Malfinder: Accelerated Malware Classification System through Filtering on Manycore System

Taegyu Kim[1], Woomin Hwang[1], Chulmin Kim[1], Dong-Jae Shin[1], Ki-Woong Park[2] and Kyu Ho Park[1]
[1]*Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Republic of Korea*
[2]*Daejeon University, Daejeon, Republic of Korea*

Keywords: Malware Variant Classification, Identical Structured Control Flow, Table Division, Dynamic Resource Allocation, NUMA (Non Uniform Memory Access).

Abstract: Control flow matching methods have been utilized to detect malware variants. However, as the number of malware variants has soared, it has become harder and harder to detect all malware variants while maintaining high accuracy. Even though many researchers have proposed control flow matching methods, there is still a trade-off between accuracy and performance. To solve this trade-off, we designed Malfinder, a method based on approximate matching, which is accurate but slow. To overcome its low performance, we resolve its performance bottleneck and non-parallelism on three fronts: I-Filter for identical string matching, table division to exclude unnecessary comparisons with some malware and dynamic resource allocation for efficient parallelism. Our performance evaluation shows that the total performance improvement is 280.9 times.

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 (OKane et al., 2011). Due to the availability of such tools, malware authors can easily create malware variants that are slight modifications of existing malware. In addition, the number of new malware variants has increased at an exploding pace. According to statistics of the AV-TEST[1], approximately 80 million new malware samples 88% of which are malware variants (Cesare et al., 2013) appeared in 2013, and this exploding appearance speed has continued to increase.

Due to the incapability in detecting malware variants through signature-based detection, Malwise (Cesare et al., 2013) has proposed control flow matching methods that classify malware variants by measuring similarities in existing malware samples. Their approaches are 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. Therefore, in order to achieve high accuracy and performance and apply parallelism, we chose to accelerate approximate matching on our platform, MN-MATE (Park et al., 2012), which has higher accuracy than exact matching.

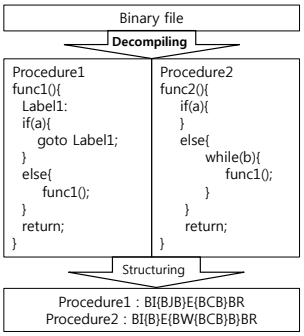
This study is an extension of our previous work, I-Filter, (Kim et al., 2014) which represented I-Filter, and focus on acceleration of the approximate matching method through fast identical procedure-level control flow string matching. Our objective in this study is to devise additional acceleration of approximate matching through database optimization and efficient parallelism. For database optimization, we suggest the table division method which reduce unnecessary comparisons by decreasing the large number of entries in malware databases because such entries contain malware samples that cannot be similar. For efficient parallelism, we propose dynamic resource allocation for efficient resource utilization in parallel analysis. We integrate the above components into our work, Malfinder. As a result, we gained on average 280.9 times total performance improvements in our experiments.

[1]AV-TEST. http://www.av-test.org

Kim T., Hwang W., Kim C., Shin D., Park K. and Park K...
Malfinder: Accelerated Malware Classification System through Filtering on Manycore System.
In *Proceedings of the 1st International Conference on Information Systems Security and Privacy*, pages 17-26
ISBN: 978-989-758-081-9

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

Figure 1: Conversion: Decompiling and Structuring(Cesare et al., 2010).

2 BACKGROUND

2.1 Conversion and Matching

Malwise (Cesare et al., 2013) proposed procedure-level control flow matching methods to which we mainly refer. Its work flow consists of conversion and similarity measurement.

For conversion, the malware classifier of Malwise judges whether an input binary is malicious based on the similarity of control structures between the input binary and malware samples. In order to measure similarities of control structures, Malwise represents them in a structured control flow string (SCFS) form which expresses control structures in a high-level language (Sharir, 1980) is used. To generate SCFSs, the malware classifier converts malware through three stages: unpacking, decompiling and structuring. Unpacking is for extraction of malicious codes hidden by packers (OKane et al., 2011), decompiling is for converting the input binary into codes written in a high-level language, and structuring is for generating SCFSs from decompiled codes. The malware classifier uses grammars for the structuring procedure in Figure 1a and the example of conversion is described in Figure 1b. After conversion, the malware classifier measures string-to-string (S2S) similarities between SCFSs of the input binary and that of pre-analyzed malware samples in databases. Then, the malware classifier calculates the set similarity meaning how many characters of all SCFSs are matched in the right order by summating and normalizing S2S similarities (Cesare et al., 2010). Based on the set similarity, the malware classifier determines whether the input binary is malicious.

