*Journal of Universal Computer Science, vol. 26, no. 7 (2020), 805-826*
*submitted: 23/11/19, accepted: 12/4/20, appeared: 28/7/20* *CC BY-ND 4.0*

# Utilizing Debugging Information of Applications in Memory Forensics

**Mohammed I. Al-Saleh, Ethar Qawasmeh**
(Jordan University of Science and Technology
Department of Computer Science
P.O. Box 3030
Irbid, Jordan 22110
misaleh@just.edu.jo, eaqawasmeh16@cit.just.edu.jo)

**Ziad A. Al-Sharif**
(Jordan University of Science and Technology
Department of Software Engineering
P.O. Box 3030
Irbid, Jordan 22110
zasharif@just.edu.jo)

**Abstract:** The rapid development in the digital world has contributed to the dramatic increase in the number of cybercrimes. An application's volatile data that is kept in memory (RAM) could give clues on how a criminal has been using the application up to acquisition time. Unfortunately, application-level memory forensics has been conducted in an ad hoc manner because a forensic investigator has to come up with a new technique for a new application. This process has become problematic and exhausting. This paper proposes a general solution to investigate any application in memory. We heavily utilize applications' debugging information generated by compilers in our solution. Furthermore, we extend Volatility[Walters, 2007], an open-source memory forensic framework, by developing and integrating a plugin to investigate applications in memory. We design several experiments to evaluate the effectiveness of our plugin. Interestingly, our plugin can parse debugging information and extract variables' names and data types regardless of their scope and complexity. In addition, we experimented with a real world application and succeeded in collecting vital information out of it. By accurately computing the Virtual Addresses (VA) of variables along with their allocated memory sizes based on their types, we are able to extract their values out of memory. In addition, we trace call stacks as per threads to extract local variables' values. Finally, direct and indirect pointers are successfully dereferenced.
**Key Words:** Debugging information, Memory forensics, Application forensics
**Category:** D.4.1, D.4.6, D.4.9

## 1 Introduction

Recent technologies have led to a significant increase in the number of cybercrimes. Intruding into others' machines to steal their valuable information, executing malicious programs, spying on users' activities or causing damage to systems are examples of cybercrimes. Digital forensics is inevitable in investigating cybercrimes and resolving cases. It follows standard methodologies to

maintain the soundness of the whole investigation process. Digital artifacts can be utilized in convicting cyber criminals and exposing their activities. Various digital storage media can be inspected to extract digital evidence such as Hard Drives (HD), Solid State Drives (SSD), memory (RAM), and networks devices hd1,hdssd,IMpc,ssd1, memory1, memory2, memory3,memory4,net1,net2, net3.

Memory Forensics (MF) is vital in the investigation process for the wealth of information it comprises lit14,lit30. For example, it holds information about open files, processes, browsing history, passwords, encryption keys, and network connections. Moreover, MF plays an essential role in malware analysis and reverse engineering. Finally, there are some cases where information can only be found in memory lit35.

Various applications might be involved in criminal activities. Researchers have developed techniques to investigate well-known applications such as web browsers and VoIP applications lit33,lit31. Despite being useful, these techniques are ad hoc in the sense that the corresponding forensic technique only targets a specific version of an application by collecting strings and network data without a priori knowledge of the application's internal structures. When a new version of an application is developed or a new application has come out then the old technique might be rendered useless. This paper, however, aims to provide a unified solution to investigate any application in memory. Our framework utilizes information about the applications' structures and variables. In particular, compilers generate various kinds of information about applications. For example, a compiler generates what is called debugging information to help debuggers trace running applications. Debugging information includes defined data structures and variables along with their virtual memory addresses.

The ability to investigate any application in memory helps refine the investigation process and reach a better understanding of an application's usage. Our ultimate goal is to make the whole memory investigation process more efficient and effective so that cases can be resolved easily and quickly.

The rest of this paper is organized as follows: Section 2 presents related work. Section 3 provides a background about Volatility framework, Portable Executable file (PE) and Program Database (PDB) file. Our methodology is explained in Section 4. The experimentation and results are presented in Section 5. Section 6 discusses limitations and future work, followed by the conclusion.

## 2 Related work

Memory keeps valuable information that can be utilized for forensics purposes [Hausknecht et al., 2015, Solomon et al., 2007, Schuster, 2008a, Walters and Petroni, 2007, Inoue et al., 2011]. In fact, many activities might involve memory-only information [Al-Saleh and Al-Sharif, 2013]. The main objective of memory forensics is to analyze volatile data in order to extract digital artifacts.

[Bugcheck, 2006] proposed GrepEXEC forensic tool to extract executive objects such as `ETHREAD` and `EPROCESS` from a memory image by searching for their recognizable signatures. Furthermore, earlier studies [Beebe and Dietrich, 2007, Beebe and Clark, 2007] utilized text-mining techniques to reduce the information retrieval overhead. [Dolan-Gavitt, 2007], used the Virtual Address Descriptor (VAD) tree structure in Windows in memory analysis. In addition, the effectiveness of Microsoft Windows pool allocation methods on memory forensics and incident response procedures were studied by [Schuster, 2008b]. Boosting confidence in the reliability of evidence could be achieved by analyzing the memory data from different consecutive RAM dumps as proposed by [Law et al., 2010].

[Al-Saleh and Al-Sharif, 2012] showed that TCP buffers might still keep data for a long time. With the increasing adoption of virtualization, memory forensics had become an imperative need to engage. [Graziano et al., 2013] proposed a set of techniques to extend the memory forensic domain towards virtual machines and hypervisors analysis. They implemented a new open source forensic tool that was extended to the Volatility framework [Walters, 2007]. Their tool aimed to detect any hypervisor that uses Intel VT-x technology. In addition, their proposed tool had the ability to reconstruct the address space of the virtual machine to support any Volatility plugin aiming to expand the analysis scope of virtual environments. [Dolan-Gavitt, 2009] supports hibernation files analysis in Volatility. In the recent versions of Microsoft Windows OSes, Microsoft changed the hibernation file format, which led to breaking all existing forensic tools. [Sylve et al., 2017] proposed the analysis of a new format of the hibernation file that is used in Windows 8, 8.1 and 10.

[Olajide et al., 2009] analyzed common Windows applications. Their experiments showed that user information such as documents and web pages could be extracted from various memory areas for the tested applications. [Said et al., 2011] studied the effectiveness of the privacy mode feature on widely used web browsers. [Ohana and Shashidhar, 2013] explored memory artifacts from private and portable web browsing sessions. [Al-Khaleel et al., 2014] examined memory artifacts of the Tor bundle. [Pulley, 2013] and [Pshoul, 2017] built Volatility plugins for Windows x86 environment named `exportstack` and `callstacks` respectively. Recently, [Otsuki et al., 2018] implemented a stack trace method as a plugin in Rekall memory analysis framework [Inc, 2017]. A program's state can be explored by utilizing the source code [Al-Sharif et al., 2017, Al-Sharif et al., 2018]. Their results showed that a program's states could still be extracted even after the garbage collector was invoked.

