IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 1, 2019, accepted December 19, 2019, date of publication January 6, 2020, date of current version March 9, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2964260

Redundancy Analysis and Elimination on Access Patterns of the Windows Applications Based on I/O Log Data

JUN-HA LEE[1] **AND HYUK-YOON KWON**[1,2]

[1]Department of Industrial and Systems Engineering, Seoul National University of Science and Technology (SeoulTech), Seoul 01811, South Korea
[2]Research Center for Electrical and Information Technology, Seoul National University of Science and Technology (SeoulTech), Seoul 01811, South Korea

Corresponding author: Hyuk-Yoon Kwon (hyukyoon.kwon@seoultech.ac.kr)

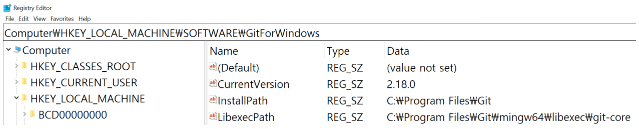
This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2018R1C1B5084424), and in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2019R1A6A1A03032119).

ABSTRACT In this paper, we analyze I/O log data monitored in the Windows operating system for improving the system performance. Especially, we focus on the I/O operations to the Windows registry. As a result, we identify redundant access patterns of the Windows applications. To find all the possible redundant patterns from the large-scale log data, we propose the redundancy detection algorithm. Then, we propose the two-level redundancy elimination method to remove unnecessary redundant operations. We also present an event-driven method that guarantees that the result of redundancy elimination is equivalent to that of the original program. Through experiments, we show that the proposed redundancy elimination method improves the performance of the original program having redundant access patterns by up to 90.25% for individual access patterns; by 8.93% ∼ 26.21% when the multiple programs having combined access patterns are running concurrently.

INDEX TERMS I/O log data, access pattern analysis, redundancy elimination, Windows registry.

I. INTRODUCTION

Log data is a collection of recording the events that are occurred from the operating systems or applications or the messages that communicate between applications [1]. There have been many research efforts for utilizing various types of log data. Mafrur *et al.* have used event log data generated in smartphones to build the human behavior model [2]. Kankane and Garg have used Web log data to analyze the usage patterns to the Web sites [3]. Chung *et al.* have used life-log data collected from the patients to better understand the patient values [4]. In this paper, we analyze I/O log data monitored in the Windows operating system. Especially, we focus on I/O operations to the Windows registry, which has not yet been considered in the previous work. The Window registry is a database that stores crucial information for the Windows operating systems and the Windows applications [5]–[7]. The Windows registry is structured in a tree format; each node in the tree stores information in a form of the key and value pair.



Computer₩HKEY_LOCAL_MACHINE₩SOFTWARE₩GitForWindows

- Computer
  - HKEY_CLASSES_ROOT
  - HKEY_CURRENT_USER
  - HKEY_LOCAL_MACHINE
    - BCD00000000

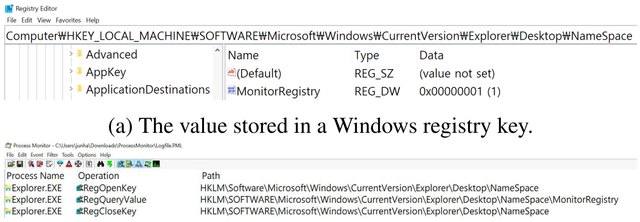
Name	Type	Data
(Default)	REG_SZ	(value not set)
CurrentVersion	REG_SZ	2.18.0
InstallPath	REG_SZ	C:₩Program Files₩Git
LibexecPath	REG_SZ	C:₩Program Files₩Git₩mingw64₩libexec₩git-core

FIGURE 1. **The registry keys and values.**

Fig. 1 shows the registry keys and registry values monitored by Registry Editor [8], which is a built-in registry editor in Windows. In the left panel, the registry keys are stored in a form of the tree; in the right panel, the registry keys and values are stored. In Fig. 1, we have a registry key, “Computer\HKEY_LOCAL_MACHINE\SOFTWARE\GitForWindows\InstallPath,” and its associated registry value, “C:\Program Files\Git.”

The Windows applications store necessary information in the registry and use them by calling registry operations provided by the Windows operating system. The example registry operations are **RegOpenKey**, which opens a registry key to use, **RegCloseKey**, which closes a registry key, and **RegQueryValue**, which obtains the registry value for

The associate editor coordinating the review of this manuscript and approving it for publication was Bijoy Chand Chand Chatterjee.



```
Registry Editor
File  Edit  View  Favorites  Help
Computer₩HKEY_LOCAL_MACHINE₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩Explorer₩Desktop₩NameSpace
  > Advanced                    Name               Type      Data
  > AppKey                      (Default)          REG_SZ    (value not set)
  > ApplicationDestinations     MonitorRegistry    REG_DW    0x00000001 (1)
```

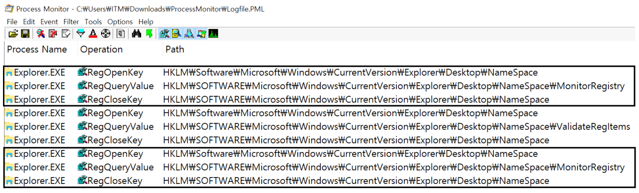
(a) The value stored in a Windows registry key.

```
Process Monitor - C:\Users\junha\Downloads\ProcessMonitor\Logfile.PML
File  Edit  Event  Filter  Tools  Options  Help
Process Name   Operation        Path
Explorer.EXE   RegOpenKey       HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\Desktop\NameSpace
Explorer.EXE   RegQueryValue    HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Desktop\NameSpace\MonitorRegistry
Explorer.EXE   RegCloseKey      HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Desktop\NameSpace
```

(b) The registry operations to access to a Windows registry key.

FIGURE 2. A use case of the Windows registry.

a specific registry key. Fig. 2 shows a use case of the Windows registry. Fig. 2 (a) shows an actual value stored in a registry key; Fig. 2 (b) shows a sequence of registry operations to access to the value stored in a registry key, which is observed by Program Monitor [9]. Specifically, a Windows application, "Explorer.exe," is trying to obtain the value, "0x00000001," stored in a registry key, "Computer\HKEY_LOCAL_MACHINE\Software\Microsoft \Windows\CurrentVersion \Explorer\Desktop\NameSpace \MonitorRegistry," by calling **RegOpenKey, RegQueryValue,** and **RegCloseKey** in turn.

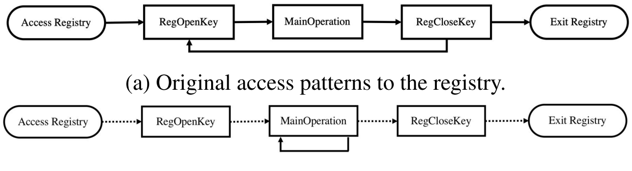


```
Process Monitor - C:₩Users₩ITM₩Downloads₩ProcessMonitor₩Logfile.PML
File  Edit  Event  Filter  Tools  Options  Help
Process Name   Operation        Path
Explorer.EXE   RegOpenKey       HKLM₩Software₩Microsoft₩Windows₩CurrentVersion₩Explorer₩Desktop₩NameSpace
Explorer.EXE   RegQueryValue    HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩Explorer₩Desktop₩NameSpace₩MonitorRegistry
Explorer.EXE   RegCloseKey      HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩Explorer₩Desktop₩NameSpace
Explorer.EXE   RegOpenKey       HKLM₩Software₩Microsoft₩Windows₩CurrentVersion₩Explorer₩Desktop₩NameSpace
Explorer.EXE   RegQueryValue    HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩Explorer₩Desktop₩NameSpace₩ValidateRegItems
Explorer.EXE   RegCloseKey      HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩Explorer₩Desktop₩NameSpace
Explorer.EXE   RegOpenKey       HKLM₩Software₩Microsoft₩Windows₩CurrentVersion₩Explorer₩Desktop₩NameSpace
Explorer.EXE   RegQueryValue    HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩Explorer₩Desktop₩NameSpace₩MonitorRegistry
Explorer.EXE   RegCloseKey      HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩Explorer₩Desktop₩NameSpace
```

FIGURE 3. An example of redundant access patterns to a registry.

In this paper, we analyze access patterns of Windows applications to the registry and identify redundant access patterns. Fig. 3 shows an example of redundant access patterns. It shows that the same access pattern (i.e., a pattern of **RegOpenKey, RegQueryValue,** and **RegCloseKey**) is repeated for a registry key (i.e., HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\Desktop\NameSpace\MonitorRegistry). This implies that a Windows application, Explorer.exe, utilizes the data retrieved from the registry repeatedly. Here, we note that the repeated accesses to the same registry are not required because we can retrieve it once and utilize it repeatedly.

In this paper, we propose the redundancy elimination method that removes redundant access patterns while guaranteeing the equivalent results. Fig. 4 shows a flow chart to describe the concept of the redundancy elimination method. Fig. 4 (a) is a flow chart that describes the repeated access patterns observed in the original access patterns to the registry; Fig. 4 (b) is a flow chart that removes the repeated access patterns. In Fig. 4 (a), both the operations to the registry (e.g., a sequence of **RegOpenKey, RegQueryValue,** and **RegCloseKey**) and the main operation (i.e., utilizing the data retrieved from the registry) are repeated; however, in Fig. 4 (b), only the main operation is repeated, and the operations to the registry are performed once.



(a) Original access patterns to the registry.

(b) Redundancy elimination of the access patterns to the registry.

FIGURE 4. Original redundant access patterns and the redundancy elimination.

Now, we need to guarantee that the result of redundancy elimination is equivalent to that of the original access pattern. For this, we should consider the case of updating the registry values while we are utilizing them repeatedly in the main operation. In the original access patterns, the main operation is able to utilize the updated data because the updates could be caught by the repeated registry access. However, in the redundancy elimination method, the updates that are occurred after we retrieve the data from the registry could not be reflected. To solve this problem, we investigate a method to detect if the updates are occurred. Specifically, we present an *event-driven method* that is able to catch the updates on a specific registry as soon as the target registry is updated. For this, we have to register an event handler to catch the updates on a target registry. Then, if the updates to the registry are occurred, the registered event handler will read newly updated value from the registry. Consequently, the proposed event-driven method can completely remove the side effect of the redundancy elimination method.

The contributions of the paper are summarized as follows.

1) We analyze I/O log data of the Windows operating system and identify the redundant access patterns to the Windows registry. Especially, we verify it in the level of assembly codes by disassembling an actual Windows application. Then, we classify redundant access patterns into the internal redundancy and the outer redundancy.
2) We propose the *redundancy detection algorithm* that finds all the possible redundant patterns from the large-scale log data. By identifying all the redundant patterns by the proposed algorithm, we show that the internal redundancy is about 59.21% and the outer redundancy is about 57.50%, which implies that we can improve the performance of accessing to the registry.
3) We propose the *two-level redundancy elimination method* to remove the internal and outer redundancy. Especially, the proposed method enhances the effect of the outer redundancy elimination by eliminating the outer redundancy after eliminating the internal redundancy first. We also present *an event-driven method* that guarantees the correctness of the redundancy elimination method by catching the updates on the registry instantly.

4) Through experiments, we show that the proposed two-level redundancy elimination approach improves the performance of the original program having redundant access patterns by up to 90.25% for individual access patterns; by 8.93% ∼ 26.21% when the multiple programs are running concurrently.

The organization of the paper is as follows. In Section II, we introduce the Windows registry and Process Monitor, which is used to monitor I/O operations occurred in the Windows operating system, as the background of the paper. In Section III, we analyze redundant access patterns and verify them in the level of assembly codes. In Section IV, we propose the redundancy detection algorithm. In Section V, we propose the two-level redundancy elimination method and event-driven method. In Section VI, we present the experimental results to show the performance improvement of the redundancy elimination method. In Section VII, we present the related work. In Section VIII, we conclude the paper.

TABLE 1. Registry operations for accessing the Windows registry [10].

Operations	Description
RegOpenKey	Open a specified registry key
RegCloseKey	Close the handle to a specified registry key
RegCreateKey	Create a specified registry key. If the key exists, **RegOpenKey** is called.
RegDeleteKey	Delete a specified key with its subkeys and values
RegQueryValue	Retrieve the registry value associated with an specified registry key
RegSetValue	Set a specified value for a registry key
RegDeleteValue	Delete the registry value associated with a specified registry key
RegEnumKey	Enumerate the subkeys of a specified registry key
RegEnumValue	Enumerate the values for a specified registry key

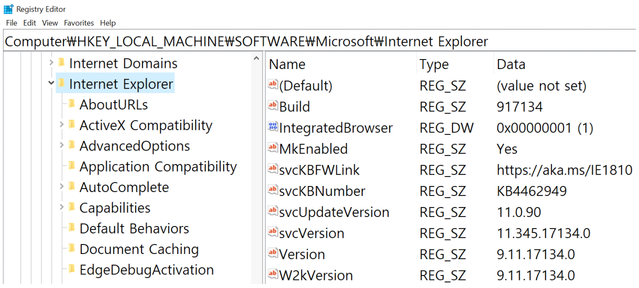
II. BACKGROUND

A. WINDOWS REGISTRY

Table 1 shows the registry operations to access to the Windows registry [10]. Let us consider an example to update the version information of Internet Explorer stored in the registry. Fig. 5 (a) shows a version information of Internet Explorer stored in the Windows registry; Fig. 5 (b) illustrates a source code that updates the version using the registry operations. First, we open the registry key representing the version of Internet Explorer by executing the operation **RegOpenKey**. Second, we update the current version to a new value, "9.11.17134.0," for a registry key, "Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft \Internet Explorer\Version," by executing the operation **RegSetValue**. Third, we close the handle to the registry key by executing the operation **RegCloseKey**.