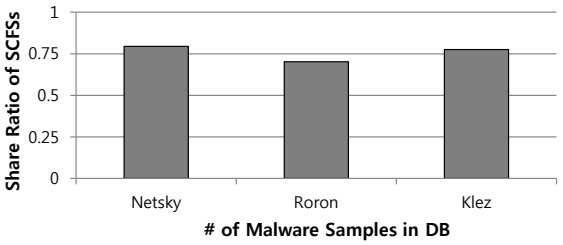


Figure 2: Share ratio of identical SCFSs.

2.2 MN-MATE

We implemented our malware variant classification system on MN-MATE (Park et al., 2012). MN-MATE is a resource management system for a many-core architecture to enhance performance and fairness among virtual machines (VMs). MN-MATE partitions and balances resources among VMs while considering NUMA architecture and the changing memory demands of VMs.

3 MOTIVATION

3.1 Inefficient SCFS Matching

Before measuring set similarities, we need to measure S2S similarities through character-to-character (C2C) matching based on the edit distance algorithm. 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 similarity there is between two SCFSs. 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 similarity there is between them 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. According to our preliminary experiments, malware variants in the same family share many identical SCFSs. In Figure 2, an average of 75.7% of identical SCFSs are shared in *Netsky*, *Klez* and *Roron*.

3.2 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.

3.3 Non-parallelized Malware Analysis

In 2013, malware authors created about 80 million malware samples 88% of which are malware variants, 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 (Cesare et al., 2013; Kim et al., 2014). 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

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 Overview

We implemented the malware variant classification system on MN-MATE (Park et al., 2012). Our malware variant classification system consists of three parts: Convertor, Analyzer and Resource Manager. Both Convertor and Analyzer work on VMs but Resource Manager works on dom0, the priviledged VM that can control hypervisor (Paul et al., 2003). We describe our architecture in Figure 3 and the flow chart in Figure 4.

Convertor Convertor is responsible for converting input binaries into SCFSs. This conversion task is composed of unpacking, decompiling and structuring. 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 (character-to-character) 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 (Kim et al., 2014). 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.

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

In Section 3, we point 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 (Kim et al., 2014) 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.

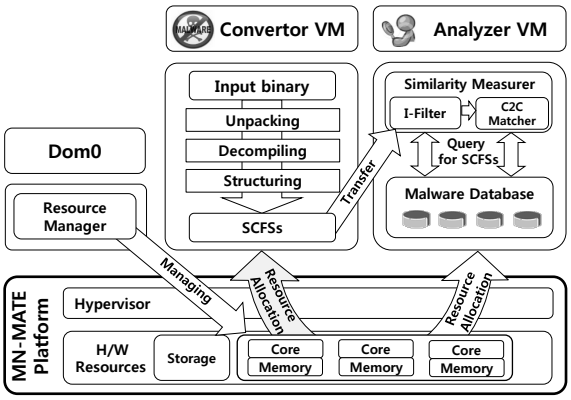


Figure 3: Overview of our system on MN-MATE.

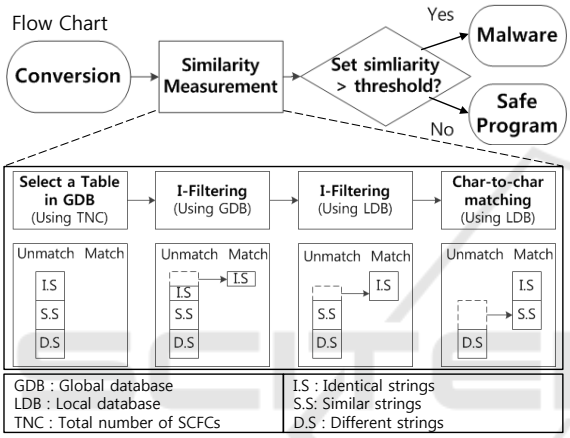


Figure 4: Whole analysis flow chart.

