Journal of Universal Computer Science, vol. 25, no. 9 (2019), 1174-1198
submitted: 22/12/18, accepted: 9/7/19, appeared: 28/9/19

The Effects of Platforms and Languages on the Memory Footprint of the Executable Program: A Memory Forensic Approach

Ziad A. Al-Sharif, Mohammed I. Al-Saleh
Yaser Jararweh, Luay Alawneh, Ahmed S. Shatnawi
(Jordan University of Science and Technology, Irbid, 22110, Jordan
{zasharif, misaleh, yijararweh, lmalawneh, ahmedshatnawi}@just.edu.jo)

Abstract: Identifying the software used in a cybercrime can play a key role in establishing the evidence against the perpetrator in the court of law. This can be achieved by various means, one of which is to utilize the RAM contents. RAM comprises vital information about the current state of a system, including its running processes. Accordingly, the memory footprint of a process can be used as evidence about its usage. However, this evidence can be influenced by several factors. This paper evaluates three of these factors. First, it evaluates how the used programming language affects the evidence. Second, it evaluates how the used platform affects the evidence. Finally, it evaluates how the search for this evidence is influenced by the implicitly used encoding scheme. Our results should assist the investigator in its quest to identify the best amount of evidences about the used software based on its execution logic, host platform, language used, and the encoding of its string values. Results show that the amount of digital evidence is highly affected by these factors. For instance, the memory footprint of a Java based software is often more traceable than the footprints of languages such as C++ and C#. Moreover, the memory footprint of a C# program is more visible on Linux than it is on Windows or Mac OS. Hence, often software related values are successfully identified in RAM memory dumps even after the program is stopped.

Key Words: Digital Forensics, Memory Forensics, Runtime Behavior, RAM Dumps
Category: H.2, H.3.7, H.5.4

1 Introduction

Criminal's actions may involve computer programs to perform or cover their misdeeds. Locating the software on their machine's hard disk might not be sufficient to establish the confidence in its potential usage. Often, a definite Digital Evidence (DE) is needed to prove that the perpetrator has actually used the software [Salajegheh et al., 2018, Al-Sharif, 2016]. This DE can be found in several places, one of which is the volatile main memory (RAM). Thus, RAM contains vital information about the current state –often recent states– of a system such as the used processes; including parts of their instructions and data. This ensures the importance of Memory Forensics (MF) and its value in the investigation of cybercrimes [Schatz and Cohen, 2017, Case and Richard, 2017, Bobowska et al., 2018, Montasari and Hill, 2019].

Generally, different software varies in their dependency on memory, CPU, disk I/Os, and networks [Sitaraman and Venkatesan, 2005, Ligh et al., 2014]. A software source-code and its execution behavior, including its control and data flow, might depend on various variables and their values that are stored in different RAM locations. MF of a software that is developed using an object oriented language can be based on variable's scopes, access modifiers, execution states and their lifetimes. A variable's scope and access modifier determine the visibility of its value and where it can be accessed within the program's source-code. Accordingly, a variable's storage (i.e. stack and heap) determines the duration in which the values are allocated and/or deleted [Pridgen et al., 2017, Yang et al., 2007]. For example, constant variables cannot be modified once are initialized, most of which are often set with literals that might be unique to the running program and its execution path. Moreover, many of the non-constant variables are often initialized with literals too. Thus, variables' values can be retained in RAM dumps and used to establish the DE about the software usage and its distinct execution path [Tien et al., 2017, Al-Sharif et al., 2017].

Assuming no information is available from the OS; only raw RAM dumps. This paper identifies the DE that can be employed to confirm the software usage and its association with the crime [Butler, 2016, Subedi et al., 2018]. In order to verify our research methodology, different experiments and scenarios are assumed, for each of which a RAM dump is created and various variables' scopes and memory types are analyzed.

Our results show that regardless of whether the process is active or just stopped, the Memory Investigator (MI) can employ knowledge about the program's source-code such as class level (i.e. static) and instance level variables and their potential values to confirm the actual usage of the software. Furthermore, results show that values of instance variables are successfully located when their corresponding stack frames are active and beyond. Additionally, literal values, static variables and local variables of static methods have longer duration in memory than the instance related values. Furthermore, in most cases, dynamically allocated values (object level) can be identified in memory dumps even after the Garbage Collector (GC) is explicitly invoked or even after the program is terminated. Thus, whenever the original source code is not available, reverse engineering and disassembling tools can be applied on the binary to obtain its equivalent source code or at least its assembly. Obviously, the success of this process depends on various factors, including the binary itself and the language used [Kruegel et al., 2004, Bauman et al., 2018].

The rest of this paper is organized as follows: Section 2 presents the motivation of this research whereas Section 3 presents some of the fundamentally needed background knowledge. Section 4 presents our investigation model that employs information available in the program's source-code and binary-code to confirm its usage. Section 5 describes our experiments whereas our results are presented in Section 6. The results are explained on three levels; subsection 6.1 compares the DE between three languages, subsection 6.2 compares the DE between three platforms, and finally subsection 6.3 compares the DE between two different encoding schemes. These comparisons are intended to guide the MI during the investigation process in order to help them obtain the best results based on the used platform and programming language. Section 7 presents the related work. Finally, our overall discussions and planned future work is presented in Section 8 whereas Section 9 concludes our work.

2 Motivation

This paper investigates the differences in program's execution traces in RAM and its utilization in MF and the DE collection process. The goal is to identify the presumed software that is used in the cybercrime. These types of DE can be easily influenced by the host platform and the used programming language. Our approach is to experiment with different implementations of three of the most widely used mainstream languages: Java, C++, and C# and to compare the execution footprints of similar programs that run on the same and different platforms. For instance, Java and C++ are supported on all of Windows, Mac OS, Linux, and more. Whereas C# is increasingly becoming a cross-platform language by its support from the open-source cross-platform Mono framework [1]. Interestingly enough, these three languages are highly different in their underlying support and memory management. In particular, Java is supported by the JVM and its runtime system whereas C++ depends heavily on the host platform and its memory management. In contrast, the rigorous implementation of the cross-platform open-source Mono project increasingly supports C# to become a cross-platform language. This Mono framework replaces C# support by the native Microsoft $.Net$ framework on different platforms such as Mac OS and Linux in addition to the MS Windows. This motivates us to investigate these languages and find the impact of using software written in any of them on different platforms and the effect of these combinations on the MF case and the DE that can be identified in RAM. Often cybercrimes involving software [Leukfeldt et al., 2017]; this research is aims at assisting the MI in its quest to ensure the actual software usage by the perpetrator.

3 Background

A software process may employ various variables (memory storage). In OOP languages, variables can be classified into class level and method level. Class

[1] Open-source Cross-platform .Net framework, `www.Mono-project.com`

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 RAM. 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, most operating systems provide services to programs they run. For example, in a UNIX based system, when the Kernel executes a C/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 Java and C#, are supported by runtime systems, frameworks, or virtual platforms. In particular, C# is supported by the Microsoft $.Net$ framework [2], 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 on Windows Machines. This $.Net$ 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 a *.exe* file, this file does not contain native executable code; it contains information needed by the JIT compiler to build the executable code [Esposito and Ciceri, 2016]. Additionally, C# is increasingly becoming a cross-platform mainstream language supported by the rigorous implementation of the open-source cross-platform Mono project, which replaces the native $.Net$ framework on other platforms such as Mac and Linux.

By contrast, Java programs are loaded and executed by the Java Virtual Machine (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 bytecodes. The internals of JVM's execution engine can vary based on different implementations and host platforms [Lindholm et al., , Schildt, 2014].

However, in general, 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

[2] Microsoft $.Net$, `https://docs.microsoft.com/en-us/dotnet/`

these sections are independent of the RAM page limits [Stevens and Rago, 2013]. During execution, different variables are stored in RAM into various logically classified segments such as heap and stack [Bovet and Cesati, 2005]. Heap is a memory segment that provides dynamic 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 languages such as Java and C#, the heap memory allocations and deallocations are automatically managed by the runtime system and its Garbage Collector (GC) whereas in C/C++, the programmer writes a dedicated code to ensure the correct memory management for these dynamically allocated variables and objects. Thus, often an explicit request can be triggered to release the unused memory to the system using special calls to the GC.

