Memory Forensics Comparison of Apple M1 and Intel Architecture Using Volatility Framework

Joshua Duke
Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_theses

Part of the Information Security Commons, and the Systems Architecture Commons

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_theses/5477

This Thesis is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Master's Theses by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
MEMORY FORENSICS COMPARISON OF APPLE M1 AND INTEL ARCHITECTURE USING VOLATILITY FRAMEWORK

A Thesis
Submitted to the Graduate Faculty of the Louisiana State University and Agricultural and Mechanical College in partial fulfillment of the requirements for the degree of Master of Science in The Department of Computer Science and Engineering

by Joshua Alexander Duke
B.S., Louisiana State University, 2016 December 2021
Acknowledgments

I would like to thank Dr. Golden G. Richard III for all his help during my time at LSU. His expertise and knowledge of memory forensics as well as his advice was very helpful throughout my research. I would like to thank Andrew Case for all his help and knowledge throughout this thesis. I would like to thank Dr. Anas Mahmoud and Dr. Gerald Baumgartner for being a part of my thesis committee. I would like to thank my friends in the cyber lab for their support throughout my degree and in my research.
# Table of Contents

Acknowledgements ................................................................................. ii

List of Tables ....................................................................................... iv

List of Figures ........................................................................................ v

Abstract .................................................................................................. vii

Chapter 1. Introduction .......................................................................... 1  
  1.1. Memory Forensics ............................................................... 1  
  1.2. History of Apple Processors ............................................. 1  
  1.3. CISC vs. RISC processors .................................................. 3  
  1.4. Research Importance ......................................................... 5  
  1.5. Analyzing Output from Memory Samples ....................... 5  
  1.6. Outline ................................................................................. 6

Chapter 2. Volatility and Memory Sample Acquisition .......................... 7  
  2.1. Volatility ............................................................................. 7  
  2.2. Memory Management in MacOS ....................................... 7  
  2.3. Memory Sample Acquisition ............................................. 9

Chapter 3. Design and Implementation .............................................. 10  
  3.1. Setting up Volatility Environment .................................. 10  
  3.2. Execution of Plugins ......................................................... 11  
  3.3. How Analysis was Done .................................................. 11

Chapter 4. Results ............................................................................. 12  
  4.1. Compiling the Results ...................................................... 12  
  4.2. Discussion ............................................................................. 22

Chapter 5. Conclusion and Future Work .......................................... 24  
  5.1. Conclusion ............................................................................. 24  
  5.2. Future Work .......................................................................... 24

References ............................................................................................ 25

Vita ........................................................................................................... 27
List of Tables

4.1 Volatility Plugins that Worked on Only One Sample ....................... 14
4.2 Volatility Plugins that Resulted in Errors for Both Intel and M1 Samples ..... 18
List of Figures

1.1 Apple’s CPU Performance and Power Consumption Comparison ............... 4
2.1 Volatility mac_pstree Output Example ........................................ 7
2.2 MacOS Memory Management ....................................................... 8
2.3 Example of Higher Memory Pressure on a System ............................. 9
2.4 Surge Collect Running on M1 ....................................................... 9
3.1 Surge meta.json Snippet Showing Shift and DTB Values ..................... 10
3.2 mac_moddump Plugin Command for Intel Memory Sample .................... 11
3.3 mac_psxview Command for M1 Memory Sample .............................. 11
4.1 Intel mac_memdump Output Snippet ............................................. 12
4.2 M1 mac_memdump Output Snippet ............................................... 13
4.3 Intel mac_procdump Output Snippet ........................................... 13
4.4 M1 mac_procdump Output Snippet .............................................. 13
4.5 M1 mac_check_syscall_shadow Error Message ................................. 15
4.6 M1 mac_check_sysctl Error Message ........................................... 15
4.7 M1 mac_check_trap_table Error Message ..................................... 15
4.8 M1 mac_malfind Error Message .................................................. 16
4.9 M1 mac_arp Output Snippet ....................................................... 16
4.10 Intel mac_arp Error Message ..................................................... 17
4.11 Intel mac_check_mig_table Error Message .................................... 17
4.12 Intel mac_lsof Error Message ................................................... 17
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.13</td>
<td>M1 <code>mac_apihooks_kernel</code> Error Message</td>
<td>18</td>
</tr>
<tr>
<td>4.14</td>
<td><code>mac_get_profile</code> Error Message for Both Samples</td>
<td>19</td>
</tr>
<tr>
<td>4.15</td>
<td><code>mac_compressed_swap</code> Error Message for Both Samples</td>
<td>19</td>
</tr>
<tr>
<td>4.16</td>
<td><code>mac_dump_file</code> Error Message for Both Samples</td>
<td>19</td>
</tr>
<tr>
<td>4.17</td>
<td><code>mac_kevents</code> Error Message for Both Samples</td>
<td>20</td>
</tr>
<tr>
<td>4.18</td>
<td>Intel <code>mac_dead_procs</code> Error Message</td>
<td>20</td>
</tr>
<tr>
<td>4.19</td>
<td>M1 <code>mac_dead_procs</code> Error Message</td>
<td>20</td>
</tr>
<tr>
<td>4.20</td>
<td>Intel <code>mac_timers</code> Error Message</td>
<td>21</td>
</tr>
<tr>
<td>4.21</td>
<td>M1 <code>mac_timers</code> Error Message</td>
<td>21</td>
</tr>
</tbody>
</table>
Abstract