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 (Cesare et al., 2010). In order to accelerate matching for SCFSs, all SCFSs are stored in the BK-tree (Baeza-Yates et al., 1998) indexed malware database in the previous work (Cesare et al., 2013). However, it is time-consumming to find 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-chracter comparison. In addition, the number of comparions 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 $O(\boldsymbol{1})O(\log \boldsymbol{s}) = O(\log \boldsymbol{s})$. 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 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 Figure 5. 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 Figure 6. 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 (Cesare et al., 2010). As described in Figure 6, 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

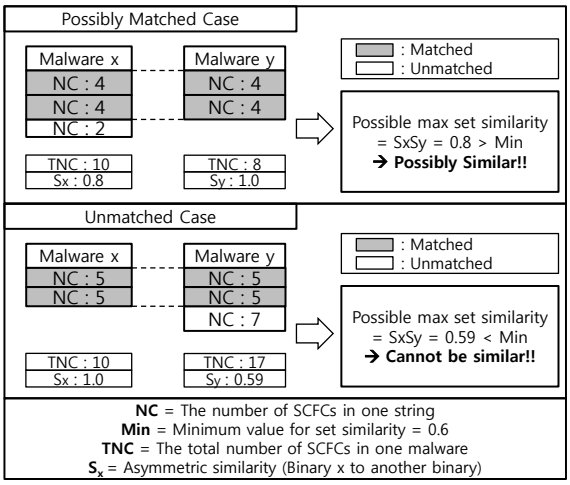


Figure 5: Max similarity according to the total number of SCFCs.

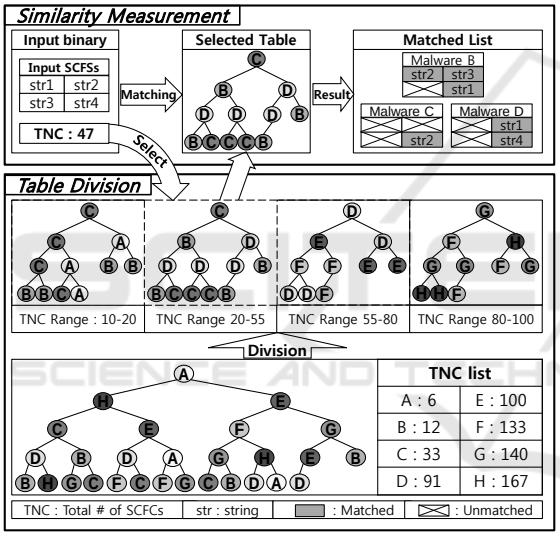


Figure 6: An example of table division application.

included in the divided 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%.

4.4 Dynamic Resource Allocation

Our system consists of two main processing parts, Convertor and Analyzer. We describe this system in Figure 7. Because processing speeds vary according to which binaries are analyzed, unbalanced workloads can waste resources. 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. In detail, (1) of Figure 7 shows that Resource Manager allocates VCPUs to Convertor and operates one more convertor process since there is no more converted input binaries for additional analysis. On the other hand, (2) of Figure 7 shows the case in which Q value higher than the threshold value indicates too many binaries were converted. In this case, Resource Manager reallocates VCPUs to Analyzer and operates one more analyzer process according to CPU usage. For more efficient memory 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.

4.5 Implementation

Convertor. Convertor is responsible for unpacking, decompiling and structuring. For unpacking, we use the unpacking function of UPX[2] because malware authors widely use it to pack malware programs. After the unpacking process, we decompile unpacked binaries using REC decompiler[3]. Then, we convert decompiled binaries into SCFSs using the rule in Figure 1a.

Analyzer. This module measures similarities between input binaries and malware samples. We describe the detailed procedure in algorithms 1 and 2. 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

[2]Ultimate Packer for eXecutables. http://upx.sourceforge.net

[3]Reverse Engineering Compiler (REC). http://www.backerstreet.com

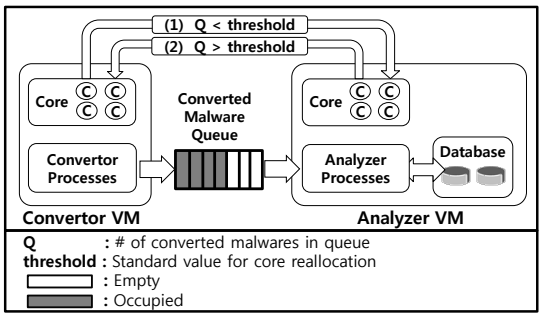


Figure 7: Dynamic core allocation model.

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 (Cesare et al., 2010). 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. We currently allocate an additional core to the Analyzer VM when Q is higher than 40 and to the Convertor VM when Q is lower than 20. 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.