On the other 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 method and its stack frame [Nadi and Holt, 2014, Josey et al., 2004]. Hence, the MI can utilize the memory of various variables' scopes, types and values to locate a DE and infer the actual software usage.

3.1 Encoding of Literal Strings: UTF8 vs. UTF16

Compilers and interpreters translate program's source-code into an equivalent form of executable binary format. For example, translating a Java program generates a cross-platforms binary called *bytecode*, whereas translating a C++ program generates native machine level instructions that are heavily depend on the physical architecture and the host OS. Nonetheless, our experimentation methodology employs a *bit-by-bit* copy of the complete RAM of the used machine. For potential values to be successfully identified within RAM dumps, the MI needs to respect their encoding schemes; internal representation. For example, to identify potential string values, the UNIX based *strings* [3] command is used to pre-process the memory dumps and extract all of the readable characters in both UTF8 and UTF16 formats. By default, this command looks for sequences of at least 4 printable characters that are terminated by a *null*, other options can be used to find characters in other formats Thus, during our experimentation, each memory dump is preprocessed and all source-code/binary-code related values are identified using various encoding schemes; including UTF8 and UTF16.

[3] GNU Binutils, `www.gnu.org/software/binutils/`

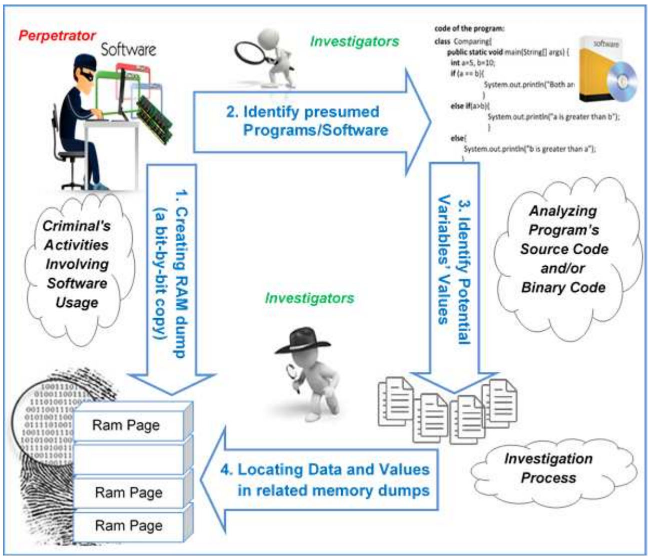


Figure 1: Investigation Model. The MI examines the digital crime scene. *Step 1*: a memory dump is captured. *Step 2*: the MI analyzes the presumed software (source-code and/or binary-code). *Step 3*: software related literals and unique values are identified. *Step 4*: values are located in corresponding RAM dumps.

4 Investigation Model

Our threat model assumes that the perpetrator uses a software to perform the offense [Kälber et al., 2013] or to hide its remnant [Conlan et al., 2016]. Figure 1 shows our investigation model, in which the MI points at the presumed software and assumes its source-code, if possible, otherwise, a reverse engineering tool can be used to disassemble the binary-code and obtain its original source-code or at least its assembly format (or logically equivalent); this can assist in comprehending the execution logic and identifying used literals. Hence, many reverse engineering tools are available for languages such as Java [Mesbah et al., 2017, Sen and Mall, 2016, Moser et al., 2017] and C# [Jensen et al., 2017].

For instance, to some extent, it is relatively easy to reverse the Java bytecode into its original source-code. Additionally, the MSIL used by C# simplify the reverse engineering process. However, native machine code obtained from languages such as C and C++ are often harder to reverse back to their originals, but it is often feasible to get its equivalent assembly code, infer its execution logic and obtain its dynamic data structures [Rupprecht et al., 2017, Haller et al., 2016, Schwartz et al., 2018]; free and shareware tools are available such as IDA pro [4].

[4] IDA, `www.hex-rays.com/products/ida/`

The MI goal is to validate whether the subject software is (or was) used on the captured machine. This can be reached by various means, one of which is to search the contents of the RAM for objects, execution states, or unique variables' values that can be used to confirm the actual software usage. These identified states and unique values can be considered a contributing DE towards the legal proof that the software is actually used by the criminal [Craiger, 2005, Thomas et al., 2013, Sali and Khanuja, 2019, Kao et al., 2019].

Three different languages are used to prepare three versions of the same program [5], all of which are semantically identical with respect to the syntactic language differences; all programs have the same behavior (i.e. coding structure, control flow, data flow, etc). These programs and corresponding memory dumps are designed to build a comparison based on various factors such as the execution scenario (state) and the host platform.

Furthermore, our analysis explores the DE and its influence by the software implementation including the: 1) member access modifier, 2) level in which the variable is defined, 3) class and method type and usage, 4) allocation and deallocation of objects that are instantiated from a base or derived class and their number, 5) and whether the program is still running or just terminated.

5 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 a DE related to the execution state based on information and variables' values from the program's source-code or its disassembled binary. A total of 12 different variables' types (scopes) are defined and experiments are prepared to inspect their potential values; see Table 1. During the investigation process, 21 different scenarios are outlined. These scenarios are based on the possible usage of object oriented features including variables' types, levels or scopes, and object states; see Table 2. All experiments assume no information is provided by the OS about the investigated process. A memory dump is captured for each of the 21 execution states, each state is tested within each of the three investigated languages and on each of the three investigated platforms. For programs written in C# and running on Windows, one set of experiments is performed using the $.Net$ framework and the same set is repeated using the Mono framework. A combined total of 210 distinct memory dumps are created. Then, each of these dumps is analyzed and searched for potential values related to the execution states of the presumed program; see Figure 2. Results are presented in Section 6.

[5] Programs, `https://github.com/zasharif/MemoryForensics--OS-Language`

Table 1: The investigated scopes.

Var #	Description
V_1	Static base class level variables
V_2	Variables of a currently active static method
V_3	Variables of a never active static method
V_4	Instance base class level variables
V_5	Variables of a currently active base class instance method
V_6	Variables of in a never active base class instance method
V_7	Instance derived class level variables
V_8	Variables of a never used overridden base class method
V_9	Variables of an overridden virtual method when its override is active
V_{10}	Variables of a never used override of a derived class method
V_{11}	Variables of an override currently active method of a derived class
V_{12}	Variables of a new currently active method of a derived class

Table 2: The investigated runtime states.

State.	Description
S_1	Program is currently running, but the investigated class is not referenced yet
S_2	A static variable of the base class is used (referenced)
S_3	A static method is being used (called and currently active)
S_4	The static method is returned (returned and currently inactive)
S_5	An instance reference of a base class is defined but not allocated yet
S_6	An instance reference of a derived class is defined but not allocated yet
S_7	An instance of the base class is allocated
S_8	An instance variable of the base class is used (referenced)
S_9	A method of the base class is being called (called and currently active)
S_{10}	The method of the base class is returned (returned and currently inactive)
S_{11}	Derived class instant is allocated and assigned to a variable of the same type
S_{12}	An instance variable of the derived class is used (referenced)
S_{13}	A new method of the derived class is being called (called and currently active)
S_{14}	The new method of the derived class is returned (currently inactive)
S_{15}	Derived class instant is allocated but up-casted to a reference of the base class
S_{16}	Virtual method of the derived class is being called (called and currently active)
S_{17}	The virtual method of the derived class is returned (currently inactive)
S_{18}	The second instance of the derived class is not referenced any more
S_{19}	References are *null* and the GC (or *delete*) is invoked explicitly
S_{20}	All objects are out of scope and not accessible any more
S_{21}	The program is terminated

5.1 Experimentation Setup

Figure 3 shows the configurations of three different Virtual Machines (VMs) that are used during our experiments. These VMs are prepared using Oracle's VirtualBox that is hosted on a Fedora Linux machine. VM_1 runs Windows 10 Pro (64-bit), VM_2 runs Mac OS 10.11.6 (El Capitan), and VM_3 runs Fedora 26 (64-bit). The RAM settings of these VMs are kept to the minimum hardware requirements that are recommended for each platform. In particular: $2GB$ of RAM for each of Windows and Mac OS and $1GB$ for Fedora Linux. Moreover,