B. PROCESS MONITOR

In this paper, we use Process Monitor [9] to collect I/O log data generated in the Windows operating system. Process Monitor is a tool provided by Microsoft for monitoring Windows I/O events that are occurred on the registries, file systems, processes, and networks [9]. Fig. 6 shows a sample of log data on I/O activities monitored by Process Monitor.



Computer₩HKEY_LOCAL_MACHINE₩SOFTWARE₩Microsoft₩Internet Explorer

- Internet Domains
- Internet Explorer
  - AboutURLs
  - ActiveX Compatibility
  - AdvancedOptions
  - Application Compatibility
  - AutoComplete
  - Capabilities
  - Default Behaviors
  - Document Caching
  - EdgeDebugActivation

Name	Type	Data
(Default)	REG_SZ	(value not set)
Build	REG_SZ	917134
IntegratedBrowser	REG_DW	0x00000001 (1)
MkEnabled	REG_SZ	Yes
svcKBFWLink	REG_SZ	https://aka.ms/IE1810
svcKBNumber	REG_SZ	KB4462949
svcUpdateVersion	REG_SZ	11.0.90
svcVersion	REG_SZ	11.345.17134.0
Version	REG_SZ	9.11.17134.0
W2kVersion	REG_SZ	9.11.17134.0

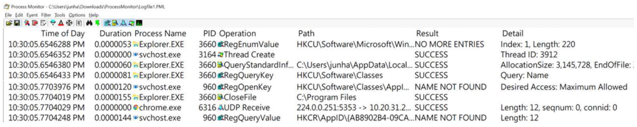
(a) Version information of Internet Explorer stored in the registry.

Updating the Version of Internet Explorer

```
RegOpenKey(HKEY_LOCAL_MACHINE,
               "SoftWare\Microsoft\Internet Explorer,
                0, KEY_ALL_ACCESS, &key_handle)

RegSetValue(key_handle,
                "Version",
                0, REG_SZ,
                "9.12.17134.0", 0 ) };

RegCloseKey(key_handle);
```

(b) The source code of updating the registry value using registry operations.

FIGURE 5. The registry and updating the registry.


Process Monitor - C:\Users\junha\Downloads\ProcessMonitor\Logfile1.PML
File Edit Event Filter Tools Options Help

```
   Time of Day       Duration  Process Name  PID Operation          Path                             Result            Detail
10:30:05.6546288 PM  0.0000053 Explorer.EXE  3660 RegEnumValue       HKCU\Software\Microsoft\Win...NO MORE ENTRIES   Index: 1, Length: 220
10:30:05.6546352 PM  0.0000000 svchost.exe   3164 Thread Create                                      SUCCESS           Thread ID: 3912
10:30:05.6546380 PM  0.0000060 Explorer.EXE  3660 QueryStandardInf... C:\Users\junha\AppData\Local... SUCCESS           AllocationSize: 3,145,728, EndOfFile:
10:30:05.6546433 PM  0.0000081 Explorer.EXE  3660 RegQueryKey        HKCU\Software\Classes            SUCCESS           Query: Name
10:30:05.7703976 PM  0.0000120 svchost.exe    960 RegOpenKey         HKCU\Software\Classes\Appl...    NAME NOT FOUND    Desired Access: Maximum Allowed
10:30:05.7704019 PM  0.0000155 Explorer.EXE  3660 CloseFile          C:\Program Files                 SUCCESS
10:30:05.7704029 PM  0.0000000 chrome.exe    6316 UDP Receive        224.0.0.251:5353 -> 10.20.31.2... SUCCESS          Length: 12, seqnum: 0, connid: 0
10:30:05.7704248 PM  0.0000144 svchost.exe    960 RegQueryValue      HKCR\AppID\{AB8902B4-09CA...NAME NOT FOUND    Length: 12
```

FIGURE 6. Windows I/O log data monitored by Process Monitor.

The specific operations are as follows: 1) operations on the file system such as creating files (i.e., **CreateFile**), writing a value into the file (i.e., **WriteFile**), and reading a value from the file (i.e., **ReadFile**), 2) operations on the network such as sending and receiving data based on TCP/UDP (i.e., **TCPSend / UDPSend** and **TCPReceive / UDPReceive**), 3) operations on the process such as creating processes or threads (i.e., **ProcessCreate** and **ThreadCreate**) and starting or stopping processes (i.e., **ProcessStart** and **ProcessExit**).

The types of data to be monitored by Process Monitor are as follows: 1) Time of Day (i.e., time to call the operation), 2) Duration (i.e., elapsed times spent to conduct the operation), 3) Process Name (i.e., the name of the process that calls the operation), 4) PID (i.e., the ID of the process), 5) Operation (i.e., the called operation), 6) Path (i.e., the path on which the operation is conducted), 7) Result (i.e., the result of calling the operation), and 8) Detail (i.e., the details of the result).

III. ANALYZING OF ACCESS PATTERNS TO THE WINDOWS REGISTRY

A. LOG DATA ON WINDOWS I/O OPERATIONS

In this paper, we analyze the log data for optimizing the registry accesses. The reason why we are using the log data

TABLE 2. The characteristics of the collected Window I/O log data.

Windows versions	Number of events	Number of information	Total collection time (seconds)	Dataset size
Windows 10	5, 000, 000	8	856.48	666.7MB
Windows 8.1	5, 000, 000	8	954.27	645.9MB
Windows 7	5, 000, 000	8	752.04	622.6MB
Total	15, 000, 000	8	2562.79	1935.2MB

TABLE 3. The occurrence and duration by the operation type.

Operation types	Occurrences	Duration (seconds)
Registry	11,463,692 (76.43%)	824.25 (31.42%)
File System	3,426,005 (22.84%)	1799.22 (68.58%)
Network	93,651 (0.62%)	0.0001 (0.000004%)
Process	16,652 (0.11%)	0.0003 (0.00001%)
Total	15,000,000 (100.00%)	2,623.47 (100.00%)

instead of using source codes or binary codes is summarized as follows. First, most Windows applications do not disclose source codes. This requires the reverse engineering for analyzing and optimizing the binary codes. Second, if we have source codes, coding styles are quite various even if they are equivalent. Third, the goal for the optimization in this paper is a quite specific. That is, we focus on optimizing of the registry accesses. All the registry accesses can be caught as the system events and can be collected. Thus, we can easily analyze the log data compared to analysis on the binary codes or source codes. Fourth, we can clearly predict the effect of redundancy elimination by using the log data, which will be actually shown in this paper later.

We collect all the I/O operations occurred in the Windows operating system using Process Monitor. To cover various environments, we collect the log data from various Windows versions on the most popular three ones: Windows 10, Windows 7, and Windows 8.1. According to the statistics for Windows version market share,[1] those three versions occupy about 95.42%. To build a common workload environment in the Windows operating system, we execute 10 common user processes, i.e., iexplorer.exe, notepad.exe, explorer.exe, taskmgr.exe, calculator.exe, mspaint.exe, regedit.exe, word.exe, powerpoint.exe, and excel.exe while the default system processes such as svchost.exe, and searchindexer.exe are running. From each version of Windows, we collect 5 million events. Totally, we collect 15 million events from three different Windows versions; total sizes are almost about 2GB; total time for collection takes about 42.7 minutes. Table 2 shows the characteristics of the collected log data. The information for the eight columns is the same as explained in Fig. 6.

Table 3 is a result of analyzing the entire I/O log data by the I/O type, i.e., registry, file system, process, and network. It shows the occurrences and duration of the operations by the I/O type. Here, we note that all the operations are concentrated on the registry and file system. Moreover, the operations for the registry occupy about 76.43% in occurrences and 31.42% in duration. This implies that we can improve the overall system performance only if we make the registry accesses efficient.

Table 2 and Table 3 imply that I/O operations occupy most of the total time for the processes. That is, the total duration of the I/O operations in Table 3 is almost close to the total collection time in Table 2. The reason why the total duration of the I/O operations is even greater than the total collection time is as follows. In this paper, we analyze redundancy in the I/O operations by a group of operations, e.g., starting from **RegOpenKey** ending to **RegCloseKey**. Thus, if some operations in a group are included in the collected operations, then we include the entire operations in the group. Here, the important thing is that I/O operations occupy most of the total time of the process, and the registry operations occupy significant portion out of the entire I/O operations. As a result, we can significantly improve the process performance by making the registry access of the process efficient.

TABLE 4. The occurrences and duration of the registry operations.

Operations	Occurrences	Duration (seconds)
RegOpenKey	3,398,806 (29.65%)	460.35 (55.85%)
RegQueryKey	2,957,266 (25.80%)	94.42 (11.46%)
RegQueryValue	2,127,163 (18.56%)	85.58 (10.38%)
RegCloseKey	1,878,538 (16.39%)	48.98 (5.94%)
RegSetInfoKey	394,141 (3.44%)	8.64 (1.05%)
RegEnumValue	337,098 (2.94%)	51.70 (6.27%)
RegEnumKey	229,246 (2.00%)	8.12 (0.99%)
RegCreateKey	92,557 (0.81%)	18.54 (2.25%)
RegSetValue	39,739 (0.35%)	19.24 (2.33%)
RegDeleteValue	3,329 (0.03%)	0.72 (0.09%)
RegQueryKeySecurity	2,906(0.03%)	0.03 (0.004%)
RegDeleteKey	1649 (0.01%)	0.52 (0.06%)
RegLoadKey	705 (0.01%)	22.89 (2.78%)
RegQueryMultipleValueKey	383 (0.003%)	0.16 (0.02%)
RegSetKeySecurity	142 (0.00%)	0.08 (0.01%)
RegFlushKey	22 (0.00%)	4.25 (0.52%)
RegRenameKey	2 (0.00%)	0.04 (0.01%)
Total	11,463,692	824.25

Table 4 is a result of analyzing the registry operations. It shows the occurrences and duration of the registry operations. We can indicate that the top-4 registry operations with the highest occurrences are **RegOpenKey, RegQueryKey, RegQueryValue**, and **RegCloseKey**. Especially, **RegOpenKey** and **RegCloseKey** occupy about 46.04% of the total occurrences and about 61.79% of the total duration. We can expect a considerable improvement of the overall performance only if we can reduce the calling of **RegOpenKey** and **RegCloseKey**. We also note that the portion of read operations such as **RegQueryKey** and **RegQueryValue** is much higher than that of write operations such as **RegCreateKey** and **RegSetValue**. That is, the read operation occupies about 49.34% in the occurrences and about 31.90% in the duration; the write operation occupies about 4.64% and about 6.31%.

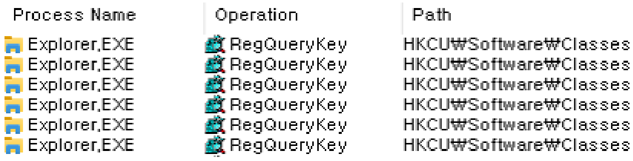
[1] https://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide

B. CASE STUDY OF REDUNDANT ACCESS PATTERNS

In this section, we introduce inefficient access patterns that have redundant accesses to the registry, which could be eliminated. The *basic structure* for accessing to the registry consists of the following three steps: 1) **RegOpenKey**, 2) a series of read/write operations to the registry, and 3) **RegCloseKey**. Thus, we classify the overall inefficient access patterns into two categories: 1) internal redundancy, which occurs within the basic structure, and 2) outer redundancy, which occurs between the basic structures. Specifically, in the internal redundancy, the same operation is repeated in multiple times between **RegOpenKey** and **RegCloseKey**; in the outer redundancy, the whole structure starting from **RegOpenKey** ending to **RegCloseKey** is repeated.

1) INTERNAL REDUNDANCY

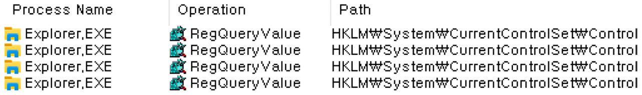
The access patterns in the internal redundancy retrieve or list the same registry keys or values repeatedly. They do not need to be repeated unless the target registry keys or values are not updated. However, in most cases, regardless of updating of the registry keys or values, most Windows applications are accessing them repeatedly according to the analysis of log data.



Process Name	Operation	Path
Explorer.EXE	RegQueryKey	HKCU₩Software₩Classes
Explorer.EXE	RegQueryKey	HKCU₩Software₩Classes
Explorer.EXE	RegQueryKey	HKCU₩Software₩Classes
Explorer.EXE	RegQueryKey	HKCU₩Software₩Classes
Explorer.EXE	RegQueryKey	HKCU₩Software₩Classes
Explorer.EXE	RegQueryKey	HKCU₩Software₩Classes

FIGURE 7. Internal redundancy of RegQueryKey in a consecutive way.

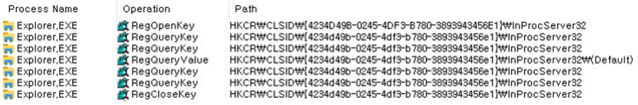
We introduce the observed three cases for the internal redundancy. Fig. 7 represents a case of the internal redundancy. It is a redundancy of **RegQueryKey**, and it is represented in a consecutive way, i.e., there are no operations between two consecutive **RegQueryKey** operations. We note that no operations to update the registry key have been called while **RegQueryKey** is repeated. This means that the operations from the second **RegQueryKey** are not necessary in this case, and only main operations that utilize the registry key retrieved from the first **RegQueryKey** need to be repeated.