Memory forensics allows an investigator to get a full picture of what is occurring on-device at the time that a memory sample is captured and is frequently used to detect and analyze malware. Malicious attacks have evolved from living on disk to having persistence mechanisms in the volatile memory (RAM) of a device and the information that is captured in memory samples contains crucial information for full forensic analysis by cybersecurity professionals. Recently, Apple unveiled computers containing a custom designed system on a chip (SoC) called the M1 that is based on ARM architecture. Our research focused on the differences in the Volatility memory analysis framework between Apple’s new M1 SoC and its previous Intel-based CPUs due to the new architecture. We extracted memory samples from a MacBook Air equipped with a M1 SoC and a Intel-based Mac virtual machine. Using those samples, we ran all the Volatility plugins available for Mac against each memory sample, taking note of any differences or errors that occurred because of the shift in architecture. This is foundational memory forensics work on the M1 ARM platform that will allow future research and improvements to be made on Volatility for M1.
Chapter 1. Introduction

1.1. Memory Forensics

Memory forensics has become widely used by cybersecurity professionals as cyber attacks and investigations become more common. Memory forensics is the analysis of data that is contained in a computer’s volatile memory (RAM) to determine the state of the machine at the time that the memory sample was taken. The memory sample is then processed with memory forensics tools and frameworks and evidence is gathered to assist in investigations. This evidence could be recovered files, malicious processes that are running, artifacts of a cyber attack, and much more. Malware authors in many cases have shifted from persistence mechanisms on disk to persistence mechanisms in volatile memory, making memory forensics even more valuable in investigations.

Memory samples can be obtained using either open-source or commercially available tools, and the size of those memory samples depends on the amount of RAM that is on the system. Additionally, multiple samples can be obtained in order to do analysis of the system from two different points in time. This allows investigators to determine if there was any significant change in behavior of the system between the two samples in order to discover malicious activity.

Volatility is a memory forensics framework that is the most widely used by cybersecurity professionals to analyze memory samples. It has many different plugins that extract a large amount of information about the system, and additional plugins can be written to extend the functionality of Volatility. This research is focusing on how Volatility interacts with memory samples taken from Apple’s new ARM-based M1 SoC and comparing its outputs to a memory sample taken from an Intel-based Apple system.

1.2. History of Apple Processors

The chip that was used in the Apple 1 and Apple 2 computers was the MOS 6502, an 8-bit processor that was in use from 1976 to 1979 [8]. With the release of the Apple
Lisa in 1983 and later the Apple Machintosh in 1984, Apple transitioned to a new chip, the Motorola 68000 (68k). This was the chip that was used in Apple’s computers until it was discontinued 1995 due to the shift to PowerPC. In 1994, Apple introduced the Power Macintosh, moving to the PowerPC architecture which utilized reduced instruction set computing (RISC). To ease the transition to the new hardware, Apple shipped a 68k emulator with MacOS so the vast majority of software would be compatible with the latest Macs that had PowerPC hardware.

The move to Intel x86 chips began in 2006, with the release of an iMac and MacBook Pro that had Intel processors. The reason for this shift was due to issues with PowerPC chips being able to match Intel processors’ speeds because of manufacturing and design issues as well as a higher power consumption compared to Intel’s chips [5]. To help with this transition, Apple released emulation software called Rosetta so a program’s PowerPC instructions could be translated into x86 instructions. These were the chips that were used in all Apple computers until Apple announced the transition to custom built Apple silicon starting in 2020 with the release of the M1 system on a chip in the Macbook Air, Macbook Pro, and Mac Mini. These devices also shipped with emulation software called Rosetta 2 to translate from x86 instructions to ARM instructions so the transition could be much smoother to the new devices.

1.2.1. Apple Silicon

Apple said the transition to Apple silicon would take about two years and all of its Mac products would use Apple silicon by the end of that period [2]. The M1 is the first in the lineup of Apple silicon chips for Mac. Because the M1 uses ARM, which the iPhone and iPad both use as well, the architecture across Apple’s products is unified which allows iPhone and iPad apps to run on Mac systems without modifications. This is one of the many benefits for Apple as it shifts to ARM in the Mac lineup [1].

Apple’s M1 has a few major differences from the Intel CPUs that were in Apple systems. The first major difference is that the M1 is a system on a chip (SoC) whereas the Intel
processors were just the CPU. The M1 has CPU cores, GPU cores, DRAM, and more all on one chip. It is the first system on a chip for Apple’s Mac systems [2]. Another major difference is the architecture; the Intel CPUs are based on Intel’s x86 architecture and the M1 SoC is based on the ARM architecture. Additionally, Intel’s CPUs are complex instruction set computing (CISC) processors whereas the M1 soc is a RISC processor because it is using the ARM architecture. A comparison of CISC and RISC processors will be covered in a later section. The shift to Apple silicon is similar to the shift when Apple began creating its own chips for the iPhone and iPad in 2010 with the release of the A4 system on a chip [18].

