Future Generation Computer Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Live forensics of software attacks on cyber–physical systems

Ziad A. Al-Sharif [a,*], Mohammed I. Al-Saleh [b], Luay M. Alawneh [a], Yaser I. Jararweh [b], Brij Gupta [c]

[a] Software Engineering Department, Jordan University of Science and Technology, Irbid, 22110, Jordan
[b] Computer Science Department, Jordan University of Science and Technology, Irbid, 22110, Jordan
[c] National Institute of Technology, Kurukshetra, India

ARTICLE INFO

Article history:
Received 31 December 2017
Received in revised form 5 May 2018
Accepted 14 July 2018
Available online xxxx

Keywords:
Digital forensics
Memory forensics
Program's execution behavior
Execution state
Execution path
Digital evidence
Evidence collection process

ABSTRACT

Increasingly, Cyber–physical Systems are expected to operate in different environments and interconnect with a diverse set of systems, equipment, and networks. This openness to heterogeneity, diversity, and complexity introduces a new level of vulnerabilities, which adds to the consistent need for security including the digital forensics capabilities. Digital investigators utilize the information on the attacker's computer to find clues that may help in proving a case. One aspect is the digital evidence that can be extracted from the main memory (RAM), which includes live information about running programs. A program's states, represented by variables' values, vary in their scope and duration. This paper explores RAM artifacts of Java programs. Because JVMs can run on various platforms, we compare the same program on three different implementations of JVM from forensic perspectives. Our investigation model assumes no information is provided by the underlying OS or JVM. Our results show that a program's states can still be extracted even after the garbage collector is explicitly invoked, the software is stopped, or the JVM is terminated. This research helps investigators identify the software used to launch the attack and understand its internal flows. Investigators can utilize this information to accuse the perpetrators and recover from attacks.

1. Introduction

Cyber–physical Systems (CPS) can be defined as the result of integration between computations and physical processes, in which computations affect the physical processes and vice versa [1–3]. However, due to the recent technological advances in computations and communications, more of these CPS systems are connected to the Internet and communicating with each other and with traditional computing machines [4]. These machines can be hijacked and compromised by perpetrators, who may intend to do harmful actions or compromise the functionalities of these CPS systems. This mandates the need to elevate the security measurements, including the Digital Forensics (DF) capabilities. Digital Investigators (DI) need to be able to detect the perpetrator's activities; including the ability to find and identify any possible harmful or illegal concession made on the system and to uncover the perpetrator.

Though, often these perpetrators utilize computers and software to execute or cover their attacks. Locating the presumed software on their machine's hard disk might not be a strong evidence about its actual usage. A definite evidence might be needed to prove that the perpetrator has actually used the software [5]. This evidence can be found in several places, one of which is the Main Memory (RAM) of the used machine. RAM encompasses vital information about the very recent states of a system such as its active processes and their execution states [6,7].

A software current execution state resides in RAM. However, often program's execution states are formalized by the execution paths of the encountered source code. Identifying this path can help in detecting the control-flow and data-flow of the presumed software and then characterizing the possible values of various variables that ought to be scattered over different RAM pages. Obviously, the use of object oriented programs increases the complexity of execution states, evidenced in part by the difficulty of its software testing process [8]. Software variables of an 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. Hence, a variable's storage, which is defined by the memory type such as stack and heap, determines the duration in which its value is allocated and released [9]. On the other hand, variables can be classified based on whether they are allowed to be changed during execution and how these changes

* Corresponding author.
E-mail addresses: zasharif@just.edu.jo (Z.A. Al-Sharif), misaleh@just.edu.jo (M.I. Al-Saleh), lmalawneh@just.edu.jo (L.M. Alawneh), yijararweh@just.edu.jo (Y.I. Jararweh), bbgupta@nitkkr.ac.in (B. Gupta).

https://doi.org/10.1016/j.future.2018.07.028
0167-739X/

are visible within various parts of the program's source code. For example, in Java, the program do not change variables' values that are marked with *final* once they are assigned, such variables are often initialized with literal values. These literals might be unique to the executable program and its execution state. Moreover, many other non-constant variables might be assigned with literal values too. When these literals are unique to the presumed program's source code and some of its execution paths, then retaining these values in RAM can be used to establish the Digital Evidences (DE), define the actual software usage and assert its distinct execution path.

Common DF tools and techniques are not designed and developed to stop an attack, but to identify the source of an incident, determine its type, preserve the DE, and analyze the findings [10]. This paper focuses on the Memory Forensics (MF) of the Java-based software usage. It identifies various DE that would be employed to confirm the software usage and its association with the crime. The research methodology presented in this paper is relying solely on mining the core dumps of the physical memory of the host machine and assuming no information is available from the Operating System (OS) or the used Java Virtual Machine (JVM).

In order to verify our approach, various experiments and scenarios are established, for each of which a RAM dump is created and analyzed. During the analysis process, various variables' scopes and memory types are assumed. Our experiments are designed to establish the DE between source code related values that obtained from the execution states of a running Java program and compare these DE when the internal implementation of the used JVM is different. JVMs that are running on all of MS Windows, Mac OS X, and Fedora Linux are investigated and the obtained DE are compared between similar scenarios.

Our results show that regardless of whether the program is running, the JVM is active or just stopped, the DI can employ knowledge from the presumed program's source code such as static and instance level variables and their potential values in order to confirm the actual usage of the software; based on identifying these values in corresponding RAM dumps. Hence, some values of local variables are not expected to be successfully located when their corresponding stack frames are not active, though most of which are successfully identified. Additionally, values of static variables and local variables of static methods are often found to have longer duration in memory than the instance related values. Furthermore, in most cases, dynamically allocated values are identified in RAM dumps even after the Garbage Collector (GC) is explicitly invoked or even after the program is terminated. A comprehensive comparison between our findings during various possible scenarios and variable states are ascertained on all investigated platforms.

The rest of this paper is organized as follows. Section 2 presents a deeper look at CPS systems, in which our research motivations are presented and the needed security measurements are highlighted. Section 3 reviews some of the background knowledge used in this paper. Section 4 presents our investigation model and explains how it employs information available in the program's source code to confirm that the program is actually used in the attack. Section 5 describes our experiments whereas our results are presented in Section 6. Section 7 presents our related work. Finally, our planned future work is presented in Section 8 whereas Section 9 concludes our findings.

2. CPS systems: New usage & security needs

The idea of Cyber–physical Processes (CPP) is not new. The term *embedded systems* is used, long before CPS, to describe engineered systems that combine physical parts of a system with a software that is built into it. Recently, this term is modernized to include the applications that are parts of aircrafts, autonomous cars, home appliances, weapons, robots, and more [11]. Hence, when such systems are closed boxes that do not expose their computing capabilities to the outside world, they are isolated and then protected from the perpetrators.

2.1. Motivation

Networking CPS systems is mandated by the new usage scenarios that are inspired by the various recent technological advances. For example, the ongoing increase in the number of physical things such as sensors, actuators, smart phones, tablets, gaming consoles, traditional desktop and laptop computers, along with the explosive increase in the usage of online networking services and applications, all of this motivates the development and the use of new CPS systems [12].

Therefore, this generates an emerging need to widen the CPS definition to include into consideration the environment, which incorporates the different types of computing devices including the Internet of Things (IoT) [13], which will conclusively combine daily used things of computers, devices, appliances, and machines that are connected to the Internet [14,15]. The utilization of these systems encouraged new services and applications and revolutionized our world in different fields through their tight interactions and automated decisions. All this is envisioned to be facilitated by the new generation of the 5G technologies that will be able to expand the interoperability between this diverse of devices and applications [16–21].

Moreover, the movement from cloud based to fog based computing or edge computing [22,23], in which the application logic including storages and analytics are placed on the actual device instead of a centric server. This serves better the heterogeneous communication network, but opens the door for vulnerabilities and endangers security [24,25]. Though, CPS forensic studies can benefit from the diverse literature in the field of digital forensic science, which refers to the mathematical, statistical, and computer science methods that are employed in order to collect, identify, analyze, and interpret digital evidences [26]. Thus, digital forensics would ensure the ability to identify, collect, and analyze malicious sources from involved nodes and infrastructures and help react to criminals' attacks.

2.2. Security threat

The technological advances will keep equipping our world with new CPS systems that will keep formulating new usage scenarios. Therefore, such a large scale would make it nearly impossible to guarantee security for every single subsystem, unless new and innovative security measurements are implemented [27–33].

For example, in this wide definition of CPS systems, the security and privacy issues are not fully defined. Additionally, issues relating to the confidentiality, integrity, and availability are not fully described [34]. Nonetheless, this opens a considerable threat that would shake the level of our trust and dependability on such systems [35]. Thus, prominence and superiority in such a future world that is filled with CPS systems will depend on the success on a variety of directions, one of which demands the evolving of security measurements including the DF and MF capacities that would facilitate the ability to digitally detect and identify perpetrators in order to apprehend them and stop their criminal acts.

This paper targets perpetrators who use software to organize their attacks on various CPS systems. It targets attackers who use software that are written in the Java programming language that is famous in its design goal of *write-once, run-anywhere* [36], which makes it one of the mostly used programming languages [37]. Our goal is to forensically experiment with different implementations

Please cite this article in press as: Z.A. Al-Sharif, et al., Live forensics of software attacks on cyber–physical systems, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.07.028.

of JVM and to compare the execution footprints of the same program on different platforms, these comparisons are made from the MF perspective and the DI point of view. Our aim is to investigate the differences in program's execution traces in RAM and its utilization in MF cases and the DE collection and establishing process. Even though Java is the target of our experimentation, we expect most of our findings to be significantly pertinent to similar programming languages and execution environments.