Process Name	Operation	Path
Explorer.EXE	RegQueryValue	HKLM₩System₩CurrentControlSet₩Control
Explorer.EXE	RegQueryValue	HKLM₩System₩CurrentControlSet₩Control
Explorer.EXE	RegQueryValue	HKLM₩System₩CurrentControlSet₩Control
Explorer.EXE	RegQueryValue	HKLM₩System₩CurrentControlSet₩Control

FIGURE 8. Internal redundancy of RegQueryValue in a consecutive way.

Fig. 8 represents another case of the internal redundancy where **RegQueryValue** is redundant in a consecutive way. Again, no operations to update the registry key have been called while **RegQueryValue** is repeated. Thus, we can eliminate the operations from the second **RegQueryValue**.



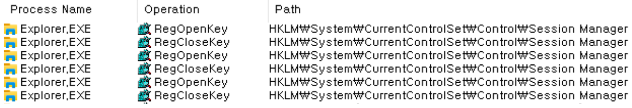
Process Name	Operation	Path
Explorer.EXE	RegOpenKey	HKCR₩CLSID₩{4234D49B-0245-4DF3-B780-3893943456E1}₩InProcServer32
Explorer.EXE	RegQueryKey	HKCR₩CLSID₩{4234d49b-0245-4df3-b780-3893943456e1}₩InProcServer32
Explorer.EXE	RegQueryKey	HKCR₩CLSID₩{4234d49b-0245-4df3-b780-3893943456e1}₩InProcServer32
Explorer.EXE	RegQueryValue	HKCR₩CLSID₩{4234d49b-0245-4df3-b780-3893943456e1}₩InProcServer32₩(Default)
Explorer.EXE	RegQueryKey	HKCR₩CLSID₩{4234d49b-0245-4df3-b780-3893943456e1}₩InProcServer32
Explorer.EXE	RegQueryKey	HKCR₩CLSID₩{4234d49b-0245-4df3-b780-3893943456e1}₩InProcServer32
Explorer.EXE	RegCloseKey	HKCR₩CLSID₩{4234d49b-0245-4df3-b780-3893943456e1}₩InProcServer32

FIGURE 9. Internal redundancy of RegQueryKey in an inconsecutive way.

Fig. 9 represents the last case of the internal redundancy where **RegQueryKey** is redundant in an inconsecutive way. That is, **RegQueryKey** is repeated in multiple times between **RegOpenKey** and **RegCloseKey**, but the other operation, i.e., **RegQueryValue**, is interleaved with **RegQueryKey**. Again, no operations to update the registry key have been called while **RegQueryKey** is repeated. Thus, we can eliminate the operations from the second **RegQueryKey**. The other operation **RegQueryValue**, which is performed while **RegQueryKey** is repeated, does not affect the result of **RegQueryKey** because both are read operations.

2) OUTER REDUNDANCY

In the outer redundancy, the whole pattern starting from **RegOpenKey** and ending to **RegCloseKey** is repeated. Between **RegOpenKey** and **RegCloseKey**, any series of registry operations could be placed. In this case, we note that **RegOpenKey** and **RegCloseKey** do not need to be repeated because the closed registry key will be opened again; only the registry operations between **RegOpenKey** and **RegCloseKey** are needed to be repeated. Here, a very complicated combination of the registry operations could be placed between **RegOpenKey** and **RegCloseKey**. Thus, we focus on the elimination of **RegOpenKey** and **RegCloseKey** in the outer redundancy because the elimination for the combination of complex registry operations could incur unexpected side effects. Moreover, most redundancy between **RegOpenKey** and **RegCloseKey** could be handled when we eliminate the internal redundancy.

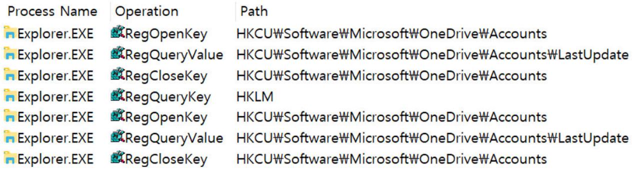


Process Name	Operation	Path
Explorer.EXE	RegOpenKey	HKLM₩System₩CurrentControlSet₩Control₩Session Manager
Explorer.EXE	RegCloseKey	HKLM₩System₩CurrentControlSet₩Control₩Session Manager
Explorer.EXE	RegOpenKey	HKLM₩System₩CurrentControlSet₩Control₩Session Manager
Explorer.EXE	RegCloseKey	HKLM₩System₩CurrentControlSet₩Control₩Session Manager
Explorer.EXE	RegOpenKey	HKLM₩System₩CurrentControlSet₩Control₩Session Manager
Explorer.EXE	RegCloseKey	HKLM₩System₩CurrentControlSet₩Control₩Session Manager

FIGURE 10. Outer redundancy of RegOpenKey - RegCloseKey.

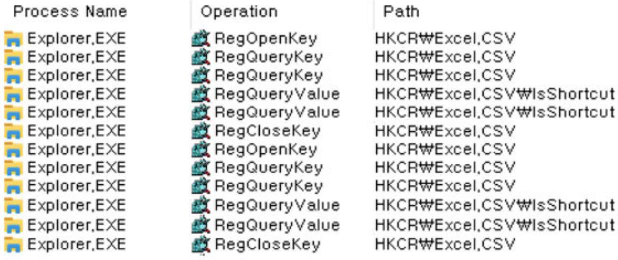
We will introduce three cases for the outer redundancy. Fig. 10 represents the simplest case of the outer redundancy. It is a redundant access pattern of **RegOpenKey - RegCloseKey**. Even if there are no registry operations between **RegOpenKey** and **RegCloseKey**, we observe that **RegOpenKey** and **RegCloseKey** are repeated in many times, which may be incurred from careless programming without considering the efficiency of accessing to the registry. Here, we can eliminate the redundancy by performing the first **RegOpenKey** and the last **RegCloseKey**.

Fig. 11 represents another case of the outer redundancy where an access pattern of **RegOpenKey - RegQueryValue - RegCloseKey** is redundant. This is similar to the previous case, but a read operation **RegQueryValue** is called between **RegOpenKey** and **RegCloseKey**, which is a more general



Process Name	Operation	Path
Explorer.EXE	RegOpenKey	HKCU₩Software₩Microsoft₩OneDrive₩Accounts
Explorer.EXE	RegQueryValue	HKCU₩Software₩Microsoft₩OneDrive₩Accounts₩LastUpdate
Explorer.EXE	RegCloseKey	HKCU₩Software₩Microsoft₩OneDrive₩Accounts
Explorer.EXE	RegQueryKey	HKLM
Explorer.EXE	RegOpenKey	HKCU₩Software₩Microsoft₩OneDrive₩Accounts
Explorer.EXE	RegQueryValue	HKCU₩Software₩Microsoft₩OneDrive₩Accounts₩LastUpdate
Explorer.EXE	RegCloseKey	HKCU₩Software₩Microsoft₩OneDrive₩Accounts

FIGURE 11. Outer redundancy of RegOpenKey - RegQueryValue - RegCloseKey.



Process Name	Operation	Path
Explorer,EXE	RegOpenKey	HKCR₩Excel,CSV
Explorer,EXE	RegQueryKey	HKCR₩Excel,CSV
Explorer,EXE	RegQueryKey	HKCR₩Excel,CSV
Explorer,EXE	RegQueryValue	HKCR₩Excel,CSV₩IsShortcut
Explorer,EXE	RegQueryValue	HKCR₩Excel,CSV₩IsShortcut
Explorer,EXE	RegCloseKey	HKCR₩Excel,CSV
Explorer,EXE	RegOpenKey	HKCR₩Excel,CSV
Explorer,EXE	RegQueryKey	HKCR₩Excel,CSV
Explorer,EXE	RegQueryKey	HKCR₩Excel,CSV
Explorer,EXE	RegQueryValue	HKCR₩Excel,CSV₩IsShortcut
Explorer,EXE	RegQueryValue	HKCR₩Excel,CSV₩IsShortcut
Explorer,EXE	RegCloseKey	HKCR₩Excel,CSV

FIGURE 12. Outer redundancy of RegOpenKey - RegQueryKey - RegQueryKey - RegQueryValue - RegQueryValue - RegCloseKey.

case to access to the registry. Here, we can eliminate the redundancy by repeating only **RegQueryValue** while performing the first **RegOpenKey** and the last **RegCloseKey**.

Fig. 12 represents the last case of the outer redundancy where an access pattern **RegOpenKey - RegQueryKey - RegQueryKey - RegQueryValue - RegQueryValue - RegCloseKey** is redundant. This is a quite complex pattern compared to the previous two patterns. Even this complex access pattern is redundant in many times. That is, according to the result of our redundancy detection algorithm presented in Section V, the number of redundancy for this pattern is 51,379.

When we detect the redundancy in access patterns, we need to consider very complex cases including the case as in Fig. 12. All the possible cases of the access patterns between **RegOpenKey** and **RegCloseKey** can be obtained by Eq. (1) if we simply assume that at most one operation could appear in the pattern. According to Table 4, 15 operations have been observed between **RegOpenKey** and **RegCloseKey**, i.e., $n = 15$ in Eq. (1). This implies that it is impossible to manually detect all the redundant access patterns. Consequently, we need an automatic method to detect the redundancy completely including even complex cases.

$$\sum_{i=1}^{n} P(n, i) \tag{1}$$

3) DISCUSSION TO WRITE OPERATIONS

We can also observe the redundant patterns on write operations to the registry such as **RegCreateKey** and **RegSetInfoKey** even if their portions are not significant. We can consider its redundancy similarly. However, we exclude the case involving the write operations because the elimination of its redundancy may incur the side effects in the internal redundancy. Specifically, even the same registry operation may change the registry state into different one. For example, two operations of **RegSetInfoKey** may change the key information into a different key.

However, in the outer redundancy, we allow the write operations between **RegOpenKey** and **RegCloseKey** when we eliminate the redundancy because our targets to eliminate the redundancy are **RegOpenKey** and **RegCloseKey**, not the write operations, which will be explained in Section IV.

C. VERIFICATION OF THE REDUNDANT ACCESS PATTERNS

In this section, we verify the redundant access patterns identified in Section III-B by disassembling the binary of the Windows program to check if they actually exist in the level of assembly codes. To disassemble the binary of the Windows application, we use IDA [11], which is a representative static analysis disassembler for binaries. We choose Explorer.exe as a Windows application to disassemble, which has been used to show redundant access patterns in Section III-B.

Fig. 13 introduces two assembly codes having redundant access patterns that have been identified in Section III-B. Fig.13 (a) is assembly codes corresponding to the internal redundancy in Fig. 8. That is, the access pattern **RegQueryValue** is redundant for the registry key "SYSTEM\Setup". Fig. 13 (b) is assembly codes corresponding to the outer redundancy in Fig. 11. That is, the access pattern **RegOpenKey - RegQueryValue - RegCloseKey** is redundant for the registry key "SOFTWARE\Microsoft\Windows\CurrentVersion\Policies."

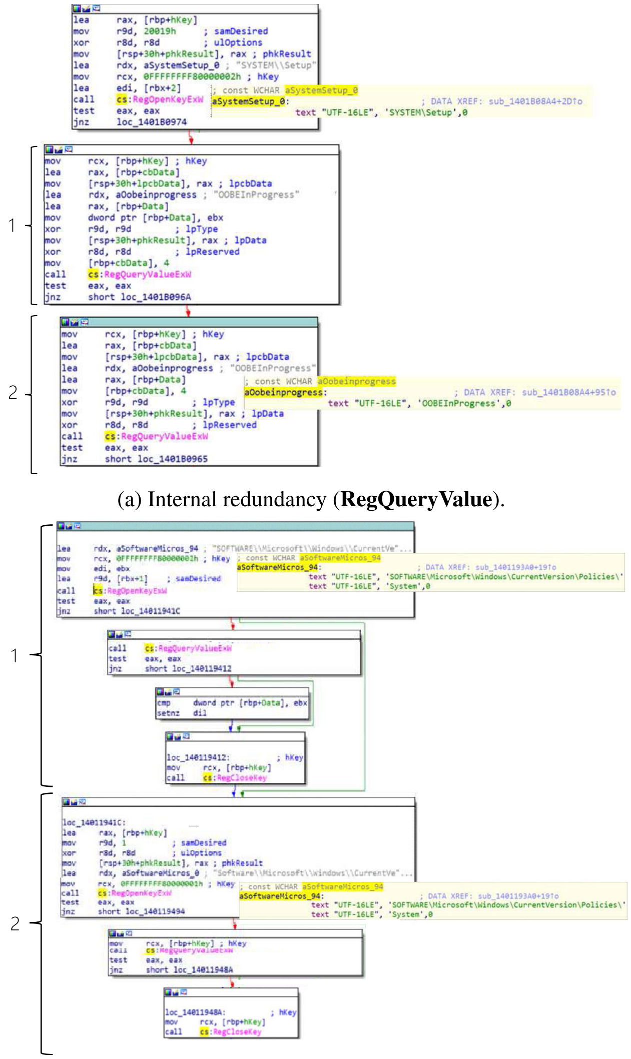
IV. REDUNDANCY DETECTION ALGORITHM

In this section, we propose the redundancy detection algorithm that automatically detects all the possible redundant access patterns from the large-scale log data. Redundancy is detected on the internal and outer duplication, respectively.

