MalCore: Toward a Practical Malware Identification System Enhanced with Manycore Technology

Taegyu Kim[1(✉)] and Ki Woong Park[2]

[1] School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA
tgkim@purdue.edu
[2] Daejeon University, Daejeon, Republic of Korea
woongbak@dju.kr

Abstract. Many conventional control flow matching methods work well, but lead to obstructive latency for the operations as the number of malware variants has soared. Even though many researchers have proposed control flow matching methods, there is still a trade-off between accuracy and performance. To alleviate this trade-off, we present a system called MalCore, which is comprised of the following three novel mechanisms, each of which aims to provide a practical malware identification system: I-Filter for identical structured control flow string matching, table division to exclude unnecessary comparisons with some malware, and cognitive resource allocation for efficient parallelism. Our performance evaluation shows that the total performance improvement is 280.9 times. This work was undertaken on a real manycore computing platform called MN-MATE.

1 Introduction

Antivirus vendors have detected malware through signature-based detection. However, such malware detection has become ineffective as malware variant generation tools have been available [15]. Due to the availability of such tools, malware authors can easily create malware variants that are slight modifications of existing malware. Additionally, the number of new malware variants has increased at an exploding pace.

Malware classification and identification is therefore of immense importance to enable assessing damages after detection, and reinforcing disinfection mechanisms [14]. In addition, understudying groups and classes of malware would enable malware researchers to concentrate their efforts on specific sets of families to understand their intrinsic characteristics, and to develop better detection and mitigation tools for them.

As a remedy to this problem, Malwise [8] has proposed structured control flow string (SCFS) matching methods at a procedure level that classify malware variants by measuring similarities in existing malware samples. Their approaches are

O. Camp et al. (Eds.): ICISSP 2015, CCIS 576, pp. 31–48, 2015.
DOI: 10.1007/978-3-319-27668-7_3

effective in detecting malware variants because, unlike signatures, control flows of malware variants are much less changeable. Its authors have proposed two control flow matching methods. One of them is exact matching and the other one is approximate matching. However, there is a trade-off between the two methods. Exact matching is faster but less accurate than approximate matching because it is only necessary to check whether each control flow is identical. On the other hand, approximate matching is more accurate but has lower performance since this method compares all parts of each control flow in a fine-grained manner. In addition, both neither method considers parallelism even though many resources are available in recent high performance computers.

This study is an extension of our previous work [10,11], in which we focused on the conceptual design and implementation such as an acceleration of the approximate matching method and efficient parallelism. Our objective in this study however, is to achieve high accuracy and performance and apply parallelism, and integrate the overall components into MN-MATE platform, a novel resource management techniques for virtualization [16]. Consequently, MalCore acts as the key primitive for a practical malware identification system enhanced with manycore technology. As a result, we gained on average 280.9 times total performance improvements in our experiments.

The remainder of the paper is organized as follows: In Sect. 2, we review related works and analyze existing malware classification and identification systems. In Sect. 3, we present our motivation of this work. In Sect. 4, we illustrate the overall system design and components of the proposed system. In Sect. 5, we evaluate the performance of the proposed system. Finally, in Sect. 6, we present our conclusions.

2 Related Work

Malware identification through matching control flows has been proposed in order to solve the problem of not being able to detect malware variants. Of various analysis approaches, one of them is to match SCFSs of binaries [8]. The authors represented procedure-level control flows in a SCFS form and measure similarities to existing malware samples in databases. If the most similar malware is larger than the threshold value, the input binary is considered malicious. They suggested two matching methods: exact matching and approximate matching. However, exact matching has a lower accuracy, and approximate matching has a lower performance.

To increase the performance of string matching, bioinformatic researchers developed the fast string matching method to find identical strings to which proteins were converted. However, the conventional character-to-character (C2C) string matching is time-consuming due to large string sizes. In order to resolve this performance bottleneck, they proposed short filtering [13]. According to this algorithm, if a string shares a certain number of substrings, the pair is considered identical. Consequently, they could skip many character-to-character comparisons in the middle of matching processes. However, this approach is

not applicable to matching malware programs because patterns of substrings in SCFSs depend on variable authors' coding styles.

From the view point of parallelism and resource management, there have been several approaches for large workload distributions in scientific calculation, such as matrix calculation [9]. It distributes workloads to multiple virtual machines (VM) which is useful to fully utilize computing resources. However, we distribute virtual CPUs (VCPU) instead of workloads. In an approach similar to our work, some researchers have proposed dynamic resource allocation [12]. These studies model workloads using resource usages, such as CPU usage, memory usage and so on. Our work utilizes an easier modeling variable, Q, which indicates how many workloads are distributed as well as CPU usage.

3 Motivation

In an attempt to achieve a practical malware identification system, we thoroughly analyze conventional operational routines for similarity measurement and identify three critical mismatches in the conventional routines. They are summarized as follows.

- **Inefficient SCFS Matching:** Before measuring set similarities, we need to measure string-to-string (S2S) similarities through C2C matching based on the edit distance algorithm [7]. However, this procedure is the main bottleneck of similarity measurements because C2C matching requires many computations. To resolve such a performance bottleneck, we found that there was a potential for improvement in matching identical SCFSs. The purpose of C2C matching is to find similar strings and measure how much similar two SCFSs are. When we determine whether SCFSs are identical, it is necessary to know whether they are identical to each other but unnecessary to measure how much similar they are because the similarity between matched identical SCFSs is 100 %. This approach can be frequently applied to C2C matching because malware variants in the same family share many identical SCFSs.
- **Brute-force Malware Comparison:** In the similarity measurement procedure, we need to match SCFSs of an input binary with all pre-analyzed malware samples in databases. However, the large number of malware samples causes the performance bottleneck. In order to reduce such comparison overhead, a rule to exclude malware samples that cannot be similar to an input binary before starting similarity measurements is necessary. Without such an exclusion rule, it is necessary to compare all malware samples in databases. This is because they are possibly similar.
- **Non-parallelized Malware Analysis:** According to the AV-TEST [1], malware authors created about 140 million new malware samples in 2014, and 88 % of them are malware variants [8], but it is hard to analyze all malware variants with the optimized methods because of the significant number of malware samples. However, we can utilize many resources in high performance computers to gain higher throughput. One way to use all resources for this

purpose is parallelization of analysis which was not considered in the previous work [11]. Even though this is a valid approach to increasing total analysis throughput, this trial can waste resources without proper management. Therefore, we need find a way to efficiently use such resources for optimized parallelism.