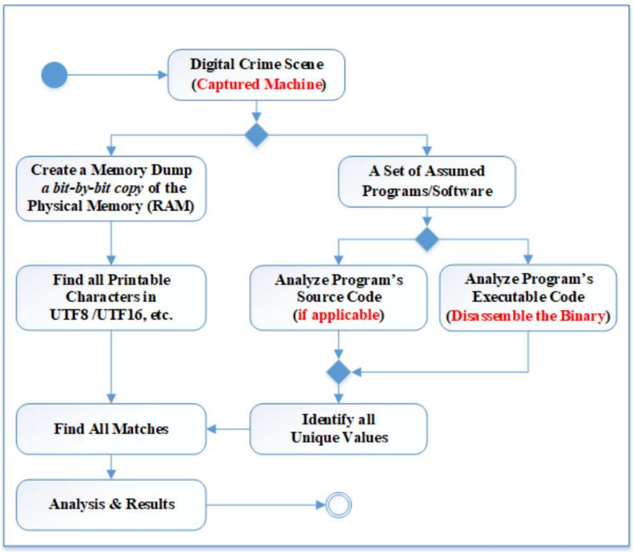


Figure 2: Experimentation model and the analysis activities.

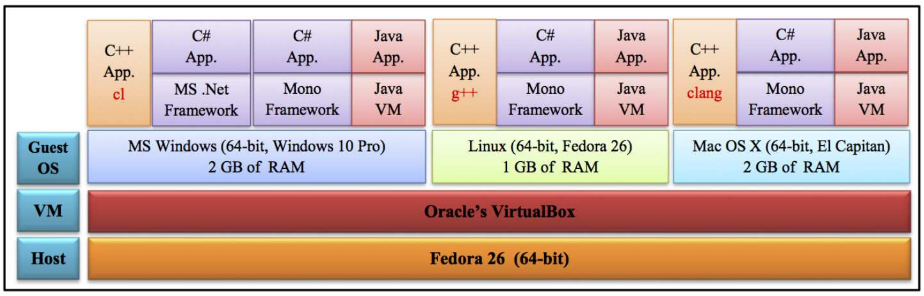


Figure 3: Experimentation Setup: VMs are created using Oracle's VirtualBox. Three different platforms are used, on each of them all three different languages are investigated. Moreover, C# is investigated twice on Windows; once using the MS *.Net* and the other using the Mono framework.

the same program is prepared in the three different languages: Java, C++, and C#. The execution of these programs are ensured to maintain the same structure, execution behavior and control flow. Additionally, the same version of JVM and Mono framework is installed on each of these three VMs. For programs written in C++, the native and most common compiler on each platform is used to prepare the executables. For instance, the *cl.exe* compiler is used on Windows. On Mac OS, *LLVM* and its *clang* front end is used to build the same C++ program. Finally the $GNUg++$ is used to build the same C++ programs under Fedora.

In all three programs, the 21 specific scenarios presented in Table 2 are used to capture the core dumps for each program independently. A machine restart is applied before each run.

6 Results

Results are discussed at three different levels. *Level 1*: Language based comparison. *Level 2*: OS based comparison. *Level 3*: Encoding based comparison.

6.1 Level 1: Language Based Results

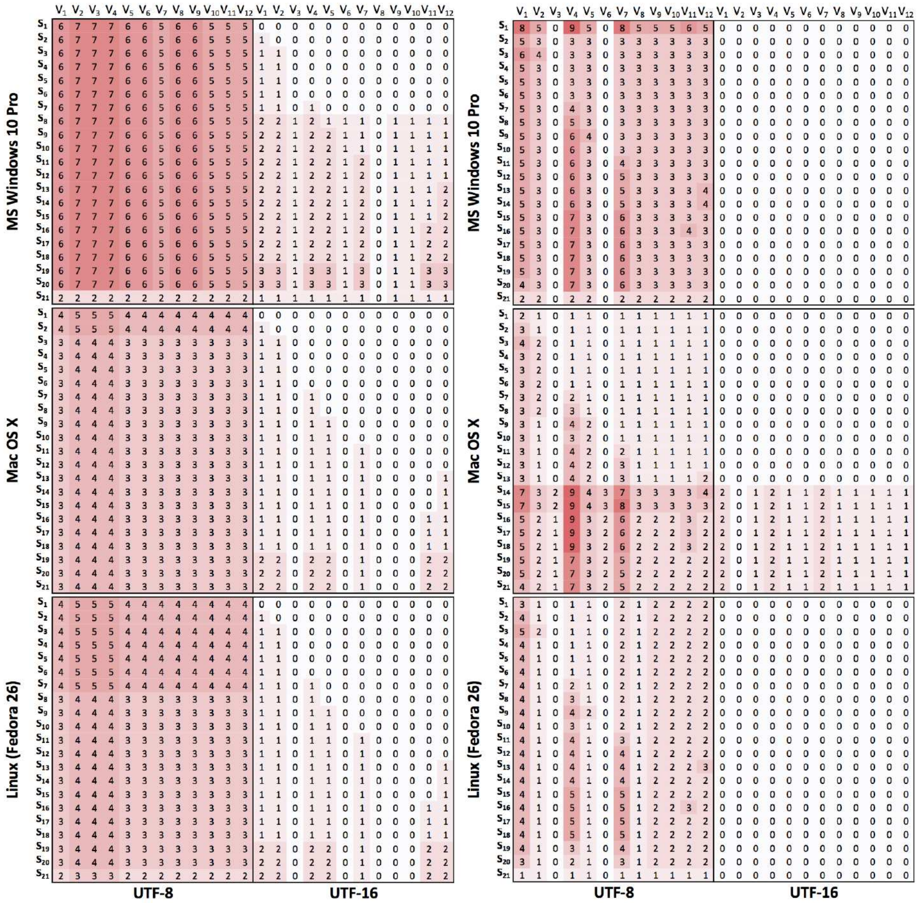
This level compares between the DE from the same language on three platforms.

6.1.1 Java Based Results

Figure 4a presents the heatmap with the number of occurrences for each variable's value V_i on each state S_i, which represents the memory $dump_i$ that was obtained for the same Java program when it was running on all of Windows (the upper part), Mac OS (the middle part) and Fedora Linux (the lower part), respectively. Additionally, the same set of values is identified using UTF8 and UTF16 encoding schemes; left versus right sides respectively.

Looking at the *left* side of Figure 4a that represent the number of successfully identified occurrences of the values obtained from the subject program's source-code using the UTF8 formats, we find that in all 12 types of variables, the MI have a good chance locating uniquely identified values that belongs to the subject program's source-code and its execution states on all three investigated platforms. Even though each of these values are appeared only once in the subject program's source-code, most of these identified values (V_i) in the corresponding RAM dumps are successfully located. Additionally, the left side of this figure shows that the number of occurrences for all variable types V_1 to V_{12} are always greater than 1; even after the program is terminated, this applies for all execution states (scenarios S_1 to S_{21}) and on all platforms. Knowing that each of these values only appeared once in the subject program's source-code, this is a very promising finding to the MI; especially the occurrences of values in the UTF8 format that are obtained from the Windows machine.