Memory forensics plays an important role in incident response and malware analysis [Schuster, 2006]. Memory analysis had been successfully utilized to detect malware. [Cohen, 2017] applied Yara signatures to memory contents in an

efficient manner. [Lapso et al., 2017] produced a visualization tool to improve the memory analysis methods. Recently, [Al-Saleh et al., 2019] utilized memory forensics to detect network reconnaissance attacks.

Microsoft Visual Studio compiler generates debugging information during the compilation process in the form of a Program DataBase (PDB) file. The idea of utilizing PDB files in digital forensics has been introduced by the authors [Qawasmeh et al., 2019]. Although the format of the PDB files is not officially documented by Microsoft, [Schreiber, 2001] described the internal structure of the PDB files. [Okolica and Peterson, 2010] proposed a memory analysis tool that utilizes the PDB files to extract important kernel structures such as processes, registries and network communication. [Okolica and Peterson, 2011] leveraged the Microsoft's online server PDB files to extract useful system information in both *user32.dll* and *win32k.sys*. [Cohen and Metz, 2014] implemented a PDB parser to calculate kernel symbol addresses. Their work is integrated as a plugin (named *mspdb*) in the Rekall memory analysis framework. [Cohen, 2015] studied the potential differences between Microsoft kernel versions and their impact in memory analysis by the means of PDB files.

The lifetime of data in memory is vital from privacy, forensics, and security perspectives. Information about processes stay in memory for more than a day even after a process terminates [Schuster, 2008a]. Allocated buffers could also keep data after being deallocated [Farmer and Venema, 2004]. [Al-Saleh and Jararweh, 2020] can detect violating machines by collecting magic values in memory. Only small portions of memory in idle machines are changed [Walters and Petroni, 2007]. Dynamic information flow tracking system is used to get a better understanding of data lifetime in memory [Chow et al., 2004]. When sensitive information stays in memory for longer than expected, this creates security and privacy problems [Garfinkel et al., 2004, Engler et al., 2001, Chow et al., 2005, Broadwell et al., 2003]. Consequently, [Chow et al., 2005] proposed a solution for secure memory deallocation. Different programming languages can leave different memory footprints [Al-Sharif et al., 2019]. The artefacts of deleted user accounts can be recovered from different storage devices including memory [Al-Saleh and Al-Shamaileh, 2017]. PDF files can also be carved from memory [Al-Sharif et al., 2015]. Antiviruses could have an impact on the digital evidence [Al-Saleh, 2013].

## 3 Background

This section provides an overview about Volatility framework, Portable Executable (PE) structure, and Program Database (PDB) files. It is not meant to be complete by any means. We aim to highlight important parts that help identify the main thrust of this work.

### 3.1 Volatility

Several memory analysis tools have been developed to help investigators find digital evidence out of volatile data. Among such tools are Volatility lit30, memparser b1, and PTFinderlit9. Volatility is an open source framework that is implemented in Python. As far as we know, Volatility is the most mature open-source memory forensics tool. It has useful functionalities that are provided through its plugin system. The typical way to extend its functionalities is by developing a new plugin and integrating it into Volatility. Volatility has the capability to analyze memory captures that are sourced from different platforms such as Microsoft Windows, Linux, Mac OS X, and Android operating systems. Volatility is utilized for various purposes such as digital forensics, incident response, and malware analysis lit14.

### 3.2 Portable Executable (PE) File

Microsoft introduced the PE file as a common format for Windows applications. The PE file consists of several sections. The **table section** provides information about the other sections such as their names, locations, and sizes. For example, the global variables along with their values are located in the `.data` section. The `.text` section contains the program's code. Each section has a virtual address that is relative to the beginning of the PE file.

### 3.3 Program Database File (PDB)

In Microsoft Windows, debugging information is created by the compiler in a separate file, called Program Database (PDB) file. Basically, PDB files are to locate symbols and related source code information for debugging purposes. Although the PDB format is undocumented, it contains valuable information that can be utilized for forensic purposes. The information inside PDB file can be extracted using DIA (Debug Interface Access), or by using third-party tools.

The internal structure of a PDB file is logically divided into streams. Each stream has a unique number and an optional name. Here we provide brief information about some streams that we utilize in this work:

**Type Information stream (TPI)**: This stream holds information about all data types, known as leaf types. For example, the LF_STRUCTURE type refers to a struct data type and the LF_CLASS type refers to a class data type. Parsing data types depends on the leaf type. For primitive data types, we can look up the whole information in one TPI entry. However, we need more than one step to look up non-primitive types. For example, we lookup the class data type in one entry. This entry includes more information about the members of the class such as the number of its members. In addition, it has a reference that

is labeled LF_FIELDLIST that enables us to access the members' information. The reference is used to lookup another TPI entry.

**Global symbol stream:** This stream includes information about global variables and functions, such as their offsets, names and data types.

**Section header stream:** This stream provides information about the PE file sections.

**Symbols stream:** This stream provides information about the functions, local variables and parameters. This includes the function's start address and size. In addition, the names, data types, and offsets of local variables and parameters are given.

## 4 Methodology

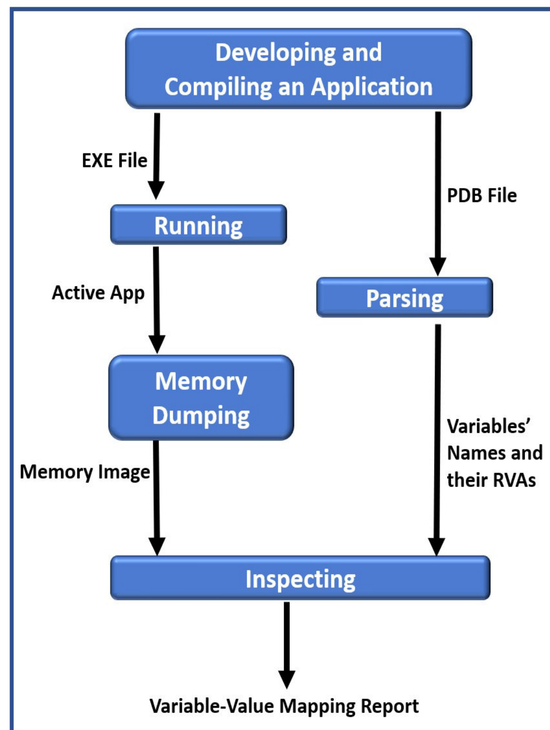


**Figure 1:** Our methodology

Our purpose is to provide a mechanism that enables us to investigate any application in memory. As Volatility affords a rich environment for memory

analysis, we want to utilize it as much as possible. In addition, we want to develop a plugin and integrate it into Volatility. In the end, we want the usage of Volatility to be as simple as running it in its normal usage but providing it with a PDB file, application name, and the name of the newly developed plugin as inputs. The following command usage illustrates how to run Volatility with the new plugin:

```
$python vol.py -f [memory-image] --profile=[OS-profile]
[our-plugin] -a [app-name] --input-file=pdb-file
```

Figure 1 shows the procedure we follow to develop and evaluate our product. As a running example, we used Microsoft Visual Studio as a development environment and the C++ programming language to test our plugin.