4 Design of MalCore Identification System

The design of our system is motivated by three points as follows: inefficient SCFS matching, brute-force malware comparison and non-parallelized malware analysis. In this section, we describe the overview of our system and then how to solve these problems.

4.1 Overall System Design

We implemented the malware variant identification system on MN-MATE [16]. Our malware variant identification system consists of three parts: Convertor, Analyzer and Resource Manager. Both Convertor and Analyzer work on VMs but Resource Manager works on dom0, the privileged VM that can control hypervisor [17]. We describe our architecture in Fig. 1 and the flow chart in Fig. 2.

- **Convertor:** Convertor is responsible for converting input binaries into SCFSs. This conversion task is composed of unpacking, decompiling and structuring. Unpacking is for extracting hidden malicious codes, decompiling is to convert binary codes into high-level codes like C, and structuring is to represent branch instructions in decompiled codes into SCFSs. We describe the structuring rule in Fig. 3a and the example of SCFS conversion in Fig. 3b. After finishing the conversion process, converted SCFSs are sent to Analyzer.
- **Analyzer:** Analyzer plays a role in deciding whether input binaries are malicious through measuring set similarities with existing malware samples in databases. Analyzer uses SCFSs obtained from Convertor for similarity measurements. We designed Analyzer with three components: malware databases, I-Filter and C2C matcher. Malware databases consist of multiple tables, and we store pre-converted SCFSs and their metadata such as hash values in these tables. The role of I-Filter is to match identical SCFSs of an input binary with those in the databases. C2C matcher is responsible for measuring similarities of the remaining SCFSs that are not matched through I-Filter [11]. For malware databases, we use two types of databases: the global database and local database. We used the global database to match identical SCFSs through I-Filter. This database consists of several tables covering malware samples in certain ranges of the total number of SCFCs. Each table stores SCFSs and metadata of covered malware families, variant names, hash values and their total numbers of SCFCs of malware samples. The local database consists of multiple tables and stores the same data but only that of malware samples in one malware family. We store indexed hash values in both types of databases to use I-Filter more efficiently.

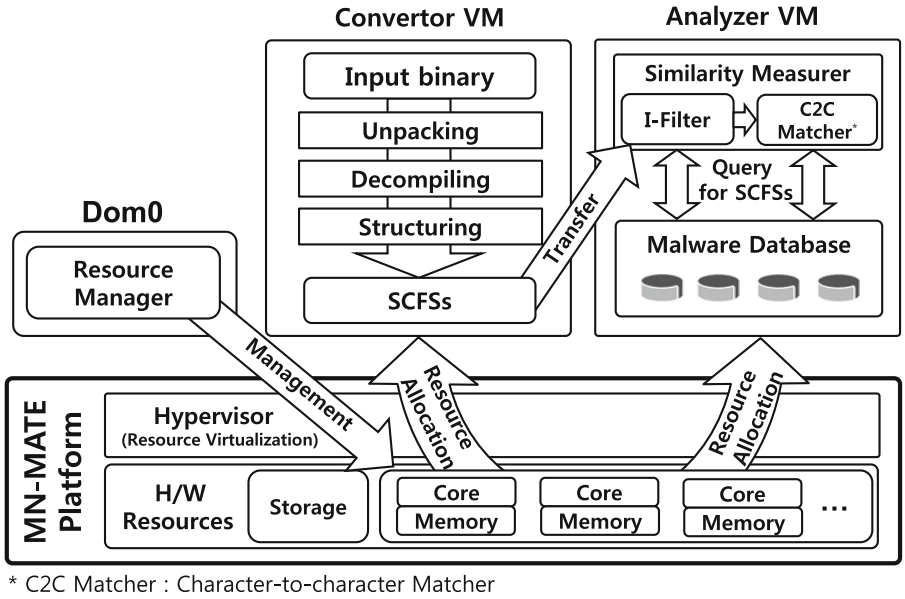


Fig. 1. Overview of our system on MN-MATE.

– **Resource Manager:** Each VM is responsible for conversion and analysis. However, their workloads vary according to the situation in which Analyzer does not work due to there being no SCFSs or Convertor generates so many SCFSs that Analyzer cannot process all of them. To prevent such a waste of resources, Resource Manager allocates a proper amount of resources to each VM. Therefore, we can conserve resources through manipulation of the processing speed of each VM through resource allocation. Also, we utilize VCPU pinning to dedicated nodes to enhance memory access performance through local memory access instead of remote memory access.

4.2 I-Filter

We pointed out that S2S matching for identical SCFSs is inefficient despite the high share ratio of identical SCFSs. In order to enhance the performance of S2S matching, we use I-Filter [11] to match identical SCFSs through hash value comparisons and then match only remaining SCFSs through edit distance algorithm. We use CRC-64 for generation of hash values.

Efficiency of I-Filter can be seen through comparisons between time complexities of both methods. In previous approach, all matching is done through edit distance algorithm. Its time complexity between two SCFSs is $O(\boldsymbol{mn})$. Both m and n are lengths of SCFSs, and their minimum value is 10 [7]. In order to accelerate matching for SCFSs, all SCFSs are stored in the BK-tree [6] indexed malware database in the previous work [8]. However, it is time-consuming to find