There are many different advantages for Apple to shift towards Apple silicon, with the primary benefits being performance, power consumption, and greater control over their systems. Because the M1 is a SoC, the multiple chips that Apple would use to perform different tasks on previous Mac products can all be combined into the single M1 SoC, resulting in better performance and better energy efficiency [2]. Additionally, ARM processors not only result in better battery life due to the single clock cycle instruction execution, but they also result in cooler running temperatures, which is why the M1 MacBook Air has no active cooling [7]. With the release of the M1 Pro and M1 Max SoCs in October 2021, Apple has continued to raise the bar on performance. The M1 Pro has over two times the number of transistors, 33.7 billion, compared to the M1, and the M1 Max has three-and-a-half times the number of transistors, 57 billion, compared to the M1, allowing Apple to pack even more performance into their chips as shown in figure 1.1 [3]. Apple silicon also includes a Neural Engine which aims to improve performance of machine learning tasks by a large margin. According to Apple, “the Neural Engine in M1 enables up to 15x faster machine learning performance”, further increasing performance gains for Apple’s Mac systems [2].

1.3. CISC vs. RISC processors

This section will break down the major difference between CISC and RISC processors and a comparison of the total number of instructions between the Intel x86 architecture
and the ARM architecture. RISC processors are designed to have simpler instruction sets so that, though it may take more instructions for a certain task, each instruction executes faster than instructions on CISC processors. Instructions on CISC processors may take multiple clock cycles to complete, whereas ARM-based RISC processors use only one clock cycle to complete. CISC is based on complex hardware that is able to handle those complex instructions in order to make the assembly programs simpler by using fewer total instructions. For RISC it is the opposite, it is based on simple hardware that can handle the reduced instruction set, but results in more complex assembly programs. [7].

Because the instructions take fewer clock cycles to complete, there is less energy wasted which increases the battery life for mobile devices such as the iPhone, iPad, Apple Watch, and Apple Silicon powered laptops, and power consumption for desktop devices like the M1 iMac is lowered as well. That is why Apple is able to get greatly improved performance from Apple silicon without the need to draw more power than comparable PC laptop chips.

There is also a large difference between the total number of instructions for Intel’s 64-bit instruction set that is used on their processors and the 64-bit ARM instruction set. When
comparing the two reference manuals, the Intel reference manual Volume 2 has 1984 pages for the A-Z instruction set reference [6]. In comparison, the Armv8 reference manual has 1579 pages for the A-Z descriptions for the A64 instruction set. That means there are 405 more pages with additional instructions and their descriptions in the Intel reference manual compared to the Armv8 manual [10]. There are also formatting differences between the two manuals that obscure the true difference between the total instructions. For instance, in the Armv8 manual, there are seven pages dedicated to the ADD instruction and its variants such as ADD extended and ADD immediate. In comparison, there are only two pages in the Intel reference manual that address the ADD instruction and all its variants. Taking that into account further underscores the fact that ARM consists of far fewer total instructions than Intel.

1.4. Research Importance

This architecture shift from Intel to ARM means that existing cybersecurity tools such as Volatility will need to make adjustments for the new systems to ensure that as much information is able to be extracted for investigations. Our research is important because there has already been documented malware that has been compiled natively for Apple silicon, the first of which was just months after the first M1 systems were released [12]. This highlights the need for cybersecurity tools to adapt quickly to the architecture shift so investigators are able to detect and prevent these attacks from happening. Additionally, due to this shift, we had to acquire the memory sample using Volexity’s Surge software as it had been recently updated to support memory sample extraction for the M1 SoC and there were no other viable alternatives such as virtualization for easy memory sample acquisition [17].

1.5. Analyzing Output from Memory Samples

The memory images from the Intel and M1 Macs were run through every Volatility plugin available that supported MacOS, a total of seventy-seven plugins, in order to deter-
mine major behavioral differences between the two architectures and if any errors occurred as a result of the architecture shift. Once the plugins were executed, the outputs and error messages were examined to make determinations about shifts between the two architectures. This is important because if there are Volatility plugins that do not work on the new architecture, or behave differently than expected, that limits cybersecurity professionals' ability to locate indicators of compromise when conducting a forensics investigation. Any findings would help the Volatility Foundation keep Volatility up to date and ensure that investigators are able to continue using the framework on the latest computer architectures.