However, searching the same memory dumps for occurrences in UTF16 format shows the variables' usage, in which there is a difference between the used variable types and the occurrences of their values that are impacted by these various execution states. The *right* side of Figure 4a shows the exact occurrences for these values during each of the used states and on each of the investigated platforms. For example, it is clear that there is a difference between the memory footprint for the same Java program when it runs on different platforms. The



(a) Java Based Heatmap (b) C++ based Heatmap

Figure 4: The occurrences of each variable V_i during each state S_i. (a): Behavior of **Java** on different platforms. (b): Behavior of **C++** on different platforms.

right upper side of Figure 4a shows these occurrences on a Windows machine, most of the values are successfully identified in corresponding memory dumps (states) after S_7, except for the V_8, which represents *local values in an overridden base class virtual method that is never used.* However, knowing that all platforms are using the same version of the JVM, the middle and bottom right sides of Figure 4a show the increasing number of values that are *unlikely* to be found using this UTF16 format on both Mac OS and Linux machines, respectively. For example, values of V_3, V_6, V_8, V_9, and V_{10} are never found using UTF16 format in any of the memory dumps of both Mac OS and Linux machines. Other values of variable types can be found, but highly impacted by the execution scenario (state). For example, only V_1 and V_2 are found after states S_1 and S_2, respectively. Other variables' values such as V_4 and V_5 are only found after states S_6 and S_8, respectively.

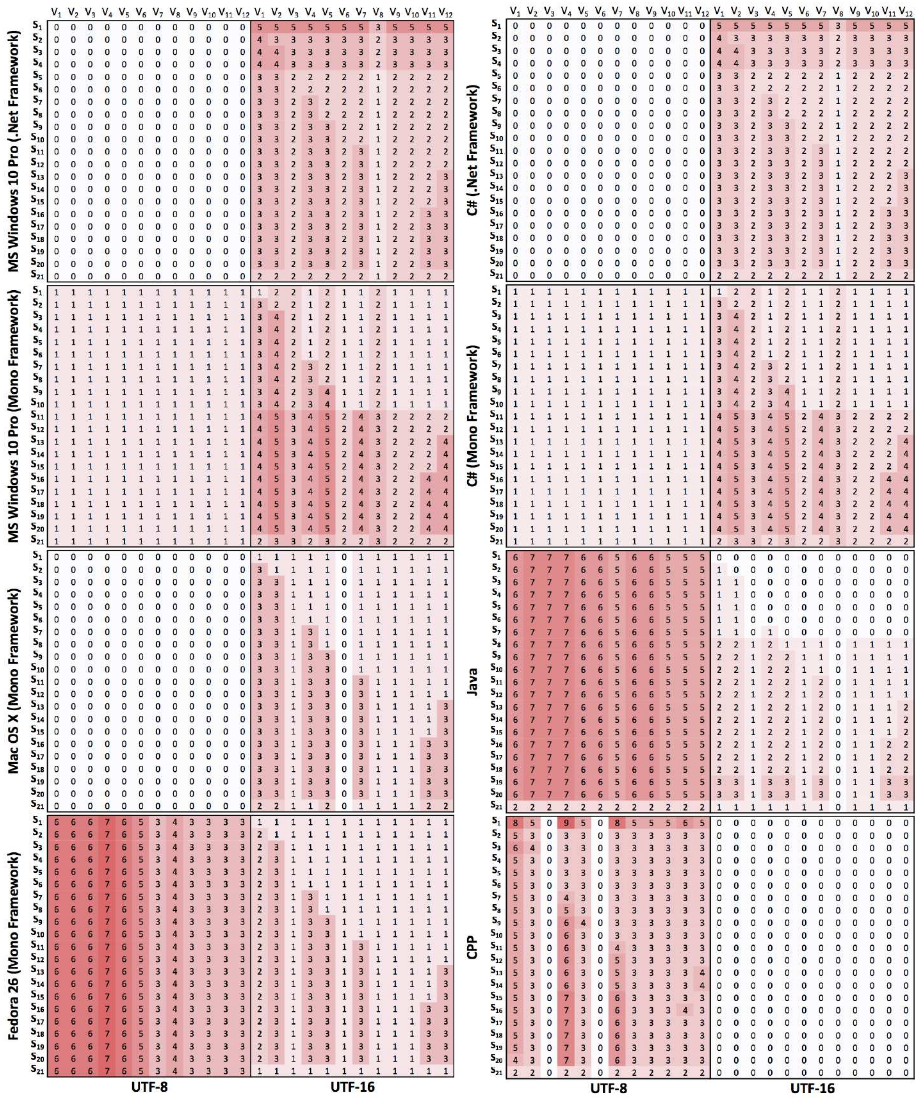
Additionally, Java results that are obtained from Mac OS and Linux machines are almost very similar. Whereas the results obtained from the Windows machine are better in the number of occurrences than those that are obtained from the other two platforms. This means, obtaining a DE from a Java program a Windows machine is better than obtaining the same DE from the same Java program on either Linux or Mac OS.

6.1.2 C++ Based Results

Figure 4b presents the heatmap with the number of occurrences for each variable type V_1 to V_{12} and execution state S_1 to S_{21}, each S_i represents the $dump_i$ on all of Windows, Mac OS and Linux, respectively. The *left* sides of this heatmap shows these successfully identified occurrences in UTF8 format whereas the *right* side presents these found occurrence for the UTF16 format. It is clear that most (not all) of the 12 types of variables provide the MI with a good chance of locating uniquely identified C++ values using the UTF8 format. These values belong to the presumed C++ program's execution state on all three investigated platforms. However, looking at the left side, we find that the number of occurrences for all variables' types (except V_3 and V_6) on the Windows machine is generally better than those same corresponding occurrences for the same C++ program when it runs on either of Mac OS or Linux machines. However, on Windows and Linux platforms, both of V_3 and V_6 are unlikely to be found in all 21 execution scenarios S_1 to S_{21}. In contrast, the number of occurrences for these V_3 and V_6 are getting better for the MI after S_{14} on the Mac OS machine.

On the other hand, identifying these same values using the UTF16 shows that it is highly unlikely to locate or identify any of the investigated variables V_1 to V_{12} during all of the execution scenarios S_1 to S_{21} on both of Windows and Linux machines; see the right side of Figure 4b. It is clear that the number of occurrences obtained using the UTF16 format are all zeros on both of Windows and Linux machines. However, the number of occurrences for these same variables' types are showing some potential availability on the Mac OS machine after S_{13}; except for variable type V_2 that remains unlikely to be found in all execution states. The number of occurrences in these states is ranging from 1 to 2, which can be considered a reasonable situation; knowing that the occurrences of each variable in the subject program's source-code is appeared only once.

Additionally, the number of occurrences obtained from Windows machine is better for the C++ program than this same program that runs on the other two platforms. However, its results obtained using the UTF16 is not promising at all on both Windows and Linux machines.



(a) C# Based Heatmap (b) MS Windows Based Heatmap

Figure 5: The occurrences of each variable V_i during each state S_i. (a) Behavior of **C#** on different platforms. (b) Behavior of different languages on **Windows**

6.1.3 C# Based Results

The third language based comparison targets C# on different platforms and frameworks. This comparison is applied on three sub-levels. *First*, it compares between the same C# program when it runs with the support from the *.Net* framework on Windows against its run on the same Windows machine with the support from the Mono framework. *Second*, it compares between this C# program when it runs with the support from the Mono framework on both Mac OS and Linux machines. *Finally*, it compares this same C# program when it runs with the support from Mono framework on a none Windows machine (Mac OS and Linux) against its run on Windows with the support from the same Mono framework. Results are discussed below.

6.1.3.1 First, C# with .Net vs. Mono on Windows:

Figure 5a shows the heatmap for the occurrences of the values obtained from the same C# program on different platforms during various execution states. The *upper* two parts of this heatmap show the difference between two different runs of the same C# program on a Windows 10 Pro machine, once when it runs with the support from the *.Net* and the other when it runs with the support from the Mono framework (both on the same machine). The *left* side of this heatmap shows the obtained results using the UTF8 format whereas the *right* side shows the results of these occurrences using the UTF16 format. Thus, it is clear that the number of occurrences of the UTF8 for the same C# program are all zeros when it runs with the support from the native *.Net* framework and all are ones when it runs with support from the Mono framework. Hence, knowing that each of these values are only appeared once in the subject C# program's source-code, this means that these results form the Mono support is not that bad. However, the *right* side of Figure 5a shows the number of occurrences using the UTF16 format. It is clear that the results from the *.Net* framework and the Mono framework are highly promising to the MI when these C# values are identified using the UTF16 (on Windows machines). In all scenarios and variable types, the number of occurrences is ranging from 1 to 4 and sometimes to 5 in the case of the native *.Net* support.

