive alternate to the state-of-the-art analysis framework CMM and easy to be realized. However, the main advantage of CBM is its ability in supporting local deployment. We hope CBM can replace the non-open source online services in large-scale malware dynamic analysis. CBM needs to be improved in several aspects including the monitoring range, the stability to various malware, and so on.

Acknowledgments This work was supported by NSFC under grants No. 61170286 and No.61202486.

References

1. Willems C, Holz T, Freiling F (2007) Toward automated dynamic malware analysis using cwsandbox. IEEE Security and Privacy, IEEE Computer Society, pp 32–39
2. Bayer U, Moser A, Kruegel C, Kirda E (2006) Dynamic analysis of malicious code. J Comp Virol, Springer, 2(1): 67–77
3. Rieck K, Trinius P, Willems C, Holz T (2011) Automatic analysis of malware behavior using machine learning. J Comp Virol, IOS Press, 19(4): 639–668
4. Lo RW, Levitt KN, Olsson RA (1995) MCF: A malicious code filter. Computers and Security, Elsevier, 14(6): 541–566
5. Christodorescu M, Jha S (2006) Static analysis of executables to detect malicious patterns. DTIC Document
6. Preda MD, Christodorescu M, Jha S, Debray S (2007) A semantics-based approach to malware detection. ACM SIGPLAN Notices, ACM, vol 24. pp 377–388
7. Popov IV, Debray SK, Andrews GR (2007) Binary obfuscation using signals. Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, USENIX Association, pp 19
8. Ferrie P (2009) Anti-unpacker tricks 2 part seven. June
9. Martignoni L, Christodorescu M, Jha S (2007) Omniunpack: Fast, generic, and safe unpacking of malware. Computer Security Application Conference, 2007. ACSAC 2007. Twenty-Third Annual, IEEE, pp 431–441
10. Sharif M, Lanzi A, Giffin J, Lee W (2009) Automatic reverse engineering of malwar emulators. Security and Privacy, 2009 30th IEE Symposium on, IEEE, pp 94–109
11. Song, D, Brumley D, Yin H, Caballero J, Jager I, Kang M, Liang Z, Newsome J, Poosankam P, Saxena P (2008) BitBlaze: A new approach to compute security via binary analysis. Information Systems Security, Springer, pp 1–25

12. Trinius P, Willems C, Holz T, Rieck K (2009) A malware instruction set for behavior-based analysis. Technical Report TR-2009-005, University of Mannheim
13. Alazab M, Venkataraman S, Watters P (2010) Towards understanding malware behaviour by the extraction of API calls. Second cybercrime and trustworthy computing workshop, pp 52–59
14. Dong X, Zhao Y, Yu X (2012) A Bot Detection Method Based on Analysis of API Invocation. Recent Advances in Computer Science and Information Engineering, Springer, pp 603–608