3. Background

A software process may employ various variables. In OOP languages such as Java, these 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 RAM. For example, the values of static variables are allocated by the runtime system before instances (objects) 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. Instance variables' duration in RAM is limited to the exact object instantiated from that class; each object has its own set of values. Moreover, local variables (method level) are allocated on the stack and their visibility is limited to the correspondent method or code block, in which they are initially defined.

3.1. OS support

Typically, software processes are served by their host OS, each of which provides services to the programs that it runs. For example, in a UNIX based system, when the Kernel executes a C program, a special routine, which is known as the startup routine, is automatically invoked to set up the command line arguments and the environments. Then, the *main*() function is invoked. Generally, the OS Kernel manages software processes, each of which is provided a set of RAM pages. When the executable starts, various sections are loaded into these pages; the starts and ends of these sections are independent of the RAM page limits [38].

During execution, different variables are stored in RAM into various logically classified segments such as heap and stack [39]. Program's data that lives in heap can be referenced outside the function scope. On the other hand, stack is a memory segment allocated when the program starts. When the program's execution stack and heap are not managed by the OS Kernel, they are automatically managed by the host Virtual Machine (VM) or the runtime system of the used language.

3.2. JVM support

Java is a modern high level programming language that is supported by its own machine, called the JVM virtual machine. A Java program is loaded and executed by this JVM, which is an abstract computing machine with its own interpreter, instruction set and runtime system that is called Java Runtime Environment (JRE). Simply, this JVM loads class files and executes their bytecode, it automatically manages the required memory areas of a running program internally and away from the OS [40].

In particular, the execution stacks of a running Java program are managed by the host JVM; not by the host OS. Hence, an execution stack consists of blocks called activation records or frames, each of which represents a call to a function and provides a storage for its corresponding local and formal parameters' values or their references. The duration (lifetime) of variables allocated on the stack is same as the scope in which they are declared; mostly the method and its stack frame [41,42]. Additionally, the heap is a logical memory segment that is allocated when the process starts. It provides dynamic memory allocation for variables and their values as needed during execution. In Java, these heap memory allocations and deallocations are automatically managed by the JVM's runtime system and its GC.

Fig. 1. Threat model. A perpetrator uses software to organize an attack that targets a CPS System. The DI job is to find the DE that would prove the software usage and its connection with the attack and the perpetrator.

This JVM differentiates Java from other mainstream OOP languages such as C++ or even C# in its underlying support and memory management, in which Java programs are executed in a layer away from the OS, whereas in C++, the programmer writes a dedicated code and uses system dependent OS APIs to pragmatically manage these dynamically allocated variables and objects. However, in Java an explicit request can be triggered to release the unused memory to the JVM system using special calls to the corresponding GC API. However, the implementation of JVM's execution engine might vary based on the host platform and the used system architecture [43].

This motivates us to investigate Java and find the effects of platform specific JVM on the MF case and its impact on the collected DE that can be identified in RAM to ensure the actual software usage on various platforms. Thus, the Memory Investigators (MI) can utilize the memory of different variables' values to collect DE about the actual software usage.

4. Investigation model

Our investigation model aims at identifying the software that is used during the illegal attack on a CPS system and to confirm its definite usage. Fig. 1 shows a threat model that would be assumed by the DI, in which a software is allegedly used to organize the attack or to hide the invasion by applying any of the anti-forensic techniques [44–46].

The DI goal is to validate whether the subject software has been used during this attack. This validation can be reached by various means, one of which is to search the contents of the RAM dump that is obtained from the captured machine for objects, execution states, or unique variables' values that can be used to evidence the actual usage of the presumed software; see Fig. 2. When the DI captures the perpetrator's machine and in order to preserve the evidence intact for further investigation, a memory dump is created; a bit-by-bit copy for the complete RAM contents is saved to preserve the live DE that resides in RAM. Then, the DI points at the presumed software (or set of software) and assumes its source code. The source code of these software are analyzed and their potential usage scenarios are defined. Then, possible execution paths and their uniquely identified values are classified based on these various possible execution states; see Fig. 3.

This investigation model builds on the base that a variable scope affects its visibility within the program source code and a variable

Please cite this article in press as: Z.A. Al-Sharif, et al., Live forensics of software attacks on cyber–physical systems, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.07.028.

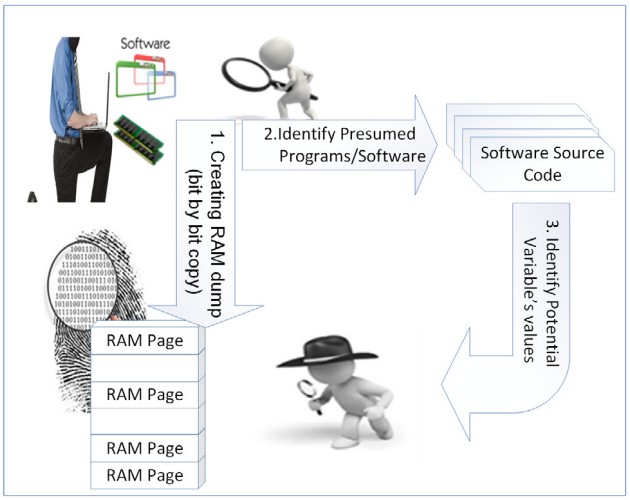


Fig. 2. Investigation model: The perpetrator uses software during the offense. *Step 1*: the investigator creates a memory dump to preserve the system's current state before it is turned off. *Step 2*: the investigator analyzes the source code of the presumed software and identifies potential unique values related to specific execution paths. Finally, *Step 3*: the investigator tries to locate these values within the preserved RAM dumps.

storage affects its duration in RAM. Hence, during various execution states, different scopes and storages affect the survivability of a variable's value in RAM. Accordingly, locating these values in RAM can be used to prove that the perpetrator has actually used the presumed software.

5. Experiments

Our experimentations explore the possibility of locating potential variables' values, which are used within an object oriented program that is written in Java. The fundamental information from the source code and its object oriented principles are used to identify different execution states and potential values in the RAM dump of the used machine. These identified states can be used to evidence the software usage in a legal demonstration.

5.1. Experimentation objectives

One of the objectives of our experimentations is to find the impact of various execution scenarios on different platforms and how these variations may affect the ability to locate the RAM related DE, which can be influenced by:

- Class usage; i.e. the class is being referenced versus when it is never referenced
- Class member visibility; i.e. the use of member access modifiers commonly used in object oriented languages such as public, private, and protected
- Class member variable type; i.e. the class level versus instance level variables such as static versus non-static member variables
- Class member method type; i.e. the use of local variables in a class level method (static method) versus the use of local variables in an instance level method
- Method usage; i.e. the method is never active, currently active, or used to be active
- Base class usage; i.e. the base class object is allocated versus when such an object is never allocated
- Derived class usage; i.e. the derived class object is allocated versus when such an object is never allocated

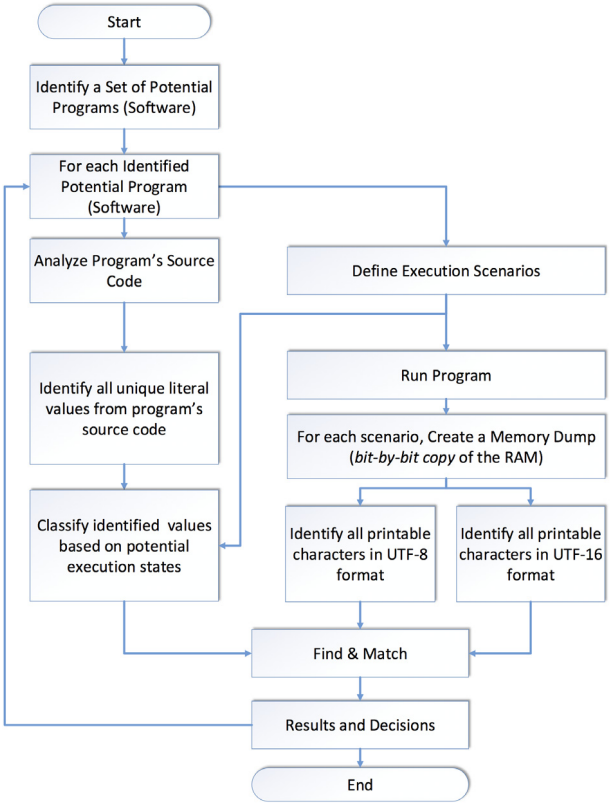


Fig. 3. Investigation Activities, the analysis activities used during our experimentation.

- Base class method usage; i.e. the base class method is used from within a derived object versus when such a method is never used
- Derived class virtual method usage; i.e. the overridden virtual method is used versus when such a method is never used
- Derived class new method usage; i.e. the newly introduced derived class method is used versus when it is never used
- Object usage; i.e. the created object is still referenced and alive versus when it is not referenced anymore and the GC is explicitly invoked
- Program status; i.e. the used program is still running versus when it is terminated or even the JVM is shutdown

The above considerations are used to investigate the duration of various variables' values and their impact by the used execution scenario, each of which are described in the following two subsections.

