Towards the Memory Forensics of OOP Execution Behavior

Ziad A. Al-Sharif1, Mohammad I. Al-Saleh2, Luay Alawneh1
Software Engineering Department1, Computer Science Department2
Jordan University of Science and Technology
P.O. Box 3030
Irbid, 22110, Jordan
Email: {zasharif, misaleh, lmalawneh}@just.edu.jo

Abstract—Perpetrators might employ computer programs (software) to perform their offenses or to cover their wrongdoings. Program’s source code, in particular variables and their values diverge in their scope and duration in RAM. Object oriented languages encapsulate variables and operations in classes and allow for objects to be instantiated, which simplify software design but add to the complexity of its execution behavior and its data and control flow. This paper explores execution behaviors and information left by program execution in support for legal actions against perpetrators in the court of law. Our investigation model assumes no information is provided by the operating system; only raw RAM dumps. Our methodology employs information from the presumed program source code and its object oriented design. It explores various execution states and scenarios to uncover the evidence of potential software usage. These scenarios are designed to show that scope, access modifiers, and storage information of various source code variables can be used to identify program’s activities. Results show that investigators have good chances locating various variables’ values that are uniquely corresponded to the presumed software and its execution states. In some cases, values are successfully identified in memory dumps even after the process is stopped.

Index Terms—Digital Forensics, Memory Forensics, Object Oriented Programming, Program Execution Behavior, Java Programs, C# Programs.

I. INTRODUCTION

Offender’s exertion may involve computers and computer programs (software) to execute or cover their misdeeds. Locating the software on the machine’s hard disk might not be sufficient to establish the confidence in its potential usage. A definite evidence might be needed to prove that the perpetrator is actually used the software [1]. This evidence can be found in several places, one of which is the RAM of the target machine. This underlines the importance of Memory Forensics (MF) and its value in crime investigation.

Generally, different software varies on their dependency on memory, CPU, disk I/Os, and networks [2]. A software source code and its execution behavior including its control and data flow might highly depend on various variables and their values that are stored in different memory locations (RAM). Thus, Main Memory or RAM encompasses vibrant information about a system such as the active processes and their execution states and behaviors.

Software variables of an Object Oriented Program (OOP) can be categorized based on their scopes, access modifiers, and execution lifetimes. A variable’s scope and its access modifier determine the visibility of a variable and where it can be accessed within the program’s source code. In contrast, variable’s storage (memory type) determines the duration in which its value is created and destroyed or deleted. Additionally, variables can be classified based on whether they are allowed to be changed during execution. Constant variables are those that cannot be changed once are assigned, most of which are often assigned with literal values. These literals might be unique to the executable program and its execution state. Moreover, many other non-constant variables might be initialized literals. These literals can be retained in RAM to establish the evidence about the software usage and its potential execution path.

Assuming no information is available from the operating system; only raw RAM dumps. This paper identifies evidences that would be employed to confirm the software usage and its association with the crime. In order to verify our research methodology, various experiments and scenarios are established, then RAM dumps are created and analyzed. Various variables’ scopes and memory types are assumed during the analysis process.
The software source code is assumed to be written in Java or C# programming languages, all of which runs under MS Windows 10 Pro operating system. However, most of our findings are equally applicable for other languages and operating systems.

Our results show that regardless of whether the process is active or just stopped, the memory investigator can employ knowledge about the program source code such as class level and instance level variables and their potential values to confirm the actual usage of the software. Hence, values of instance variables are successfully located when their corresponding stack frames are still active. On the other hand, static variables and local variables of static methods have longer duration in memory than the instance related values. In some cases, dynamically allocated values (object level) can be identified in memory dumps even after the Garbage Collector (GC) is explicitly invoked.

The rest of this paper is organized as follows. Section II highlights some of the background knowledge used in this paper. Section III presents our investigation model and explains how it employs information available in the program source code to confirm that the program is actually used. Section IV describes our experiments whereas Section V presents our promising results. Section VI presents our related work. Finally, our planned future work is presented in Section VII whereas Section VIII concludes our findings.