Algorithm 1: Similarity measurement.

```
freq       : duplication level of SCFSs
gtb(freq)  : selected global table
ltb(m)     : local table for one malware family
m          : a malware
i          : an input binary
cand       : similar malware candidate
pcands     : possibly similar malware samples
matching   : update similarity using algorithm2
scfs       : unmatched SCFS
TNC        : total number of SCFCs
T          : set similarity threshold
minT       : safe program threshold
id         : identical matching
app        : approximate matching
setSim[m]  : set similarity with a malware

Measure TNC of input and select gtb using TNC;
while scfs_i in input SCFSs do
    while scfs_m in gtb(unique) do
        matching(scfs_i, scfs_m, setSim[m], id)
        if any setSim[m] ≥ T then
            pcands ← m
        end
    end
end
while scfs_i in input SCFSs do
    while scfs_m in gtb(duplicate) do
        matching(scfs_i, scfs_m, setSim[m], id)
        if any setSim[m] ≥ T then
            pcands ← m
        end
    end
end

if any setSim[m] ≤ min_t then
    return Result(Safe, 0)
end
pcands ← Top5MostSimlar

while cand in pcands do
    while scfs_i in input SCFSs do
        while scfs_m in ltb(mcand) do
            matching(scfs_i, scfs_m, setSim[cand], id)
        end
    end
    while scfs_i in input SCFSs do
        while scfs_m in ltb(mcand) do
            matching(scfs_i, scfs_m, setSim[cand],
            app)
        end
    end
    if setSim[m] ≥ T then
        return Result(cand, setSim[cand])
    end
end
return Result(Safe, 0)
```

Algorithm 2: Match.

```
m          : a malware
S_i        : similarity from input to malware
S_m        : similarity from malware to input
scfsSim    : similarity of a SCFS pair
setSim[m]  : set similarity between input and malware
nc[scfs]   : the number of characters in scfs
T          : string-to-string similarity threshold
hash[scfs] : hash value of a SCFS

if matching method == identical then
    if hash[scfs_i] == hash[scfs_m] then
        S_i ← S_i + nc[scfs_i]
        S_m ← S_m + nc[scfs_m]
        setSim[m] ← S_i · S_m
    end
else
    Measure scfsSim between scfs_i and scfs_m;
    if scfsSim ≥ T then
        S_i ← S_i + nc[scfs_i] · scfsSim
        S_m ← S_m + nc[scfs_m] · scfsSim
        setSim[m] ← S_i · S_m
    end
end
```

5 EXPERIMENT

This section presents module performance improvements, total performance improvements, similarities between malware variants and validation of safe program threshold. The experimental environment consists of the AMD Opteron Processor 6282SE 64 core 2.6Ghz, 128GB RAM, SAS 10kbytes HDD, Cent OS 6.4 64bit with Kernel 3.8.2 version, MN-MATE (Park et al., 2012) and MySQL 14.14 for the database, but we utilized 16 cores and 16GB RAM. We implemented our databases on MyISAM[4] which is a type of disk-based database. On the other hand, Malwise (Cesare et al., 2013) consists of BK-tree (Baeza-Yates et al., 1998) indexed memory databases. In all experiments, Malfinder 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[5] and generated additional malware variants using the code mutation tool[6].

[4]MySQL reference. http://dev.mysql.com/doc/refman/5.7/en/index.html

[5]Offensive computing. http://www.offensivecomputing.net

[6]Code pervertor. http://z0mbie.host.sk

5.1 Module Performance Improvements

In this section, we evaluate the performance of each module in malware variant classification 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 Figure 8. For the Analyzer performance, the performance improvements between approximate matching of Malwise (Cesare et al., 2013) and application of all of our techniques on MN-MATE are from 512 to 657 times. For comparison between I-Filter application and Malfinder, 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 Figure 9.

We describe the performance improvements in Figure 10, 11 and 12. 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 demonstrate the performance of Convertor in Figure 13. There is no performance variation according to the number of malware samples in the databases because the operation is independent of databases. 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 classification systems. We

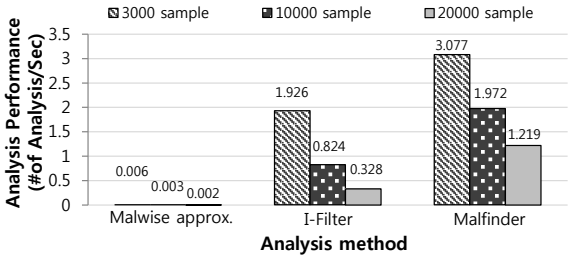


Figure 8: Analyzer performance with a single core

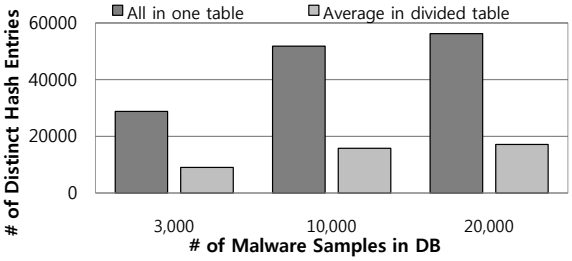


Figure 9: The number of hash entries in each database.