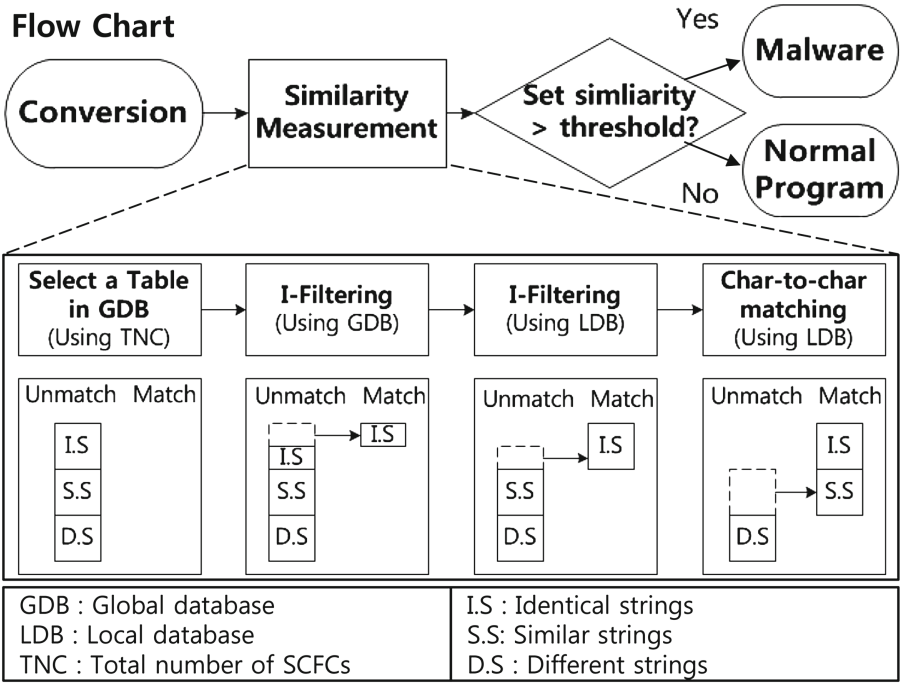
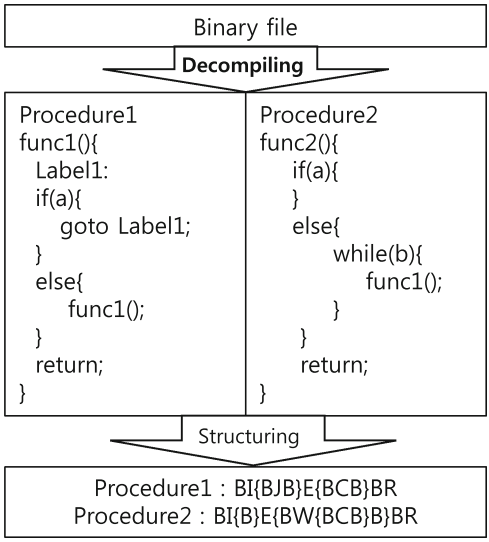


Fig. 2. Whole analysis flow chart.

Control flow	Result
Basic Block	B
Procedure Call	C
Break	K
Continue	T
Return	R
Goto	J
If	I
Else	E
Open Brace	{
Close Brace	}
Branch Condition: AND	A
Branch Condition: OR	O
Loop: Pre-tested	W
Loop: Post-tested	D
Loop: Infinite	F

(a) Grammar for structuring



(b) Example of conversion

Fig. 3. Conversion rule and example [7].

valid SCFSs because each character of SCFSs should be checked. On the other hand, the searching time complexity of I-Filtering is $O(log\ \boldsymbol{s})$ where s is the number of SCFSs. For S2S matching for each SCFS, each matching is processed through hash value matching whose time complexity is $O(\boldsymbol{1})$ without character-to-character comparison. In addition, the number of comparisons is $O(log\ \boldsymbol{s})$ because we stored the hash values in B-tree. Therefore, we can induce the time complexity for finding one SCFS is $O(log\ \boldsymbol{s})$ from $O(\mathbf{1})O(log\ \boldsymbol{s}) = O(log\ \boldsymbol{s})$. However, checks for identicalness are required to prevent hash collisions for all SCFSs whose hash values are identical. The time complexity for hash collision checking is $O(\boldsymbol{m})$ which is proportional to the lower length m in a string pair.

4.3 Table Division

When we match SCFSs in the global database, unnecessary comparisons with malware samples that cannot be similar cause redundant overhead costs. In order to reduce such costs, we make a rule for excluding malware samples that cannot be similar before starting similarity measurements. Because the set similarity, the final similarity result [7], is directly related to the total number of SCFCs, we can exclude such malware samples through dividing tables in the global database. Therefore, we can exclude many malware candidates through comparisons of the total number of SCFCs of an input binary. We describe such cases in Fig. 4. In the first case, malware x can be similar to malware y if all their SCFSs are matched. In the second case, malware x and y however are definitely dissimilar even if malware x and y consist of only identical SCFSs. Thus, malware x is eligible for comparison but malware y is ineligible according to the malware exclusion policy.

In order to apply the above policy, we divide the table of the global database into smaller tables according to the total number of SCFCs. Because our divided tables store only possibly similar malware samples, it is possible to compare a smaller number of entries. We describe the example of table division in Fig. 5. Before we analyze input SCFSs, we select one of tables in the global database based on the total number of SCFCs of each input binary. Although this selection may result in a small cost, we can gain greater performance benefits from it. Since each malware has on average 94 SCFSs in our malware samples, we can avoid comparisons of 94 SCFSs of the input binary with those of malware samples that cannot be similar in databases through one table selection. Through table division, we can reduce comparisons due to reduced depths of B-trees and I/O requests for loading unnecessary malware data from databases.

However, table division should guarantee that all possibly similar malware samples are in each divided table. This guarantee is based on the set similarity threshold value, 0.6 [7]. As described in Fig. 5, if the selected table covers malware D, E and F with the total number of SCFCs from 55 to 80, this table should have malware samples with the total number of SCFCs from 55 by 0.6 to 80 by 1.67. In such cases, we call the total number of SCFCs from 55 to 80 the cover range and from 33 to 55 and from 80 to 134 the guarantee range. Malware samples covered by guarantee range should be included in the divided