II. BACKGROUND

A software process may employ various variables (memory storage). In OOP languages, variables can be classified into class level and method level. Class level variables can be further categorized into instance and static. Most of the time, a variable’s type defines the duration of its value in main memory. For example, static data is allocated by the runtime system for a program before instances are created. These variables might be initialized with default values whenever they are not explicitly initialized by the programmer.

However, unlike instance variables, the duration of static variables does not depend on the objects created from that class. Furthermore, static values are shared by all objects; all point to the same value. Instance variables’ visibility is limited to the object instantiated from that class; each object has its own set of values. Often, local variables (method level) are allocated on the stack and their visibility is limited to their methods or code blocks.

Typically, all operating systems provide services to programs they run. For example, in a Unix based system, when the Kernel executes a C program, a special routine (known as the startup routine) is automatically invoked to set up the command line arguments and the environments. Then, the main() function is called.

Modern high level programming languages, such as C# and Java, are supported by runtime systems, frameworks, or platforms. For example, C# is supported by the MS .Net framework ¹, which is a software layer that sets on top of the Windows OS. This .Net framework supports a Common Language Runtime system (CLR), which allows C# programs to compile and execute. The .Net framework supports its own intermediate language called Microsoft Intermediate Language (MSIL). The compiler reads the C# source code and produces MSIL (the .exe program). When this executable runs, the Just-In-Time (JIT) compiler that is part of the .Net framework reads the MSIL code and produces an executable application in memory. Even though the MSIL code is stored in an .exe file, this file does not contain native executable code; it contains information needed by the JIT compiler to execute the code [3], [4]. By contrast, the Java programs are loaded and executed by the Java Virtual Machine (JVM); Java programs run on top of the JVM, which is an abstract computing machine with its own instruction set. It internally manipulates various memory areas at run time. Simply, it loads class files and execute their bytecode. JVM’s execution engine can vary based on different implementations and target platforms [5], [6].

Generally, the OS Kernel manages software processes, each of which is provided a dedicated memory space in RAM. When the executable starts, various sections are allocated and loaded into RAM, the starts and ends of these sections are independent of the RAM page limits [7]. During execution, different variables are stored in RAM into various logically classified segments such as heap and stack [8]. Heap is a memory segment allocated when the process starts. It provides runtime memory allocation for variables and their values as needed during execution. Program’s data that lives in heap can be referenced outside the function scope. In both Java and C# languages, the heap memory allocations and deallocations are automatically managed by the runtime system and its Garbage Collector (GC). However, an explicit request can be triggered to release the unused memory to the system using special API. On the other hand,

hand, stack is a memory segment allocated when the program starts. When the program’s execution stack is not managed by the OS Kernel, it is automatically managed by the Virtual Machine (VM) or the runtime system of the used language. Usually, a stack consists of blocks called activation records or frames, each of which represents a call to a function and provides storage for its corresponding local and formal parameters. The lifetime (duration) of variables allocated on the stack is same as the scope in which they are declared; mostly the function/method and its stack frame [9], [10]. Hence, memory investigators can utilize the memory of various variables’ values to locate evidences about the actual use of the software.

III. INVESTIGATION MODEL

A variable scope affects its visibility within the program source code and a variable storage affects its duration. Hence, different scopes and storages might affect the survivability of a variable’s value in memory during various execution states. Accordingly, locating these values in a RAM memory dump can be used as evidence to prove that the user is actually used the presumed program. Our investigation model studies the possibility of locating these variables’ values that are used within two different object oriented languages: Java and C#.

Figure 1 shows our investigation model, in which a perpetrator uses a software (computer program) to perform his/her offense or to hide the remnant of his/her wrongdoings. The investigator points at the presumed software and assumes its source code. The investigator’s intention is to validate whether the subject software is or was used on the captured machine. This can be reached by searching the contents of the RAM of the target machine for objects, execution states, or unique variables’ values that can be used to confirm the actual usage of the presumed software. The fundamental information from the source code is used to identify various execution states and potential values in the RAM dump of the captured machine. These identified states are considered as evidence to prove that the software is actually used.