1. We develop and compile an application in order to get both executable and PDB files. In Visual Studio, the debugging information that is contained in PDB files can be produced when choosing Debug mode during the compilation process. See Section 3.3 for more information about the PDB file format.

2. We take the executable file and run it in a production environment. While the application is running, a memory dump of the machine is taken. We use Oracle VirtualBox for experimentation and memory acquisition. VirtualBox has built-in features that enable us to capture bit-by-bit copies of the memory. VirtualBox has the capability to capture memory without freezing or interfering with the Virtual Machine. Our Volatility plugin investigates the memory dump in an investigation environment (not in the production environment) and thus no changes are made to the production environment by our plugin.

3. Now that the memory is captured from the production environment, we take the memory dump into the investigation environment. In addition, we take the PDB file from the development environment into the investigation environment.

4. Our Volatility plugin is now ready to perform the analysis step. It is designed to achieve the following functionalities (see **Algorithm 1**):

   (a) It parses the PDB file to extract all variable names and their Relative Virtual Addresses (RVA). It also analyzes their data types. We integrate and use microsoft-pdb parser to parse the PDB file.

   (b) It utilizes Volatility API to obtain the virtual address space (VAS) of the target application out of the memory dump programmatically. Volatility extracts the page tables of the target application and then it walks through them to obtain the virtual address space of the application.

**Algorithm 1** The outline of our Volatility plugin

**Require:** $MemoryImage$, $PDBfile$, $PID$
1: $BaseAddress \leftarrow PID.ImageBaseAddress$
2: $GlbVariables, FunctionsInfo \leftarrow ParsePDB$ ($PDBfile$,$BaseAddress$)
3: //Extract info about functions and global variables
4: $TaskSpace \leftarrow PID.GetProcessAddressSpace$
5: //Parsing PE Info to get VA of .data section ($VAPE$)
6: **for each** $variable$ in $GlbVariables$ **do**
7: $variable.RVA \leftarrow variable.offset + VAPE + BaseAddress$
8: //Extract global variables values
9: Call ExtractVarValue(variable.Name, variable.RVA,
10: variable.DataType, $TaskSpace$)
11: **end for**
12: **for each** $thread$ in $PID$.ThreadListHead **do**
13: //Identify the stack region for each thread
14: $ebp \leftarrow thread.Tcb.TrapFrame.Ebp$
15: $i \leftarrow 0$
16: **while ebp** is Found **do**
17: $Fun \leftarrow FunctionsInfo$[i]
18: //Extract Local variables values for each called function
19: Call ExtractVarValue(Fun.variable.Name, Fun.variable.RVA,
20: Fun.variable.DataType, $TaskSpace$)
21: $ebp \leftarrow OldEbp$
22: $i \leftarrow i$+1
23: **end while**
24: **end for**

(c) It inspects the VAS of the application by looking up the information parsed out of the PDB file. The data types of the variables is crucial in the process of extracting their values.

5. Variable-Value mapping table will then be reported. **Algorithm** 2 explains the process of extracting the values of the variables out of the memory.

## 5 Experimentation and Results

In this Section, we design experiments to validate our methodology. Then, we present an experiment with a real world application.

We proceed with the following procedure:

**Algorithm 2** The outline of extracting variables' Values

```
1: procedure ExtractVarValue(Name, RVA, DataType, TaskSpace )
2:     if DataType is Primitive then
3:         Interpret(format, TaskSpace.Read(RVA,length))
4:     else
5:         LeafTypeInfo ← DataType.LeafType
6:         //Parsing Info from PDB.TPI stream depends on the LeafType
7:         for each variable in LeafTypeInfo do
8:             //Update variable Info
9:             variable.RVA ← variable.offset + RVA
10:            Call ExtractVarValue(variable.Name, variable.RVA,
11: variable.DataType, TaskSpace)
12:        end for
13:    end if
14: end procedure
```

- We develop an application that has specific constructs to verify the ability to extract the values of these constructs out of memory. This is done in a development environment that consists of Microsoft Windows 7 and Visual Studio 2017.
- The application's PDB file is taken so it will be used as an input to our plugin in the investigation environment. In addition, the exe file is taken to the production environment, which is a VirtualBox virtual machine (VM) that runs Microsoft Windows 7 and has 2GB of memory.
- The exe file is executed in the production environment.
- The memory in the production environment is captured into a dump file. We make sure that the application is active (i.e., not terminated) at the memory capture time. We take the memory dump into the investigation environment.
- We run our Volatility plugin in the investigation environment. The memory dump and the PDB file are given to the plugin as inputs.
- The plugin produces variable-value output for the tested application.

Figures (2-7) show code snippets that highlight the constructs we want to investigate. Below that is the parsed information and RVA computation.

### 5.1 Example 1

Figure 2 shows an example of declaring primitive data type variables in the global scope. The primitive data types include boolean, character, integer, float

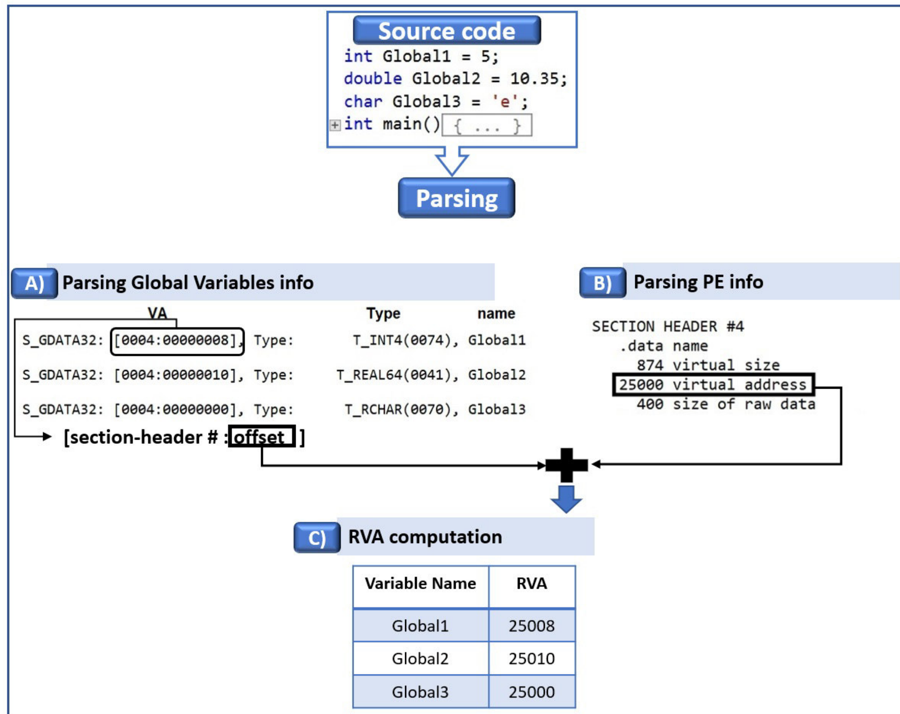


**Figure 2:** Example 1 (primitive data types).

and double data types. Step A in the parsing stage extracts the information of the global stream of a PDB file.