5.2. Investigated variables

During our validation process, a set of experimental steps are designed with the aim at exposing the evidence related to the execution state based on information and variables' values from the program's source code. A total of 26 different variables' types (V_1 *to* V_{26}) are defined and experiments are established to inspect the durations of their potential values in corresponding RAM dumps that are captured during various execution scenarios. These variables are categorized based on their utilization and usage objectives into *ten* different types, each of which is marked with a letter *from A to J* and described below:

A: Static base class level variables (class level variables)

Please cite this article in press as: Z.A. Al-Sharif, et al., Live forensics of software attacks on cyber–physical systems, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.07.028.

V_1: Public static base class level variables
V_2: Protected static base class level variables
V_3: Private static base class level variables

B: Locals of an active static base class method (class level methods)

V_4: Local variables in a public currently active static base class method
V_5: Local variables in a protected currently active static base class method
V_6: Local variables in a private currently active static base class method

C: Locals of a never been active static base class method (class level methods)

V_7: Local variables in a public but never active static base class method
V_8: Local variables in a protected but never active static base class method
V_9: Local variables in a private but never active static base class method

D: Instance base class level variables (object level variables)

V_{10}: Public instance base class level variables
V_{11}: Protected instance base class level variables
V_{12}: Private instance base class level variables

E: Locals of an active instance base class method (object level methods)

V_{13}: Local variables in a public currently active instance base class method
V_{14}: Local variables in a protected currently active instance base class method
V_{15}: Local variables in a private currently active instance base class method

F: Locals of a never been active instance base class method (object level methods)

V_{16}: Local variables in a public but never active instance base class method
V_{17}: Local variables in a protected but never active instance base class method
V_{18}: Local variables in a private but never active instance base class method

G: Instance derived class level variables (object level variables)

V_{19}: Public instance derived class level variables
V_{20}: Protected instance derived class level variables
V_{21}: private instance derived class level variables

H: Locals of an overridden virtual method (base class methods)

V_{22}: Local variables in an overridden virtual method that is never used
V_{23}: Local variables in an overridden base class virtual method that its override is currently active

I: Locals of an overridden virtual method (derived class methods)

V_{24}: Local variables in an override of a derived class virtual method that is never used
V_{25}: Local variables in an override of a derived class virtual method that is currently active

J: Locals of a newly introduced method (derived class methods)

V_{26}: Local variables in new derived level class method that is currently active

We start our experimentation steps by preparing a Java program and predefine a set of various execution states and paths that can be assumed during the execution of this subject program. These execution states are used to identify the capturing points of the corresponding RAM dumps, each of which are analyzed and the results are compared to investigate the physical memory behavior of this Java program that runs on each of MS Windows, Mac OS X, and Fedora Linux; the 64-bit version of these platforms is used during all experiments. The aim is to forensically compare the results of different runs, each on one of the three platforms. Additionally, we compare the forensic evidence that is collected from the RAM of the used machine and identify these values in various formats, including the *UTF8* and *UTF16* character encoding format of strings' values, see Fig. 3.

5.3. Investigated execution scenarios

Because our investigation targets the Java programming language and the implementation of its JVM and JRE on three different platforms, experiments are designed to explore and compare the potential DE that would be used to evidence the actual software usage during various execution states (scenarios) on any of these platforms. These scenarios are designed to find the effects of different platforms on the MF and its DE that can be collected and identified from the RAM of the machine used during the attack. These scenarios are defined based on the potential usage of object oriented programming features. During our validation process, each of these 21 different scenarios is designed to investigate its impact on the duration of the assumed variable's value in RAM and its association with the variable's type, scope, and state. These scenarios are:

S_1: The program is currently running, but the investigated class is never referenced
S_2: A static variable of the base class is read; used or 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 declared but not allocated yet
S_6: An instance reference of a derived class is declared but not allocated yet
S_7: An instance of the base class is allocated
S_8: An instance variable of the base class is read; used or 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}: An instance of the derived class is allocated and assigned to a variable of the same type; derived class
S_{12}: An instance variable of the derived class is used; used or 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; returned and currently inactive
S_{15}: A second instance of the derived class is allocated but up-casted to a reference of the base class type

Please cite this article in press as: Z.A. Al-Sharif, et al., Live forensics of software attacks on cyber–physical systems, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.07.028.

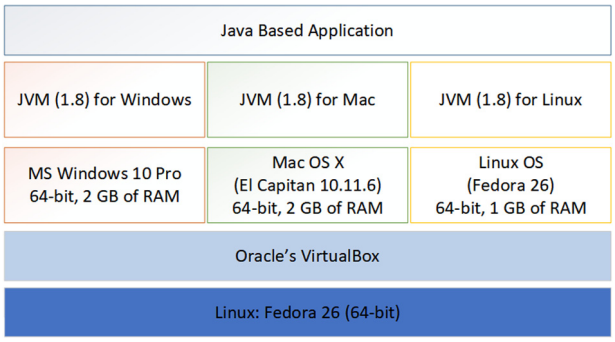


Fig. 4. Experimentation Setup: the VMs are created using Oracle's VirtualBox. Three different platforms are used, each of which has its platform dependent JVM that is downloaded from Oracle.

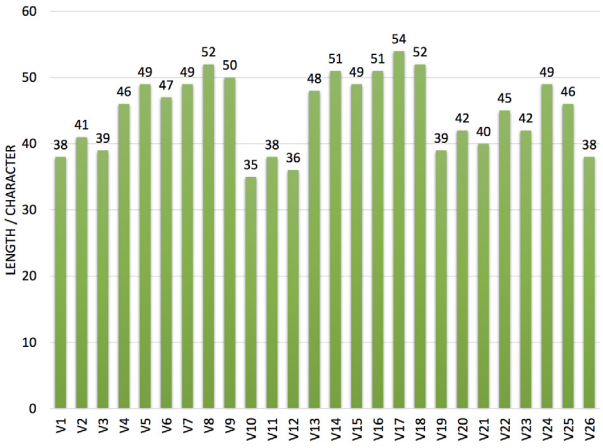


Fig. 5. The used string literals. It shows the length of these 26 (V_1 *to* V_{26}) potential string values that are used during our experimentation.

S_{16}: A virtual method of the derived class is being called; called and currently active
S_{17}: The virtual method of the derived class is returned; returned and currently inactive
S_{18}: The second instance of the derived class is not referenced any more
S_{19}: All references are assigned *null* and the GC is invoked explicitly
S_{20}: All objects are out of scope and not accessible any more
S_{21}: The program is stopped and the JVM is terminated

Additionally, all experiments assume no information is provided by the OS or the used JVM about the investigated process. The RAM dumps are created after each one of the above 21 scenarios (execution states); a dump is created for each state and on each one of the three platforms. The combined total is 63 distinct dumps, each of which is analyzed and searched for potential values related to the same program's source code; see Fig. 3. The results of our findings are presented in Section 6.

5.4. Experimentation setup

The Oracle's VirtualBox[1] is used during all experiments. Fig. 4 shows three different Virtual Machines (VMs) that are prepared and hosted on a Fedora 26 (64-bit) machine. The details of these VMs are:

VM_1: running Windows 10 Pro (64-bit)
VM_2: running Mac OS X 10.11.6 (El Capitan)
VM_3: running Fedora 26 (64-bit)

The RAM size for each of these VMs is kept to the minimum recommended configuration for each platform; based on its hardware specification. In particular, a 2 GB of RAM are allocated for MS Windows 10 Pro and another 2 GB of RAM memory is allocated for Mac OS X 10.11.6. However, only 1 GB of RAM memory is allocated for Fedora 26 (64-bit). Additionally, the same version of JVM (Java 1.8) is used on each of these three platforms. Moreover, the 21 specific points, execution states or scenarios explained above, are identified in advance to create the RAM dumps on each state. Hence, a fresh machine start is ensured before each experiment. Additionally, in the used Java program, different strings and string literal values are used for each of the predefined 26 variables (V_1 *to* V_{26}), Fig. 5 shows the length of these strings, each of which is ensured to be unique in its value in the original program's source code.

Algorithm 1 Find & Match: Evidence Identification
Takes a set of potential values from the assumed program states and tries to locate them within a set of RAM dumps from the execution scenarios.

Input 1: *ValSet*: A set of potential variables' values
Input 2: *DumpSet*: A Set of captured memory dumps
Result: # Occurrences of ($V_j \in ValSet_i$) *In* $Dump_i$
for each $Scenario_i$ **do**
  for each $Dump_i \in Scenario_i$ **do**
    for each $Val_j \in ValSet_i$ **do**
      ▷ *Find all occurrences of* Val_j *in* $Dump_i$
      if Val_j is found in $Dump_j$ **then**
        Add Val_j to *Found Vars*. *In RAM* $Dump_i$;
      end if
    end for
  end for
end for

5.5. Encoding of string values: UTF8 vs. UTF16

In Java, when a program's source code is compiled, it generates a binary format called *bytecode*, which is to be interpreted by the JVM. Different platforms are supported by a compatible version of the JVM. However, the memory dump is created for the complete RAM of the host machine; physical machine, not for the program nor for the JVM. These dumps are exact copy of the physical RAM used by the host machine. Thus, it represents the current system state. Searching these dumps for string values related to the executable Java program can be implemented by different tools, some of which may utilize the *grep* command or simply looking for a substring within the RAM dump as one string. However, the size of these dumps can be reduced using the UNIX based *strings*[2] command, which can be employed to preprocess the captured memory dumps. This command can be used to extract printable strings and save them into a simple text file for further investigation. Table 1 presents the sizes of these preprocessed memory dumps. This *strings* command is mainly used to extract human readable text from the binary format of the RAM dump.[3] By default, this command looks for sequences of at least 4 printable characters that are terminated by a *null* character, other options can be used to collect characters in other encoding formats. For

[1] Oracle VirtualBox, www.virtualbox.org.