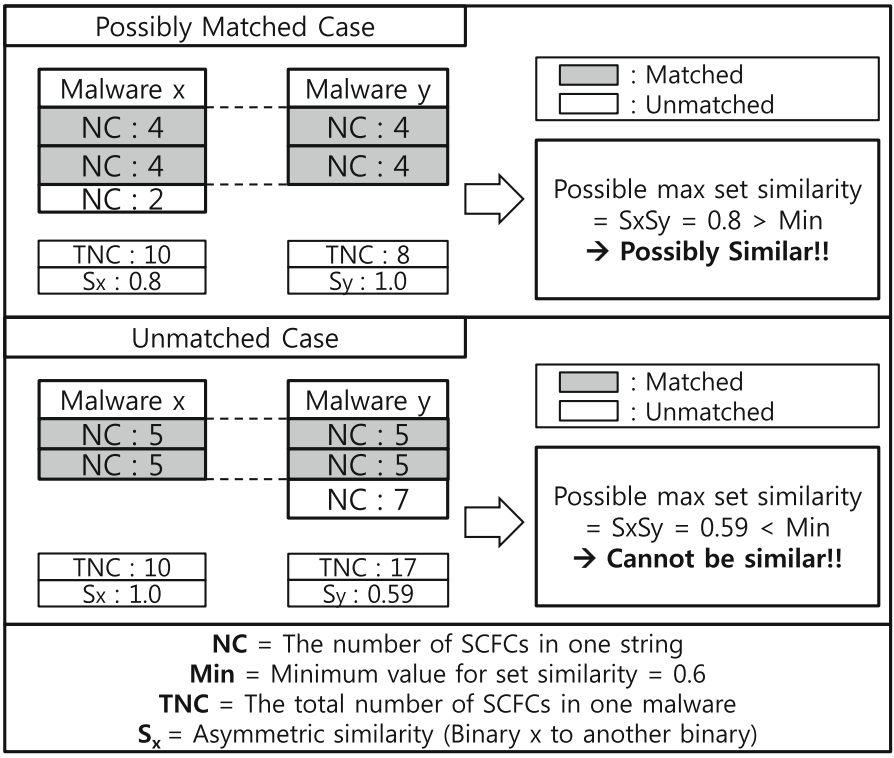


Fig. 4. Max similarity according to the total number of SCFCs.

tables. Otherwise, comparison only with a table cannot guarantee that all possibly similar malware samples are stored. From the perspective of performance, we need to divide a table into smaller tables because the number of hash entries can change according to sizes of cover ranges. However, excessive table division causes storage redundancy for guarantee ranges. Furthermore, they can be much larger than cover ranges if the cover ranges are too small. Therefore, we set the cover range from one of 3,000, 10,000 and 20,000 and dynamically divide tables to avoid excessive storage redundancy while maintaining a certain level of performance. If the difference in the number of hash entries is smaller than 110 % of the total number of hash entries with a larger cover range, we set the larger cover range since this 10 % difference does not cause meaningful performance degradation. After applying our table division policy, storage redundancy is not large compared to storage capacity of HDD. As a result of table division, depths are reduced from 50 % to 80 % and their average depth is 33 %.

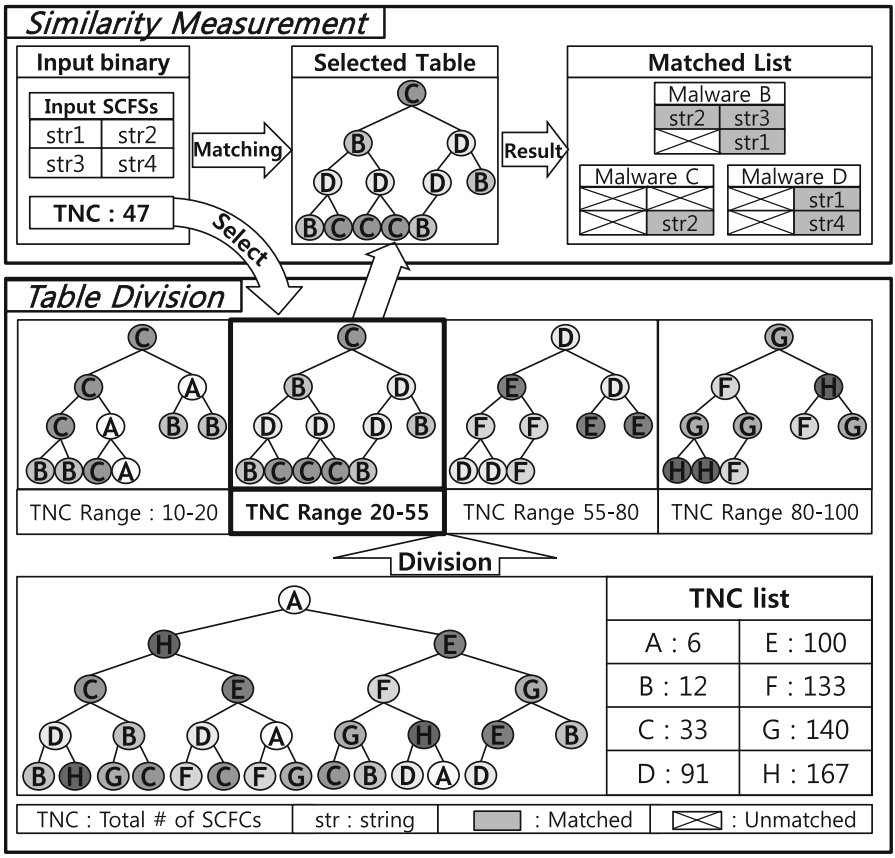


Fig. 5. An example of table division application.

4.4 Dynamic Resource Allocation

Our system consists of two main processing parts, Convertor and Analyzer. We describe this system in Fig. 6. There are two problems causing unbalanced core allocation. First one is processing speeds vary according to which binaries are analyzed, and the other one is that the required number of cores is not a natural number unlike the actual number of allocated cores. Therefore, naive static core allocation cause waste of CPU power. To prevent such a situation, we dynamically allocate cores according to the number of converted binaries. We define the value Q as the representation of the number of such binaries for dynamic core allocation modeling. To figure out whether core allocation is appropriate, we also define *Low* and *High* which are threshold values of Q which is periodically checked by Resource Manager. We categorize four states to determine whether we need core reallocation according to Q.

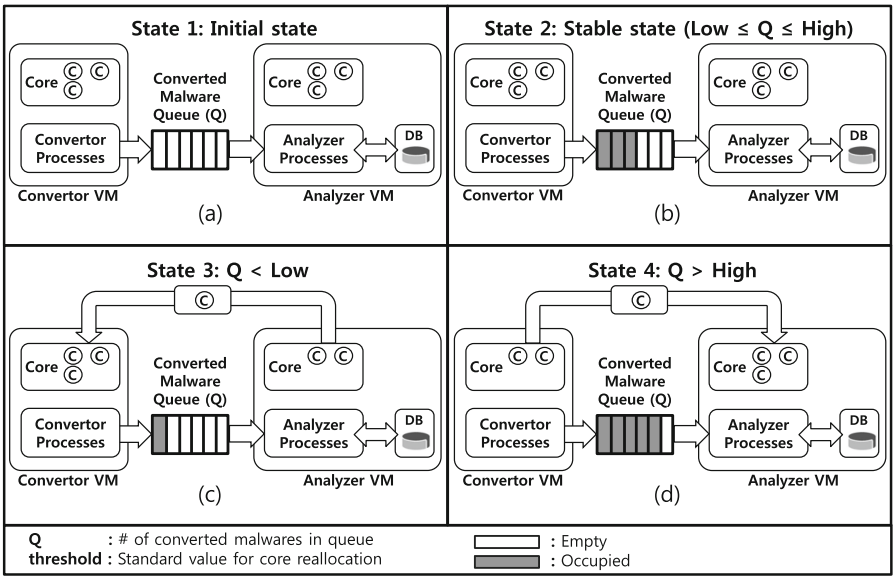


Fig. 6. Dynamic core allocation.