1.6. Outline

Chapter 2 covers how Volatility works, how MacOS manages memory, and how the memory samples were obtained for analysis. Chapter 3 talks about the analysis of the memory samples using Volatility. Chapter 4 is a compilation of the results from the Volatility plugins and a discussion about the findings. Chapter 5 concludes the research and focuses on future work of memory forensics on M1.
Chapter 2. Volatility and Memory Sample Acquisition

2.1. Volatility

The Volatility framework is an open-source memory analysis tool that can analyze the data found in the volatile memory (RAM) of a system [14]. Volatility supports memory analysis for all major operating systems such as Windows, Mac, and Linux. In order to run the plugins that Volatility supports, specific profiles are needed that match the version of the current operating system that the machine is running [15]. Volatility has a variety of plugins that can extract different types of information from the memory sample. As an example of a plugin for Mac, the `mac_pstree` plugin is able to construct a tree of running processes on the machine at the time that the memory sample was taken in order to determine the parent and child relationships between processes [9].

```
$ python vol.py --profile=MacMountainLion_10_8_3_AMDx64
 -f 10.8.3x64.vmem mac_pstree
Volatility Foundation Volatility Framework 2.4
Name      Pid   Uid
kernel_task  0     0
.launchd     1     0
..launchd    213   89
...mdworker  227   89
...cfprefsd  217   89
...distnoted 216   89
..coresymbolicacio 211   0
..com.apple.audio. 203   202
<snip>
..launchd    133   501
...Terminal   199   501
....login    218   0
......bash   219   501
.......sudo   222   0
.......dtrace 223   0
```

Figure 2.1. Volatility `mac_pstree` Output Example

2.2. Memory Management in MacOS

On MacOS, there are three categories of physical memory: Memory Used, Cached Files, and Swap Used. The Memory Used category consists of three distinct categories:
App Memory, Wired Memory, and Compressed Memory. App memory is the memory that is currently being used by apps that are running on the system. Wired memory is memory that cannot be cached and is required to run the system. It must remain in RAM so other apps on the system will not use it. Compressed memory allows the system to free up more RAM as it begins to approach the maximum memory capacity. RAM used by inactive apps is compressed, which frees up more RAM for other active apps [4].

![Memory Pressure Graph](image)

Figure 2.2. MacOS Memory Management

Cached files are stored in unused regions of memory to help improve performance of reopening apps on the system. These files will remain cached unless the region of memory is overwritten. Swap used refers to the amount of storage space that is currently in use on disk to swap unused files from RAM to disk. All of this is done to allow the operating system to manage the RAM as efficiently as possible to achieve the best performance [4]. Swap space could contain useful forensic evidence in an investigation, but is often difficult to analyze reliably. Additionally, with the use of encrypted swap files, extracting that information becomes even more complicated. Memory smearing during capture creates another issue, causing inconsistencies between the swap file and process page tables and kernel structures in charge of virtual memory. This makes extracting useful information from the swap files, in practice, unpractical. The use of compressed memory means that pages that would have been swapped out become compressed, allowing for Volatility plugins to be developed to extract information that would have otherwise been difficult to extract [13].

The memory pressure graph is a representation of the system’s ability to manage the memory used by applications. As memory pressure increases, the system could suffer performance degradation as there are not enough resources for all the applications [4].
2.3. Memory Sample Acquisition

While Volatility is able to analyze memory samples, it does not have memory sample acquisition capabilities. There are both open-source and commercially available memory acquisition programs that can be used to extract the data from RAM. Because our research focused on two different architectures and because the M1 was so new, we needed to use two different methods to extract the memory samples. The Intel-based Mac system was a virtual machine running on Mac hardware, so the memory sample was obtained by taking a snapshot of the virtual machine. For the ARM-based M1 Mac, the memory sample had to be obtained using Volexity’s Surge Collect software [17]. This is because the architecture was new, and there was not any other readily-available software that had been updated to support the new M1 Mac. Once the memory samples were obtained, analysis with Volatility could begin.

```bash
Josh@Josh-Air.eng.:~$ sudo ./surge-collect-md-new /path/to/your/sample /path/to/output/
Password: ...
```

Figure 2.4. Surge Collect Running on M1
Chapter 3. Design and Implementation

3.1. Setting up Volatility Environment

Because the M1 Mac architecture is ARM-based, we had to set up two different Volatility environments in order to do the analysis. One was the Volatility Foundation’s normal Volatility repository, and the other was a branch of Volatility that has added in arm64 support [16].

In addition to the two Volatility environments, the correct profiles needed to be obtained so analysis could be done. Both the Intel and ARM-based Mac systems were on the same build version of MacOS, Big Sur 11.4 20F71, to ensure the systems were as similar as possible before the memory samples were taken. That way the only major differences in the outputs would be a result of Volatility’s operation on the samples and the structure of the memory samples themselves. These profiles were provided by Volexity.

Once the profiles were created, the Intel memory sample was ready for analysis, but the M1 sample was not ready yet. That is because there are additional arguments that had to be passed in the Volatility command-line for ARM: the kernel address space layout randomization (KASLR) shift address and the directory table base (DTB) address as shown in figure 3.1. The KASLR value is needed because the offset specified in the profile and the actual offset of where the variables are in the memory sample are different. The DTB value is used for address translation [9]. These values are extracted automatically for the Intel sample, but the process to do that for ARM is not currently implemented, so the values are taken from the metadata file.

```
{"darwin":{"KASLRShift":485703680,"PageSize":16384},"files":{},"format":2,"memory":
{"dtb":34487664640,"segments": [{"address":34386755584,"size":16502620160}],"platform":
{"TimeZone":"CDT","offset":-18000},"architecture":arm64,"arguments":[]}
```

Figure 3.1. Surge meta.json Snippet Showing Shift and DTB Values
3.2. Execution of Plugins

Once the Volatility environments were set up, each of the Mac-compatible Volatility plugins were run on the memory samples. For each plugin, the output was piped into a file, and any error messages that occurred were also piped into a file for analysis after the fact. For plugins that extracted files, a separate directory was created to store the extracted data for both memory samples. Once every plugin was run for both memory samples, analysis of any major differences between the two systems could begin.