In step A of Figure 2, the first column, S_GDATA32, indicates that this symbol is a global variable. The second bracketed column is used in the virtual address computation. The left part of it (here 0004) refers to the PE section header number that contains this global variable. As mentioned in Section 3.2, the global variables are stored in the `.data` section in the PE file. By referring to the section header stream in the PDB file, we can get the full details about the PE section headers associated with the variables. The relative virtual address of the `.data` section can be extracted from the section header information. As shown in step B of Figure 2, it is (0x25000). The right side of the second column is the variable offset into the `.data` section. To get the RVA of a variable, its offset is added to the RVA of the section it resides in. The third column is the type of the variable and the last column is the variable name. The VA address of the global variable can be obtained by adding its computed RVA to the base

address of the application (`task.Peb.ImageBaseAddress`).

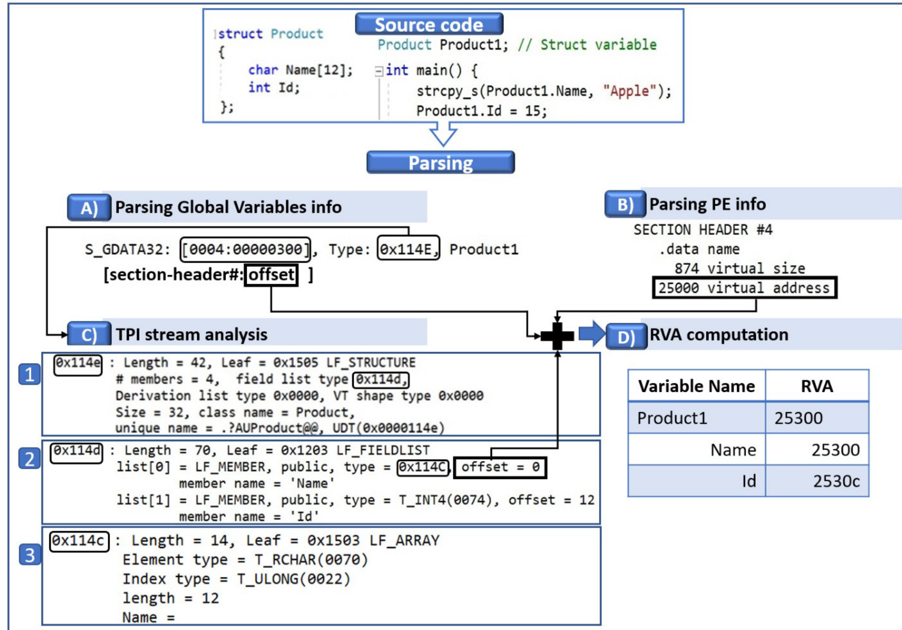


**Figure 3:** Example 2 (struct data type).

## 5.2 Example 2

Figure 3 shows an example of defining a struct data type (named Product) and declaring a global variable of this type (Product1). The defined struct has four members that consist of an array of characters, integer, double and float data types, respectively. The data members are given values in the main function. As explained before, the global information is obtained from the PDB's global stream (step A). However, unlike the primitive data types where the whole information can be obtained from the global stream, in the non-primitive data type more steps are needed. First, a reference address from the global stream is taken (0x114e) and indexed in the TPI stream to get the full information of the variable (step C(1)). To access the struct's members, the reference address (0x114d) is followed as shown in step C(2). Each member has an offset within the struct itself. Consequently, the RVA of each member is computed by adding its offset, the struct's offset (taken from step A), and the VA of `.data` section

(taken from step B). The computed RVAs are listed in the figure. The VA of each member is computed by adding its RVA with the base address of the running application. All the values of the struct' members were successfully extracted out of memory.

### 5.3 Example 3

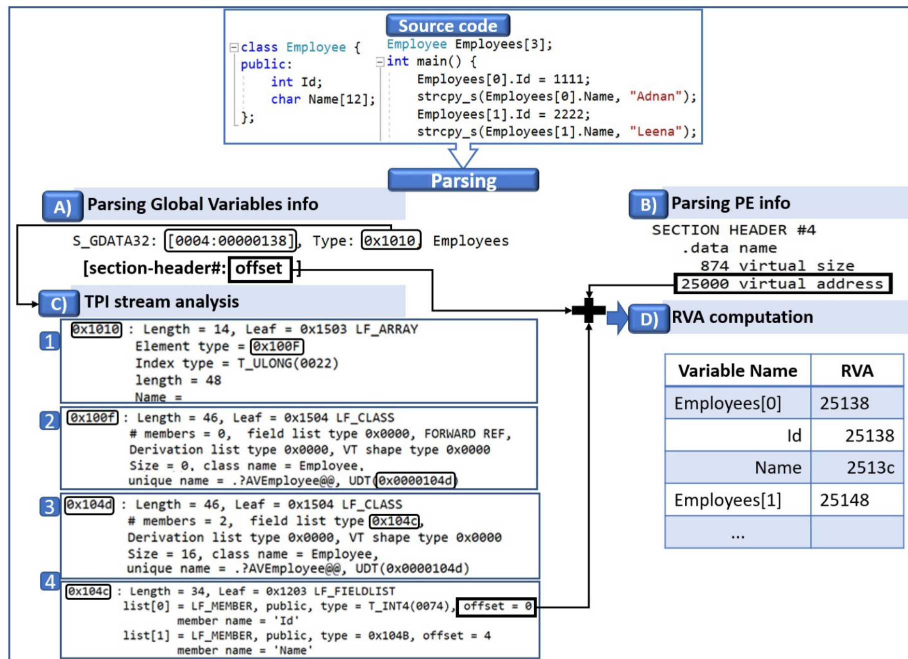


**Figure 4:** Example 3 (array of objects).

Figure 4 shows an example of defining an array of objects in the global scope. The process of identifying the details of a non-primitive data type variable is already explained in Example 2. However, in this example more complexity is added. Investigating an array of objects creates more lookup stages in the TPI stream as shown in the figure. A reference to the array is looked up. We continue to analyze the first element of the array and then its members. All the values of the objects in the array were successfully extracted.