6.1.3.2 Second, C# on a None Windows Machine:

The *lower* two parts Figure 5a present the heatmap that compares between two different runs of the same C# program; when it runs with the support from the Mono framework on Mac OS and Fedora Linux. Looking at the results obtained from these memory dumps, we find that identifying C# values using the UTF8 on a Mac OS machine is *highly unlikely*; all 12 variables' types (V_1 to V_{12}) are not found on all of the dumps that are captured for the corresponding execution states (S_1 to S_{21}). In contrast, on the Fedora Linux machine, the number of occurrences for the same set of variables from the same C# program is ranging from 3 to 7. Hence, knowing that each of these values are only appeared once in the subject program's source-code, this is very promising to the MI. See the *left* side of the Mac OS and Fedora Linux parts of Figure 5a for the exact numbers of occurrences. On the contrary, identifying the same set of values using the UTF16 format shows almost similar occurrences for these values on both Mac OS and Linux machine; with the exception of V_6 variable type that its value was not found in any of the execution states on a Mac OS machine. See the *right* side of Figure 5a for the exact number of occurrences. Hence, the only small difference appears on the last execution state S_{21}, which represents the scenario in which *the presumed program was already terminated when the corresponding memory dump is captured*; this S_{21} state produces better occurrences for some variables on Mac OS than it is on the Linux machine.

6.1.3.3 Finally, C# with Mono Framework:

This part compares between the results obtained from the C# program when it runs on Windows with the support from Mono framework, the *upper* part of Figure 5a, against its run on both Mac OS and Linux machines with the support from the same Mono framework, the *lower* part of Figure 5a. It is clear that the occurrences of values on the Linux machine are fare much better than those obtained from the same C# program when it runs on Windows with Mono framework support; in which the number of UTF8 occurrences for all 12 values V_1 to V_{12} and all execution states S_1 to S_{21} are 1, but all occurrences are zeros on Mac OS, see the *left* side of Figure 5a. Hence, these values from the Windows and Linux machines are very processing to the MI; knowing that these values V_1 to V_{12} are only occurred once in the subject C# program's source-code. In contrast, identifying the same set of values in dumps S_1 to S_{21} using the UTF16 shows that the results for the Mono framework running on Windows is relatively better than those occurrences obtained from both Mac OS and Linux machines, see the *right* side of Figure 5a.

6.2 Level 2: OS Based Results

This level compares between similar programs that are written in three different languages when each is running on the same platform.

6.2.1 MS Windows Based Results

Figure 5b shows the heatmap with the number of concurrences obtained from the set of similar programs with respect to language differences when each is running on the same Windows machine. The *left* side of Figure 5b represent occurrences that are identified using the UTF8 format whereas the *right* side of Figure 5b represent occurrences identified using the UTF16 format. This heatmap shows that Java is the best when it comes to the number of occurrences that are identified using the UTF8 on a Windows machine. Whereas the C# program that runs with the support from the $.Net$ framework is the worst for the MI. However, C# programs that are supported using the Mono framework and C++ programs show moderate results (number of occurrences), in which the occurrences from the C# program are all 1 and the occurrences from the C++ program are relatively better with the exception that V_3 and V_6 are unlikely to be found in the RAM of the Windows machine using the UTF8 format. However, this heatmap shows that C# programs are the best in terms of the occurrences of their values when these values are identified using the UTF16 format. In contrast, identifying values using the UTF16 shows moderate occurrences on the Mac OS machine whereas it is highly unlikely to be able to identify any of these values using the UTF16 from any C++ program on a Windows machine.

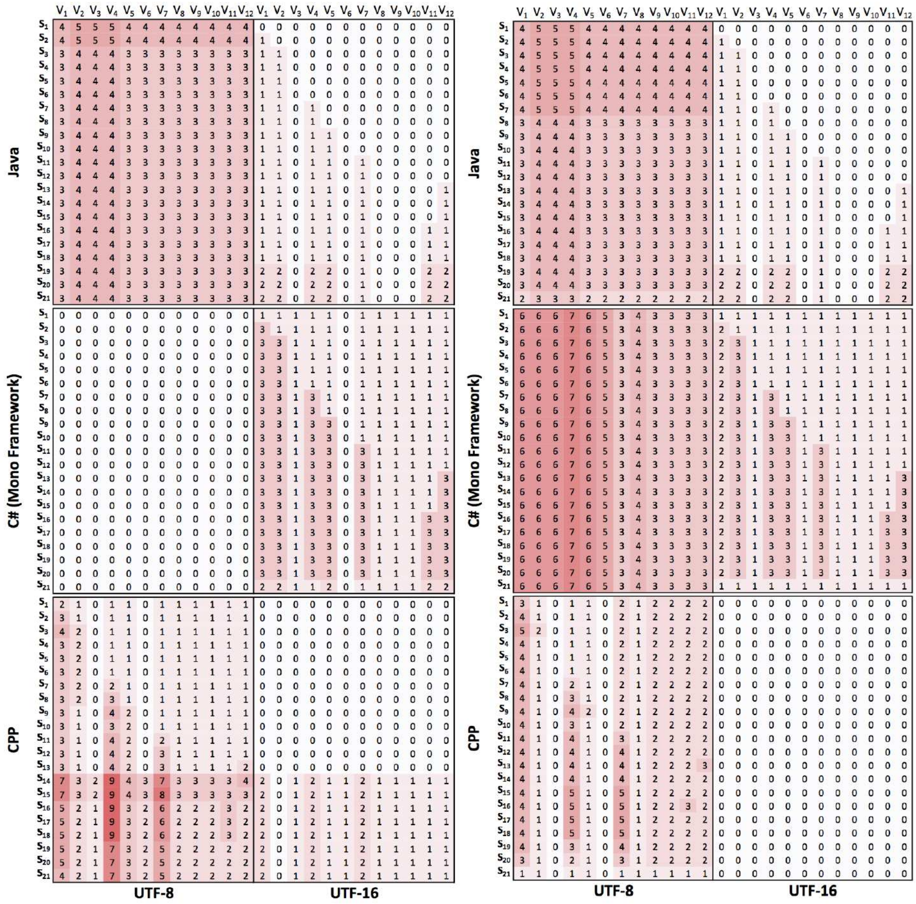
6.2.2 Mac OS Based Results

Figure 6a presents the heatmap with the number of occurrences obtained from the same set of similar programs, all of which run on the same Mac OS machine. This heatmap shows that the best results, for the MI, can be obtained from a Java program; when these values are identified using the UTF8 format, see the *left* side of the Java part of Figure 6a. Whereas the worst results that can be identified using the UTF8 formats are obtained from the C# program, in which all UTF8 values are zeros. The results from the C++ language shows moderate number of occurrences that are successfully found using the UTF8 format.

However, the *right* side of Figure 6a shows that C# presents the best occurrences when these values are identified using the UTF16 format; only variable V_6 was unlikely to be identified. In contrast, identifying values using the UTF16 shows moderate occurrences on the Mac OS machine for Java programs and bad results for the C++ programs, which are highly unlikely to identify values using the UTF16 from any C++ program before state S_{14}.