A. INTERNAL REDUNDANCY DETECTION

Fig. 14 shows an algorithm for detecting the internal redundancy. We have a function called, *data_preprocessing*, for preprocssing the entire log data collected from Process Monitor. In the function, we extract the associated registry operations for each process from the entire data. Here, we assign the identifier for each basic structure (simply, *BS_ID*) into each registry operation. Each basic structure is defined as a sequence of the registry operations starting from **RegOpenKey** and ending to **RegCloseKey** corresponding to **RegOpenKey**. We can identify the basic structure using *BS_ID* when we eliminate the redundancy. Each element of the result consists of (*BS_ID*, *Operation*, *Path*), which will be used for both the internal and outer redundancy.

The algorithm for detecting the internal redundancy is called by passing the log data for each process, which has been obtained by the *data_preprocessing* function, as the parameter. First, we find the basic structure. In each basic structure, we identify the internal redundancy and count the number of the redundancy. Here, we distinguish the paths that are accessed by the registry operation. That is, we need to



```
lea     rax, [rbp+hKey]
mov     r9d, 20019h         ; samDesired
xor     r8d, r8d            ; ulOptions
mov     [rsp+30h+phkResult], rax ; phkResult
lea     rdx, aSystemSetup_0 ; "SYSTEM\\Setup"
mov     rcx, 0FFFFFFFF80000002h ; hKey
lea     edi, [rbx+2]        ; const WCHAR aSystemSetup_0
call    cs:RegOpenKeyExW    aSystemSetup_0:                 ; DATA XREF: sub_1401B08A4+2D↑o
test    eax, eax                            text "UTF-16LE", 'SYSTEM\Setup',0
jnz     loc_1401B0974

mov     rcx, [rbp+hKey] ; hKey
lea     rax, [rbp+cbData]
mov     [rsp+30h+lpcbData], rax ; lpcbData
lea     rdx, aOobeinprogress ; "OOBEInProgress"
lea     rax, [rbp+Data]
mov     dword ptr [rbp+Data], ebx
xor     r9d, r9d            ; lpType
mov     [rsp+30h+phkResult], rax ; lpData
xor     r8d, r8d            ; lpReserved
mov     [rbp+cbData], 4
call    cs:RegQueryValueExW
test    eax, eax
jnz     short loc_1401B096A

mov     rcx, [rbp+hKey] ; hKey
lea     rax, [rbp+cbData]
mov     [rsp+30h+lpcbData], rax ; lpcbData
lea     rdx, aOobeinprogress ; "OOBEInProgress"
lea     rax, [rbp+Data]     ; const WCHAR aOobeinprogress
mov     [rbp+cbData], 4     aOobeinprogress:                ; DATA XREF: sub_1401B08A4+95↑o
xor     r9d, r9d            ; lpType    text "UTF-16LE", 'OOBEInProgress',0
mov     [rsp+30h+phkResult], rax ; lpData
xor     r8d, r8d            ; lpReserved
call    cs:RegQueryValueExW
test    eax, eax
jnz     short loc_1401B0965
```

(a) Internal redundancy (**RegQueryValue**).

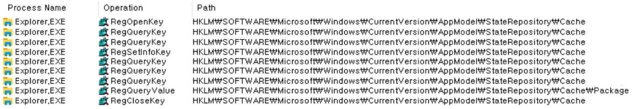
```
lea     rdx, aSoftwareMicros_94 ; "SOFTWARE\\Microsoft\\Windows\\CurrentVe"...
mov     rcx, 0FFFFFFFF80000002h ; hKey  ; const WCHAR aSoftwareMicros_94
mov     edi, ebx                aSoftwareMicros_94:             ; DATA XREF: sub_1401193A0+19↑o
lea     r9d, [rbx+1]    ; samDesired    text "UTF-16LE", 'SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\'
call    cs:RegOpenKeyExW                text "UTF-16LE", 'System',0
test    eax, eax
jnz     short loc_14011941C

call    cs:RegQueryValueExW
test    eax, eax
jnz     short loc_140119412

cmp     dword ptr [rbp+Data], ebx
setnz   dil

loc_140119412:                  ; hKey
mov     rcx, [rbp+hKey]
call    cs:RegCloseKey

loc_14011941C:
lea     rax, [rbp+hKey]
mov     r9d, 1              ; samDesired
xor     r8d, r8d            ; ulOptions
mov     [rsp+30h+phkResult], rax ; phkResult
lea     rdx, aSoftwareMicros_0 ; "Software\\Microsoft\\Windows\\CurrentVe"...
mov     rcx, 0FFFFFFFF80000001h ; hKey  ; const WCHAR aSoftwareMicros_94
call    cs:RegOpenKeyExW        aSoftwareMicros_94:             ; DATA XREF: sub_1401193A0+19↑o
test    eax, eax                        text "UTF-16LE", 'SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\'
jnz     short loc_140119494             text "UTF-16LE", 'System',0

mov     rcx, [rbp+hKey] ; hKey
call    cs:RegQueryValueExW
test    eax, eax
jnz     short loc_14011948A

loc_14011948A:                  ; hKey
mov     rcx, [rbp+hKey]
call    cs:RegCloseKey
```

(b) Outer redundancy (**RegOpenKey – RegQueryValue – RegCloseKey**).

FIGURE 13. Verification of the redundant access patterns in the level of assembly codes.

```
Internal redundancy detection
1  def data_preprocessing (process) :
2      data = entire log data collected from ProcessMonitor
3      process_data = [] // each element consists of (BS_ID, Operation, Path)
4      BS_ID = 0
5      for i in range(0, len(data) ) :
6          if (data[i]['ProcessName'] == process) :
7              if (data[i]['Operation'] == 'RegOpenKey') :
8                  j = i
9                  while ( ! (data[j]['Operation'] == 'RegCloseKey'
                          and data[j]['Path'] == data[i]['Path']) ) :
10                     process_data.append((BS_ID, data[j]['Operation'], data[j]['Path']))
11                     j = j + 1
12         BS_ID = BS_ID +1
13     return process_data
14
15 def detect_internal_redundancy (process_data) :
16     internal_redundant_patterns = {}
17     i = 0
18     while ( i < len(process_data) ) :
19         if (process_data[i]['Operation'] == 'RegOpenKey') :
20             j = i
21             while ( ! (process_data[j]['Operation'] == 'RegCloseKey'
                       and process_data[j]['Path'] == process_data[i]['Path'] )
                       and process_data[j]['Operation'] != 'Reg_WriteOperations') :
22                 if (process_data[i]['Path'] in process_data[j]['Path']):
23                     internal_pattern = (process_data[j]['BS_ID'], process_data[j]['Operation'],
                                           process_data[j]['Path'])
24                     if (process_data[j] not in internal_redundant_patterns.keys()) :
25                         internal_redundant_patterns.append(internal_pattern, 1)
26                     else :
27                         (internal_pattern, count) = internal_redundant_patterns.search(internal_pattern)
28                         internal_redundant_patterns.update(internal_pattern, count+1)
29                 j = j + 1
30         i = i + 1
31     return internal_redundant_patterns
```

FIGURE 14. Internal redundancy detection algorithm.



Process Name	Operation	Path
Explorer.EXE	RegOpenKey	HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩AppModel₩StateRepository₩Cache
Explorer.EXE	RegQueryKey	HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩AppModel₩StateRepository₩Cache
Explorer.EXE	RegQueryKey	HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩AppModel₩StateRepository₩Cache
Explorer.EXE	RegSetInfoKey	HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩AppModel₩StateRepository₩Cache
Explorer.EXE	RegQueryKey	HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩AppModel₩StateRepository₩Cache
Explorer.EXE	RegQueryKey	HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩AppModel₩StateRepository₩Cache
Explorer.EXE	RegQueryKey	HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩AppModel₩StateRepository₩Cache
Explorer.EXE	RegQueryValue	HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩AppModel₩StateRepository₩Cache₩Package
Explorer.EXE	RegCloseKey	HKLM₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩AppModel₩StateRepository₩Cache

FIGURE 15. The case where both the read and write operations are combined.

differentiate the same registry operation accessing to different registry paths. For each identified pattern (i.e., a registry operation), we obtain (*BS_ID*, *Operation*, *Path*, *Count*). Fig. 7, Fig. 8, and Fig. 9 are the examples for the internal redundancy where the patterns are represented in a consecutive way or an inconsecutive way. Using this detection algorithm, we can detect any patterns including those cases.

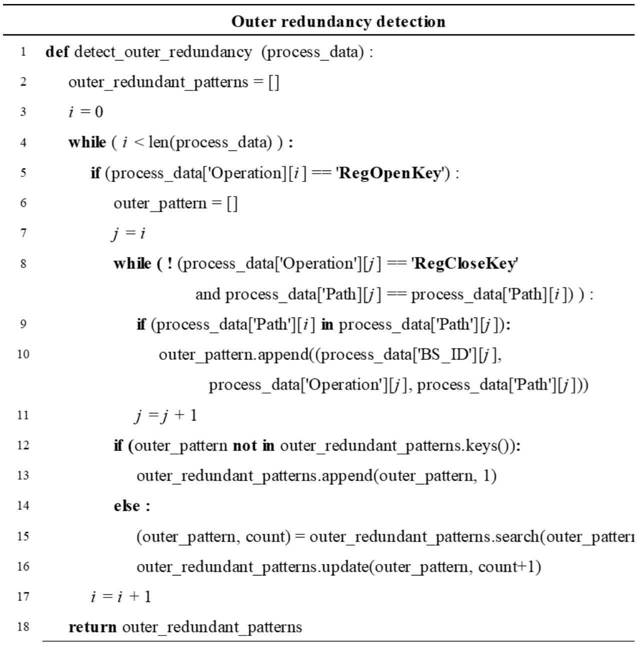
As discussed in Section III-B, we do not consider the write operations in the internal redundancy. When both read operations and write operations are combined in the pattern, the read operations after the write operations cannot be removed because the write operations could affect the keys or values for the read operations. Fig. 15 shows the example of this case. In this example, **RegQueryKey** is repeated between **RegOpenKey** and **RegCloseKey**. Here, the registry key is updated by **RegSetInfoKey** while **RegQueryKey** is performed. In this case, **RegQueryKey** after **RegSetInfoKey** is necessary because the key updated by **RegSetInfoKey** should be reloaded by **RegQueryKey**. But, to maximize the effect of eliminating the redundancy, we detect the redundant read patterns before the write operations appear. In Line 21 of the algorithm, we have the condition for this.

B. OUTER REDUNDANCY DETECTION

Fig. 16 shows an algorithm for detecting the outer redundancy. The basic logic of the outer redundancy detection is similar to that of the internal redundancy detection. The main difference is that we define the pattern for the outer redundancy as a sequence of all the operations between **RegOpenKey** and **RegCloseKey** as in Lines 8∼11 and check if the defined patterns are redundant as in Lines 12∼16. Fig. 10, Fig. 11, and Fig. 12 are the examples of this case. Using this algorithm, we can detect any complex patterns including those examples.

TABLE 5. All the access patterns having the internal redundancy (Occurrences).

Patterns	Total (Portion)	Consecutive redundancy	Inconsecutive redundancy	Total redundancy	Portion of redundancy
RegOpenKey	3,398,806 (29.65%)	83,922	1,813,408	1,897,330	55.82%
RegQueryKey	2,957,266 (25.80%)	669,249	1,666,708	2,335,957	78.99%
RegQueryValue	2,127,163 (18.56%)	332,372	764,171	1,096,543	51.55%
RegCloseKey	1,878,538 (16.39%)	4,950	937,221	942,171	50.15%
RegEnumValue	337,098 (2.94%)	269,906	45,535	315,441	93.58%
RegEnumKey	229,246 (2.00%)	84,882	113,461	198,343	86.52%
RegQueryKeySecurity	2,906 (0.03%)	1,189	774	1,963	67.55%
RegLoadKey	705 (0.01%)	12	192	204	28.94%
RegQueryMultipleValueKey	383 (0.00%)	94	134	228	59.53%
Others (Write Operations)	531,581 (4.64%)	0	0	0	0.00%
Total	11,463,692	1,446,576	5,341,604	6,788,180	59.21%



Outer redundancy detection

```
def detect_outer_redundancy (process_data) :
    outer_redundant_patterns = []
    $i$ = 0
    while ( $i$ < len(process_data) ) :
        if (process_data['Operation'][$i$] == 'RegOpenKey') :
            outer_pattern = []
            $j$ = $i$
            while ( ! (process_data['Operation'][$j$] == 'RegCloseKey'
                        and process_data['Path'][$j$] == process_data['Path'][$i$]) ) :
                if (process_data['Path'][$i$] in process_data['Path'][$j$]):
                    outer_pattern.append((process_data['BS_ID'][$j$],
                        process_data['Operation'][$j$], process_data['Path'][$j$]))
                $j$ = $j$ + 1
            if (outer_pattern not in outer_redundant_patterns.keys()):
                outer_redundant_patterns.append(outer_pattern, 1)
            else :
                (outer_pattern, count) = outer_redundant_patterns.search(outer_patter
                outer_redundant_patterns.update(outer_pattern, count+1)
        $i$ = $i$ + 1
    return outer_redundant_patterns
```

FIGURE 16. Outer redundancy detection algorithm.