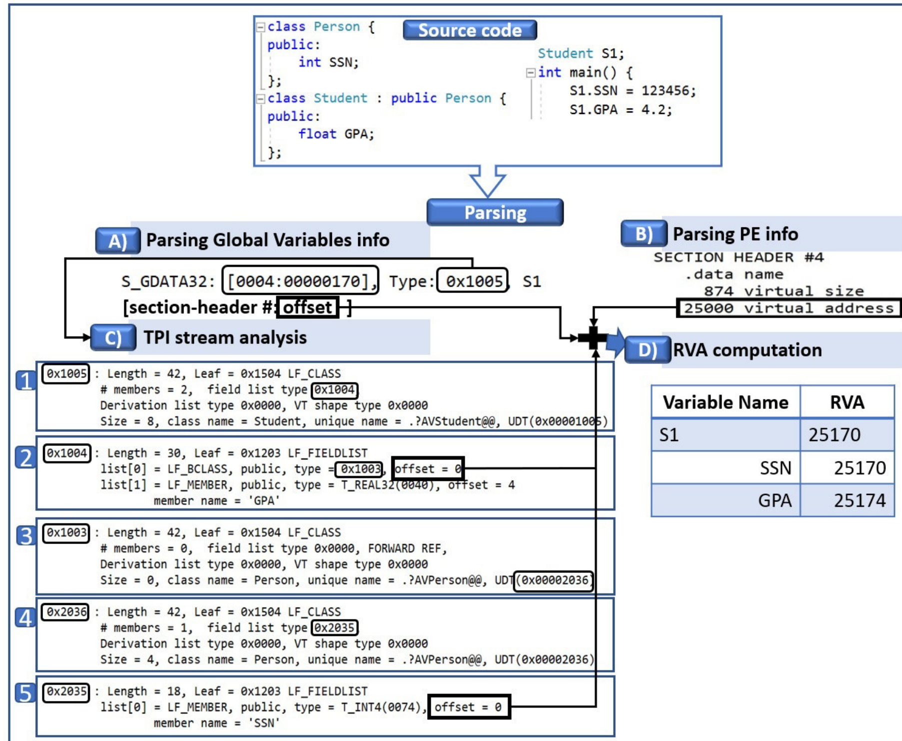


**Figure 5:** Example 4 (class inheritance).

### 5.4 Example 4

In this example, we add one more layer of complexity by using the concept of class inheritance. Figure 5 shows an example of defining a class variable with inheritance. As explained in the previous examples, the information of global variables are shown in the parsing stage (steps A and B in Figure 5). The S1 is a variable of a non-primitive data type. Thus, in step A, the type of non-primitive variable is given by a reference address (0x1005). Therefore, we have to analyze the reference address from the TPI stream to get more information.

In step C(1) of Figure 5, the parsed information of the reference address (0x1005) indicates that the variable is defined from a class data type (LF_CLASS) and its name is Student. To get more information of the class members, the reference address (0x1004) is followed as shown in step C(2). It shows the class members' names, types and offsets. It also indicates that the class inherits from another class by defining it as LF_BCLASS. The reference address to the base

class Person is (0x1003). Following the reference address (0x1003) we obtain the base class's information as shown in steps C(3) and C(4). Finally, step C(5) shows the information of the Person class's members along with names, types and offsets.

The RVA of each class member is computed by adding its offset (step C(2) and step C(5)), the variable's offset (step A), and the RVA of `.data` section (step B). The target VA addresses of a variable can be obtained by adding its RVA to the base address of the target application. Interestingly, all the values of the class members were successfully extracted.

### 5.5 Example 5

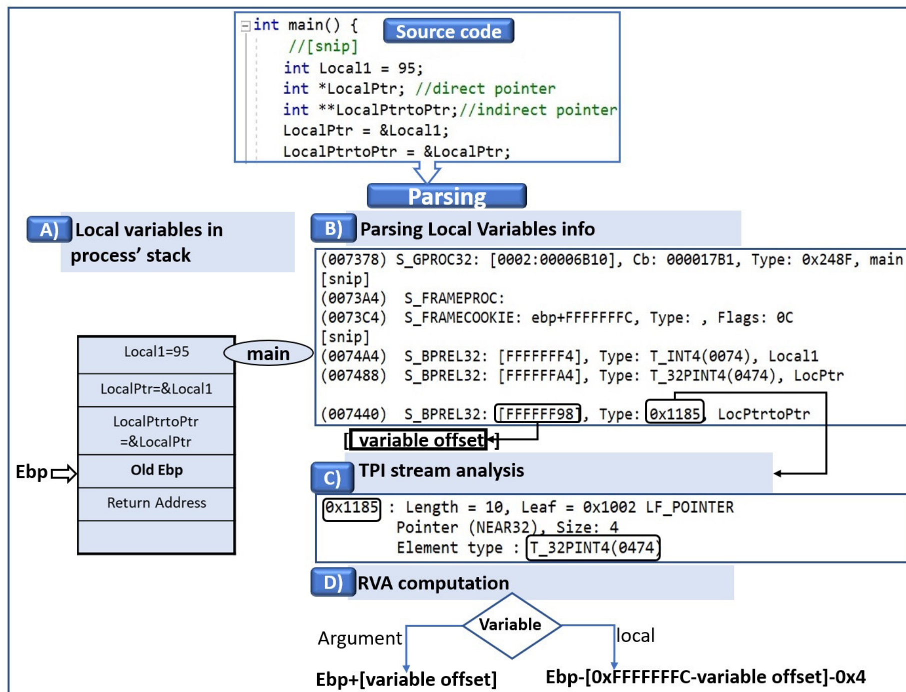


**Figure 6:** Example 5 (local variables).

In Figure 6, local variables are created in different functions. All information about the functions and their local variables are obtained in step B. Local variables and function arguments are placed in positions relative to the ebp's value

of the function frame. The value of the ebp can be obtained using our Volatility plugin by accessing (`thread.Tcb.TrapFrame.Ebp`). The RVA of an argument is computed by adding its offset to the value of the ebp register. The RVA of a local variable is computed by subtracting its offset from S_FRAMECOOKIE (0xFFFFFFFC). Then, the result is subtracted from the ebp value. The following formula is used to compute the RVAs of local variables:

RVA = ebp - (oxFFFFFFFC - offset) - 0x4

All the values of the local variables were successfully extracted.