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

[3] GNU Binary Utilities, https://sourceware.org/binutils/docs-2.29/binutils/strings.html.

Please cite this article in press as: Z.A. Al-Sharif, et al., Live forensics of software attacks on cyber–physical systems, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.07.028.

Table 1
The size in MB for the striped down RAM dumps during the 21 different execution states for all of Windows, Mac OS X, and Fedora Linux. (i) represents the striped down memory dumps using the UTF8 encoding format, whereas the (ii) represents the size of the striped down memory dumps using the UTF16 encoding format.

Execution	MS Windows 10 Pro.		Mac OS X 10.11.6		Linux: Fedora 26	
Scenarios	(i)	(ii)	(i)	(ii)	(i)	(ii)
S_1	110.85	39.49	255.67	5.77	58.35	0.29
S_2	111.67	39.22	252.18	7.44	58.04	0.31
S_3	109.11	37.35	243.20	10.23	58.08	0.31
S_4	111.22	37.81	241.00	10.29	58.06	0.31
S_5	110.74	39.15	240.94	10.13	58.11	0.31
S_6	111.38	38.87	240.78	10.16	58.32	0.31
S_7	110.38	37.47	240.78	10.24	59.84	0.26
S_8	114.25	36.42	240.43	10.23	60.88	0.25
S_9	118.19	37.07	240.10	10.20	60.86	0.25
S_{10}	120.50	37.33	242.33	10.25	60.87	0.25
S_{11}	128.60	38.15	241.92	10.25	60.88	0.25
S_{12}	127.64	34.52	242.07	10.29	60.97	0.26
S_{13}	127.67	34.46	241.70	10.17	61.27	0.27
S_{14}	126.02	34.54	240.26	10.11	61.53	0.28
S_{15}	125.93	34.45	240.60	10.10	60.74	0.27
S_{16}	125.88	34.42	240.64	10.15	60.89	0.28
S_{17}	125.89	34.42	240.75	10.25	60.90	0.28
S_{18}	125.89	34.40	240.75	10.29	60.90	0.28
S_{19}	125.90	34.43	240.90	10.46	60.86	0.31
S_{20}	125.91	34.53	240.54	10.48	60.86	0.31
S_{21}	125.26	33.34	240.40	10.48	60.63	0.30

example, the "$-el$" option can be used to extract only strings that are encoded using the 16-bit little endian (UTF16 or Unicode).

Thus, we applied this command on each memory dump twice, once with the default settings and the other with the "$-el$" option. Additionally, based on our knowledge of the source code of the used program, we extracted all literal values from the source code of the investigated program and formatted them in both UTF8 and UTF16. Then, both of these encoded literals are located and their number of occurrences are identified in the target memory dump. Algorithm 1 is applied on the set of memory dumps presented in each column of Table 1, in which a cell represents the size of the preprocessed RAM dump that is captured during scenario (S_i) on one of the used platforms. Table 1 shows the sizes of these striped down memory dumps using UTF8 and UTF16 from all three platforms. It is clear that the use of this *strings* command reduced the search space from 2 GB to less than 130 MB and 245 MB on both Windows and Mac OS X, respectively. It also reduced the size of the memory dump from 1 GB to less than 65 MB on Linux machines. Fig. 6 presents a comparison for these striped down memory dumps during each execution scenario (S_i) and for each of the investigated platforms.

Algorithm 1 employs a simple string search algorithm to locate the potential variables' values and their number of occurrences during each scenario on a specific platform. Timewise, this algorithm is used to spend more than one hour to locate the values and their occurrences when it is applied on a set of raw RAM dumps; the unprocessed memory dumps. Whereas, applying the same algorithm on the set of memory dumps that are preprocessed using the *strings* command drastically reduces the search time. Fig. 7 presents a time based comparison that compares between the search time in seconds that is spent during identifying the DE in each of the investigated platforms. The longest time is a bit more than 600 seconds that is needed to analyze the Mac OS X related RAM dumps (UTF8 based), which is the largest set of dumps in size. Additionally, even though all these literal values appeared only once in the original program's source code, surprisingly enough, often their occurrences in the corresponding memory dump are identified on more than one location. In some scenarios, the number of occurrences reached 7 or 8 times; especially when they are located using the UTF8. However, the number of occurrences goes down dramatically when they are located using the UTF16. More results are presented in Section 6.

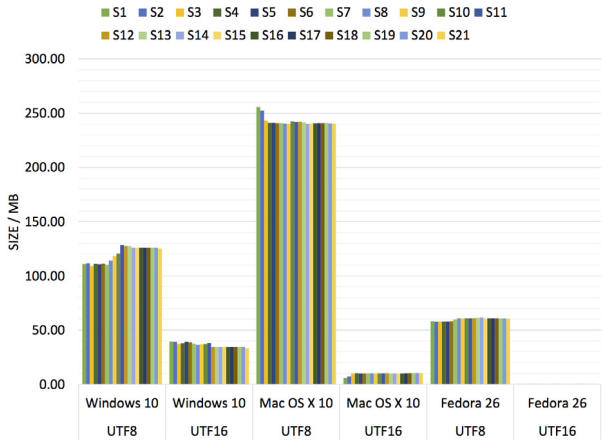


Fig. 6. The size of the striped down memory dumps. It shows the size of each memory dump during each execution scenario (S_i) on each of the three expected platforms.

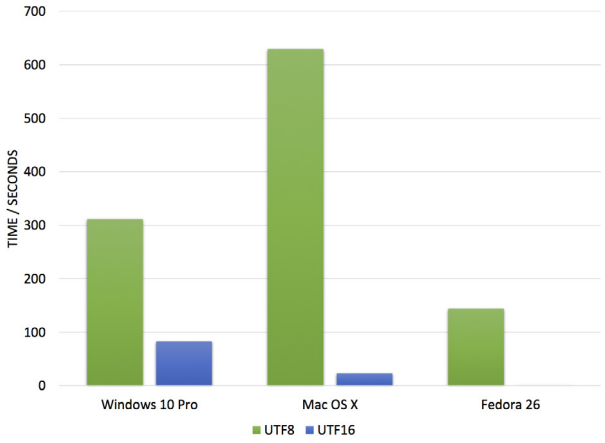


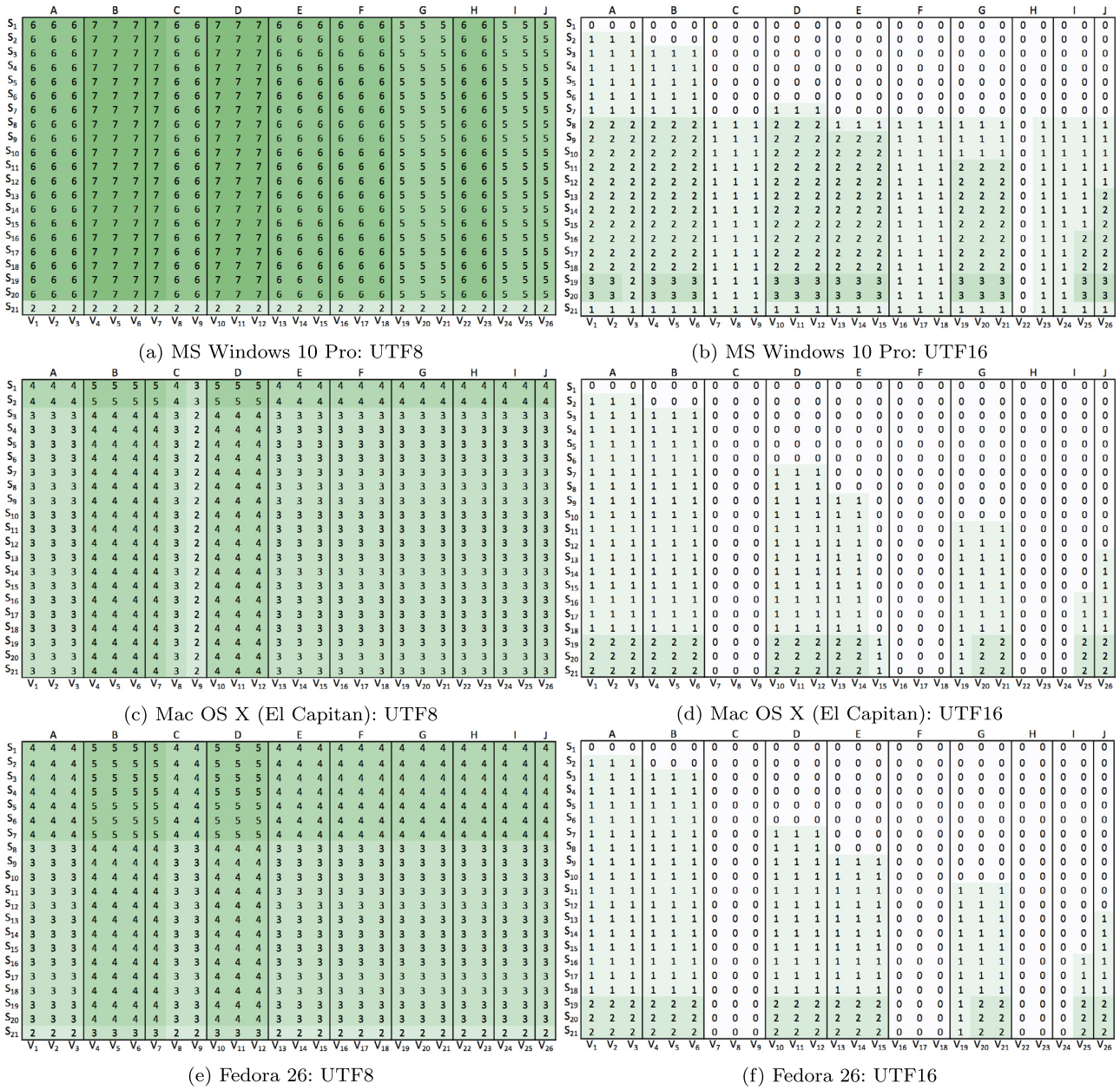
Fig. 7. Experimentation Time. It shows the total time in second that is spent during the searching process for these 26 potential values within the corresponding preprocessed (striped down) memory dumps that was captured during each of the 21 execution scenarios.