C. ANALYSIS ON THE REDUNDANCY DETECTION

Now, we summarize the results of the redundancy detection algorithm. Table 5 shows the results of all the access patterns having the internal redundancy. Due to the redundancy detection algorithm, we can efficiently and effectively detect the redundancy from large-scale log data. We note that all the patterns we have observed in Section III-B, i.e., **RegQueryKey** and **RegQueryValue** are included in the detected patterns. In addition, we find all the other patterns that have the internal redundancy. To see the effects of both consecutive and inconsecutive redundancy, we count them separately. As shown in the table, the patterns with the high occurrences are **RegOpenKey**, **RegQueryKey**, **RegQueryValue**, and **RegCloseKey**, which occupy about 90.40%. In addition, we note that they have many redundancies of 50.15%∼78.99%. The total internal redundancy for all the patterns is about 59.21%. We can eliminate them only if we consider the case where the updates to the registry are occurred by the other processes. In Section V-C, we will discuss it and present the method to remove the side effects completely according to the redundancy elimination.

Table 6 shows the top-10 access patterns having the outer redundancy. In the detected patterns, all the patterns we have observed in Section III-B such as **RegOpenKey - RegQueryValue - RegCloseKey** and **RegOpenKey - RegQueryKey - RegCloseKey** are included. In addition, we find all the other patterns that have the outer redundancy. We indicate that many various patterns are detected because the others except for the top-10 access patterns still occupy about 22.21%. As shown in the table, the top-3 patterns with the high occurrences are **RegOpenKey - RegSetInfoKey - RegQueryValue - RegCloseKey**, **RegOpenKey - RegCloseKey**, and **RegOpenKey - RegQueryValue - RegCloseKey**, which occupy about 43.48%. In addition, we note that they have many redundant patterns of 43.43%∼69.19%. The total redundancy for all the patterns is about 57.50%. We can effectively eliminate the redundancy without the side effects by eliminating the redundant pattern of **RegOpenKey** and **RegCloseKey** while we remain the registry operations between **RegOpenKey** and **RegCloseKey**.

Table 7 shows the analyzed result for the top-10 processes frequently accessing to the registry to check the I/O operation portion by the process. The table shows the portion by the I/O operation type of the processes. The result indicates that I/O operations are concentrated on the registry and file system in terms of both occurrences and duration; occurrences and duration for network and process are negligible. The occurrences for the registry operation occupy 35.83% ∼ 96.55%, and 76.92% on average; the duration for the registry 7.11% ∼ 79.54%, and 32.06% on average. The portion of redundancy is 47.70% ∼ 82.41%, and 71.49% on average. As a result, we conclude that we can improve the performance of the process by eliminating the redundant accesses to the registry.

V. REDUNDANCY ELIMINATION ON THE ACCESS PATTERNS TO THE WINDOWS REGISTRY

A. TWO-LEVEL REDUNDANCY ELIMINATION METHOD

In this section, we propose a two-level redundancy elimination method for eliminating redundant and unnecessary

TABLE 6. Top-10 access patterns having the outer redundancy (Occurrences).

Pattern No.	Patterns	Total (Portion)	Redundancy	Portion of redundancy
$Outer_1$	**RegOpenKey-RegSetInfoKey-RegQueryValue-RegCloseKey**	181,253 (15.08%)	105,425	58.16%
$Outer_2$	**RegOpenKey-RegCloseKey**	181,178 (15.07%)	125,353	69.19%
$Outer_3$	**RegOpenKey-RegQueryValue-RegCloseKey**	160,261 (13.33%)	69,600	43.43%
$Outer_4$	**RegOpenKey-RegQueryKey-RegOpenKey-RegQueryValue-RegCloseKey**	107,535 (8.95%)	82,468	76.69%
$Outer_5$	**RegOpenKey-RegOpenKey-RegCloseKey**	85,109 (7.08%)	64,090	75.30%
$Outer_6$	**RegOpenKey-RegQueryKey-RegQueryKey-RegQueryValue-RegQueryValue-RegCloseKey**	66,781 (5.56%)	51,379	76.94%
$Outer_7$	**RegOpenKey-RegQueryKey-RegOpenKey-RegCloseKey**	58.197 (4.84%)	48,241	82.89%
$Outer_8$	**RegOpenKey-RegQueryValue-RegQueryValue-RegCloseKey**	56,020 (4.66%)	2,565	4.58%
$Outer_9$	**RegOpenKey-RegSetInfoKey-RegCloseKey**	19,943 (1.66%)	14,610	73.26%
$Outer_1 0$	**RegOpenKey-RegQueryKey-RegQueryKey-RegCloseKey**	18,820 (1.57%)	6,326	33.61%
Others		267,052 (22.21%)	121,158	45.37%
Total		1,202,149	691,215	57.50%

TABLE 7. Analysis on I/O operation portion for the top-10 processes frequently accessing to the registry.

Processes	Registry			File system		Network		Process	
	Occurrences (Portion)	Duration (Portion)	Redundancy (Portion)	Occurrences (Portion)	Duration (Portion)	Occurrences (Portion)	Duration (Portion)	Occurrences (Portion)	Duration (Portion)
trustedinstaller.exe	2,831,412 (72.63%)	203.58 secs (26.65%)	1,780,305 (62.88%)	1,067,219 (27.37%)	160.29 secs (73.35%)	0 (0.00%)	0 secs (0.00%)	21 (0.00%)	0.00 secs (0.00%)
explorer.exe	2,123,175 (87.39%)	152.66 secs (48.78%)	1,698,110 (79.98%)	305,209 (12.56%)	160.29 (51.22%)	253 (0.01%)	0.00 (0.00%)	912 (0.04%)	0.00 (0.00%)
iexplorer.exe	1,687,270 (75.21%)	121.32 secs (32.86%)	1,390,553 (82.41%)	472,062 (21.04%)	247.91 secs (67.14%)	81,720 (3.64%)	0.00009 secs (0.00%)	2,302 (0.10%)	0.00004 secs (0.00%)
excel.exe	830,263 (84.19%)	121.32 secs (32.86%)	607,116 (73.12%)	153,792 (15.59%)	80.77 secs (57.50%)	1,168 (0.12%)	0.00 secs (0.00%)	1,003 (0.10%)	0.00001 secs (0.00%)
winword.exe	694,664 (71.68%)	49.95 secs (25.83%)	518,767 (74.68%)	273,044 (28.17%)	143.39 secs (74.17%)	735 (0.08%)	0.00 secs (0.00%)	665 (0.07%)	0.00001 secs (0.00%)
powerpnt.exe	694,075 (82.80%)	49.90 secs (40.04%)	519,134 (74.80%)	142,322 (16,98%)	74.74 secs (59.96%)	1,063 (0.13%)	0.00 secs (0.00%)	810 (0.10%)	0.00001 secs (0.00%)
svchost.exe	563,521 (78.14%)	40.52 secs (34.05%)	272,043 (48.34%)	149,403 (20.72%)	78.46 secs (65.95%)	6,012 (0.83%)	0.00001 secs (0.00%)	2,189 (0.30%)	0.00004 secs (0.00%)
mspaint.exe	219,108 (89.78%)	15.75 secs (54.90%)	171,557 (78.30%)	24,646 (10.10%)	12.94 secs (45.10%)	0	0.00 secs (0.00%)	302 (0.12%)	0.00001 secs (0.00%)
sihost.exe	176,700 (96.55%)	12.70 secs (79.54%)	102,697 (58.12%)	6,222 (3.40%)	3.27 secs (20.46%)	0 (0.00%)	0.00 secs (0.00%)	87 (0.05%)	0.00 secs (0.00%)
msmpeng.exe	170,113 (35.83%)	12.23 secs (7.11%)	81,151 (47.70%)	304,474 (64.13%)	159.90 secs (92.89%)	23 (0.00%)	0.00 secs (0.00%)	181 (0.04%)	0.00 secs (0.00%)

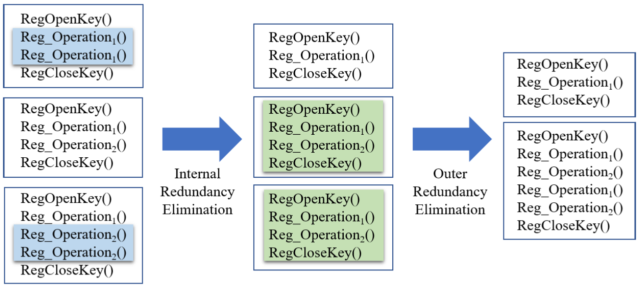
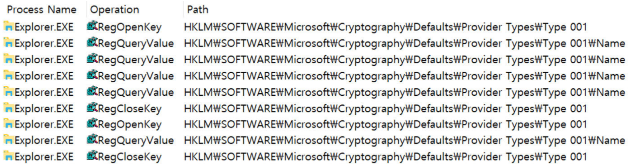


FIGURE 17. The concept of the two-level redundancy elimination method.

access patterns effectively. Fig. 17 shows the concept of the two-level redundancy elimination method. The basic idea is applying the redundancy elimination in two-level: 1) to the internal redundancy (i.e., internal redundancy elimination) and 2) to the outer redundancy (i.e., outer redundancy elimination).

In the two-level redundancy elimination method, the order of applying the internal and outer redundancy elimination is important because each one affects the other. To enhance the effect of the redundancy elimination, we first apply the internal redundancy elimination, and then, the outer redundancy elimination based on the result of the internal redundancy elimination. We call this order of redundancy elimination *internal-then-outer redundancy elimination*. This stems from the fact that the internal redundancy elimination converges the multiple different original patterns into the same pattern. As a result, we can enhance the effect of the outer redundancy elimination. Fig. 18 represents the actual example that shows this effect. Here, we have two different patterns **RegOpenKey - RegQueryValue - RegQueryValue - RegQueryValue - RegQueryValue - RegCloseKey** and **RegOpenKey - RegQueryValue - RegCloseKey**. If we apply the internal redundancy elimination to the first pattern, then it becomes the same pattern as the second one. Then, we can eliminate the outer redundancy.



```
Process Name   Operation       Path
Explorer.EXE   RegOpenKey      HKLM₩SOFTWARE₩Microsoft₩Cryptography₩Defaults₩Provider Types₩Type 001
Explorer.EXE   RegQueryValue   HKLM₩SOFTWARE₩Microsoft₩Cryptography₩Defaults₩Provider Types₩Type 001₩Name
Explorer.EXE   RegQueryValue   HKLM₩SOFTWARE₩Microsoft₩Cryptography₩Defaults₩Provider Types₩Type 001₩Name
Explorer.EXE   RegQueryValue   HKLM₩SOFTWARE₩Microsoft₩Cryptography₩Defaults₩Provider Types₩Type 001₩Name
Explorer.EXE   RegQueryValue   HKLM₩SOFTWARE₩Microsoft₩Cryptography₩Defaults₩Provider Types₩Type 001₩Name
Explorer.EXE   RegCloseKey     HKLM₩SOFTWARE₩Microsoft₩Cryptography₩Defaults₩Provider Types₩Type 001
Explorer.EXE   RegOpenKey      HKLM₩SOFTWARE₩Microsoft₩Cryptography₩Defaults₩Provider Types₩Type 001
Explorer.EXE   RegQueryValue   HKLM₩SOFTWARE₩Microsoft₩Cryptography₩Defaults₩Provider Types₩Type 001₩Name
Explorer.EXE   RegCloseKey     HKLM₩SOFTWARE₩Microsoft₩Cryptography₩Defaults₩Provider Types₩Type 001
```

FIGURE 18. The example of the improved effect of the redundancy elimination due to the internal-then-outer redundancy elimination.

Original pattern

```
// Perform both registry operations and
// main operation for n times
RegOpenKey()

for i in n :
    Reg_Operations()
    MainOperations()

RegCloseKey()
```

Internal redundancy elimination

```
// Perform registry operations once
RegOpenKey()
Reg_Operations()

// Perform only main operations for n times
for i in n :
    MainOperations()

RegCloseKey()
```

FIGURE 19. The algorithm of the internal redundancy elimination.

From now on, let us explain the internal and outer redundancy elimination algorithm, respectively. Fig. 19 shows the algorithm of the internal redundancy elimination. Here, we define the registry operations in Table 5 (e.g., **RegQueryKey** or **RegEnumKey**) as **Reg_Operations**. We can easily eliminate redundancy on **Reg_Operations** by performing **Reg_Operations** once. This redundancy elimination will not incur any side effects if updates on the registry are not occurred by other processes while main operations are repeated. However, if updates are occurred on the registry, this could make a different result from the original access pattern. We will discuss this case in Section V-C and present a method that eliminates side effects completely.

Original pattern

```
// Perform both registry operations and
// main operation for n times

for i in n:
    RegOpenKey()
    Reg_Operations()
    MainOperations()
    RegCloseKey()
```

Outer redundancy elimination

```
// Perform RegOpenKey once
RegOpenKey()

// Perform Reg_Operations and
// main operations for n times
for i in n :
    Reg_Operations()
    MainOperations()

// Perform RegCloseKey once
RegCloseKey()
```

FIGURE 20. The algorithm of the outer redundancy elimination.