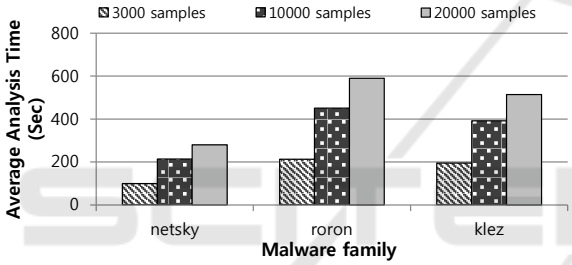


Figure 10: Analyzer performance with approximate matching of Malwise.

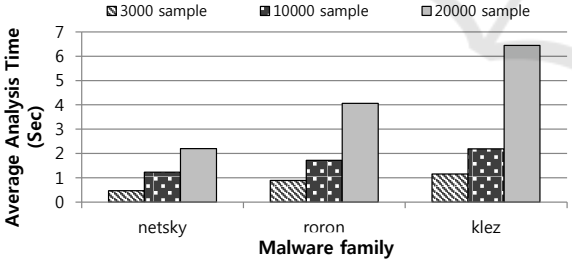


Figure 11: Analyzer performance with approximate matching and I-Filter.

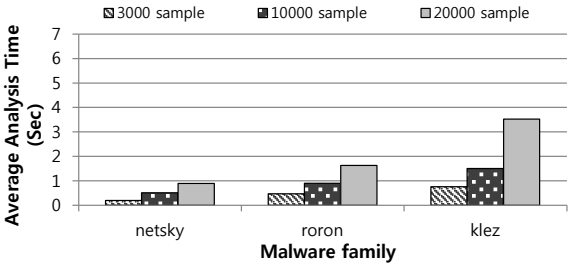


Figure 12: Analyzer performance with Malfinder.

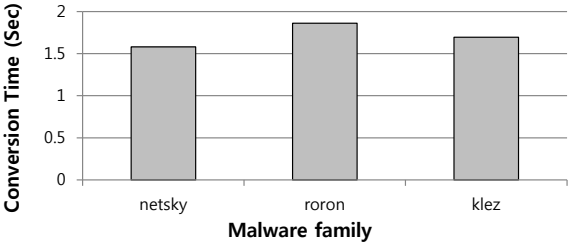


Figure 13: Convertor performance.

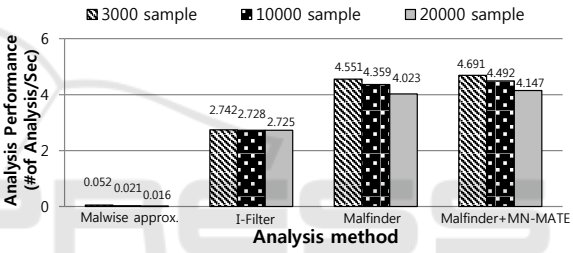


Figure 14: Total performance (50% of resource to Convertor and 50% of resource to Analyzer).

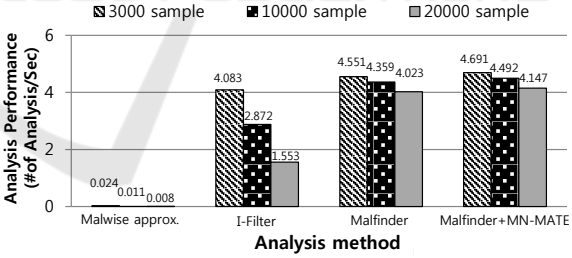


Figure 15: Total performance (75% of resource to Convertor and 25% of resource to Analyzer).

applied our Convertor to Malwise (Cesare et al., 2013) because Malwise does not have dynamic resource allocation functions. We randomly choose malware samples for our experiments and show the performance evaluation in Figure 14 and 15.

Performance improvements of Malfinder 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 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 (Cesare et al., 2013). Table 1 shows similarities between malware variants in *Klez*, *Roron* and *Netsky* malware families.