6.2.3 Linux Based Results

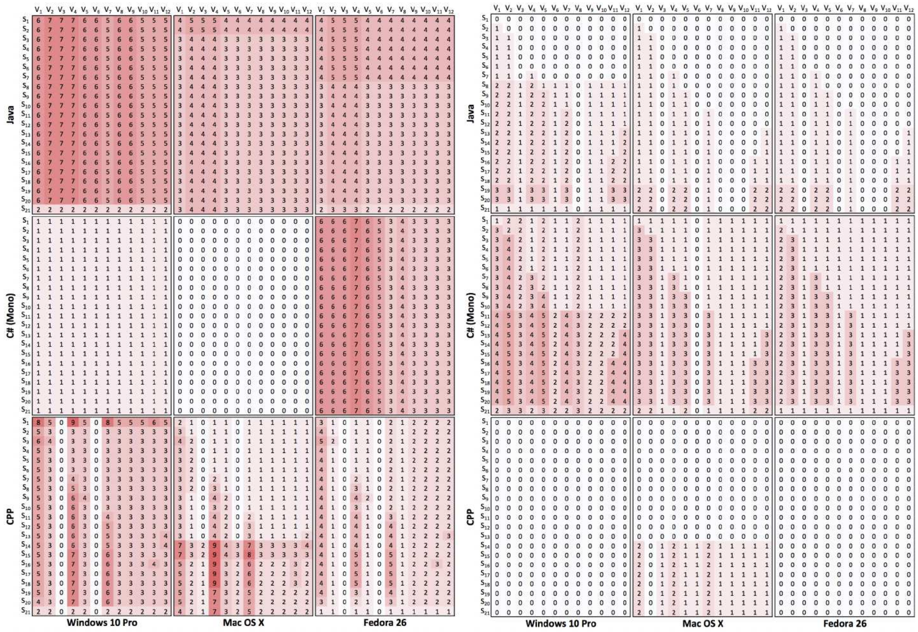
Figure 6b presents the heatmap with the number of occurrences obtained from the same set of similar programs, all of which run on the same Fedora 26 Linux machine. It shows that the best results, for the MI, can be obtained from a C# program that is running on a Linux machine with the support from the Mono framework. The *left* side of Figure 6b presents these occurrences in UTF8 format whereas the *right* side of Figure 6b shows these results for the UTF16 format. Occurrences show promising results that can be found using UTF8 and UTF16 encoding formats; only for the C# program. The second best results on a Linux machine can obtained from a Java program, especially for source-code related values that are identified using the UTF8 format, see the left side of the Java part of Figure 6b. Hence, C++ shows a reasonable results using the



(a) Mac Based Heatmap (b) Linux Based Heatmap

Figure 6: The occurrences of each variable V_i during each state S_i. (a): Behavior of different languages on **Mac OS**. (b): Behavior of different languages on **Linux**.

UTF8 format, but not for all types of variables; values for variables of type V_3 and V_6 are unlikely to be found using the UTF8 format from the C++ program on the Linux machine. On the contrary, identifying values using the UTF16 format shows good results from the C# programs on the Linux machine. In contrast, C++ shows that values in the UTF16 are unlikely to be found in the RAM of the used Linux machine. However, the results from the Java program shows moderate results when these values are identified using the UTF16 format, except for variables such as V_3, V_6, V_8, V_9, and V_{10} that are unlikely to be found in any of the 21 execution states.



(a) UTF8. (b) UTF16.

Figure 7: UTF8 vs. UTF16 Heatmap: occurrences of each variable V_i during each execution state S_i from all investigated languages and on all used platforms. Different languages utilize different encoding schemes on different platforms.

6.3 Level 3: Text-Encoding Based Results

Figure 7 shows that generally identifying values using the UTF8 format is better than identifying these same values using UTF16. The presented heatmap in Figure 7a is relatively darker than the one presented in Figure 7b. Obviously, this is valid with with some exceptions. For example, the MI is unlikely to identify UTF8 values from the memory dumps of a C# program running on Mac OS. Figure 7a shows that Java values identified using the UTF8 presents the best results on all platforms. Whereas, identifying the same set of values from the Java program using the UTF16 format shows a moderate occurrences, which are ranging from 0 to 3. Nevertheless, the UTF16 in some cases is better than the UTF8. For example, C# with the Mono project shows better results with UTF16 over UTF8 on both Windows and Mac. However, it is clear that C# on Linux has more identified occurrences with UTF8 than it is identified with the UTF16, for the same set of values. Java and C++ languages show better occurrences with UTF8 in all platforms over UTF16. Thus, we can not prefer any encoding over the other, it depends on the platform and the used language. Figure 8 presents a combined heatmap view for all investigated languages on all

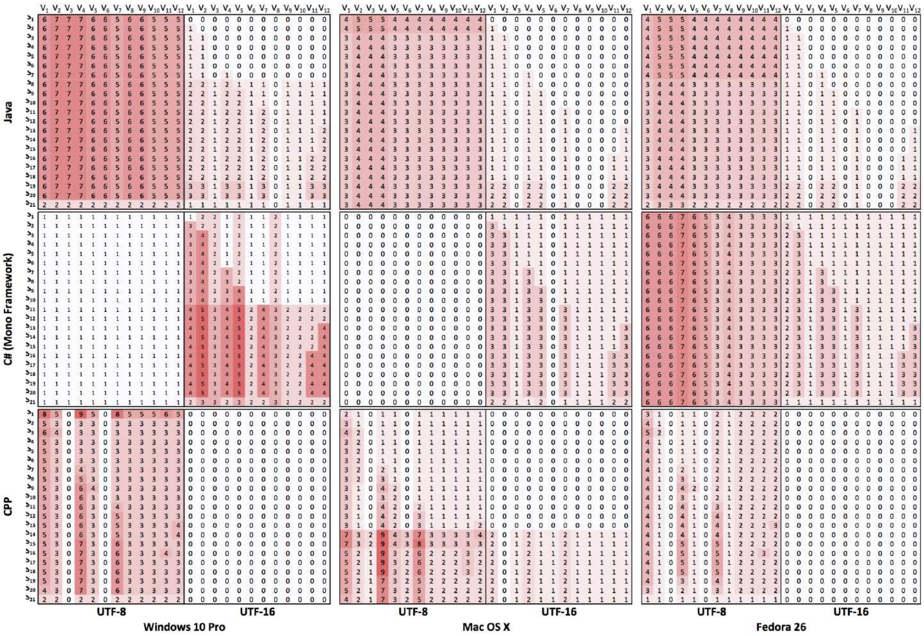


Figure 8: Combined Heatmap: Language vs. OS: the occurrences of each variable V_i during each execution state S_i from all investigated languages and platforms. It is clear that the same language behaves differently on different platforms and different languages behave differently on the same platform.

platforms, each of which is vertically divided for occurrences that are identified using the UTF8 format and the UTF16 format.

7 Related Work

RAM memory contains a vital source of information [Mazurczyk et al., 2013, Dezfoli et al., 2013] that can be used in support for legal actions against perpetrators in digital forensic cases [Al-Sharif et al., 2015]. Thus, many researchers investigate various situations and scenarios to identify artifacts that would be used as a DE. Many of these researchers focus on the potential software that can be used during the offense [Rafique and Khan, 2013, Al-Sharif et al., 2018a, Al-Sharif et al., 2018b, Al-Saleh and Al-Sharif, 2013]. Otsuki et al. introduced the design and implementation for a method that builds the stack traces from a memory dump of the Windows x64 environment [Otsuki et al., 2018]. Malachi et al. utilized the binary analysis techniques and provided an automated in-memory malware detection that employing machine learning [Jones, 2018]. Stelly et al. discussed the design and implementation of a domain specific language (DSL) called *nugget*, which aims to enable the practical formal specification of digital forensic computations in a tool-agnostic fashion [Stelly and Roussev, 2018]. Johannes et al. proposed a technique to investigate firmware and its major components. He proposed a new technique that improves forensics imaging based on PCI introspection and page table mapping [Stüttgen et al., 2015]. Shashidhar et al. targeted the *prefetch* folder and its potential value to the MI. This folder is used to speed up the startup time of a program on a Windows machine [Shashidhar and Novak, 2015]. Joseph et al. provided a summarized study of the forensics methods and their domains including the anti-forensic techniques [Joseph and Norman, 2019]. Singh et al. presented a taxonomy of various program execution artifacts and evaluated eleven of them that are running on Windows and presented their forensic values [Singh and Singh, 2018].

However, this paper utilizes the source-code and/or binary-code related values to identify the software usage on different platforms. Its goal is to identify and validate whether the presumed software is actually used during the digital crime. The study presented in this paper compares between three diffident languages, each of which is evaluated on three different platforms.