In detail, Fig. 6a refers to *State 1*, the initial core allocation state. In this state, we initially allocate cores using average core allocation of the previous result. If there is no such result, we can arbitrarily allocate cores. Even if it is wrong allocation, Resource Manger will appropriately allocate cores. When Q reaches the average value of *Low* and *High*, the current state will be *State 2*. In Fig. 6b, we illustrate *State 2*. In this case, we consider cores are properly allocated. Because the case where Q is higher than or equal to *Low* and lower than or equal to *High* indicates that jobs are appropriately assigned to Convertor and Analyzer. Consequently, Resource Manger does not reallocate cores. We describe *State 3* where Q is lower than *Low* in Fig. 6c. In this state, we reallocate cores to Convertor since Analyzer occupies too many cores compared to workloads. However, it is possible that frequent and unnecessary core reallocation can occur with this policy. To prevent such situation, Resource Manager temporarily store Q in *Low* when core reallocation occurs. In the next period, Resource Manger continuously control the number of cores with a new *Low* until the state is changed to *State 2*. On the other hand, *State 4* is the case where Q is lower than *Low* as described in Fig. 6d. In this situation, we reallocate cores since Convertor processed too many binaries. Similarly, Resource Manager temporarily store Q in *High*, when core reallocation occurs. Then, Resource Manger continuously maintains a new *High* until the state is changed to *State 2*.

4.5 Implementation

- **Convertor:** Convertor is responsible for unpacking, decompiling and structuring. For unpacking, we use the unpacking function of UPX [5] because malware authors widely use it to pack malware programs. After the unpacking process, we decompile unpacked binaries using REC decompiler [4]. Then, we convert decompiled binaries into SCFSs using the rule in Fig. 3a.
- **Analyzer:** This module measures similarities between input binaries and malware samples. The matching process starts with the global database. We first select a table in the global database based on the total number of input SCFCs. With the selected global database, we match only identical SCFSs through I-Filter. In this step, we process near unique strings first and then match duplicated SCFSs. Because such near unique strings are not normally shared, they are useful in determining a specific malware candidate. If the set similarity exceeds the set similarity threshold T, matching processes are performed on the local database whose tables cover respective malware families. If the highest set similarity is lower than min_t even after all SCFSs are processed, we consider this binary unmalicious. Otherwise, the top five candidate malware samples with similarities higher than the others are selected. With the local database, we apply I-Filter first because all SCFSs could not be matched through I-Filtering. Then, the similarity of the remaining SCFSs is measured through C2C matching. We consider the target binary is malicious if its similarity is larger than T.

  In the above procedure, we define several parameters. We choose min_t as 0.1 and determine the number of candidates as 5 based on our experiments. We use 0.9 for T and 0.6 for t as used in related work [7]. However, we can change these values according to additional experiments.
- **Resource Manager:** In Resource Manager, we use the Q variable to predict workloads between Convertor and Analyzer. For actual parameters, we currently define *High* as 30 and *Low* as 10. With these values, there was no waste of resources, such as too many converted SCFSs or no SCFSs for similarity measurements, during our experiments. If we increase this value, the occurrence wasted resources will be reduced. In this case, even though more binaries will not be analyzed, its effect is negligible in the long run. However, we should consider that the most important factor for threshold values of Q is whether their values can guarantee avoidance of unbalanced resource distribution. We can change threshold values considering such conditions. Also, we define the period of checking Q as 0.3 seconds. Although it can be changed, we should choose checking period while considering core reallocation frequency. With a small value, it cause performance degradation because core reallocation flushes caches. With a large value, it cannot balance core distribution.

  When we reallocate cores, we reassign VCPUs to allocate cores to VMs. For efficient resource utilization, we assign memory on one node to each VM and set the VCPU affinity to the node in order to avoid remote memory access. Moreover, Resource Manager reallocates VCPUs to Analyzer and operates one more analyzer process according to CPU usage.

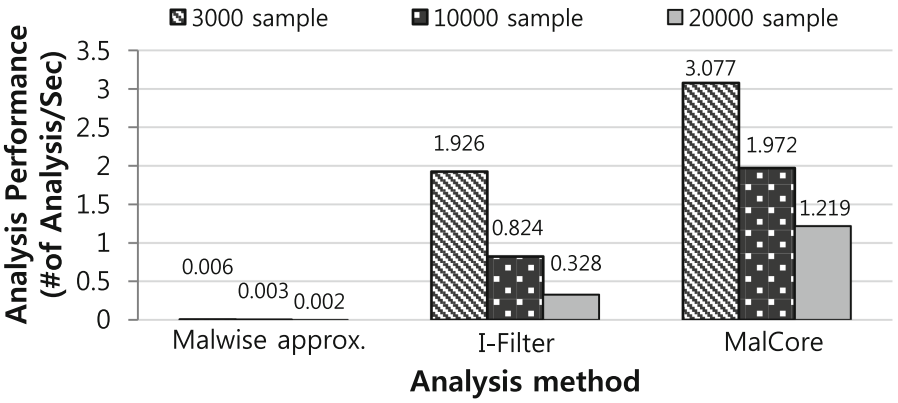


Fig. 7. Analyzer performance with a single core.