Table 1: Similarities between Malware Variants.

: Matched : Unmatched

	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

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 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 (Cesare et al., 2013), 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.

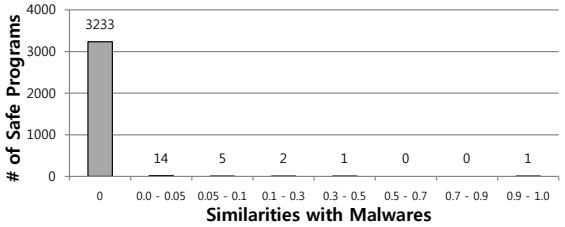


Figure 16: Similarity between safe programs and malwares.

5.4 Validity of Safe Program Threshold

As we mentioned in implementation of Analyzer, we use a safe program threshold, 0.1. If the set similarity is lower than 0.1, we consider the input as a safe program after matching identical SCFSs with the global database. To validate our parameter, we measure similarities of 3,256 safe 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 safe programs exceeded 0.1 as shown in Figure 16.

6 RELATED WORK

Malware classification 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 (Cesare et al., 2013). 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 string matching is time-consuming due to large string sizes. In order to resolve this performance bottleneck, they proposed

short filtering (Li et al., 2006). 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 (Gusev et al., 2012). It distributes workloads to multiple VMs. However, we distribute VCPUs instead of workloads. In an approach similar to our work, some researchers have proposed dynamic resource allocation (Kundu et al., 2010). 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.

7 CONCLUSION

Our main goal was to accelerate approximate matching, which cannot classify numerous malware variants, its performance is too low. To accomplish our objective, we proposed Malfinder with I-Filter, table division and dynamic resource allocation which focuses on acceleration of Analyzer and apply them incrementally. 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.

ACKNOWLEDGEMENT

This work was supported by Ministry of Knowledge Economy, Republic of Korea (Project No. 10035231).

REFERENCES

Baeza-Yates, R. and Navarro, G. (1998). Fast Approximate String Matching in a Dictionary. In *Proceedings of A South America Symposium on String Processing and Information Retrieval*, SPIRE 1998, pages 14-22, IEEE.

Cesare, S. and Xiang, Y. (2010). Classification of Malware Using Structured Control Flow. In *Proceedings of Australasian Symposium on parallel and Distributed Computing*, AusPDC 2010, pages 61-70, ACM.

Cesare, S., Xiang, Y. and Zhou, W. (2013). Malwise–An Effective and Efficient Classification System for Packed and Polymorphic Malware. *IEEE Transactions on Computers*, 62(6):1193-1206.

Gusev, M. and Ristov, S. (2012). Matrix multiplication performance analysis in virtualized shared memory multiprocessor. In *Proceedings of 35th International Convention*, MIPRO 2012, pages 251-256, IEEE.

Kephart, J.O. and Arnold, W.C. (1994). Automatic Extraction of Computer Virus Signatures. *Virus Bulletin Conference*, 1994, pages 178-184.

Kim, T., Hwang, W. Park, K. W. and Park, K. H. (2014). 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.

Kundu, S., Rangaswami, R., Dutta, K. and Zhao, M. (2010). Application performance modeling in a virtualized environments. In *Proceedings of 16th International Symposium on High Performance Computer Architecture*, HPCA 2010, pages 1-10, IEEE.

Li, W. and Godzik, A. (2006). Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics*, 22(13):1658-1659.

OKane, P., Sezer, S. and McLaughlin, K. (2011). Obfuscation: The Hidden Malware. *IEEE Security & Privacy*, 9(5):41-47.

Park K. H., Park S. K., Hwang W., Seok H., Shin D. J., and Park K. W. (2012). Resource Management of Manycores with a Hierarchical and a Hybrid Main Memory for MN-MATE Cloud Node. In *Proceedings of Eighth World Congress on Services*, SERVICES 2012, page 301-308, IEEE.

Paul B., Boris D., Keir F., Steven H., Tim H., Alex H., Rolf N., Ian P., Andrew W. (2003). Xen and the art of virtualization. In *Proceedings of the 19th ACM symposium on Operating systems principles*, SOSP 2003, pages 164-177, ACM.

Sharir, M. (1980). Structural Analysis : A new approach to flow analysis in optimizing compiler. *Computer Languages*, 5(3-4):141-153.

Ukkonen, E. (1986). Algorithms for approximate string matching. *Information and Control*, 61(1-3):100-118.