6. Results

All captured memory dumps are analyzed and a comparison is made between the findings of Java based source code related values from the program execution states in corresponding RAM dumps that are captured during various scenarios. These findings are compared on each of the investigated platforms; same program and execution scenario but different implementations of the JVM that are hosted on different platforms.

In particular, a comparison is made, from the MF point of view, between the memory behavior of a Java program and how it differs in its memory footprint when it runs on either of Windows, Mac OS X, and Fedora Linux. Fig. 8 presents a combined heatmap for the collected DE of the same program on all platforms. Parts of Fig. 8 show the individual sub-heatmap for the number of occurrences from each experiment, on each platform and for each execution state; including the value of each variable in that state, its internal encoding format, and the number of occurrences of its value in RAM. Heatmap colors go from the darkest green shade into a complete white; the darker shade of green cells indicates an increase

Please cite this article in press as: Z.A. Al-Sharif, et al., Live forensics of software attacks on cyber–physical systems, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.07.028.



(a) MS Windows 10 Pro: UTF8

(b) MS Windows 10 Pro: UTF16

(c) Mac OS X (El Capitan): UTF8

(d) Mac OS X (El Capitan): UTF16

(e) Fedora 26: UTF8

(f) Fedora 26: UTF16

Fig. 8. Experimentation results on all of Windows, Mac, and Fedora Linux; all experiments are performed using the same program, same version of the JVM (64-bit of JVM 1.8), and the 64-bit of the host platform. Parts (a) & (b) compares between UTF8 and UTF16 on Windows 10 Pro. Parts (c) & (d) compares between UTF8 and UTF16 on Mac OS X 10.11.6. Finally, Parts (e) & (f) compares between UTF8 and UTF16 on Fedora 26 (Linux 64-bit). Shaded cells in all figures represent the heatmap for the number of occurrences on all platforms and in both UTF8 and UTF16; the darker the shade the higher the frequency of the found object. It is clear that UTF8 is better than UTF16 and Windows is better than Linux from the MI point of view.

in the frequency of the found values compared to the completely not found values in the white cells. Fig. 8a and b represent the occurrences of these values in the RAM of the Windows machine. Fig. 8c and d represent the occurrences of these values in the RAM of the Mac machine. Finally, Fig. 8e and f represent the occurrences of these values in the RAM of the Fedora 26 Linux machine.

Furthermore, Fig. 8 shows that the number of identified occurrences on the Windows machine is more than in the Mac or Linux machines; taking into consideration that the same program and JVM version is used on all platforms. Additionally, more occurrences are successfully identified using the UTF8 format than it is successfully identified using the UTF16 format; this is true for all platforms as well. Moreover, each part of Fig. 8 are labeled with a letter from *A* to *J*. The sub-columns in each labeled major column are almost identical during the same execution state (S_1 to S_{21}), this is only valid within the same platform. This means, the member access modifiers (public, protected, and private) do not affect the frequency and the availability of the occurrences for these values in RAM.

6.1. Literal values in source code (UTF8)

Java *String* type including string literals are objects not arrays of bytes. Thus, objects are created in heap; *String* instances and source code literals are no exception. In particular, Fig. 8a, c, and e show the heatmaps of our findings of the source code related values (UTF8) in the RAM of the Windows, Mac, and Linux machines, respectively. These heatmaps present the number of occurrences

for each literal value that are assigned to a variable in the source code; 26 unique values are used, each of which is assigned to one variable only (V_1 to V_{26}).

Even though one unique value is used for each variable, unexpectedly, during various execution scenarios or states (S_1 to S_{21}), these figures show that the same variable value, which only occurred once in the source code related variable, often has more occurrences than once in the captured RAM dumps of Windows, Mac, and Linux machines. Furthermore, the number of occurrences on the Windows machine are more than those occurrences on either of Mac or Linux machines. For example, the maximum number of occurrences for variables in column *A* of Fig. 8a of the Windows machine is 6 whereas the maximum number of occurrences from the same column on Mac (Fig. 8c) or Linux (Fig. 8e) is 4. Hence, this difference is almost repeated in all columns between the Windows machine on one side and the Mac and Linux machines on the other side. Furthermore, it is highly noticeable that the results from Mac and Linux machines are almost identical in the number of occurrences for each of these variables (V_1 to V_{26}) and during each of the corresponding states (S_1 to S_{21}).

Additionally, looking at parts of Fig. 8 from the perspective of the MI, it is clear that in all used platforms the investigators have a good chance locating the values of these literals that are obtained from the program source code in the execution states that are captured in corresponding RAM dumps, even after the GC is explicitly invoked, the program is terminated, or even after the host JVM is shutdown. However, the number of occurrence might go down. For example, during our experiments of the Windows machine, the number of occurrences went down from 7 or 6 when the JVM was running to 2 occurrences after the JVM is shutdown. Thus, these occurrences can go down from 5 or 4 to 2 on Mac and Linux. Though, it is still a good chance for the MI; remembering that each of these values occurred only once in the original program source code.

6.2. Literal values during the execution (UTF16)

In Java, the internal representation of the literal strings used during the execution (program run) are based on the UTF16 encoding format. This means that identifying these values in RAM dumps that are captured during the execution of the running program can help the investigator collect more links to evidence the actual software usage by the perpetrator. Thus, the captured RAM dumps are preprocessed using the UNIX based *strings* command with the *−el* option to extract only the UTF16 strings from the binary of the corresponding RAMs. Then, literal values that are extracted from the original program source code are located in these corresponding preprocessed RAM dumps and their number of occurrences are identified. The following three subsections present our results for all investigated variables' values and during all of the execution scenarios; on all of Windows, Mac, and Linux machines.

6.2.1. Class based members

This category represents class level static variables and the locals of static methods. Looking at Fig. 8b, d and f, each of which represents the heatmap with the number of occurrences for the corresponding variables (columns), in each of the associated states (rows), on all of Windows, Mac, and Linux machines; respectively. It is clear that columns labeled with *A* and *B* present better occurrences across all states except S_1 and S_2. Remembering that S_1 represents the *class is not referenced yet* whereas S_2 represents the *class is referenced but the method is not used yet*. This means, the values of static variables and those of local variables of static methods will not be available in RAM until the class is used; referenced during the execution. However, these values will remain in memory long after their usage time. Additionally, the local references in static methods will not be available in RAM until their corresponding method is used. Hence, these values will stay in RAM for a longer duration than the time of their method usage. Looking at column *A* and *B* of Fig. 8b, d, and f, it is clear to see that there are still occurrences for these values in RAM even after the program is terminated (S_{21}) and the JVM is shutdown.

However, local values of static methods, which are not used, is never found in any of the execution states that are captured from either Mac or Linux machines; see column *C* of Fig. 8d and f. On the contrary, on the Windows machine, occurrences for local values of static methods that are never used appeared only once in RAM dumps after state S_8, which represents: *an object is allocated and an instance-based member variable of that object is used (read)*. See column *C* of Fig. 8b. Remembering that these values belong to static methods and these methods are never used, not from the class nor in the object, it is a valuable consideration for the MI on Windows machines.

Finally, the number of occurrences for these sub-columns under the major columns that are labeled with *A*, *B* and *C* are almost identical for the same execution states. This means, once again, the member access modifier (public, protected, and private) does not affect their number of occurrences.

6.2.2. Instance based members (Super Class)

This category represents instance based variables and locals of instance methods. Looking at columns labeled with *D*, *E*, and *F* of Fig. 8b, d and f. These columns present occurrences of the variables' values that are related to the instance (object) that is created from the base class. Fig. 8b presents a heatmap for the occurrences of the identified values in the RAM dump of a Windows machine, whereas Fig. 8d and f present a heatmap for the number of occurrences for the identified values in the RAM of both Mac and Linux machines, respectively. Column *D* shows that occurrences start showing valuable identification in corresponding RAM dumps after state S_7 on all of Windows, Mac, and Linux. Column *D* represents *the class level instance variables* in three access modifiers (V_{10}: public, V_{11}: protected, and V_{12}: private). However, column *E* shows the occurrences of local values in an active instance level method, whereas column *F* shows the occurrences of these literal values of the local variables in a never active method. Column *E* starts showing valuable forensic usage after state S_7 on Windows machines and after state S_9 on both Mac and Linux machines. Column *F* starts showing valuable forensic usage after state S_7 on Windows machines, but its values are never found on both Mac and Linux machines.