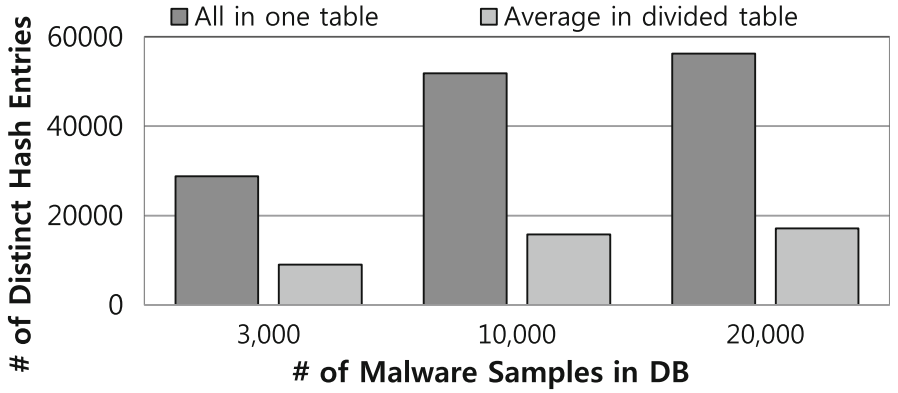


Fig. 8. The number of hash entries in each database.

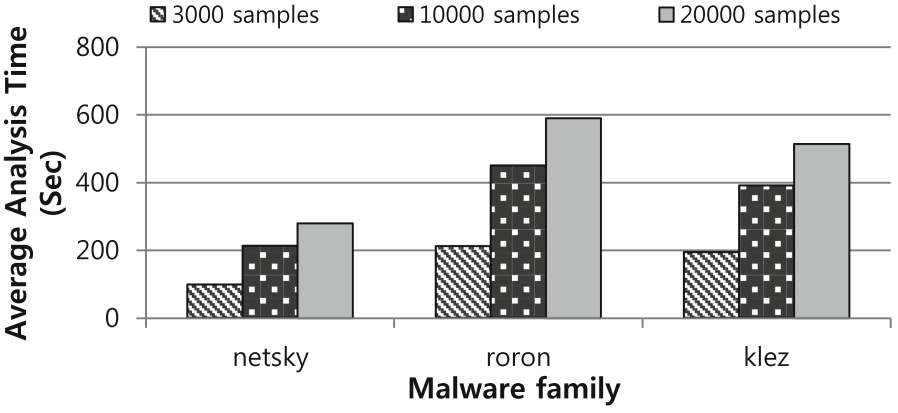


Fig. 9. Analyzer performance with approximate matching of Malwise.

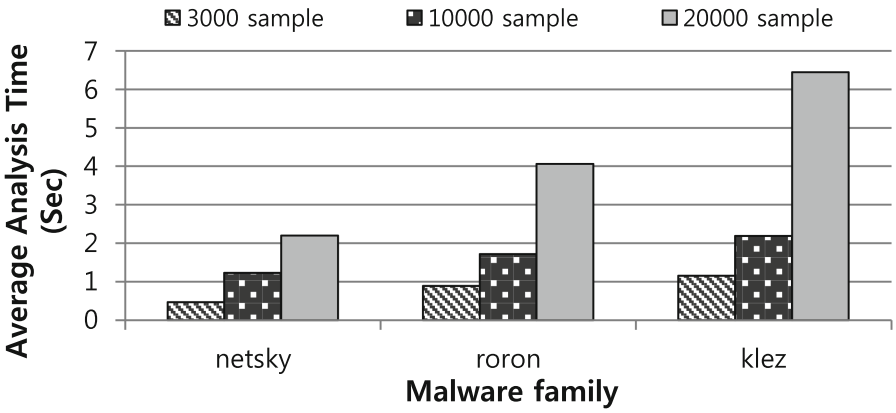


Fig. 10. Analyzer performance with approximate matching and I-Filter.

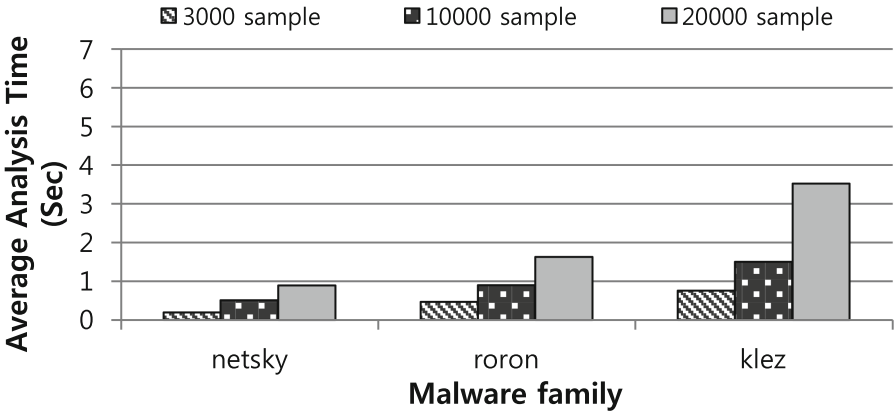


Fig. 11. Analyzer performance with MalCore.

5 Performance Evaluation

This section presents module performance improvements, total performance improvements, similarities between malware variants and validation of normal program threshold. The experimental environment consists of the AMD Opteron Processor 6282SE 64 core 2.6 Ghz, 128 GB RAM, SAS 10kbytes HDD, Cent OS 6.4 64 bit with Kernel 3.8.2 version, MN-MATE [16] and MySQL 14.14 for the database, but we utilized 16 cores and 16 GB RAM. We implemented our databases on MyISAM [2] which is a type of disk-based database. On the other hand, Malwise [8] consists of BK-tree [6] indexed memory databases. In all experiments, MalCore refers to application of I-Filter, Table division and Dynamic Resource Allocation. But Dynamic Resource Allocation is not applied to single process experiments. Also, MN-MATE means that we experimented on MN-MATE. Without MN-MATE, we experimented on Xen 4.2.1. Finally, we used

3,000 malware samples [3] and generated additional malware variants using the code mutation tool.

5.1 Module Performance Improvements

In this section, we evaluate the performance of each module in malware variant identification systems. This evaluation does not reflect the effect of dynamic core allocation because it focuses on each module with a single core, but, we consider the effect of VCPU pinning. First, we compare the performance of analyzer in Fig. 7. For the Analyzer performance, the performance improvements between approximate matching of Malwise [8] and application of all of our techniques on MN-MATE are from 512 to 657 times. For comparison between I-Filter application and MalCore, the performance improvements are from 60 % to 272 % and the improvement increases as the number of malware samples increases. These improvements are largely from table division because table division reduces on average 33 % numbers of table entries for identical SCFSs matching procedure as described in Fig. 8.