IV. EXPERIMENTS

In order to validate our investigation model, a set of experiments are designed to investigate the footprints of various execution behaviors of an object oriented program in RAM. Our methodology aims to find evidence related to the execution state based on information and variables’ values from the program source code. Six different variables’ states are identified and experiments are established to inspect their potential variables’ values. These variables are:

- SV₁: Class level static variables
- SV₂: Local variables of static methods that are currently active in the execution stack frame or used to be active
- SV₃: Local variables of static methods that are never been active in the execution stack frame (never called)
- IV₁: Class level instance variables
- IV₂: Local variables of instance methods that are currently active in the execution stack frame or used to be active
- IV₃: Local variables of instance methods that are never been active in the execution stack frame (never called)

Our experiments are designed to explore the potential evidence that would prove the actual software usage during various execution states. Our experiments attempt to identify the possibility of locating these six variables’ values that are used within different scenarios and ex-
execution states. Identified values are used as possible confirmations that the program was running when the machine is captured. The scenarios are defined based on the potential usage of object oriented programming features. During the investigation process, ten different scenarios are defined. These execution states (scenarios) depend on the variable’s type, scope, and state, in which a variable’s value belongs to:

- **S₁**: A class that is never referenced
- **S₂**: A class that is referenced but never allocated
- **S₃**: A member static method that is currently active (and the class level static variable is being used)
- **S₄**: A member static method that is currently inactive (used to be active but not at dump time)
- **S₅**: One object is allocated and the default constructor is used
- **S₆**: Two objects are allocated and the default constructor are used
- **S₇**: A member instance method that is currently active (and the instance variable is being used)
- **S₈**: A member instance method that is currently inactive (used to be active but not at dump time)
- **S₉**: Objects that are no longer referenced; the Garbage Collector (GC) is explicitly invoked
- **S₁₀**: A program that is no longer active; program is terminated.

It is worth mentioning that all experiments assume no information is provided by the operating system about the investigated process. A memory dump is created after each one of the above ten scenarios. Then, each of these dumps is analyzed and searched for potential values related to the source code of the presumed program. The results of our findings are presented in Section V.

### A. Experimentation Setup

In all experiments, we used a Windows virtual machine that is created using VirtualBox. The VM runs Windows 10 Pro with 2 GB of RAM memory. This VM is hosted on a Mac OS X 10.11.6. See Figure 2.

### V. RESULTS

Results presented in Table I is valid for both Java and C# programs. These results are obtained from experiments and RAM memory dumps of identical programs written in both Java and C#. Interestingly, their results are also identical. In particular, Table I shows that the values of static variables ($SV_1$) are identifiable in memory from state $S_3$ until state $S_9$. This means that as soon as the static variable is used in the program, its value stays in memory for the life of the program. Table I also shows that local references (variables’ values) of static methods ($SV_2$) are available in memory from state $S_3$ until state $S_9$. This means that as soon as the static method is called, its references of local variables and their values stays in memory and these values are not released after the function is returned or even after the

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>Variables</th>
<th>$SV_1$</th>
<th>$SV_2$</th>
<th>$SV_3$</th>
<th>$IV_1$</th>
<th>$IV_2$</th>
<th>$IV_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$S_3$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$S_4$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$S_5$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$S_6$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$S_7$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$S_8$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$S_9$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$S_{10}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

TABLE II

The number of occurrences of a string literal in the RAM of a running Java program. These strings are found using the UTF-8 encoding format.

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>Variables</th>
<th>$SV_1$</th>
<th>$SV_2$</th>
<th>$SV_3$</th>
<th>$IV_1$</th>
<th>$IV_2$</th>
<th>$IV_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$S_3$</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>$S_4$</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>$S_5$</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>$S_6$</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>$S_7$</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>$S_8$</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>$S_9$</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$S_{10}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
GC is explicitly invoked.