Interestingly enough, results from the execution footprint in the RAM of both Mac and Linux are different, to some extent, from the corresponding execution footprint of the Windows machine. In particular, column *F* (variables: V_{16}, V_{17}, and V_{18}) that represent *the values of local variables in methods that are never active during the program execution*, shows that it is unlikely to find these values in the RAM dumps of the host machine when it is running on Mac or Linux machines, but it can be found on the corresponding Windows machine during the same states. Additionally, on both Mac and Linux machines, the number of found occurrences in the lower parts of columns *D* and *E* (S_{19}, S_{20}, &S_{21}) are better than those corresponding occurrences in states before them.

6.2.3. Instance based members (Derived Class)

This category includes instance based members of a derived class including local variables of overridden virtual methods and newly introduced methods. Looking at columns labeled with *G*, *H*, *I*, *and J* (V_{19} *to* V_{26}) of Fig. 8b. It shows that, on Windows, the investigator has a good chance locating variables' values from instances that are created from the derived class right after state S_8, which represents *an instance variable of the base class is used*.

Please cite this article in press as: Z.A. Al-Sharif, et al., Live forensics of software attacks on cyber–physical systems, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.07.028.

However, this set of variables belongs only to the derived class or to the newly introduced method in the derived class. Hence, on all three platforms, the only variable that is unlikely to be found is V_{22}, which represents *a local variable in an overridden virtual method that is never used*. This is expected because the function is never used during any of the execution states (S_1 to S_{21}); considering the dynamic binding.

On the contrary, the same Java program behaves differently on both Mac and Linux machines. In particular, variables V_{19} to V_{21}, which are derived class level variables (public, protected, and private, respectively) are more likely to be found on UNIX based machines after state S_{11}, see column *G* of Fig. 8d and f. Remember that S_{11} represents *an instance of the derived class is allocated*. Surprisingly enough, Mac and Linux machines behave almost the same whereas both of them behave completely different from the Windows machine. For example, the values of V_{23} and V_{24} are successfully found after state S_8 on the Windows machine whereas these same variables' values are unlikely to be found during any of the investigated execution states (S_1 to S_{21}) on Mac nor Linux. These results are highly expected on Mac and Linux, because the values of V_{23} represent a variable in *an overridden base class virtual method that its override is currently active, but not the actual function that has the value*. Additionally, the value of V_{24} represents the value of *a local variable in a derived class virtual method that is never used*. However, unexpectedly, these same values are found in the RAM of the Windows machine. To find the literal value of these variables on Windows, it is something strange to the behavior of the JVM on of a Windows machine after state S_8. Additionally, V_{25} and V_{26} are found on Windows after state S_8 too, but they are unlikely to be found on a Mac or Linux based machines before states S_{16} and S_{13}, respectively.

7. Related work

Many researchers are making use of existing forensic research in related areas such as cloud forensics, mobile forensics, virtualization forensics, and storage forensics in order to facilitate the forensics of CPS Systems [22,23,47]. Others, are focused on the forensics of a particular type of CPS systems. For example, Hahn et al. evaluated a smart grid security testbed, in which several potential attack scenarios are assumed to explore the security of the CPS system [48]. Zonouz et al. introduced a Security-oriented Cyber–physical State Estimation (SCPSE) in order to detect the malicious activities within the CPS system [49]. Grispos et al. integrates forensics principles, concepts, design and development of the Medical Cyber–physical Systems (MCPS) in one framework in order to strengthen the investigation capabilities [50]. Eli Sohl et al. studied various forensics tools and techniques that focus on the fundamentals of digital forensics in the electrical power grid. They found that the security of industrial control systems in the power grid appears to be much less mature than that of the general-purpose computing [51]. Boyer et al. shows that Supervisory Control And Data Acquisition (SCADA) system is one of the common monitoring and control systems that can be used in the power grid [52], it is also used in other industrial infrastructures such as oil refineries, water treatment facilities, and transportation systems. Parry et al. presented a high-speed network forensics tool specifically designed for capturing and replaying data traffic dedicated toward special-purpose network interface card that can be found in CPS systems [53]. Taylor et al. reviewed some of the challenges and highlighted the security risks that are needed to be covered to increase the CPS security [54]. Giraldo et al. provided a taxonomy that classifies all of CPS domains, attacks, defenses, research-trends, network-security, security level implementation, and computational strategies [55].

On the other hand, 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 forensics cases [56–65]. For example, 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 [66]. Ellick et al. introduced ForenScope a RAM forensics 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 [67]. Olajide et al. uses RAM dumps to extract user's input information from Windows applications [68]. 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 [69]. Shashidhar et al. targeted the *prefetch* folder and its potential value to the investigator. This prefetch folder is used to speed up the startup time of a program on a Windows Machine [70].

However, this paper utilizes the OOP concepts and its potential source code usage in Java in order to identify and validate whether the presumed software is actually used during the attack on a CPS system.

8. Discussions and future work

In this paper, in order to facilitate the ability to find live evidence about the actual software usage that are used to organize an attack on a CPS system, experimentally we established the approach of utilizing potential variables' values from the program's source code and their associations with various in RAM program execution states to evidence the actual software usage. Based on the possible data-flow and control-flow of the subject program, we investigate an object oriented program that is written in Java and experiment with its MF on three different platforms; including Windows 10 Pro, Mac OS X, and Fedora 26 (Linux 64-bit). Hence, the same program's source code is used during all experiments, the only difference is the underneath JVM that encapsulates these differences to accommodate the host platform. Experimental results show that these different implementations affect the RAM memory footprint of the used program from the MF point of view.

Nonetheless, theoretically the used programming language (Java) operates in a layer that is distant from the OS Kernel. Practically, this means that a specific memory area that is used to handle program's execution and its runtime storage (i.e. stack and heap) are kept away from the OS. The GC picks up the unused memory (garbage), but often this memory will not become immediately available to the OS. Therefore, in RAM blind program's execution traces are expected to behave differently on different platforms for other cross-platform languages as well. Additionally, Java is a strictly typed language, however other languages support dynamic typing such as Visual Basic and the scripting languages such as Python, Icon/Unicon, Ruby and others. Thus, for future research, it would be highly interesting to compare our experimental results from this paper with the results from similar experimentations that are based on these dynamically typed languages.

Additionally, our experimentations are conducted to practically determine the digital forensics process and the technical considerations in regards to the investigations of the allegedly used software during the attack on a CPS system. Locating the presumed software on the perpetrator's machine hard disk might not be a strong digital evidence to establish the actual software usage. A definite legal evidence might be needed to prove that the perpetrator has actually used the software. This paper experimentally established that this evidence can be found in the RAM of the perpetrator's machine. However, additional research is needed to address the scalability of these fundamental concepts. In particular,

Please cite this article in press as: Z.A. Al-Sharif, et al., Live forensics of software attacks on cyber–physical systems, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.07.028.

during our experimentations, we covered only 26 different types of variables that would cover the principles of the Java based OOP variables and their usage within 21 basic execution scenarios (states). However, in bigger programs, this number of variables' values and potential execution states can easily become much bigger than our basic proof of concept used in this paper. Thus, an issue with the performance can be easily raised in regards to various directions, each of which may open the opportunity to new research directions. Some of these directions can be summarized at least in threefold:

First, in a large software, the potential variables' values will reach a limit that would require improving the searching algorithm by various means such as the employment of parallel algorithms or the utilization of GPUs. Additionally, RAMs used in our daily used computers are getting larger and larger. This implies an issue with the performance of the evidence identification algorithm is on the horizon [47].

Second, the increase in the number of potential variable states and their values will enlarge the number of values that are needed to be identified. Additionally, the increase in size of the RAM dumps and the number of these dumps can grow to become a big data problem, in which a huge number of potential variables' values, execution states, and memory dumps are needed to be investigated. Thus, in such a scenario, our approach can benefit from the big data analytics and machine learning techniques to automatically identify and build a legal decision about the actual software usage.

Third, utilizing various software test cases, whether it is unit testing or system testing, in the process of identifying the potential variables' values and their execution states would be very interesting and highly challenging to the memory investigator. However, a solution can be envisioned by utilizing various machine learning algorithms in order to automate the digital investigation process. For examples, a research is needed to investigate the possibility of training various machine learning algorithms based on these potential test sets (test suite) in order to automatically predict or identify potential execution states that can be assumed to be used by the perpetrators during the attack. This would advance and speed up the evidence identification process.

Furthermore, our investigation process targets simple variables' values, it would be interesting to investigate similar scenarios for other complex data structures such as collections and generics. Additionally, it is highly important to investigate various variables' types and their impacts on long running programs such as servers and web services. Moreover, nowadays many mobile applications are used to interconnect, communicate, and control various CPS systems [71]. Thus, investigating these environments of small devices such as phones and tablets can be one of the future research targets.

9. Conclusion

Increasingly, our modernized aspects of life is depending on various Cyber–physical Systems (CPS). In the same time, the speed of securing these systems does not match the speed of our adaption of these CPS, which are vulnerable to be attacked and compromised by various means. This emphasizes the need to invest more in related research. The focus of this paper is the Memory Forensics and the post-mortem analysis that can be employed to investigate these attacks and help apprehend the criminals. It utilizes information from the source code of a program and employs program's execution data during various execution states to help the 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 practical experimentation is based on Java, which is one of the most widely used programming languages. Memory footprints from similar programs that are written in Java are compared on different platforms; including MS Windows 10 Pro, Mac OS X, and Fedora 26. However, experimentally, we found that when it comes to memory forensics, data persistence in memory is much longer than it is during the execution of the program that it takes different execution paths based on its control-flow and data-flow. Additionally, we found that utilizing source code information can be valuable to the investigator. It helps establish the evidence that the perpetrator has 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 identified in RAM dups that are captured during various scenarios and execution states. For example, we found that variable values of static and non-static values of a Java program running on a Windows is better than the other two platforms (Linux and Mac). Additionally, values of instance variables are likely to be found on Windows more than on machines that are running Mac or Linux.