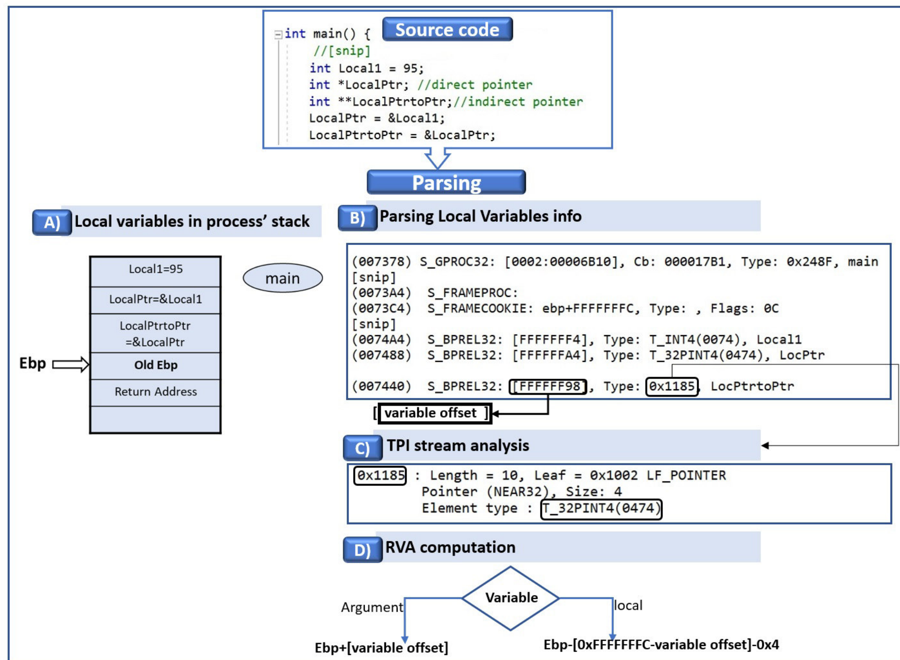


**Figure 7:** Example 6 (pointers).

## 5.6 Example 6

Figure 7 shows an example of using local direct and indirect pointers. The direct pointer LocPtr has T_32PINT4 data type, which indicates a pointer variable. In this case, its value is first de-referenced and then used as another reference to access the variable that it points to. The indirect pointer LocPtrToPtr has a reference data type that needs to be looked up in the TPI stream first. Then,

it continues as we did in the direct pointer case. All values were successfully extracted.

### 5.7 Values Extraction

Volatility provides the capability to access the memory space of the target process through `task.get_process_address_space()` API. The base address of a process is different for each run because of the Address Space Layout Randomization (ASLR) feature adopted by Microsoft Windows OSes. ASLR is a security feature that makes predicting memory addresses more difficult, making hackers' job much harder. Fortunately, we can obtain the base address of a process with Volatility's help by accessing `task.Peb.ImageBaseAddress`. The base address of the process is added to the RVA of a variable to compute the absolute VA of it.

By now, we are able to obtain the VA of the variables and we only need to extract their values out of memory. The byte representation of the variables depends on their data type. For example, variables might have different sizes in memory. The integer variable reserves 4 bytes in memory while double variable reserves 8 bytes. In addition, the format of integer is different from that of double or float. We also take care of the byte order of variables (Endianness) to obtain their values correctly.

Extracting local variables' values is more challenging because the call stack should be traced carefully. Tracing the stack trace can be achieved by following the chain of the current and stored values of the frame pointer register (ebp) and utilizing the values of the return addresses stored in the stack as well. In order to do that, we utilized the parsed symbol records to create a table of all functions in the application. The table contains functions' names, start addresses, and limit addresses. Our plugin traces the stack frames and checks return addresses against the function table we created for this purpose.

The runtime system manages function calls through the means of a call stack. Each called function has its own stack frame that is only visible to that function. Figure 8 shows the stack frame layout. When called, a function pushes its arguments, the return address, the current value of the extended base pointer (ebp) register, and the local variables into the stack. The function code can access the arguments and the local variables via the ebp register. The functions and their local variables are stored in the symbol records stream in the PDB file.

More interestingly, our plugin handles multi-threaded applications, where each thread has its own stack. We only need to identify the stack region for each thread and read its contents accordingly.

In Microsoft Windows, the kernel keeps a process's information in a data structure that is called `_EPROCESS`. Figure 9 shows the internal structure of `_EPROCESS`. The `ActiveThreads` field indicates the number of active threads

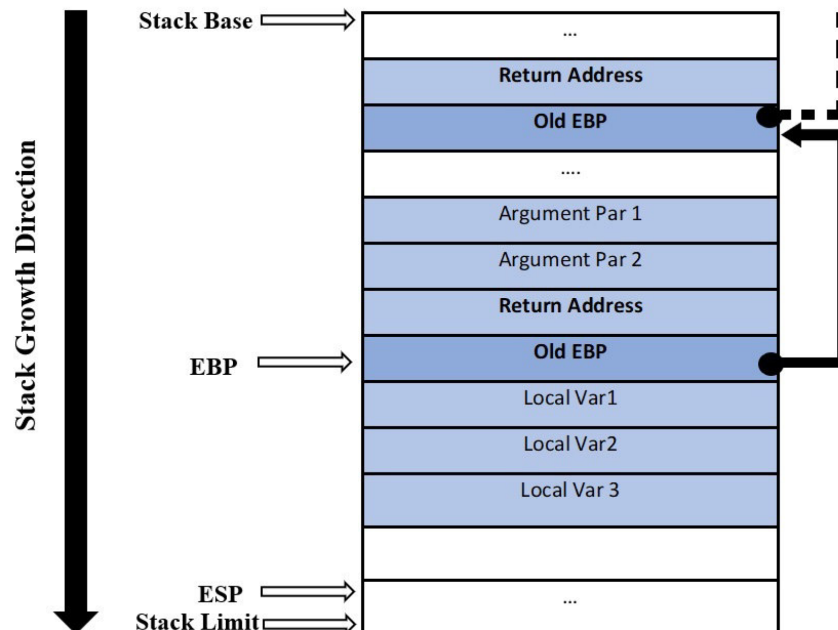


**Figure 8:** Call stack layout.

running in the process context. Threads' information is stored in a doubly linked list structure pointed to by the `ThreadListHead` pointer. Each element has a thread structure of type `_ETHREAD` as shown in figure 9. From `_ETHREAD` we can reach a pointer to `_KTRAP_FRAME` structure, where we can gain the values of the stack registers for each thread.

### 5.8 Experimenting with a real application

We evaluated the effectiveness of our plugin against a real world application that is called Frhed 1.6.0 frhed. It is an open source hex editor. We compiled the application and got its PDB file. We ran the application and loaded a picture file into it to view its hex values. While the application was running, we dumped the memory. Our Volatility plugin parsed the PDB file and utilized its contents to analyze the application. Interestingly, we extracted the values of very important variables about the loaded picture such as its contents, path, name, and size. Table 1 shows the most useful variables we extracted out of the application.