8 Discussions and Future Work

The results presented in this paper are based on our carefully designed software. We are not using a real world open-source or commercial software, that is because our experimentations are designed to evaluate the differences between the behaviors of the same language on different platforms and, also, to evaluate the differences between the behaviors of different languages on the same platform. Though, it is not that hard to find the same software running on different platforms, but it would be difficult enough to find the same exact program written in three different languages in order to be evaluated on the same platform; a state by state comparison is performed in order to find the exact differences in the corresponding memory footprints (digital evidence) [James and Gladyshev, 2015]. Thus, the research presented in this paper establishes a base case for a promising future work, in which we are planning to investigate open-source and closed-source real world software. Of course, future studies will be guided by the results that are presented in this paper.

However, in this paper, we tested the results of programs written in three different languages: Java, C++, and C#, each of which is tested running on Windows 10 Pro, Mac OS, and Fedora Linux. Hence, in all experiments, the prepared programs are semantically identical with respect to the syntactic language differences. The execution of the presumed programs are supported by different underneath implementations that could potentially affect their memory forensic evidence on different platforms.

Nonetheless, the investigated languages differ in their OS support. For instance, Java operates 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. This might explain the prominence of the digital evidence of Java based programs over other languages, even though, it shows platform differences, which is expected due to the differences in the implementation of these JVMs according the host OS. Thus, traces of various execution values (evidences) showed different footmarks on different platforms.

Additionally, whenever the source-code of the presumed software is not available, software reverse engineering and its supportive tools, such as IDA disassembler [6] and OllyDbg [7], can provide a considerable assistant to the MI in analyzing the executable. For example, when a Java source-code is compiled, it is converted into bytecode that is to be executed by the JVM. In comparison to C/C++ programs, Java's bytecode is machine independent. Thus, when it comes to understanding the binary representation, Java bytecode retain more information than the native code. Thus, it is relatively easier to decompile bytecode and with greater accuracy. Freely available tools allow the Java bytecode to be converted into Java source-code, and the resulting source-code is likely to be very similar to the original one [Eilam, 2011]. Additionally, there are tools available to obfuscate Java, thereby making the memory investigators job more challenging, but none are particularly strong—even highly obfuscated Java bytecode is generally easier to reverse than unobfuscated native machine code [Stamp, 2011].

In contrast, C# is supported by the Microsoft *.Net* framework, and its equivalent open-source cross-platform Mono framework that supports C# executables on none Windows machines. Even though the MSIL code is stored in a *.exe* file, this file does not contain native executable code; it contains information needed by the JIT compiler to execute the code [Esposito and Ciceri, 2016]. Just like in the Java case, this MSIL simplify the process of understanding the execution logic [Kumar et al., 2015].

For future work, in addition to investigating real world applications, we are planning to investigate the environments of small devices such as phones and tablets. Although, this paper targets the strictly typed languages, we are planning to investigate dynamically typed languages including the scripting languages such as Python, Ruby, R, Icon/Unicon and others. For example, JavaScript is one of the most commonly used dynamic languages that is widely used for web development, thus we plan to forensically investigate its behavior along with the HTTP protocol [Graniszewski et al., 2018]. Furthermore, we

[6] IDA disassembler, `www.hex-rays.com/index.shtml`

[7] OllyDbg, `http://www.ollydbg.de/`

are looking forward to investigate similar scenarios for other data types and 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 and web services.

9 Conclusion

This paper utilizes information from the source-code and/or the disassembled binary of a software and employs it's execution data to help the MI establishes the DE against a perpetrator. Experiments are designed to target the potential DE that would prove the actual software usage during various execution states and scenarios. Memory footprints from similar programs that are developed in different languages are compared on the same platforms. Moreover, the memory footprint of each program is compared between different platforms. Three different factors that can affect the digital evidence in memory are examined. It evaluated how the DE is impacted by the used platform, the used programming language, and finally how it is impacted by the implicitly used encoding schemes such as the UTF8 and UTF16. The result and discussion sections present in details how and what method the MI are encouraged to employ in order to obtain the best amount of digital evidence that can be identified in RAM based on the host platform, language used, and the encoding of string values and literals.

Results show that utilizing source-code and binary-code information can be valuable to the MI. Moreover, some languages show better results on specific platform over others. However, execution related data that is in memory can often be used to establish the DE that the perpetrator is actually used the software during the cybercrime. Various variables' values and string literals and non-literals related to the program execution are successfully identified during different scenarios. This should permit law enforcement agencies to take legal actions against criminals in the court of law. Finally, results show that when it comes to investigating memory footprints of a used software, the MI cannot rely on one technique always, instead, the MI need to take the environment (platform), used language, and the encoding schemes into consideration in order to successfully obtain relative and unique execution values (evidence). The results presented in this paper should assist the MI toward the right direction, in which to obtain the most valuable evidence.

References

[Al-Saleh and Al-Sharif, 2013] Al-Saleh, M. and Al-Sharif, Z. (2013). Ram forensics against cyber crimes involving files. In *The Second International Conference on Cyber Security, Cyber Peacefare and Digital Forensic (CyberSec2013)*, pages 189–197. The Society of Digital Information and Wireless Communication (SDIWC).

[Al-Sharif, 2016] Al-Sharif, Z. A. (2016). Utilizing program's execution data for digital forensics. In *The Third International Conference on Digital Security and Forensics (DigitalSec2016)*, pages 12–19.

[Al-Sharif et al., 2017] Al-Sharif, Z. A., Al-Saleh, M. I., and Alawneh, L. (2017). Towards the memory forensics of oop execution behavior. In *2017 8th International Conference on Information, Intelligence, Systems Applications (IISA)*, pages 1–6.

[Al-Sharif et al., 2018a] Al-Sharif, Z. A., Al-Saleh, M. I., Alawneh, L. M., Jararweh, Y. I., and Gupta, B. (2018a). Live forensics of software attacks on cyber–physical systems. *Future Generation Computer Systems*.

[Al-Sharif et al., 2018b] Al-Sharif, Z. A., Bagci, H., Zaitoun, T. A., and Asad, A. (2018b). *Towards the Memory Forensics of MS Word Documents*, pages 179–185. Springer International Publishing, Cham.

[Al-Sharif et al., 2015] Al-Sharif, Z. A., Odeh, D. N., and Al-Saleh, M. I. (2015). Towards carving pdf files in the main memory. In *The International Technology Management Conference (ITMC2015)*, pages 24–31. The Society of Digital Information and Wireless Communication (SDIWC).

[Bauman et al., 2018] Bauman, E., Lin, Z., and Hamlen, K. W. (2018). Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proc. NDSS*, pages 40–47.

[Bobowska et al., 2018] Bobowska, B., Choraś, M., and Woźniak, M. (2018). Advanced analysis of data streams for critical infrastructures protection and cybersecurity. *Journal of Universal Computer Science*, 24(5):622–633.

[Bovet and Cesati, 2005] Bovet, D. P. and Cesati, M. (2005). *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.".

[Butler, 2016] Butler, J. (2016). Physical memory forensics system and method. US Patent 9,268,936.

[Case and Richard, 2017] Case, A. and Richard, G. G. (2017). Memory forensics: The path forward. *Digital Investigation*, 20(Supplement C):23 – 33. Special Issue on Volatile Memory Analysis.

[Conlan et al., 2016] Conlan, K., Baggili, I., and Breitinger, F. (2016). Anti-forensics: Furthering digital forensic science through a new extended, granular taxonomy. *Digital Investigation*, 18(Supplement):S66 – S75.

[Craiger, 2005] Craiger, P. (2005). Recovering digital evidence from linux systems. In *IFIP International Conference on Digital Forensics*, pages 233–244. Springer.

[Dezfoli et al., 2013] Dezfoli, F. N., Dehghantanha, A., Mahmoud, R., Sani, N. F. B. M., and Daryabar, F. (2013). Digital forensic trends and future. *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, 2(2):48–76.

[Eilam, 2011] Eilam, E. (2011). *Reversing: secrets of reverse engineering.* John Wiley & Sons.

[Esposito and Ciceri, 2016] Esposito, A. and Ciceri, M. (2016). *Reactive Programming for. NET Developers.* Packt Publishing Ltd.

[Graniszewski et al., 2018] Graniszewski, W., Krupski, J., and Szczypiorski, K. (2018). Somsteg-framework for covert channel, and its detection, within http. *Journal of Universal Computer Science*, 24(7):864–891.