Fig. 20 shows the algorithm of the outer redundancy elimination. Here, we define a sequence of the patterns in Table 6 as **Reg_Operations**. As shown in Section III-B, **RegOpenKey** and **RegCloseKey** are not necessary to be repeated. Therefore, we can remove all the repeated **RegOpenKey** operations except for the first one and all the repeated **RegCloseKey** except for the last one. As presented in Fig. 20, all the registry operations are repeated in the original access patterns while **RegOpenKey** and **RegCloseKey** are performed once in the outer redundancy elimination. Because **RegOpenKey** and **RegCloseKey** occupy much portion as presented in Table 4 (i.e., 61.79% in the duration), we can significantly improve the performance of the program by this redundancy elimination strategy while it does not incur side effects at all.

TABLE 8. The effect of the internal redundancy elimination (Occurrences).

Patterns	Original pattern	Unique pattern	Portion of redundancy
RegOpenKey	3,398,806	1,501,476	55.82%
RegQueryKey	2,957,266	621,309	78.99%
RegQueryValue	2,127,163	1,030,620	51.55%
RegCloseKey	1,878,538	936,367	50.15%
RegEnumValue	337,098	21,657	93.58%
RegEnumKey	229,246	30,903	86.52%
RegQueryKey-Security	2,906	943	67.55%
RegLoadKey	705	501	28.94%
RegQuery-MultipleValueKey	383	155	59.53%
Others (Write Operations)	531,581	531,581	0.00%
Total	11,463,692	4,675,512	59.21%

B. ANALYZING THE EFFECT OF THE TWO-LEVEL REDUNDANCY ELIMINATION METHOD

In this section, we analyze the effect of the two-level redundancy elimination method. Table 8 shows the effect of the internal redundancy elimination. It shows the occurrences of original patterns having redundancy, occurrences of unique patterns, and the portion of the redundancy. From the portion of the redundancy for each pattern, we know the effect of the redundancy elimination. For instance, in the case of **RegOpenKey**, the occurrences in the original pattern with the redundancy were 3,398,806. They can be reduced into 1,501,476 after the internal redundancy elimination. That is, by the redundancy elimination, we can remove the redundancy about 55.82%. In Table 8, we can obtain the effect of the internal redundancy elimination on all the patterns identified in Section IV. The overall effect of the internal redundancy elimination on all the patterns is about 59.21%.

Table 9 shows the effect of the outer redundancy elimination. It shows the occurrences of original patterns having redundancy, unique patterns, and the portion of the redundancy. For instance, in the case of $Outer_1$, the occurrences in the original pattern were 181,253. They could be reduced into 75,828 after the outer redundancy elimination. That is, by the redundancy elimination, we can remove the redundancy about 58.16%. In Table 9, we can obtain the information on all the patterns identified in Section IV. The overall effect of the outer redundancy elimination on all the patterns is about 57.50%.

Table 10 shows the effect of the internal-then-outer redundancy elimination. The internal redundancy elimination tends to convert the original pattern into a simplified one. Consequently, it enhances the effect of the outer redundancy elimination. For example, $Outer_{10}$, **RegOpenKey - RegQueryKey - RegQueryKey - RegCloseKey**, could be reduced into $Outer_5$, **RegOpenKey - RegQueryKey - RegCloseKey**. Then, those access patterns, which have been different originally, are converged into the same pattern, and

TABLE 9. The effect of the outer redundancy elimination (Occurrences).

Pattern No.	Original pattern	Unique pattern	Portion of redundancy
$Outer_1$	181,253	75,828	58.16%
$Outer_2$	181,178	55,825	69.19%
$Outer_3$	160,261	90,661	43.43%
$Outer_4$	107,535	25,067	76.69%
$Outer_5$	85,109	21,019	75.30%
$Outer_6$	66,781	15,402	76.94%
$Outer_7$	58,197	9,956	82.89%
$Outer_8$	56,020	53,455	4.58%
$Outer_9$	19,943	5,333	73.26%
$Outer_{10}$	18,820	12,494	45.37%
Others	267,052	145,894	45.37%
Total	1,202,149	510,934	57.50%

TABLE 10. The effect of the internal-then-outer redundancy elimination (Occurrences).

Pattern No.	Original pattern	Outer redundancy elimination	Internal-then-outer redundancy elimination	Improved effect of internal-then-outer redundancy
$Outer_1$	181,253	75,828	13,880	81.70%
$Outer_2$	181,178	55,825	26,738	52.10%
$Outer_3$	160,261	90,661	50,371	44.44%
$Outer_4$	107,535	25,067	1,504	94.00%
$Outer_5$	85,109	21,019	14,154	32.66%
$Outer_6$	66,781	15,402	19	99.88%
$Outer_7$	58,197	9,956	3,759	62.24%
$Outer_8$	56,020	53,455	8,167	84.72%
$Outer_9$	19,943	5,333	3,757	29.55%
$Outer_{10}$	18,820	12,494	1,167	90.66%
Others	267,052	145,894	110,795	24.06%
Total	1,202,149	510,934	234,311	54.14%

they can be eliminated from the outer redundancy. Overall, the internal-then-outer redundancy elimination improves the effect of the redundancy elimination by about 54.14%.

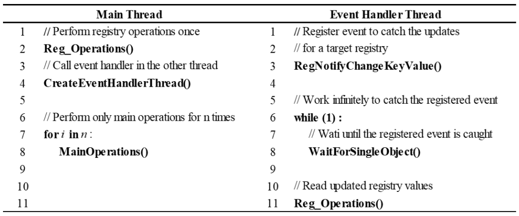
Table 11 shows the final effect of the two-level redundancy elimination method. It first eliminates the internal redundancy, which reduces the redundancy by about 59.21%; then it eliminates the outer redundancy, which reduces the redundancy additionally by about 23.81%. Here, we note that the outer redundancy elimination affects only two operations **RegOpenKey** and **RegCloseKey**. Overall, we can expect that 68.93% of redundancy is eliminated by the two-level redundancy elimination method.

C. EVENT-DRIVEN METHOD FOR CATCHING UPDATES ON THE WINDOWS REGISTRY

In Section III-B, we discussed the case where the redundancy elimination method could incur the side effect due to the updates of the registry. That is, the redundancy elimination method performs the registry operation once, and then, uses the retrieved value repeatedly in the main operations. Here, if the updates on the Windows registry occur after we retrieve the data from the registry, it could use out-of-date values. To prevent this case, we present an *event-driven method*. The basic idea is that we register an event handler to catch the updates on a target registry and read a newly updated value from the registry when the registered event occurs.

TABLE 11. The final effect of the two-level redundancy elimination (Occurrences).

Operations	Original pattern	Internal redundancy elimination	Outer redundancy elimination	Effect of two-level redundancy elimination
RegOpenKey	3,398,806	1,501,476	940,219	72.34%
RegQueryKey	2,957,266	621,309	621,309	78.99%
RegQueryValue	2,127,163	1,030,620	1,030,620	51.55%
RegCloseKey	1,878,538	936,367	384,443	79.53%
RegEnumValue	337,098	21,657	21,657	93.58%
RegEnumKey	229,246	30,903	30,903	86.52%
RegQueryKey-Security	2,906	943	943	67.55%
RegLoadKey	705	501	501	28.94%
RegQuery-MultipleValueKey	383	155	155	59.53%
Others (Write operations)	531,581	531,581	531,581	0.00%
Total	11,463,692	4,675,512	3,562,331	68.93%



Main Thread

```
// Perform registry operations once
Reg_Operations()
// Call event handler in the other thread
CreateEventHandlerThread()

// Perform only main operations for n times
for $i$ in $n$ :
    MainOperations()
```

Event Handler Thread

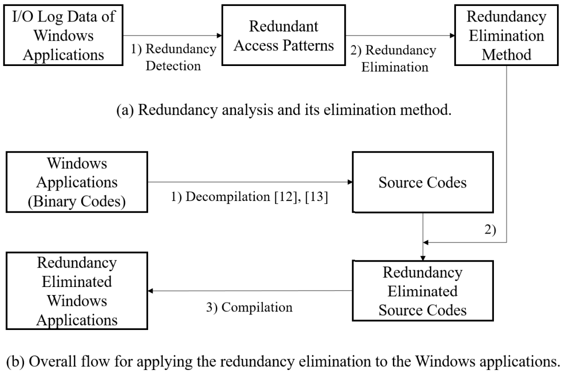
```
// Register event to catch the updates
// for a target registry
RegNotifyChangeKeyValue()

// Work infinitely to catch the registered event
while (1) :
    // Wati until the registered event is caught
    WaitForSingleObject()

// Read updated registry values
Reg_Operations()
```

FIGURE 21. The event-driven internal redundancy elimination.

Fig. 21 shows pseudo codes that apply the event-driven method when we eliminate the internal redundancy. It consists of two threads: 1) main thread and 2) event handler thread. The main thread performs registry operations once, which is the same as in the redundancy eliminated access patterns. Then, it calls an event handler that will catch the event whenever the target registry is updated. Last, it performs the main operations repeatedly. The event handler thread registers an event to catch the updates for a target registry. We use the **RegNotifyChangeKeyValue()**[2] as the event handler to catch the updates on the target registry. Then, it will work infinitely to catch the registered event. We note that, if the registered event is caught, we can read the updated registry value instantly. Specifically, while processing the main operations in the main thread, we read newly updated values by the event handler thread as soon as the updates are occurred in the target registry.

By using the event-driven method, we can completely remove the case where the programs read out-of-date values. It could occur to read out-of-date values even in the original access patterns with the internal redundancy, which read the registry keys or values repeatedly, due to the time difference between iterations in the loop. However, in the event-driven method, the programs can read newly updated values right after the updates are occurred on the registry. As a result, the event-driven method achieves 1) efficient processing due

[2] https://docs.microsoft.com/en-us/windows/desktop/api/winreg/nf-winreg-regnotifychangekeyvalue


I/O Log Data of Windows Applications
1) Redundancy Detection
Redundant Access Patterns
2) Redundancy Elimination
Redundancy Elimination Method
Windows Applications (Binary Codes)
1) Decompilation [12], [13]
Source Codes
2)
Redundancy Eliminated Windows Applications
3) Compilation
Redundancy Eliminated Source Codes

(a) Redundancy analysis and its elimination method.

(b) Overall flow for applying the redundancy elimination to the Windows applications.

FIGURE 22. Overall framework for redundancy analysis and the application of the redundancy elimination.

to the redundancy elimination and 2) guaranteeing to instantly read the newly updated data.

D. OVERALL FRAMEWORK

Fig. 22 shows the overall framework for redundancy analysis and the application of the redundancy elimination. Fig. 22 (a) shows redundancy analysis on access patterns. First, we detect the redundancy of access patterns from I/O log data of Windows applications. Next, to remove them without side effects, we propose the redundancy elimination method. We can apply the proposed redundancy elimination method into actual Windows applications by combining the existing work and the proposed redundancy elimination method. However, in this paper, we do not cover the application to the actual Windows applications because the automatic translation of the source code is out of the scope of this paper in which we focus on the redundancy analysis on access patterns and the redundancy elimination.

In this paper, we only present the overall flow for applying the proposed redundancy elimination into the actual Windows applications as shown in Fig. 22 (b). The whole process consists of three steps: 1) decompilation, 2) redundancy elimination, and 3) compilation. For applying the redundancy elimination, we first need to convert a given Windows application in the binary to the compilable source code. For this, we can adopt the existing methods that convert from the platform-dependent binary to the platform independent high-level source code [12], [13]. Next, we convert the high-level source code to the redundancy eliminated source code using the redundancy elimination method proposed in Section V-A and Section V-C. Finally, we generate the Windows application without redundancy in access patterns by compiling the redundancy eliminated source code.

VI. EXPERIMENTAL RESULTS

A. EXPERIMENTAL ENVIRONMENTS

In this section, we show the effectiveness of the redundancy elimination method by conducting three kinds of experiments. We measure the performance of the original access patterns and that of redundancy eliminated access patterns. First, we measure the performance of the program consisting of each individual pattern having the internal redundancy. We use the event-driven method that removes the side effect of the internal redundancy elimination. Here, we also measure the overhead of the event-driven method under the update intensive environments to the registry. Second, we measure the performance of the program consisting of each individual pattern having the outer redundancy. Here, we also measure the effect of the internal-then-outer redundancy elimination. Third, we measure the performance when the multiple programs having combined access patterns are running concurrently. This aims to simulate real Windows environments that run multiple programs simultaneously where each process accesses to the Windows registry.

For the first and second experiments, we determine n of the pseudo code for each pattern (i.e., Fig. 19, Fig. 20, and Fig. 21) based on analyzing of log data. Thus, we use the average number of redundancy on each pattern in Table 8 and Table 9 as n. We fix the number of operations to be performed for all the patterns because the patterns have different occurrences as shown in Table 8 and Table 9. For this fixed number, we use 10,000. In addition, we define the **MainOperations** as an operation that retrieves a registry value.

For the third experiment, we define a program to run all the patterns (i.e., 9 patterns) having the internal redundancy and top-10 patterns having the outer redundancy once and execute multiple programs concurrently. Then, as the number of the programs increases, we measure the performance of the program consisting of the original access patterns and that of the program consisting of the redundancy eliminated access patterns.

To implement each pattern, we use the APIs provided by MSDN (Microsoft Developer Network).[3] Our experiments are conducted on the machine running Windows 10 64bit, equipped with Intel Core i7-7820 @ 2.90 GHz CPU and 16GB RAM. All the source codes were implemented with C++ using Microsoft Visual Studio(MSVC) 2017.