Figure 3.2. *mac_moddump* Plugin Command for Intel Memory Sample

```
@ubuntu:~$ volatility --volatilityenv=/macOS_11.0_FinalSnapshot1.vmem --profile=MacBigSur_11_4_20F71x64 mac_moddump -D ..dumpdir/
```

Figure 3.3. *mac_psxview* Command for M1 Memory Sample

```
@ubuntu:~$ volatility --volatilityenv=/arm_volatility/volatility --profile=MacBigSur_11_4_20F71x64_t8101arm64 --shift=485703680 --dtb=34487664648 mac_psxview > ../outputs/arm_mac_psxview
```

3.3. How Analysis was Done

After every plugin was run against both samples, each output was carefully examined and differences in results, such as if a certain plugin worked for one sample but not the other, if error messages differed between samples, or if the outputs between samples were substantially different, were noted. Once this process was completed, those differences were examined more closely to determine what the causes might have been, whether that is because of the architecture difference between Intel and ARM or if Volatility plugins themselves were the cause of the differences. The ultimate goal is to carefully document issues so that patches to Volatility can be developed to enhance operation on M1 Macs.
Chapter 4. Results

4.1. Compiling the Results

There are a total of seventy-seven Mac plugins that are supported by Volatility both for the Intel and ARM systems. Each of those plugins was run on both samples, and these were the results. Of the seventy-seven plugins, thirty-five of them worked on both the Intel and ARM systems. Twelve plugins resulted in errors, but they had the same error message depending on the plugin. There were five plugins that resulted in errors on both samples, but the error messages were different. Fifteen plugins worked on Intel, but not on M1. There were eight plugins that did not work on Intel, but worked on M1. Finally, there were two plugins that had odd behavior. These two plugins were *mac_memdump* and *mac_procdump*. The *mac_memdump* plugin dumps addressable memory pages to a file and *mac_procdump* dumps the executable of a process to a file.

We will examine *mac_memdump* and *mac_procdump* first. In the case of both of these plugins, they both ran to completion without emitting error messages. For *mac_memdump* on the Intel memory sample, the files dumped correctly and there were no issues. On the M1 memory sample, however, the files were dumped into the output directory, but every file was empty. The output for *mac_memdump* can be seen in figures 4.1 and 4.2.

```
-rw-r--r-- 1 j j 2652250112 Oct 14 16:06 FFFFFFF8007AA2420.kernel_task.dmp
-rw-r--r-- 1 j j 868507648 Oct 14 16:10 FFFFFFF8680555980.DockHelper.dmp
-rw-r--r-- 1 j j 55648256 Oct 20 14:41 FFFFFFF8680E4C000.WeatherIntents.dmp
```

Figure 4.1. Intel *mac_memdump* Output Snippet

For *mac_procdump* on the Intel sample, when process ids were specified, the proper file was output correctly and there were no issues. On the M1 memory sample, output files were created, but every file with the exception of the NordLynx process with pid 649 was empty. NordLynx is a part of the NordVPN application that was installed on the M1 system [11]. The output for *mac_procdump* can be seen in figures 4.3 and 4.4.
Table 4.1 shows which plugins had different behavior between the Intel and M1 memory sample.
Table 4.1. Volatility Plugins that Worked on Only One Sample

<table>
<thead>
<tr>
<th>Plugin Name</th>
<th>Works on Intel</th>
<th>Works on M1</th>
<th>Errors on Intel</th>
<th>Errors on M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mac_arp</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_check_mig_table</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_check_syscall_shadow</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_check_sysctl</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_check_trap_table</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_devfs</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_list_kauth_listener</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_list_kauth_scopes</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_list_raw</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_lsof</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_malfind</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_pgrp_hash_table</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_pid_hash_table</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_psaux</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_psenv</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_pslist</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_recover_filesystem</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_socket_filters</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_tasks</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_threads</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_threads_simple</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_volshell</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_yarascan</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

We will look at the output for the plugins that worked for the Intel memory sample and resulted in an error for the M1 memory sample first. These plugins will be grouped together based on the error message for the M1 sample. The `mac_check_syscall_shadow` plugin resulted in the error shown in figure 4.5. The `mac_check_sysctl` plugin resulted in a different error message on the M1 sample, and it is shown in figure 4.6.