References

[1] A. Ahmad, A. Paul, M.M. Rathore, H. Chang, Smart cyber society: Integration of capillary devices with high usability based on cyber–physical system, Future Gener. Comput. Syst. 56 (2016) 493–503.
[2] A.A. Cardenas, S. Amin, S. Sastry, Secure control: Towards survivable cyber-physical systems, in: 2008 The 28th International Conference on Distributed Computing Systems Workshops, June 2008, pp. 495–500.
[3] E.A. Lee, Cyber physical systems: Design challenges, in: 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), May 2008, pp. 363–369.
[4] H. Ning, H. Liu, J. Ma, L.T. Yang, R. Huang, Cybermatics: Cyber–physical–social–thinking hyperspace based science and technology, Future Gener. Comput. Syst. 56 (2016) 504–522.
[5] B. Schatz, M. Cohen, Advances in volatile memory forensics, Digit. Investig. 20 (Supplement C) (2017) 1. Special Issue on Volatile Memory Analysis.
[6] A. Case, G.G. Richard, Memory forensics: The path forward, Digit. Investig. 20 (Supplement C) (2017) 23–33. Special Issue on Volatile Memory Analysis.
[7] C.W. Tien, J.W. Liao, S.C. Chang, S.Y. Kuo, Memory forensics using virtual machine introspection for malware analysis, in: 2017 IEEE Conference on Dependable and Secure Computing, Aug 2017, pp. 518–519.
[8] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, 2016.
[9] A. Pridgen, S. Garfinkel, D.S. Wallach, Picking up the trash: Exploiting generational gc for memory analysis, Digit. Investig. 20 (Supplement) (2017) S20–S28. DFRWS 2017 Europe.
[10] N.H.A. Rahman, W.B. Glisson, Y. Yang, K.K.R. Choo, Forensic-by-design framework for cyber-physical cloud systems, IEEE Cloud Comput. 3 (2016) 50–59.
[11] Y. Shoukry, M. Chong, M. Wakaiki, P. Nuzzo, A. Sangiovanni-Vincentelli, S.A. Seshia, J.P. Hespanha, P. Tabuada, Smt-based observer design for cyber-physical systems under sensor attacks, ACM Trans. Cyber-Phys. Syst. 2 (2018) 5:1–5:27.
[12] F. Hu, Y. Lu, A.V. Vasilakos, Q. Hao, R. Ma, Y. Patil, T. Zhang, J. Lu, X. Li, N.N. Xiong, Robust cyber–physical systems: Concept, models, and implementation, Future Gener. Comput. Syst. 56 (2016) 449–475.
[13] V.R. Kebande, I. Ray, A generic digital forensic investigation framework for internet of things (iot), in: 2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud), Aug 2016, pp. 356–362.
[14] A. Jones, S. Vidalis, N. Abouzakhar, Information security and digital forensics in the world of cyber physical systems, in: 2016 Eleventh International Conference on Digital Information Management (ICDIM), Sept 2016, pp. 10–14.
[15] V. Hahanov, E. Litvinova, S. Chumachenko, A. Hahanova, Cyber physical computing, in: Cyber Physical Computing for IoT-Driven Services, Springer, 2018, pp. 1–20.
[16] J. Wu, S. Guo, H. Huang, W. Liu, Y. Xiang, Information and communications technologies for sustainable development goals: State-of-the-art, needs and perspectives, IEEE Commun. Surv. Tutor. PP (99) (2018) 1–1.
[17] K. Hamedani, L. Liu, R. Atat, J. Wu, Y. Yi, Reservoir computing meets smart grids: Attack detection using delayed feedback networks, IEEE Trans. Ind. Inf. 14 (2018) 734–743.
[18] J. Wu, S. Guo, J. Li, D. Zeng, Big data meet green challenges: Big data toward green applications, IEEE Syst. J. 10 (2016) 888–900.
[19] J. Wu, S. Guo, J. Li, D. Zeng, Big data meet green challenges: Greening big data, IEEE Syst. J. 10 (2016) 873–887.

Please cite this article in press as: Z.A. Al-Sharif, et al., Live forensics of software attacks on cyber–physical systems, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.07.028.