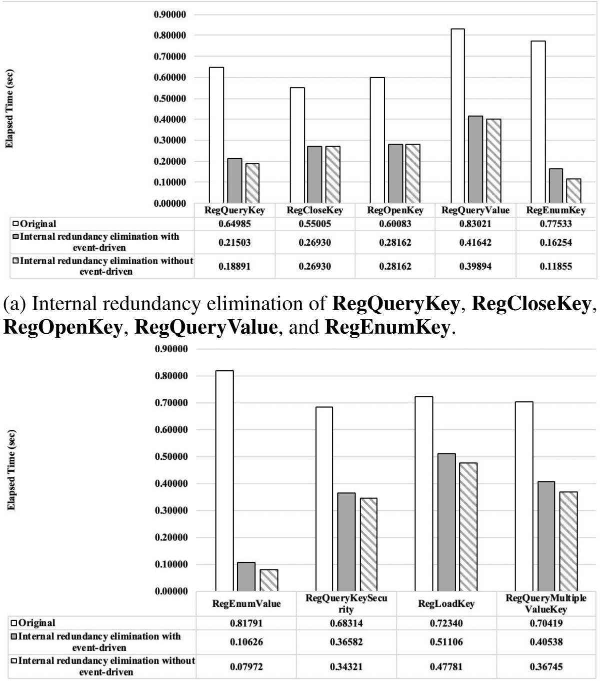
B. EXPERIMENTAL RESULTS

1) EXPERIMENTS FOR THE INTERNAL REDUNDANCY

a: THE RESULTS ON THE PATTERNS HAVING THE INTERNAL REDUNDANCY

Fig. 23 represents the comparison results between the original and internal redundancy eliminated patterns. For the internal redundancy elimination, we use the event-driven method to exclude side effects when the updates are occurred on the registry we are accessing. To check the overhead of the event-driven method, we compare the case where the event-driven method is not used and the case where the event-driven method is used. In the event-driven method, the time for registering an event handler is added, but it is negligible as shown in Fig. 23. Finally, the result shows that the internal redundancy elimination improves the performance of the original access patterns by 33.95% ∼ 90.25%. This stems

[3] https://msdn.microsoft.com



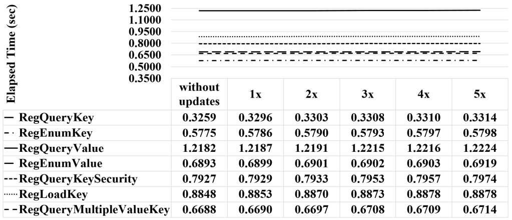
	RegQueryKey	RegCloseKey	RegOpenKey	RegQueryValue	RegEnumKey
□Original	0.64985	0.55005	0.60083	0.83021	0.77533
■Internal redundancy elimination with event-driven	0.21503	0.26930	0.28162	0.41642	0.16254
▨Internal redundancy elimination without event-driven	0.18891	0.26930	0.28162	0.39894	0.11855

(a) Internal redundancy elimination of **RegQueryKey**, **RegCloseKey**, **RegOpenKey**, **RegQueryValue**, and **RegEnumKey**.

	RegEnumValue	RegQueryKeySecurity	RegLoadKey	RegQueryMultipleValueKey
□Original	0.81791	0.68314	0.72340	0.70419
■Internal redundancy elimination with event-driven	0.10626	0.36582	0.51106	0.40538
▨Internal redundancy elimination without event-driven	0.07972	0.34321	0.47781	0.36745

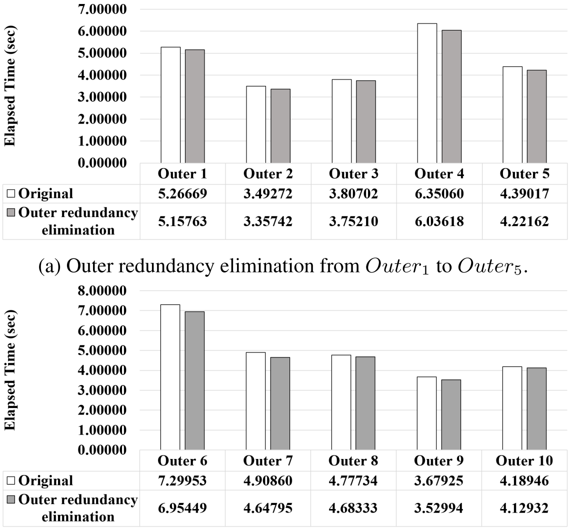
(b) Internal redundancy elimination of **RegEnumValue**, **RegQueryKeySecurity**, **RegLoadKey**, and **RegQueryMultipleValueKey**.

FIGURE 23. The comparison results of the access patterns in the internal redundancy elimination.



	without updates	1x	2x	3x	4x	5x
— RegQueryKey	0.3259	0.3296	0.3303	0.3308	0.3310	0.3314
-· RegEnumKey	0.5775	0.5786	0.5790	0.5793	0.5797	0.5798
—RegQueryValue	1.2182	1.2187	1.2191	1.2215	1.2216	1.2224
— RegEnumValue	0.6893	0.6899	0.6901	0.6902	0.6903	0.6919
---RegQueryKeySecurity	0.7927	0.7929	0.7933	0.7953	0.7957	0.7974
·····RegLoadKey	0.8848	0.8853	0.8870	0.8873	0.8878	0.8878
- -RegQueryMultipleValueKey	0.6688	0.6690	0.6697	0.6708	0.6709	0.6714

FIGURE 24. The performance evaluation of the event-driven method when update operations are occurred.



	Outer 1	Outer 2	Outer 3	Outer 4	Outer 5
□Original	5.26669	3.49272	3.80702	6.35060	4.39017
■Outer redundancy elimination	5.15763	3.35742	3.75210	6.03618	4.22162

(a) Outer redundancy elimination from $Outer_1$ to $Outer_5$.

	Outer 6	Outer 7	Outer 8	Outer 9	Outer 10
□Original	7.29953	4.90860	4.77734	3.67925	4.18946
■Outer redundancy elimination	6.95449	4.64795	4.68333	3.52994	4.12932

(b) Outer redundancy elimination from $Outer_6$ to $Outer_{10}$.

FIGURE 25. The comparison results of the access patterns in the outer redundancy elimination.

from the fact that we remove unnecessary repeated registry operations.

b: THE OVERHEAD OF THE EVENT-DRIVEN METHOD WHEN THE UPDATES ARE OCCURRED

In the previous experiment, we measure the overhead of the event-driven method, but actual updates for the registry are not occurred. To check the overhead when actual updates are occurred, we simulate the environments where the update operations are periodically repeated based on the statistics of write operations. Out of 9 patterns for the internal redundancy, we have 7 patterns that require the event-driven method: **RegQueryKey**, **RegEnumKey**, **RegQueryValue**, **RegEnumValue**, **RegQueryKeySecurity**, **RegLoadKey**, and **RegQueryMultipleValueKey**. The corresponding update operation for each operation is as follows: **RegSetInfoKey** for **RegQueryKey**, **RegEnumKey**, **RegQueryKeySecurity**, **RegLoadKey**, and **RegQueryMultipleValueKey**; **RegSetValue** for **RegQueryValue** and **RegEnumValue**. According to the statistics, the average frequency of **RegSetInfoKey** is 103.68ms, and **RegSetValue** is 2863.86ms. For the experiment, we run another separate process where the update operation is periodically executed with a given frequency. Here, we use the frequency according to the statistics as the base frequency and increase the base frequency up to five times to check the overhead in update intensive environments. Fig. 24 shows the elapsed time of the event-driven method as we increase the frequency of the update operation. We note that the performance degradation where the frequency is five times of the base frequency is only 0.38% $\sim$ 1.66% compared to the case where updates are not occurred at all. That is, even in the update intensive environments, the proposed event-driven method is still efficient while removing the side effects completely.

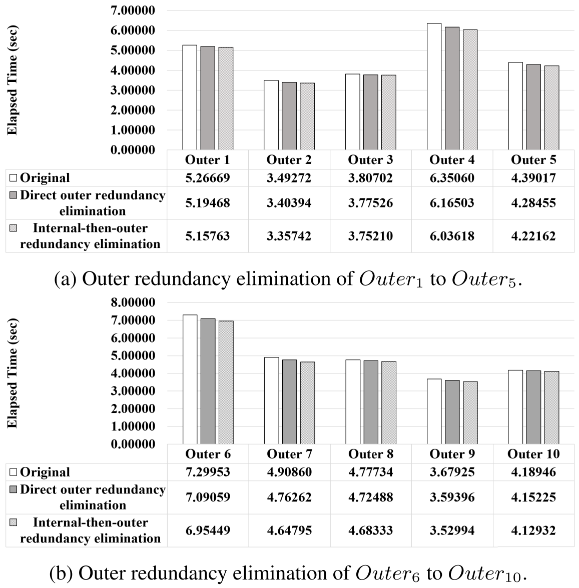
2) EXPERIMENTS FOR THE OUTER REDUNDANCY

a: THE RESULTS ON THE PATTERNS HAVING THE OUTER REDUNDANCY

Fig. 25 represents the comparison results between the original and outer redundancy eliminated access patterns. The result shows that the outer redundancy elimination method improves the performance of the original access patterns by 1.44% $\sim$ 5.31%. This stems from the fact that we remove unnecessary repeated registry operations of **RegOpenKey** and **RegCloseKey**.

b: THE EFFECT OF THE INTERNAL-THEN-OUTER REDUNDANCY ELIMINATION

In this experiment, to see the effect of the internal-then-outer redundancy elimination, we compare the result of applying the outer redundancy elimination directly into the original patterns and the result of applying the internal-then-outer redundancy elimination. Fig. 26 represents the results of



	Outer 1	Outer 2	Outer 3	Outer 4	Outer 5
□ Original	5.26669	3.49272	3.80702	6.35060	4.39017
▨ Direct outer redundancy elimination	5.19468	3.40394	3.77526	6.16503	4.28455
▨ Internal-then-outer redundancy elimination	5.15763	3.35742	3.75210	6.03618	4.22162

(a) Outer redundancy elimination of $Outer_1$ to $Outer_5$.

	Outer 6	Outer 7	Outer 8	Outer 9	Outer 10
□ Original	7.29953	4.90860	4.77734	3.67925	4.18946
▨ Direct outer redundancy elimination	7.09059	4.76262	4.72488	3.59396	4.15225
▨ Internal-then-outer redundancy elimination	6.95449	4.64795	4.68333	3.52994	4.12932

(b) Outer redundancy elimination of $Outer_6$ to $Outer_{10}$.

FIGURE 26. The comparison results to check the effect of applying the internal-then-outer redundancy elimination.

comparison of original, direct outer redundancy elimination, and the internal-then-outer redundancy elimination. The result shows that applying the internal-then-outer redundancy elimination improves the performance of direct outer redundancy elimination by 0.55% $\sim$ 2.34%. This stems from the fact that we converge multiple different patterns into the same one by the internal redundancy elimination, and it enhances the effect of the outer redundancy elimination.

3) EXPERIMENTS FOR MULTIPLE PROGRAMS HAVING COMBINED ACCESS PATTERNS

Fig. 27 shows the comparison results of the original and two-level redundancy elimination method when the multiple programs, which combine access patterns of all the access patterns having the internal redundancy (i.e., 9 access patterns) and top-10 access patterns having the outer redundancy, are running concurrently. When the multiple programs run, they will access to the registry concurrently. As the competition to the registry becomes severe, we can expect that the performance improvement of the programs where the redundancy is eliminated will be increased. The experimental results show that the performance improvement according to the redundancy elimination is increased by 8.93% $\sim$ 26.21% when the number of multiple programs is increased from 1 to 20. We note that this result shows the effectiveness of the redundancy elimination method because the redundancy elimination method not only improves the performance of the individual program but also solves the bottleneck of the system to the registry.

VII. RELATED WORK

In this paper, we propose a method to improve the performance of Windows applications by analyzing I/O log data. Especially, we eliminate the redundant access patterns to the Windows registry, which has not been considered in the previous work. We classify existing related work into the following five categories: 1) Windows registry forensics and access analysis, 2) redundancy elimination, 3) pattern-based code transformation, 4) source code transformation and optimization, and 5) binary transformation and optimization.

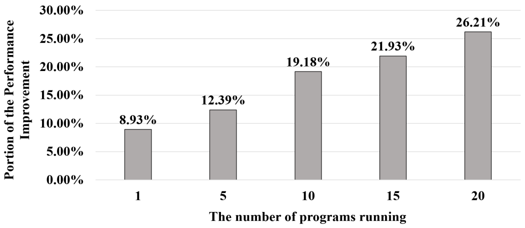


FIGURE 27. The performance improvement of the two-level redundancy elimination method compared to the original program as the number of programs running is increased.

A. WINDOWS REGISTRY FORENSICS AND ACCESS ANALYSIS

The existing researches for Windows registry forensics and access analysis have been mostly focused on the detection of malicious software. Dollan-Gavitt *et al.* presented techniques that can extract data in Windows registry from the memory dumps [5]. Apap *et al.* presented an intrusion detection system that monitors anomalous accesses to the Windows registry [14]. They trained a model of normal registry accesses and used this model to detect abnormal registry accesses. Saidi *et al.* analyzed the Windows registry to trace the artifacts left by the attacker [15]. Roy *et al.* demonstrated how to track data theft from the system via USB devices by analyzing Windows registry [16].