The `mac_check_trap_table`, `mac_devfs`, and `mac_yarascan` plugins all resulted in the same error message for the M1 sample. That error message is shown by figure 4.7. The `mac_malfind`, `mac_pgrp_hash_table`, `mac_pid_hash_table`, `mac_psaux`, `mac_psenv`, `mac_pslist`, `mac_tasks`, and `mac_threads` plugins all resulted in the same error message for the M1 sample. That error message can be seen in figure 4.8.
This next group of figures focuses on the plugins that worked on the M1 sample, but resulted in an error for the Intel sample. They will also be grouped together based on the error message for the Intel sample. Figures 4.9 and 4.10 show the output of the `mac_arp`
Figure 4.8. M1 `mac_malfind` Error Message

plugin for the M1 and Intel sample, respectively, as an example of a successful execution and the corresponding error for the other memory sample. The other plugins that had the same error message as `mac_arp` on Intel when executed were: `mac_list_kauth_listener`, `mac_list_kauth_scopes`, `mac_socket_filters`, and `mac_threads_simple`. The `mac_check_mig_table` plugin had a different error on the Intel sample. That error message is shown in figure 4.11. Both the `mac_list_raw` and `mac_lsof` plugins resulted in the same error for the Intel sample. That error message can be seen in figure 4.12.

<table>
<thead>
<tr>
<th>Source IP</th>
<th>Dest. IP</th>
<th>Name</th>
<th>Sent</th>
<th>Recv</th>
<th>Time</th>
<th>Exp.</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.168.0.15</td>
<td>00:00:00:00:00:00</td>
<td>en0</td>
<td>2</td>
<td>8</td>
<td>2021-09-15 16:34:18 UTC+0000</td>
<td>1394</td>
<td>1200</td>
</tr>
<tr>
<td>192.168.0.19</td>
<td>00:00:00:00:00:00</td>
<td>en0</td>
<td>3</td>
<td>10</td>
<td>2021-09-15 16:34:15 UTC+0000</td>
<td>1427</td>
<td>1236</td>
</tr>
<tr>
<td>192.168.0.288</td>
<td>00:00:00:00:00:00</td>
<td>en0</td>
<td>31</td>
<td>0</td>
<td>2021-09-15 16:33:43 UTC+0000</td>
<td>184</td>
<td>25</td>
</tr>
<tr>
<td>192.168.0.255</td>
<td>00:00:00:00:00:00</td>
<td>en0</td>
<td>6</td>
<td>0</td>
<td>2021-09-15 16:33:43 UTC+0000</td>
<td>159</td>
<td>0</td>
</tr>
<tr>
<td>192.168.0.17</td>
<td>00:00:00:00:00:00</td>
<td>en0</td>
<td>0</td>
<td>1</td>
<td>2021-09-15 16:33:36 UTC+0000</td>
<td>1354</td>
<td>1200</td>
</tr>
<tr>
<td>192.168.0.122</td>
<td>00:00:00:00:00:00</td>
<td>en0</td>
<td>0</td>
<td>1</td>
<td>2021-09-15 16:33:34 UTC+0000</td>
<td>1350</td>
<td>1200</td>
</tr>
<tr>
<td>192.168.0.16</td>
<td>00:00:00:00:00:00</td>
<td>en0</td>
<td>23</td>
<td>27</td>
<td>2021-09-15 16:33:34 UTC+0000</td>
<td>1350</td>
<td>1200</td>
</tr>
<tr>
<td>192.168.0.1</td>
<td>00:00:00:00:00:00</td>
<td>en0</td>
<td>0</td>
<td>0</td>
<td>2021-09-15 16:33:31 UTC+0000</td>
<td>1435</td>
<td>1200</td>
</tr>
</tbody>
</table>

Figure 4.9. M1 `mac_arp` Output Snippet
Figure 4.10. Intel mac_arp Error Message

Figure 4.11. Intel mac_check_mig_table Error Message

Figure 4.12. Intel mac_lsof Error Message
Table 4.2 shows which twelve plugins resulted in the same error and which five plugins resulted in different error messages.

Table 4.2. Volatility Plugins that Resulted in Errors for Both Intel and M1 Samples

<table>
<thead>
<tr>
<th>Plugin Name</th>
<th>Error Messages were the Same</th>
<th>Error Messages were Different</th>
</tr>
</thead>
<tbody>
<tr>
<td>mac_apihooks_kernel</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>mac_check_fop</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>mac_check_syscalls</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>mac_compressed_swap</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>mac_dead_procs</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_dead.Sockets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mac_dead_vnodes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mac_dump_file</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_get_profile</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_interest_handler</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_ip_filters</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_kernel_classes</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_kevents</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mac_list_zones</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mac_lsmod_iokit</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>mac_timers</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>mac_vfsevents</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

We will examine the output of the plugins that had the same error first. The `mac_apihooks_kernel` plugin resulted in the error shown in figure 4.13 for both the Intel and M1 memory sample. There were other plugins that resulted in this same error for both samples as well. These plugins were: `mac_check_fop`, `mac_check_syscalls`, `mac_interest_handlers`, `mac_ip_filters`, `mac_kernel_classes`, `mac_lsmod_iokit`, and `mac_vfsevents`.

Figure 4.13. M1 `mac_apihooks_kernel` Error Message
When the `mac_get_profile` plugin was executed, it also resulted in an error for both samples shown in figure 4.14. The `mac_compressed_swap` error is shown in figure 4.15. The `mac_dump_file` and `mac_kevents` errors are shown in figures 4.16 and 4.17, respectively.