We describe the performance improvements in Figs. 9, 10 and 11. We can discern several performance trends in these figures. The trends of performance improvements are different from the malware families. This difference results mainly from the number of SCFCs in each malware and each string. For Malwise, the trend of performance difference between malware families results from the fact that computation time depends on string matching measurements. On the other hand, our approach relies on the number of hash entries in databases.

Finally, we describe the performance of Convertor. With a single core, Convertor can convert on average 0.365 input malware binaries per second. As the speed of Analyzer increases, the performance of Convertor creates a larger bottleneck. Moreover, its performance trend is different from Analyzer because it depends on how many instructions; not only branch instructions but also other types of instructions, variables and other factors are included. This different trend of processing speed causes unbalanced workload distributions even with perfect static core allocation. This is why core allocation, a part of dynamic resource allocation, is necessary.

5.2 Total Performance Improvements

In this section, we evaluate the total performance of the malware variant identification systems. We applied our Convertor to Malwise [8] because Malwise does not have dynamic resource allocation functions. We randomly choose malware samples for our experiments and show the performance evaluation in Figs. 12 and 13.

Performance improvements of MalCore with MN-MATE are on average 280.9 times compared to approximate matching proposed in Malwise and 71 % improvements compared to only I-Filter application. Although improvements are mostly from I-Filter for matching identical SCFSs and table division, the performance gain is limited by Convertor performance and a waste of resources

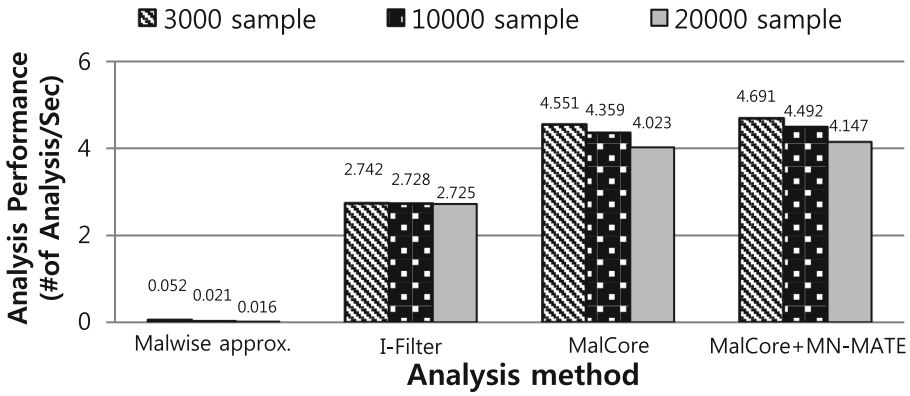


Fig. 12. Total performance (50 % of resource to Convertor and 50 % of resource to Analyzer).

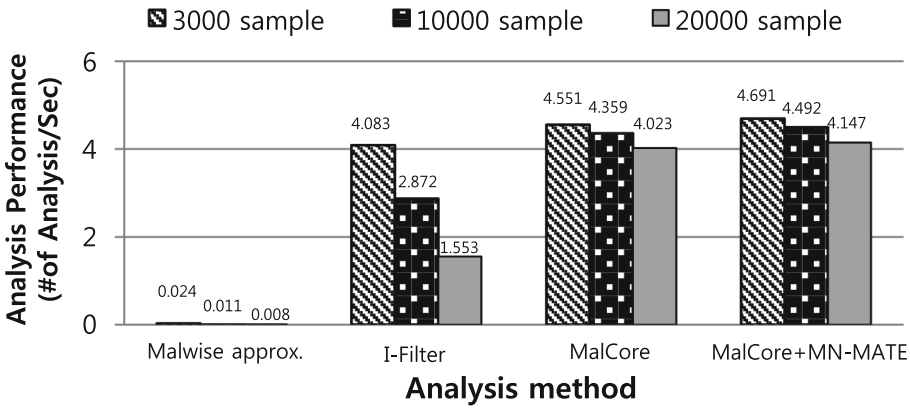


Fig. 13. Total performance (75 % of resource to Convertor and 25 % of resource to Analyzer).

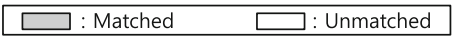
due to unbalanced resource distribution. However, our system can balance the performance of each VM with our dynamic allocation.

5.3 Similarity of Malware Variants

In this experiment, we measure similarities using our approach. To determine whether input binaries were malicious, we used the same set similarity threshold value, 0.6, used in the related work [8]. Table 1 shows similarities between malware variants in *Klez*, *Roron* and *Netsky* malware families.

According to our experiments, *Klez*, *Roron* and *Netsky* had 43, 62, 66 percent matching rates. As the matching rates increase, new malware variants will more probably be classified. However, we still can classify malware variants with low

Table 1. Similarities between malware variants.



	12	25	35	37	ao	b39	b50
12		0.5	0.53	0.53	0.66	0.5	0.39
25	0.5		0.84	0.89	0.56	0.93	0.63
35	0.53	0.84		0.94	0.64	0.9	0.63
37	0.53	0.89	0.94		0.6	0.95	0.63
ao	0.66	0.56	0.64	0.6		0.57	0.43
b39	0.5	0.93	0.9	0.95	0.57		0.63
b50	0.39	0.63	0.63	0.63	0.43	0.63	

Roron

	a	b	c	d	e	g	h	i
a		0.73	0.91	0.65	0.5	0.49	0.5	0.45
b	0.73		0.8	0.87	0.54	0.53	0.54	0.52
c	0.91	0.8		0.7	0.5	0.49	0.5	0.45
d	0.65	0.87	0.7		0.52	0.5	0.52	0.51
e	0.5	0.54	0.5	0.52		0.94	0.91	0.91
g	0.49	0.54	0.49	0.5	0.94		0.93	0.92
h	0.5	0.54	0.5	0.52	0.91	0.93		0.99
i	0.45	0.52	0.45	0.51	0.91	0.92	0.99	

Klez

	ab	b	c	k	p	u	w	x
ab		0.74	0.84	0.91	0.64	0.75	0.7	0.6
b	0.74		0.76	0.72	0.54	0.58	0.55	0.53
c	0.84	0.76		0.86	0.6	0.67	0.63	0.59
k	0.91	0.72	0.86		0.61	0.7	0.66	0.58
p	0.64	0.54	0.6	0.61		0.68	0.6	0.88
u	0.75	0.58	0.67	0.7	0.68		0.85	0.64
w	0.7	0.55	0.63	0.66	0.6	0.85		0.57
x	0.6	0.53	0.59	0.58	0.88	0.64	0.57	

Netsky

matching rates. For instance, the matching rates of the *Klez* family were only 43 percent. However, let us suppose *a*, *b*, *c* and *d* *Klez* variants are group *A* and the other ones, *e*, *g* and *i*, *Klez* variants, are group *B*. In this case, one malware sample from group *A* and the other one from group *B* are enough to classify all *Klez* malware variants in Table 1. But, there is more chance to classify unseen malware programs with higher matching rates.