[20] R. Atat, L. Liu, H. Chen, J. Wu, H. Li, Y. Yi, Enabling cyber-physical communication in 5g cellular networks: challenges, spatial spectrum sensing, and cybersecurity, IET Cyber-Phys. Syst.: Theory Appl. 2 (1) (2017) 49–54.
[21] J. An, K. Yang, J. Wu, N. Ye, S. Guo, Z. Liao, Achieving sustainable ultra-dense heterogeneous networks for 5g, IEEE Commun. Mag. 55 (2017) 84–90.
[22] R. Roman, J. Lopez, M. Mambo, Mobile edge computing fog et al.: A survey and analysis of security threats and challenges, Future Gener. Comput. Syst. 78 (2018) 680–698.
[23] Y. Liu, A. Liu, S. Guo, Z. Li, Y.-J. Choi, H. Sekiya, Context-aware collect data with energy efficient in cyber–physical cloud systems, Future Gener. Comput. Syst. (2017).
[24] C. Huang, R. Lu, K.K.R. Choo, Vehicular fog computing: Architecture, use case, and security and forensic challenges, IEEE Commun. Mag. 55 (2017) 105–111.
[25] D. Ding, Q.-L. Han, Y. Xiang, X. Ge, X.-M. Zhang, A survey on security control and attack detection for industrial cyber-physical systems, Neurocomputing 275 (2018) 1674–1683.
[26] M. Erol-Kantarci, H.T. Mouftah, Smart grid forensic science: applications, challenges, and open issues, IEEE Commun. Mag. 51 (1) (2013) 68–74.
[27] Y. Mo, T.H.J. Kim, K. Brancik, D. Dickinson, H. Lee, A. Perrig, B. Sinopoli, Cyber-physical security of a smart grid infrastructure, Proc. IEEE 100 (2012) 195–209.
[28] M. Wolf, D. Serpanos, Safety and security in cyber-physical systems and internet-of-things systems, Proc. IEEE 106 (2018) 9–20.
[29] X. Chang, Z. Ma, M. Lin, Y. Yang, A.G. Hauptmann, Feature interaction augmented sparse learning for fast kinect motion detection, IEEE Trans. Image Process. 26 (2017) 3911–3920.
[30] X. Chang, Z. Ma, Y. Yang, Z. Zeng, A.G. Hauptmann, Bi-level semantic representation analysis for multimedia event detection, IEEE Trans. Cybern. 47 (2017) 1180–1197.
[31] X. Chang, Y. Yang, Semisupervised feature analysis by mining correlations among multiple tasks, IEEE Trans. Neural Netw. Learn. Syst. 28 (2017) 2294–2305.
[32] X. Chang, Y.L. Yu, Y. Yang, E.P. Xing, Semantic pooling for complex event analysis in untrimmed videos, IEEE Trans. Pattern Anal. Mach. Intell. 39 (2017) 1617–1632.
[33] Z. Li, F. Nie, X. Chang, Y. Yang, Beyond trace ratio: Weighted harmonic mean of trace ratios for multiclass discriminant analysis, IEEE Trans. Knowl. Data Eng. 29 (2017) 2100–2110.
[34] K.K.R. Choo, M.M. Kermani, R. Azarderakhsh, M. Govindarasu, Emerging embedded and cyber physical system security challenges and innovations, IEEE Trans. Dependable Secure Comput. 14 (2017) 235–236.
[35] Y. Li, L. Zhang, H. Zheng, X. He, S. Peeta, T. Zheng, Y. Li, Nonlane-discipline-based car-following model for electric vehicles in transportation- cyber-physical systems, IEEE Trans. Intell. Transp. Syst. 19 (2018) 38–47.
[36] R.W. Atherton, Moving java to the factory, IEEE Spectr. 35 (1998) 18–23.
[37] J.F. Baquero, J.E. Camargo, F. Restrepo-Calle, J.H. Aponte, F.A. González, Predicting the Programming Language: Extracting Knowledge from Stack Overflow Posts, Springer International Publishing, Cham, 2017, pp. 199–210.
[38] W.R. Stevens, S.A. Rago, Advanced Programming in the UNIX Environment, Addison-Wesley, 2013.
[39] D.P. Bovet, M. Cesati, Understanding the Linux Kernel: From I/O Ports to Process Management, O'Reilly Media, Inc., 2005.
[40] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, The java virtual machine specification-java se 8 edition, feb. 2015.
[41] S. Nadi, R. Holt, The linux kernel: A case study of build system variability, J. Softw.: Evol. Process 26 (8) (2014) 730–746.
[42] A. Josey, D. Cragun, N. Stoughton, M. Brown, C. Hughes, et al., The open group base specifications issue 6 ieee std 1003.1, IEEE Open Group 20 (60) (2004).
[43] H. Schildt, Java: The Complete Reference, McGraw-Hill Education Group, 2014.
[44] K. Conlan, I. Baggili, F. Breitinger, Anti-forensics: Furthering digital forensic science through a new extended, granular taxonomy, Digit. Investig. 18 (Supplement) (2016) S66–S75.
[45] S. Kälber, A. Dewald, F.C. Freiling, Forensic application-fingerprinting based on file system metadata, in: 2013 Seventh International Conference on IT Security Incident Management and IT Forensics, March 2013, pp. 98–112.
[46] D. DiMase, Z.A. Collier, K. Heffner, I. Linkov, Systems engineering framework for cyber physical security and resilience, Environ. Syst. Decis. 35 (2015) 291–300.
[47] L. Caviglione, S. Wendzel, W. Mazurczyk, The future of digital forensics: Challenges and the road ahead, IEEE Secur. Privacy 15 (2017) 12–17.
[48] A. Hahn, A. Ashok, S. Sridhar, M. Govindarasu, Cyber-physical security testbeds: Architecture, application, and evaluation for smart grid, IEEE Trans. Smart Grid 4 (2013) 847–855.
[49] S. Zonouz, K.M. Rogers, R. Berthier, R.B. Bobba, W.H. Sanders, T.J. Overbye, Scpse: Security-oriented cyber-physical state estimation for power grid critical infrastructures, IEEE Trans. Smart Grid 3 (2012) 1790–1799.
[50] G. Grispos, W.B. Glisson, K.K.R. Choo, Medical cyber-physical systems development: A forensics-driven approach, in: 2017 IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE), July 2017, pp. 108–113.
[51] E. Sohl, C. Fielding, T. Hanlon, J. Rrushi, H. Farhangi, C. Howey, K. Carmichael, J. Dabell, A field study of digital forensics of intrusions in the electrical power grid, in: Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy, CPS-SPC '15, ACM, New York, NY, USA, 2015, pp. 113–122.
[52] S.A. Boyer, SCADA: Supervisory Control and Data Acquisition, International Society of Automation, 2009.
[53] J. Parry, D. Hunter, K. Radke, C. Fidge, A network forensics tool for precise data packet capture and replay in cyber-physical systems, in: Proceedings of the Australasian Computer Science Week Multiconference, ACSW '16, ACM, New York, NY, USA, 2016, pp. 22:1–22:10.
[54] J.M. Taylor, H.R. Sharif, Security challenges and methods for protecting critical infrastructure cyber-physical systems, in: 2017 International Conference on Selected Topics in Mobile and Wireless Networking (MoWNeT), May 2017, pp. 1–6.
[55] J. Giraldo, E. Sarkar, A.A. Cardenas, M. Maniatakos, M. Kantarcioglu, Security and privacy in cyber-physical systems: A survey of surveys, IEEE Des. Test 34 (2017) 7–17.
[56] M. Rafique, M. Khan, Exploring static and live digital forensics: Methods, practices and tools, Int. J. Sci. Eng. Res. 4 (10) (2013) 1048–1056.
[57] F.N. Dezfoli, A. Dehghantanha, R. Mahmoud, N.F.B.M. Sani, F. Daryabar, Digital forensic trends and future, Int. J. Cyber-Secur. Digital Forensics (IJCSDF) 2 (2) (2013) 48–76.
[58] L. Cai, J. Sha, W. Qian, Study on forensic analysis of physical memory, in: The proceedings of 2nd International Symposium on Computer, Communication, Control and Automation (3CA 2013), 2013, pp. 221–224.
[59] Z.A. Al-Sharif, Utilizing program's execution data for digital forensics, in: The Third International Conference on Digital Security and Forensics (DigitalSec2016), 2016, p. 12.
[60] M.I. Al-Saleh, Z.A. Al-Sharif, Utilizing data lifetime of tcp buffers in digital forensics: Empirical study, Digit. Investig. 9 (2) (2012) 119–124.
[61] Z.A. Al-Sharif, D.N. Odeh, M.I. Al-Saleh, Towards carving pdf files in the main memory, in: The International Technology Management Conference, (ITMC2015), The Society of Digital Information and Wireless Communication (SDIWC), 2015, pp. 24–31.
[62] M. Al-Saleh, Z. Al-Sharif, Ram forensics against cyber crimes involving files, in: The Second International Conference on Cyber Security, Cyber Peacefare and Digital Forensic, (CyberSec2013), The Society of Digital Information and Wireless Communication (SDIWC), 2013, pp. 189–197.
[63] Z.A. Al-Sharif, H. Bagci, T.A. Zaitoun, A. Asad, Towards the Memory Forensics of MS Word Documents, Springer International Publishing, Cham, 2018, pp. 179–185.
[64] M.I. Al-Saleh, F.M. AbuHjeela, Z.A. Al-Sharif, Investigating the detection capabilities of antiviruses under concurrent attacks, Int. J. Inf. Secur. 14 (4) (2015) 387–396.
[65] V.S. Harichandran, D. Walnycky, I. Baggili, F. Breitinger, Cufa: A more formal definition for digital forensic artifacts, Digit. Investig. 18 (2016) S125–S137.
[66] A.F. Shosha, L. Tobin, P. Gladyshev, Digital forensic reconstruction of a program action, in: Security and Privacy Workshops (SPW), 2013 IEEE, IEEE, 2013, pp. 119–122.
[67] E. Chan, W. Wan, A. Chaugule, R. Campbell, A framework for volatile memory forensics, in: Proceedings of the16th ACM Conference on Computer and Communications Security, 2009.
[68] F. Olajide, N. Savage, G. Akmayeva, C. Shoniregun, Identifying and finding forensic evidence on windows application, J. Internet Technol. Secured Trans. ISSN (2012) 2046–3723.
[69] J. Stüttgen, S. Vömel, M. Denzel, Acquisition and analysis of compromised firmware using memory forensics, Digit. Investig. 12 (2015) S50–S60.
[70] N.K. Shashidhar, D. Novak, Digital forensic analysis on prefetch files, Int. J. Inf. Secur. Sci. 4 (2) (2015) 39–49.
[71] X. Hu, T.H.S. Chu, H.C.B. Chan, V.C.M. Leung, Vita: A crowdsensing-oriented mobile cyber-physical system, IEEE Trans. Emerging Top. Comput. 1 (2013) 148–165.

Ziad A. Al-Sharif is currently an assistant professor at Jordan University of Science and Technology, Irbid, Jordan. He joined the Department of Software Engineering in February of 2010. Dr. Al-Sharif received his Ph.D. degree in Computer Science in December of 2009 from the University of Idaho, USA. He also received his MS. degree in Computer Science in August of 2005 from New Mexico State University, USA. His research interests are in digital forensics, software engineering, cloud computing, and collaborative virtual environments.

Please cite this article in press as: Z.A. Al-Sharif, et al., Live forensics of software attacks on cyber–physical systems, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.07.028.

Mohammed I. Al-Saleh is an associate professor in the Computer Science Department at Jordan University of Science and Technology. He earned his Ph.D. in 2011 from The University of New Mexico (UNM) and his MSc degree from New Mexico State University (NMSU) in 2007. His research interests include digital forensics, cyber security, antivirus testing and evaluation, and dynamic information flow tracking.

Luay Alawneh is an assistant professor in the Department of Software Engineering at Jordan University of Science and Technology, Irbid, Jordan. His research interests are software engineering, software maintenance and evolution, big data analytics, parallel processing and high performance computing systems. Luay received a Ph.D. in electrical and computer engineering from Concordia University. In addition to his research achievements, Luay possesses excellent industrial experience gained from prestigious North American firms.

Yaser Jararweh received his Ph.D. in Computer Engineering from University of Arizona in 2010. He is currently an associate professor of Computer Science at Jordan University of Science and Technology, Jordan. He has co-authored about seventy technical papers in established journals and conferences in fields related to cloud computing, HPC, SDN and Big Data. He was one of the TPC Co-Chair, IEEE Globecom 2013 International Workshop on Cloud Computing Systems, and Networks, and Applications (CCSNA). He is a steering committee member for CCSNA 2014 and CCSNA 2015 with ICC. He is the General Co-Chair in IEEE International Workshop on Software Defined Systems SDS-2014 and SDS 2015. He is also chairing many IEEE events such as ICICS, SNAMS, BDSN, IoTSMS and many others. Dr. Jararweh served as a guest editor for many special issues in different established journals. Also, he is the steering committee chair of the IBM Cloud Academy Conference.

B.B. Gupta received his Ph.D. degree from Indian Institute of Technology Roorkee, India in the area of Information and Cyber Security. He spent more than six months in University of Saskatchewan (UofS), Canada to complete a portion of his research work. Dr. Gupta has excellent academic record throughout his carrier, was among the college toppers, during Bachelor's degree and awarded merit scholarship for his excellent performance. In addition, he was also awarded Fellowship from Ministry of Human Resource Development (MHRD), Government of India to carry his Doctoral research work. He has published more than 70 research papers (including 01 book and 08 chapters) in International Journals and Conferences of high repute including IEEE, Elsevier, ACM, Springer, Wiley Inderscience, etc. He has visited several countries, i.e. Canada, Japan, Malaysia, Hong-Kong, etc to present his research work. His biography was selected and publishes in the 30th Edition of Marquis Who is Who in the World, 2012. He is also working principal investigator of various R&D projects. He is also serving as reviewer for Journals of IEEE, Springer, Wiley, Taylor & Francis, etc. Currently he is guiding 08 students for their Master's and Doctoral research work in the area of Information and Cyber Security. He also served as Organizing Chair of Special Session on Recent Advancements in Cyber Security (SS-CBS) in IEEE Global Conference on Consumer Electronics (GCCE), Japan in 2014 and 2015.

Please cite this article in press as: Z.A. Al-Sharif, et al., Live forensics of software attacks on cyber–physical systems, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.07.028.