<table>
<thead>
<tr>
<th>Profile</th>
<th>Shift Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>: volatility.debug : Unable to find an OS X profile for the given memory sample.</td>
</tr>
</tbody>
</table>

Figure 4.14. `mac_get_profile` Error Message for Both Samples

| ERROR   | : volatility.debug : The given memory sample does not utilize compressed swap. |

Figure 4.15. `mac_compressed_swap` Error Message for Both Samples

Traceback (most recent call last):
  File "./vol.py", line 192, in <module>
    main()
  File "./vol.py", line 183, in main
    command.execute()
  File "/home/j/volatility/volatility/plugins/mac/common.py", line 48, in execute
    commands.Command.execute(self, *args, **kwargs)
  File "/home/j/volatility/volatility/commands.py", line 147, in execute
    func(outfd, data)
  File "/home/j/volatility/volatility/plugins/mac/dump_files.py", line 59, in render_text
    for vnode_off, outfile, wrote, in data:
  File "/home/j/volatility/volatility/plugins/mac/dump_files.py", line 54, in calculate
    wrote = common.write_vnode_to_file(vnode, outfile)
  File "/home/j/volatility/volatility/plugins/mac/common.py", line 224, in write_vnode_to_file
    for (offset, page) in vnode.get_contents():
  File "/home/j/volatility/volatility/plugins/overlays/mac/mac.py", line 379, in get_contents
    menq = self_v.un.vu_ubcinfo.ul_control.mac_object.menq
  File "/home/j/volatility/volatility/obj.py", line 541, in __getattr__
    return getattr(result, attr)
  File "/home/j/volatility/volatility/obj.py", line 751, in __getattr__
    return self.m(attr)
  File "/home/j/volatility/volatility/obj.py", line 733, in m
raise AttributeError("Struct {0} has no member {1}".format(self.obj_name, attr))
AttributeError: Struct ul_control has no member mac_object

Figure 4.16. `mac_dump_file` Error Message for Both Samples

The next group of figures are the plugins that had a different error message on each sample. Figures 4.18 and 4.19 show the `mac_dead_procs` plugin output for both the Intel and M1 sample, showing the different error messages that output. There were other plugins that resulted in the same error messages for the Intel and M1 samples, respectively. These plugins were: `mac_dead_sockets`, `mac_dead_vnodes`, and `mac_list_zones`.

The `mac_timers` plugin also resulted in different error messages for the Intel and M1 sample. Figures 4.20 and 4.21 show the error messages that output for both samples.
Traceback (most recent call last):
File ".vol.py", line 192, in <module>
    main()
File ".vol.py", line 183, in main
    command.execute()
File "/home/j/volatility/volatility/plugins/mac/common.py", line 48, in execute
    commands.Command.execute(self, *args, **kwargs)
File "/home/j/volatility/volatility/command.py", line 147, in execute
    func(outfd, data)
File "/home/j/volatility/volatility/plugins/mac/kevents.py", line 104, in render_text
    filt_idx = kn.kn_kevent.filter * -1
File "/home/j/volatility/volatility/obj.py", line 751, in __getattr__
    return self.m(attr)
File "/home/j/volatility/volatility/obj.py", line 733, in m
    raise AttributeError("Struct {0} has no member {1}".format(self.obj_name, attr))
AttributeError: Struct kn_kevent has no member filter

Figure 4.17. *mac.kevents* Error Message for Both Samples

Traceback (most recent call last):
File ".vol.py", line 192, in <module>
    main()
File ".vol.py", line 183, in main
    command.execute()
File "/home/j/volatility/volatility/plugins/mac/common.py", line 48, in execute
    commands.Command.execute(self, *args, **kwargs)
File "/home/j/volatility/volatility/command.py", line 147, in execute
    func(outfd, data)
File "/home/j/volatility/volatility/plugins/mac/pplist.py", line 138, in render_text
    for proc in data:
File "/home/j/volatility/volatility/plugins/mac/pplist.py", line 40, in calculate
    for zone in zones:
File "/home/j/volatility/volatility/plugins/mac/list_zones.py", line 47, in calculate
    zones = obj.json Artículo('Array', targetType="zone", vn = self.addr_space, count = 256, offset = zone_ptr)
File "/home/j/volatility/volatility/obj.py", line 171, in Object
    offset = int(offset)
TypeError: int() argument must be a string or a number, not 'NoneType'

Figure 4.18. Intel *mac.dead.procs* Error Message

Traceback (most recent call last):
File ".vol.py", line 192, in <module>
    main()
File ".vol.py", line 183, in main
    command.execute()
File "/home/j/volatility/volatility/plugins/mac/common.py", line 48, in execute
    commands.Command.execute(self, *args, **kwargs)
File "/home/j/volatility/volatility/command.py", line 147, in execute
    func(outfd, data)
File "/home/j/volatility/volatility/plugins/mac/pplist.py", line 138, in render_text
    for proc in data:
File "/home/j/volatility/volatility/plugins/mac/pplist.py", line 40, in calculate
    for zone in zones:
File "/home/j/volatility/volatility/plugins/mac/list_zones.py", line 50, in calculate
    if zone.is_valid():
File "/home/j/volatility/volatility/plugins/overlays/mac.py", line 1576, in is_valid
    return self.elem_size > 0
File "/home/j/volatility/volatility/obj.py", line 780, in __getattr__
    return self.m(attr)
File "/home/j/volatility/volatility/obj.py", line 762, in m
    raise AttributeError("Struct {0} has no member {1}".format(self.obj_name, attr))
AttributeError: Struct Array 0 has no member elem_size