[Haller et al., 2016] Haller, I., Slowinska, A., and Bos, H. (2016). Scalable data structure detection and classification for c/c++ binaries. *Empirical Software Engineering*, 21(3):778–810.

[James and Gladyshev, 2015] James, J. I. and Gladyshev, P. (2015). Automated inference of past action instances in digital investigations. *International Journal of Information Security*, 14(3):249–261.

[Jensen et al., 2017] Jensen, D., Lundkvist, A., and Hammouda, I. (2017). On the significance of relationship directions in clustering algorithms for reverse engineering. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 1239–1244, New York, NY, USA. ACM.

[Jones, 2018] Jones, M. (2018). Automated in-memory malware/rootkit detection via binary analysis and machine learning.

[Joseph and Norman, 2019] Joseph, D. P. and Norman, J. (2019). An analysis of digital forensics in cyber security. In *First International Conference on Artificial Intelligence and Cognitive Computing*, pages 701–708. Springer.

[Josey et al., 2004] Josey, A., Cragun, D., Stoughton, N., Brown, M., Hughes, C., et al. (2004). The open group base specifications issue 6 ieee std 1003.1. *The IEEE and The Open Group*, 20(6).

[Kao et al., 2019] Kao, D.-Y., Wu, N.-C., and Tsai, F. (2019). The governance of digital forensic investigation in law enforcement agencies. In *2019 21st International Conference on Advanced Communication Technology (ICACT)*, pages 61–65. IEEE.

[Kälber et al., 2013] Kälber, S., Dewald, A., and Freiling, F. C. (2013). Forensic application-fingerprinting based on file system metadata. In *2013 Seventh International Conference on IT Security Incident Management and IT Forensics*, pages 98–112.

[Kruegel et al., 2004] Kruegel, C., Robertson, W., Valeur, F., and Vigna, G. (2004). Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18.

[Kumar et al., 2015] Kumar, K., Kaur, P., and GNDU, A. (2015). A generalized process of reverse engineering in software protection & security. *Int. J. Computer Science & mobile Computing*, 4(5):534–544.

[Leukfeldt et al., 2017] Leukfeldt, E. R., Kleemans, E. R., and Stol, W. P. (2017). Cybercriminal networks, social ties and online forums: social ties versus digital ties within phishing and malware networks. *The British Journal of Criminology*, 57(3):704–722.

[Ligh et al., 2014] Ligh, M. H., Case, A., Levy, J., and Walters, A. (2014). *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory.* John Wiley & Sons.

[Lindholm et al.,] Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. The java virtual machine specification-java se 8 edition, march 2014.

[Mazurczyk et al., 2013] Mazurczyk, W., Szczypiorski, K., Tian, H., and Liu, Y. (2013). Trends in modern information hiding: techniques, applications and detection. *Security and Communication Networks*, 6(11):1414–1415.

[Mesbah et al., 2017] Mesbah, A., Lanet, J.-L., and Mezghiche, M. (2017). Reverse engineering a code without the code: Reverse engineering of a java card dump. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, ROOTS, pages 1:1–1:8, New York, NY, USA. ACM.

[Montasari and Hill, 2019] Montasari, R. and Hill, R. (2019). Next-generation digital forensics: Challenges and future paradigms. In *2019 IEEE 12th International Conference on Global Security, Safety and Sustainability (ICGS3)*, pages 205–212. IEEE.

[Moser et al., 2017] Moser, M., Pfeiffer, M., and Pichler, J. (2017). Towards reverse engineering of intermediate code for documentation generators. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 553–554.

[Nadi and Holt, 2014] Nadi, S. and Holt, R. (2014). The linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process*, 26(8):730–746.

[Otsuki et al., 2018] Otsuki, Y., Kawakoya, Y., Iwamura, M., Miyoshi, J., and Ohkubo, K. (2018). Building stack traces from memory dump of windows x64. *Digital Investigation*, 24:S101 – S110.

[Pridgen et al., 2017] Pridgen, A., Garfinkel, S., and Wallach, D. S. (2017). Picking up the trash: Exploiting generational gc for memory analysis. *Digital Investigation*, 20(Supplement):S20 – S28. DFRWS 2017 Europe.

[Rafique and Khan, 2013] Rafique, M. and Khan, M. (2013). Exploring static and live digital forensics: Methods, practices and tools. *International Journal of Scientific &*

Engineering Research, 4(10):1048–1056.

[Rupprecht et al., 2017] Rupprecht, T., Chen, X., White, D. H., Boockmann, J. H., Lüttgen, G., and Bos, H. (2017). Dsibin: identifying dynamic data structures in c/c++ binaries. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 331–341. IEEE.

[Salajegheh et al., 2018] Salajegheh, M., Gathala, S. A. K., Das, S. M., and Islam, N. (2018). System and method of performing online memory data collection for memory forensics in a computing device. US Patent App. 15/248,178.

[Sali and Khanuja, 2019] Sali, V. R. and Khanuja, H. (2019). Ram forensics: The analysis and extraction of malicious processes from memory image using gui based memory forensic toolkit. In *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, pages 1–6. IEEE.

[Schatz and Cohen, 2017] Schatz, B. and Cohen, M. (2017). Advances in volatile memory forensics. *Digital Investigation*, (20):1.

[Schildt, 2014] Schildt, H. (2014). *Java: The Complete Reference*. McGraw-Hill Education Group.

[Schwartz et al., 2018] Schwartz, E. J., Cohen, C. F., Duggan, M., Gennari, J., Havrilla, J. S., and Hines, C. (2018). Using logic programming to recover c++ classes and methods from compiled executables. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 426–441. ACM.

[Sen and Mall, 2016] Sen, T. and Mall, R. (2016). Extracting finite state representation of java programs. *Software & Systems Modeling*, 15(2):497–511.

[Shashidhar and Novak, 2015] Shashidhar, N. K. and Novak, D. (2015). Digital forensic analysis on prefetch files. *International Journal of Information Security Science*, 4(2):39–49.

[Singh and Singh, 2018] Singh, B. and Singh, U. (2018). Program execution analysis in windows: A study of data sources, their format and comparison of forensic capability. *Computers & Security*, 74:94 – 114.

[Sitaraman and Venkatesan, 2005] Sitaraman, S. and Venkatesan, S. (2005). Low-intrusive consistent disk checkpointing: A tool for digital forensics. *Journal of Universal Computer Science*, 11(1):20–36.

[Stamp, 2011] Stamp, M. (2011). *Information security: principles and practice*. John Wiley & Sons.

[Stelly and Roussev, 2018] Stelly, C. and Roussev, V. (2018). Nugget: A digital forensics language. *Digital Investigation*, 24:S38 – S47.

[Stevens and Rago, 2013] Stevens, W. R. and Rago, S. A. (2013). *Advanced programming in the UNIX environment*. Addison-Wesley.

[Stüttgen et al., 2015] Stüttgen, J., Vömel, S., and Denzel, M. (2015). Acquisition and analysis of compromised firmware using memory forensics. *Digital Investigation*, 12:S50–S60.

[Subedi et al., 2018] Subedi, K. P., Budhathoki, D. R., and Dasgupta, D. (2018). Forensic analysis of ransomware families using static and dynamic analysis. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 180–185. IEEE.

[Thomas et al., 2013] Thomas, S., Sherly, K., and Dija, S. (2013). Extraction of memory forensic artifacts from windows 7 ram image. In *2013 IEEE Conference on Information & Communication Technologies*, pages 937–942. IEEE.

[Tien et al., 2017] Tien, C. W., Liao, J. W., Chang, S. C., and Kuo, S. Y. (2017). Memory forensics using virtual machine introspection for malware analysis. In *2017 IEEE Conference on Dependable and Secure Computing*, pages 518–519.

[Yang et al., 2007] Yang, J., Li, T., Liu, S., Wang, T., Wang, D., and Liang, G. (2007). Computer forensics system based on artificial immune systems. *Journal of Universal Computer Science*, 13(9):1354–1365.