On the other hand, Table I also shows that references of static variables (SV3) that are never used in the program and those inside static methods that are never called are not available in any of the RAM dumps that are created after each of the execution states S1 to S10. Additionally, Table I shows that the values of class level instance variables can be identified in RAM dumps right after the object is allocated and up until the end of the program (from state S5 to S9). For values of reference type local variables of instance methods, the values of these variables depend on the call to that method; they can be identified in memory dumps even after the method is returned.

Furthermore, different programming languages deal with strings in various ways. In particular, C# and the .Net applications encode strings in the Unicode format (UTF-16), whereas the Java 8 implementation uses the UTF-8 encoding. Thus, finding string literals and values of various string variables in memory dumps is affected by the encoding format of these string values. Therefore, another set of experiments is repeated for string literals. The same set of scenarios and variable types are investigated with string literals. Results for identifying string literals in memory dumps using the UTF-16 encoding is totally identical in both Java and C# to the results found in Table I. However, results of identifying literal strings of a Java program can be found in both the UTF-16 and UTF-8 formats. Surprisingly, the number of occurrences for these string literals in the UTF-8 format of Java programs are impacting the RAM dumps with a number of occurrences that ranges from 1 to 8 in some cases. Whereas, C# has zero occurrences of these UTF-8 string formats in all of the 10 investigated states. Table II shows the number of occurrences for these strings that are identified using the UTF-8 format in memory dumps created for Java programs during the 10 states.

VI. RELATED WORK

Many researchers find in the RAM memory a vital source of information that can be used in support for legal actions against criminals in digital forensic cases [11]–[19]. Ahmad Shosha et al. developed a prototype to detect different malicious programs that are regularly used by criminals. The proposed approach depends on the deduction of evidences that are extracted based on traces related to the suspect program [20]. Chan Ellick et al. introduced ForenScope [21] a RAM forensic tool that permits users to investigate a machine using regular bash-shell. It allows users to disable anti-forensic tools and search for potential evidences. In order to maintain the RAM memory intact, it is designed to work in the unused memory space of the target machine.

On the other hand, Table I also shows that references and values of class level instance variables can be identified in RAM dumps right after the object is allocated and up until the end of the program (from state S5 to S9). For values of reference type local variables of instance methods, the values of these variables depend on the call to that method; they can be identified in memory dumps even after the method is returned.

In this paper, we tested the results of programs written in C# and Java and running on a Windows 10 Pro machine, while the program’s code is identical in all experiments, the execution of the presumed programs is completely processed by different underneath implementations that could potentially behave differently on different platforms. For future work, we are planning to investigate other environments such as Mac OS X, Linux, and small devices such as phones and tablets. In particular, C# is partially supported by the Mono project under Linux and Mac OS X.

Nonetheless, theoretically the investigated languages operate to some extent in a layer that is distant from the OS Kernel (i.e. JVM). Practically, this means that a specific area of the memory is kept from this layer to handle the program execution. Therefore, while the GC picks up the garbage, the memory will not become immediately available to the OS. Thus, traces are expected to behave differently with other object oriented languages such as C++. Thus, we are looking forward to find out how the results will contrast with the ones reported in this paper.

3Cross platform .Net framework, www.mono-project.com
Additionally, our experimentation in this paper target the strictly typed languages, whereas some other languages support dynamic typing such as Visual Basic and the scripting languages such as Python, Icon/Unicon, Ruby and others. We plan to investigate the differences in the behavior of dynamically typed programming languages. Furthermore, we are looking forward to investigate similar scenarios for other data types and data structures such as collections and generics. Finally, it would be important to investigate various types and their impacts on long running programs such as servers.

VIII. CONCLUSION

This paper utilizes information from the source code of a program and employs program’s execution data during various execution states to help digital investigators establish the evidence against a perpetrator. This will permit law enforcement agencies to take legal actions against criminals in the court of law. Our experimentation is based on modern object oriented languages; in particular Java and C#. Based on these experiments, we found that utilizing source code information can be valuable to the investigator. It helps establish the evidence that the perpetrator is actually used the software to perform the crime or to cover the wrongdoing associated with the crime. Various variables’ values and string literals and non-literals related to the program execution are successfully located during various scenarios and execution states.

REFERENCES