Figure 4.19. M1 *mac.dead.procs* Error Message
Figure 4.20. Intel *mac_timers* Error Message

Figure 4.21. M1 *mac_timers* Error Message
4.2. Discussion

This section will discuss some of the main reasons that a large number of plugins error on the Intel and M1 memory samples. There are three main points that address these issues. The first is that because this research is the first to do a deep dive into a memory sample of the M1, which is ARM rather than Intel, there are bugs in Volatility’s ARM support for Mac that cause those plugins to fail. This is due to the fact that Volatility only supported Intel until recently and there are plugins that are still trying to access Intel specific information. Updates would have to be made to patch any major issues that arose as a result of testing.

The second is how Volatility profiles are created. Volatility uses information in Apple’s kernel debug kits in order to create profiles for the correct version of MacOS, but Apple began changing how they ship the debug information for their kernels. This means that Volatility does not always get the complete symbol and type information needed for many plugins. Volatility uses information such as using a symbol name to find a specific location in the memory sample and then plugins are able to extract that information. If that symbol is not present in the kernel debug kit and the plugin only relies on that specific symbol, the plugin will fail. Apple also no longer includes all the type information in the kernel debug kits, creating a similar issue when plugins rely only on specific type information to run. Because this change by Apple is more recent, the Mac plugins are not as resilient to this problem as the plugins for Windows and Linux, resulting in printing out the backtrace of the plugin rather than a graceful error message.

Finally, the algorithms and data structures that are used in each kernel version change as Apple releases updates. If Volatility has not updated to support that new version, it will try to use old methods to get the information plugins need and will fail. A good example that highlights the need to keep Volatility up-to-date to ensure as many plugins are working as possible is the `mac_tasks` plugin. This plugin resulted in an error on the M1 sample because it is trying to access information that is specific to Intel. As a result of
this error, a cascade effect of errors is caused for the plugins that inherit from \textit{mac\_tasks}. These plugins are: \textit{mac\_malfind}, \textit{mac\_psaux}, \textit{mac\_psenv}, \textit{mac\_threads}, \textit{mac\_volshell}, and \textit{mac\_yarascan}.

Here are more examples of specific reasons that certain plugins error. The functionality for the \textit{mac\_memdump} plugin, which exhibited odd behavior as shown in figure 4.2, has only been implemented for Intel. That is why the plugin outputs normally on the Intel sample but results in empty files for the M1 sample. The \textit{mac\_check\_syscall\_shadow} and \textit{mac\_check\_trap\_table} plugins use information that is specific to Intel and therefore result in error messages on the M1 sample when executed. The \textit{mac\_socket\_filters} plugin is an example of a plugin that relies on data structures present in the Intel kernel debug kits provided by Apple that have since been stripped out. Since the plugin is looking for something specific and has no other method of extracting the information, the plugin fails on the Intel sample.
Chapter 5. Conclusion and Future Work

5.1. Conclusion

With the introduction of Apple’s M1 SoC and the shift to the ARM instruction set, both forensics investigators and malware authors have to adapt. With the introduction of malware compiled natively for M1, it is vital that forensic tools such as Volatility are kept up to date in order to detect and mitigate any malicious activity on a system. We used all seventy-seven Mac compatible Volatility plugins on memory samples obtained from an Intel-based Mac system and a new M1 ARM-based system. After running all plugins against each sample, we were able to determine differences between each system as well as some of the major reasons for those differences. This will allow The Volatility Foundation to make any updates that are necessary to ensure that Volatility can continue to extract as much information as possible. It also establishes a baseline for memory forensics research on Apple’s M1 SoC.

5.2. Future Work

This research is intended to provide a foundation for memory forensics research using Volatility on Apple’s new M1 SoC. Due to this, the scope of the research was limited to testing Volatility’s plugins. In future research, new plugins could be written to extract new information from Apple’s M1 systems, or work towards updating the necessary plugins as Apple continues to update their new systems and introducing new SoCs such as the M1 Pro and M1 Max.

Additionally, other research could be done to examine how Apple’s kernel has changed as a result of this architecture. This would provide additional insight into how memory is allocated and structured, allowing for additional improvements to be made to Volatility and other forensics investigation tool sets.
References


Vita

Joshua Alexander Duke was born in Baton Rouge, Louisiana. He graduated with his B.S. in Computer Science, with a minor in Mathematics, from Louisiana State University in 2019. His exposure to cybersecurity during his bachelor’s degree led him to pursue his master’s degree in Computer Science at Louisiana State University the following year. He plans to receive his master’s this December 2021.