Furthermore, we should compare our similarity results since the purpose of our work is to accelerate Malwise. However, because we use REC decompiler which is different from Malwise [8], we measure similarities in both Malwise and our approach with REC decompiler. As a result, most similarities are identical, and they are lower than 0.01, even if the similarities are different. The reason for this small difference is that we match identical SCFSs first and then similar SCFSs but Malwise matches similar SCFSs.

5.4 Validity of Normal Program Threshold

As we mentioned in implementation of Analyzer, we use a normal program threshold, 0.1. If the set similarity is lower than 0.1, we consider the input as a normal program after matching identical SCFSs with the global database. To validate our parameter, we measure similarities of 3,256 normal programs from the Windows system folders with malware samples. The result of our experiments confirm that our threshold value is valid because set similarities of only 0.0012 % of normal programs exceeded 0.1 as shown in Fig. 14.

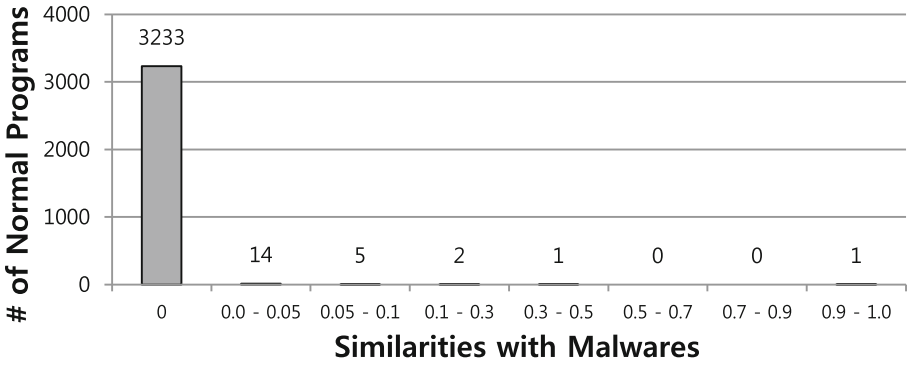


Fig. 14. Similarity between normal programs and malwares.

6 Conclusion

Many researchers have proposed many control flow matching methods. However, there is a trade-off between accuracy and performance. To solve such problems, we designed MalCore which is composed of I-Filter, table division and dynamic resource allocation and apply them incrementally on a real manycore computing platform called MN-MATE. As a result, we gained the total performance improvement of on average 280.9 times in our experiments; especially, the performance improvement of Analyzer is 593.2 times on average.

References

1. AV-TEST. http://www.av-test.org
2. MySQL reference. http://dev.mysql.com/doc/refman/5.7/en/index.html
3. Offensive computing. http://www.offensivecomputing.net
4. Reverse Engineering Compiler. http://www.backerstreet.com
5. Ultimate Packer for eXecutables. http://upx.sourceforge.net
6. Baeza-Yates, R., Navarro, G.: Fast approximate string matching in a dictionary. In: Proceedings of South America Symposium on String Processing and Information Retrieval, SPIRE 1998, pp. 14–22. IEEE (1998)
7. Cesare, S., Xiang, Y.: Classification of malware using structured control flow. In: Proceedings of Australasian Symposium on Parallel and Distributed Computing, AusPDC 2010, pp. 61–70. ACM (2010)
8. Cesare, S., Xiang, Y., Zhou, W.: Malwise–an effective and efficient classification system for packed and polymorphic malware. IEEE Trans. Comput. **62**(6), 1193–1206 (2013)
9. Gusev, M., Ristov, S.: Matrix multiplication performance analysis in virtualized shared memory multiprocessor. In: Proceedings of 35th International Convention, MIPRO 2012, pp. 251–256. IEEE (2012)
10. Kim, T., Hwang, W., Kim, C., Shin, D.J., Park, K.W. Park, K.H.: Malfinder: accelerated malware classification system through filtering on manycore system. In: Proceedings of 1st International Conference on Information Systems Security and Privacy, ICISSP 2015, pp. 1–10 (2010)

11. Kim, T., Hwang, W. Park, K.W., Park, K.H.: I-Filter: identical structured control flow string filter for accelerated malware variant classification. In: Proceedings of International Symposium on Biometrics and Security Technologies, ISBAST 2014. IEEE (2014)
12. Kundu, S., Rangaswami, R., Dutta, K., Zhao, M.: Application performance modeling in a virtualized environments. In: Proceedings of 16th International Symposium on High Performance Computer Architecture, HPCA 2010, pp. 1–10. IEEE (2010)
13. Li, W., Godzik, A.: Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. Bioinformatics **22**(13), 1658–1659 (2006)
14. Mohaisen, A., West, A.G., Mankin, A., Alrawi, O.: Chatter: Classifying malware families using system event ordering. In: Proceedings of 2nd Communications and Network Security, CNS 2014, pp. 283–291. IEEE (2014)
15. OKane, P., Sezer, S., McLaughlin, K.: Obfuscation: the hidden malware. IEEE Secur. Priv. **9**(5), 41–47 (2011)
16. Park, K.H., Hwang, W., Seok, H., Kim, C., Shin, D.J., Kim, D.J., Maeng, M.K., Kim, S.M.: MN-MATE: elastic resource management of manycores and a hybrid memory hierarchy for a cloud node. ACM J. Emerg. Technol. Comput. Syst. **12**(1), 5 (2015)
17. Paul, B., Boris, D., Keir, F., Steven, H., Tim, H., Alex, H., Rolf, N., Ian, P., Andrew, W.: Xen and the art of virtualization. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP 2003, pp. 164–177. ACM (2003)