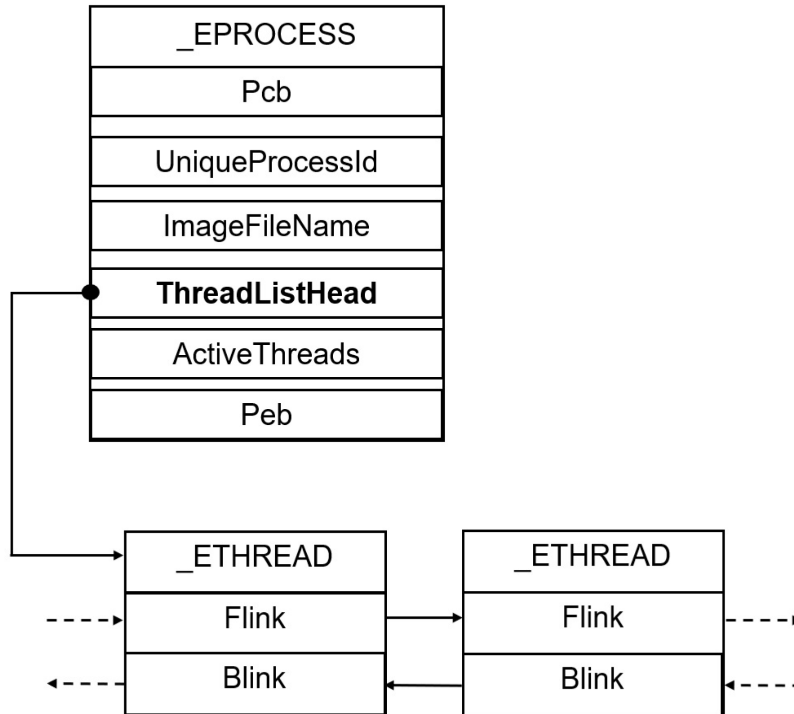


Figure 9: Threads are chained together in a doubly linked list pointed to by ThreadListHead

## 6 Discussion and future work

In this section, we present some limitations of our work and introduce future implications. The most common limitation to any memory forensic research is the volatile feature of memory, where data vanishes when a device is turned off. However, this does not stop investigators from leveraging the invaluable information that resides in memory in case a device is found to be running at the acquisition time.

In our approach, the debugging information of applications must be provided. Some developers might not cooperate to provide such information. In these cases, current forensic techniques should be utilized. Thus, getting the PDB files of an application is not guaranteed. However, in certain situations we still expect application developers to cooperate with the investigation agencies and law-enforcement officers during the investigation process.

Volatility can analyze all libraries (e.g., DLL files) that are linked with a

**Table 1:** Useful variables extracted out of the Frhed application.

| Variable | Variable Leaf Type | Description |
|---|---|---|
| pHexWnd | LF_POINTER | A pointer variable of type HexEditorWindow class. It contains information about the hex editor. |
| pHexWnd->filename | LF_ARRAY | An array variable of type character. It contains the path and name of displayed files. |
| pHexWnd-> DataArray | LF_CLASS | An object variable of type class SimpleArray. It holds the file's data along with many other properties. |
| pHexWnd-> DataArray-> m_pT | T_32PUCHAR(0420) | A pointer variable of type character. It holds the actual address of where the contents of the displayed file is placed in memory. |
| pHexWnd-> DataArray-> m_nSize | T_INT4(0074) | This An integer variable that indicates the size of the displayed file in bytes. |

program that is being investigated. The PDB files of these libraries can also be obtained to fully investigate the program in question. Furthermore, as a running example, we investigated C++ applications on Microsoft Windows OS. The same conceptual idea can also be applied to other programs written in other programming languages on other platforms. For example, DWARF debugging file format can be used in different operating systems, such as Linux and Unix, to address requirements of different programming languages such as C++ and Fortran. This information can be utilized in our plugin in the investigation process. Trying applications that are developed using other programming languages and experimenting with other platforms is a plan for the future.

## 7 Conclusion

When applications are newly developed or updated, new forensic techniques are required to evolve to handle the situation. This paper proposed a novel technique that overcomes the diversity of applications in a unified solution. We leveraged

the debugging information that is provided by compilers about applications in the investigation process. We extended Volatility by integrating a new plugin that is capable of analyzing any application in memory. The experiments we designed showed that our plugin could extract variables' values out of memory regardless of their complexity or scope. The work presented in this paper highly encourages the cooperation between investigation agencies and application developers to combat cybercrimes.

## References

[Al-Khaleel et al., 2014] Al-Khaleel, A., Bani-Salameh, D., and Al-Saleh, M. I. (2014). On the memory artifacts of the tor browser bundle. In *The International Conference on Computing Technology and Information Management (ICCTIM)*, page 41. Society of Digital Information and Wireless Communication.

[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.

[Al-Saleh, 2013] Al-Saleh, M. I. (2013). The impact of the antivirus on the digital evidence. *IJESDF*, 5(3/4):229–240.

[Al-Saleh and Al-Shamaileh, 2017] Al-Saleh, M. I. and Al-Shamaileh, M. J. (2017). Forensic artefacts associated with intentionally deleted user accounts. *IJESDF*, 9(2):167–179.

[Al-Saleh and Al-Sharif, 2012] Al-Saleh, M. I. and Al-Sharif, Z. A. (2012). Utilizing data lifetime of tcp buffers in digital forensics: Empirical study. *Digital Investigation*, 9(2):119–124.

[Al-Saleh et al., 2019] Al-Saleh, M. I., Al-Sharif, Z. A., and Alawneh, L. (2019). Network reconnaissance investigation: A memory forensics approach. In *2019 10th International Conference on Information and Communication Systems (ICICS)*, pages 36–40. IEEE.

[Al-Saleh and Jararweh, 2020] Al-Saleh, M. I. and Jararweh, Y. (2020). Fingerprinting violating machines with in-memory protocol artefacts. *International Journal of Advanced Intelligence Paradigms*, 15(4):388–404.

[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. IEEE.

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

[Al-Sharif et al., 2019] Al-Sharif, Z. A., Al-Saleh, M. I., Jararweh, Y., Alawneh, L., and Shatnawi, A. S. (2019). The effects of platforms and languages on the memory footprint of the executable program: A memory forensic approach. *Journal of Universal Computer Science*, 25(9):1174–1198.

[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.

[Beebe and Dietrich, 2007] Beebe, N. and Dietrich, G. (2007). A new process model for text string searching. In *IFIP International Conference on Digital Forensics*, pages 179–191. Springer.