B. REDUNDANCY ELIMINATION

Komondoor and Horwitz modeled the entire source code as a program dependence graph and identified duplicated nodes in the graph [17]. Ducasse *et al.* detected duplicated codes after transforming the source codes into the language-independent form [18]. Lopez *et al.* defined equivalent mutants that are functionally equivalent to the original programs and found equivalent and improved mutants [19]. Briggs *et al.* improved the effectiveness of the partial redundancy elimination in the source code by overcoming the limitation that only recognizes lexically-identical expressions [20]. Mayfield *et al.* proposed an automatic memoization method, which refers to a method that transforms an ordinary function into one that caches its results to avoid repeating of the calculation, for AI applications [21].

C. PATTERN-BASED CODE TRANSFORMATION

There have been some researches for transforming the source code based on the pattern matching. Cai *et al.* proposed a pattern-based code transformation for migrating the application into the cloud environment [22]. They applied the pattern matching method based on the regular expression into the source code and transformed the original code automatically to the target code. Preissl *et al.* detected bottleneck patterns in the message passing interface and guided the optimized source codes for them [23]. Kartsaklis *et al.* designed a code transformation system, called HERCULES, aiming to improve the code maintenance and to optimize the performance [24]. HERCULES transformed the source code according to pattern-based transformation scripts. Kessler *et al.* proposed a system that can automatically parallelize the code for distributed memory systems using the pattern-recognition tool [25]. Sangwan *et al.* proposed a method for performance tuning in the real-time imaging system through pattern-based code transformation [26].

D. SOURCE CODE TRANSFORMATION AND OPTIMIZATION

There have been many researches of source code transformation and optimization for improving the performance of the software. Chung improved the energy consumption of softwares by applying loop unrolling and loop blocking techniques to source codes [27]. Cooper *et al.* applied some optimizations more than once and found the best sequence of the optimizations for minimizing code spaces [28]. Zhao *et al.* transformed the source codes so as to maximize the parallelism for multicore architectures of new processors [29]. There have been many researches to transform the source codes for optimizing the energy consumption in embedded environments. Sushko *et al.* transformed the loop in the source codes by designing the block in the loop as the cache size [30]. Simunic *et al.* profiled the bottleneck of energy consumption in embedded systems and optimized them [31]. Fei *et al.* transformed source codes while considering the interaction between the processes and the operating system, not only for the optimization of the single process [32]. Falk *et al.* transformed the loop for the energy optimization in embedded multimedia devices [33].

E. BINARY TRANSFORMATION AND OPTIMIZATION

We can consider the binary transformation and optimization to improve the performance of Windows applications even if it is not the target of this paper. Most existing methods to improve the program performance on binaries are based on the transformation of binary codes into the intermediate representation (IR) codes or source codes. Pradelle *et al.* transformed the binary into the C source code while extracting high-level information and applied the existing parallelizer to the C source code for parallelization of binary codes [34]. Kotha *et al.* adapted the existing parallelization methods for source codes into binary codes for automatic parallelization of binary codes [35]. Sato *et al.* dynamically generated improved binaries by transforming the input binary into the IR code and by optimizing the IR code [36]. Shigenobu *et al.* transformed the binary code based on ARM machine code into the IR code for the optimization [37]. Bondhugula *et al.* decompiled loop nest regions in binaries into IR codes and applied the optimization technique to the IR codes [38].

VIII. CONCLUSION

In this paper, we have analyzed I/O log data monitored in the Windows operating system. Especially, we have focused on I/O operations to the Windows registry. As a result, we have made the following four contributions. First, we have identified redundant patterns and classified them into the internal and outer redundancy. We have verified them in the assembly codes by disassembling an actual Windows application. Second, we have proposed *the redundancy detection algorithm* that finds all the possible redundant patterns from the large-scale log data. By identifying all the redundant patterns by the proposed algorithm, we have shown that the internal redundancy is about 59.21% and the outer redundancy is about 57.50%, which implies that we can improve the performance of accessing to the registry. Third, we have proposed *the two-level redundancy elimination method* to remove the internal and outer redundancy. Especially, the proposed method enhances the effect of eliminating the outer redundancy by eliminating the outer redundancy after eliminating the internal one first. We have also presented *an event-driven method* to remove the side effect of the redundancy elimination method, which could be occurred in the case of updating the Windows registry. It guarantees the correctness of the redundancy elimination method by instantly reading newly updated data as soon as the updates are occurred. Fourth, through experiments, we have shown that the two-level redundancy elimination method improves the performance of the original program having inefficient access patterns by up to 90.25%. In addition, as the number of programs running is increased, the performance improvement of the redundancy elimination method compared to the original program becomes large from 8.93% to 26.21%.

In this paper, we have analyzed the access pattern of the Window applications to the registry. The important result is that the identified access patterns are not specific for individual programs, but affect the overall system performance. That is, the Windows applications tend to access to the Windows registry repeatedly, and consequently, the Windows registry could be a significant bottleneck due to the inefficient access patterns. Therefore, by applying the redundancy elimination method to individual programs, we can improve the overall system performance.

In this paper, we have focused on the redundancy analysis and the effect of eliminating of the redundancy. As a future work, we plan to apply the proposed redundancy elimination into the actual Windows application as presented in Fig. 22 (b). Consequently, we will develop an automatic translation system of the original Windows application into the redundancy eliminated one, which consists of three steps: 1) decompliation of Windows application (in binary) into the source code, 2) redundancy elimination in the source code, and 3) redundancy eliminated Windows application that makes the same result with the original Windows application.

REFERENCES

[1] T. Peters, "The history and development of transaction log analysis," *Library Hi Tech.*, vol. 11, no. 2, pp. 41–66, Mar. 1993.

[2] R. Mafrur and I. C. D. Nugraha, "Modeling and discovering human behavior from smartphone sensing life-log data for identification purpose," *Hum.-Centirc Comput. Inf. Sci.*, vol. 5, Oct. 2015, Art. no. 31.

[3] S. Kankane and V. Garg, "A survey paper on: Frequent pattern analysis algorithm from the Web log data," *Int. J. Comput. Appl.*, vol. 119, no. 13, pp. 27–29, Jun. 2015.

[4] C.-F. Chung, J. Cook, E. Bales, J. Zia, and S. A. Munson, "More than telemonitoring: Health provider use and nonuse of life-log data in irritable bowel syndrome and weight management," *J. Med. Internet Res.*, vol. 17, no. 8, p. e203, Aug. 2015.

[5] B. Dolan-Gavitt, "Forensic analysis of the windows registry in memory," *Digit. Invest.*, vol. 5, pp. S26–S32, Sep. 2008.

[6] Microsoft. *Registry*. Accessed: Jun. 12, 2019. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/sysinfo/registry

[7] Y. Kim and D. Hong, "Windows registry and hiding suspects' secret in registry," in *Proc. Int. Conf. Inf. Secur. Assurance*, Apr. 2017, pp. 393–398.

[8] Microsoft. *How to Open Registry Editor in Windows 10*. Accessed: Jun. 12, 2019. [Online]. Available: https://support.microsoft.com/en-us/help/4027573/windows-10-open-registry-editor

[9] Microsoft. *Process Monitor v3.50*. Accessed: Jun. 12, 2019. [Online]. Available: https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

[10] Microsoft. *Registry Functions*. Accessed: Jun. 12, 2019. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/SysInfo/registry-functions

[11] Hex-Rays. *IDA: About*. Accessed: Jun. 12, 2019. [Online]. Available: https://www.hex-rays.com/products/ida

[12] C. Cifuentes and K. Gough, "Decompilation of binary programs," *Softw., Pract. Exper.*, vol. 25, no. 7, pp. 811–829, Jul. 1995.

[13] G. Z. Chen Qi, S. Huang, K. Ni, Y. Zheng, W. Binder, and H. Guan, "A refined decompiler to generate c code with high readability," *Softw., Pract. Exper.*, vol. 43, no. 11, pp. 1337–1358, Nov. 2013.

[14] F. Apap, A. Honig, S. Hershkop, E. Eskin, and S. Stolfo, "Detecting malicious software by monitoring anomalous windows registry accesses," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, Oct. 2002, pp. 36–53.

[15] R. Saidi, S. Ahmad, N. Noor, and R. Yunos, "Windows registry analysis for forensic investigation," in *Proc. Int. Conf. Technol. Adv. Electr., Electron. Comput. Eng. (TAEECE)*, May 2013, pp. 132–136.

[16] T. Roy and A. Jain, "Windows registry forensics: An imperative step in tracking data theft via USB devices," *Int. J. Comput. Sci. Inf. Technol.*, vol. 3, no. 3, pp. 4427–4433, 2012.

[17] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proc. Int. Static Anal. Symp.*, Jul. 2001, pp. 40–56.

[18] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Aug. 1999, pp. 109–118.

[19] J. López, N. Kushik, and N. Yevtushenko, "Source code optimization using equivalent mutants," *Inf. Softw. Technol.*, vol. 103, pp. 138–141, Nov. 2018.

[20] P. Briggs and K. D. Cooper, "Effective partial redundancy elimination," *SIGPLAN Notices*, vol. 29, no. 6, pp. 159–170, Jun. 1994.

[21] J. Mayfield, T. Finin, and M. Hall, "Using automatic memoization as a software engineering tool in real-world AI systems," in *Proc. 11th Conf. Artif. Intell. Appl.*, Feb. 1995, pp. 87–93.

[22] Z. Cai, L. Zhao, X. Wang, X. Yang, J. Qin, and K. Yin, "A pattern-based code transformation approach for cloud application migration," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, Jun. 2015, pp. 30–40.

[23] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. Supinski, and D. J. Quinlan, "Using MPI communication patterns to guide source code transformations," in *Proc. Int. Conf. Comput. Sci.*, Jun. 2008, pp. 253–260.

[24] C. Kartsaklis, O. Hernandez, C. Hsu, T. Ilsche, W. Joubert, and R. Graham, "HERCULES: A pattern driven code transformation system," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, pp. 574–583, May 2012.

[25] C. Kessler, "Pattern-driven automatic program transformation and parallelization," in *Proc. Euromicro Workshop Parallel Distrib. Process.*, Jan. 1995, pp. 76–83.

[26] R. Sangwan, R. Ludwig, P. Laplante, and C. Neill, "Performance tuning of imaging applications through pattern-based code transformation," in *Real-Time Imaging IX*. Bellingham, WA, USA: SPIE, Feb. 2005, pp. 1–7. [Online]. Available: https://spie.org/Publications/Proceedings/Volume/5671

[27] E. Chung and L. M. G. Benini, "Source code transformation based on software cost analysis," in *Proc. 14th Int. Symp. Syst. Synth.*, Sep. 2001, pp. 153–158.

[28] K. Cooper, P. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proc. ACM SIGPLAN Notices*, May 1999, pp. 1–9.

[29] B. Zhao, L. Zhen, J. Ali, W. Felix, and W. Weiguo, "Dependence-based code transformation for coarse-grained parallelism," in *Proc. Int. Workshop Code Optim. Multi Many Cores*, Feb. 2015.

[30] S. Sushko and A. Chemeris, "Increasing the energy efficiency of a computational system by using automatic software optimization," in *Proc. IEEE 9th Int. Conf. Dependable Syst.*, May 2018, pp. 578–582.

[31] T. Simunic, L. Benini, M. G. De, and M. Hans, "Source code optimization and profiling of energy consumption in embedded systems," in *Proc. 13th Int. Symp. Syst. Synth.*, Sep. 2000, pp. 193–198.

[32] Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha, "Energy-optimizing source code transformations for operating system-driven embedded software," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 1, Dec. 2007, Art. no. 2.

[33] H. Falk, P. Marwedel, and F. Catthoor, "Control flow driven splitting of loop nests at the source code level," in *Proc. Conf. Design, Automat. Test Eur.*, Dec. 2005, pp. 215–229.

[34] B. Pradelle, A. Ketterlin, and P. Clauss, "Polyhedral parallelization of binary code," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, Jan. 2012, Art. no. 39.

[35] A. Kotha, K. Anand, T. Creech, K. Elwazeer, M. Smithson, G. Yellareddy, and R. Barua, "Affine parallelization using dependence and cache analysis in a binary rewriter," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 8, pp. 2154–2163, Aug. 2015.

[36] Y. Sato, T. Yuki, and T. Endo, "ExanaDBT: A dynamic compilation system for transparent polyhedral optimizations at runtime," in *Proc. Comput. Frontiers Conf.*, May 2017, pp. 191–200.

[37] K. Shigenobu, K. Ootsu, T. Ohkawa, and T. Yokota, "A translation method of ARM machine code to LLVM-IR for binary code parallelization and optimization," in *Proc. 5th Int. Symp. Comput. Netw. (CANDAR)*, Nov. 2017, pp. 575–579.

[38] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proc. 29th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2008, pp. 101–113.

JUN-HA LEE is currently pursuing the bachelor's degree in IT management programme with the Department of Industrial and Systems Engineering, Seoul National University of Science and Technology (Seoul-Tech). His research interests include log data analysis, pattern analysis, and NoSQL systems.

HYUK-YOON KWON received the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 2013. He worked at the Ministry of National Defense as a Researcher, from 2014 to 2018, and worked at KAIST as a Postdoctoral Researcher, from 2013 to 2014. He worked at Microsoft Research Asia as a Research Intern, from 2011 to 2012. He is currently an Assistant Professor in IT management programme with the Department of Industrial and Systems Engineering, Graduate School of Data Science, Seoul National University of Technology (SeoulTech). His research interests include Big data, distributed systems, parallel search engines, databases, machine learning, and Web crawler.