[Beebe and Clark, 2007] Beebe, N. L. and Clark, J. G. (2007). Digital forensic text string searching: Improving information retrieval effectiveness by thematically clustering search results. *Digital investigation*, 4:49–54.

[Broadwell et al., 2003] Broadwell, P., Harren, M., and Sastry, N. (2003). Scrash: a system for generating secure crash information. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 19–19, Berkeley, CA, USA. USENIX Association.

[Bugcheck, 2006] Bugcheck, C. (2006). Grepexec: Grepping executive objects from pool memory. In *Proc. Digital Forensic Research Workshop*.

[Chow et al., 2004] Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., and Rosenblum, M. (2004). Understanding data lifetime via whole system simulation. In *Proc. 13th USENIX Security Symposium*.

[Chow et al., 2005] Chow, J., Pfaff, B., Garfinkel, T., and Rosenblum, M. (2005). Shredding your garbage: reducing data lifetime through secure deallocation. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 22–22, Berkeley, CA, USA. USENIX Association.

[Cohen, 2017] Cohen, M. (2017). Scanning memory with yara. *Digital Investigation*, 20:34–43.

[Cohen and Metz, 2014] Cohen, M. and Metz, J. (2014). Ms pdb parser. https://github.com/google/rekall/commit/89f4f2832d99eac3b783b02ce9025806eaca6bd8.

[Cohen, 2015] Cohen, M. I. (2015). Characterization of the windows kernel version variability for accurate memory analysis. *Digital Investigation*, 12:S38–S49.

[Dolan-Gavitt, 2007] Dolan-Gavitt, B. (2007). The vad tree: A process-eye view of physical memory. *digital investigation*, 4:62–64.

[Dolan-Gavitt, 2009] Dolan-Gavitt, B. (April 2009). Add support for inactive hiberfiles to hibinfo volatilityfoundation/ volatility@552c1d8. https://github.com/volatilityfoundation/volatility/commit/552c1d813b05a0bf8d3d1ec1f64b3ba5f98403cc.

[Engler et al., 2001] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. (2001). Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 57–72, New York, NY, USA. ACM.

[Farmer and Venema, 2004] Farmer, D. and Venema, W. (2004). *Forensic Discovery*. Addison Wesley Professional.

[Garfinkel et al., 2004] Garfinkel, T., Pfaff, B., Chow, J., and Rosenblum, M. (2004). Data lifetime is a systems problem. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, EW 11, New York, NY, USA. ACM.

[Graziano et al., 2013] Graziano, M., Lanzi, A., and Balzarotti, D. (2013). Hypervisor memory forensics. In *International Workshop on Recent Advances in Intrusion Detection*, pages 21–40. Springer.

[Hausknecht et al., 2015] Hausknecht, K., Foit, D., and Burić, J. (2015). Ram data significance in digital forensics. In *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1372–1375. IEEE.

[Inc, 2017] Inc, G. (2017). Rekall memory forensic framework. http://www.rekall-forensic.com/.

[Inoue et al., 2011] Inoue, H., Adelstein, F., and Joyce, R. A. (2011). Visualization in testing a volatile memory forensic tool. *Digital Investigation*, 8(Supplement):S42–S51.

[Lapso et al., 2017] Lapso, J. A., Peterson, G. L., and Okolica, J. S. (2017). Whitelisting system state in windows forensic memory visualizations. *Digital Investigation*, 20:2–15.

[Law et al., 2010] Law, F., Chan, P., Yiu, S.-M., Tang, B., Lai, P., Chow, K.-P., Ieong, R., Kwan, M., Hon, W.-K., and Hui, L. (2010). Identifying volatile data from multiple memory dumps in live forensics. In *IFIP International Conference on Digital Forensics*, pages 185–194. Springer.

[Ohana and Shashidhar, 2013] Ohana, D. J. and Shashidhar, N. (2013). Do private and portable web browsers leave incriminating evidence?: a forensic analysis of residual artifacts from private and portable web browsing sessions. *EURASIP Journal on Information Security*, 2013(1):6.

[Okolica and Peterson, 2010] Okolica, J. and Peterson, G. L. (2010). Windows operating systems agnostic memory analysis. *Digital investigation*, 7:S48–S56.

[Okolica and Peterson, 2011] Okolica, J. and Peterson, G. L. (2011). Extracting the windows clipboard from physical memory. *digital investigation*, 8:S118–S124.

[Olajide et al., 2009] Olajide, F., Savage, N., et al. (2009). Application level evidence from volatile memory. *Journal of Computing in Systems and Engineering*, 10:171–175.

[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.

[Pshoul, 2017] Pshoul, D. (2017). community/dimapshoul at master volatilityfoundation/community github. `https://github.com/volatilityfoundation/community/tree/master/DimaPshoul`.

[Pulley, 2013] Pulley, C. (2013). Github - carlpulley/volatility: A collection of volatility framework plugins. `https://github.com/carlpulley/volatility`.

[Qawasmeh et al., 2019] Qawasmeh, E., Al-Saleh, M. I., and Al-Sharif, Z. A. (2019). Towards a generic approach for memory forensics. In *2019 Sixth HCT Information Technology Trends (ITT)*, pages 094–098.

[Said et al., 2011] Said, H., Al Mutawa, N., Al Awadhi, I., and Guimaraes, M. (2011). Forensic analysis of private browsing artifacts. In *2011 International Conference on Innovations in Information Technology*, pages 197–202. IEEE.

[Schreiber, 2001] Schreiber, S. B. (2001). *Undocumented Windows 2000 secrets: a programmer's cookbook*. Addison-Wesley Longman Publishing Co., Inc.

[Schuster, 2006] Schuster, A. (2006). Searching for processes and threads in microsoft windows memory dumps. *digital investigation*, 3:10–16.

[Schuster, 2008a] Schuster, A. (2008a). The impact of microsoft windows pool allocation strategies on memory forensics. *Digital Investigation*, 5, Supplement(0):S58 – S64. The Proceedings of the Eighth Annual DFRWS Conference.

[Schuster, 2008b] Schuster, A. (2008b). The impact of microsoft windows pool allocation strategies on memory forensics. *Digital Investigation*, 5:S58–S64.

[Solomon et al., 2007] Solomon, J., Huebner, E., Bem, D., and Sze?ynska, M. (2007). User data persistence in physical memory. *Digital Investigation*, 4(2):68 – 72.

[Sylve et al., 2017] Sylve, J. T., Marziale, V., and Richard III, G. G. (2017). Modern windows hibernation file analysis. *Digital Investigation*, 20:16–22.

[Walters, 2007] Walters, A. (2007). The volatility framework: Volatile memory artifact extraction utility framework.

[Walters and Petroni, 2007] Walters, A. and Petroni, N. L. (2007). Volatools : Integrating volatile memory forensics into the digital investigation process. *Digital Investigation*, pages 1–18.