High Performance Fuzz Testing of Memory Forensics Frameworks

Arian Dokht Shahmirza
Louisiana State University and Agricultural and Mechanical College

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

Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_theses/4968

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.
HIGH PERFORMANCE FUZZ TESTING OF MEMORY FORENSICS FRAMEWORKS

A Thesis

Submitted to the Graduate Faculty of theLouisiana State University andAgricultural and Mechanical Collegein partial fulfillment of therequirements for the degree ofMaster of ScienceinThe Department of Computer Science

byArian Dokht ShahmirzaB.Sc., Shiraz University, 2011August 2019
Acknowledgments

I would like to thank my advisor, Dr. Golden G. Richard III, for his guidance and support. His bright and unique attitude toward science has given me new insights into research. I would like to thank Andrew Case for his help throughout my work. I would like to thank Dr. Konstantin Busch and Dr. Hartmut Kaiser for their contributions as my committee members. I would like to thank my friends and my colleges for their supports and encouragements. I would like to thank my husband Mehdi for always being there for me.

This work was supported by NSF grant "SaTC: CORE: Medium: Robust Memory Forensics Techniques for Userland Malware Analysis", Award # 1703683, $1,113,426.

Portions of this research were conducted with high performance computing resources provided by Louisiana State University (http://www.hpc.lsu.edu).
Table of Contents

ACKNOWLEDGMENTS ................................................................. ii

LIST OF TABLES ................................................................. iv

LIST OF FIGURES ............................................................... v

ABSTRACT ................................................................. vi

CHAPTER
1. INTRODUCTION .............................................................. 1
   1.1. Memory Forensics ...................................................... 1
   1.2. Fuzz Testing ............................................................ 2
   1.3. Contribution of This Thesis .......................................... 2
   1.4. Outline ................................................................. 3

2. FUZZ TESTING ............................................................... 4
   2.1. Background and Literature Review .................................. 4
   2.2. Fuzzing Memory Forensics Frameworks ............................. 12
   2.3. Design and Implementation ......................................... 13

3. TEST CASE GENERATION .................................................. 20
   3.1. Background and Literature Review .................................. 21
   3.2. Design and Implementation ......................................... 25

4. HIGH PERFORMANCE FUZZING ........................................... 32
   4.1. Background and Literature Review .................................. 32
   4.2. Design and Implementation ......................................... 35

5. RESULTS ................................................................. 41

6. CONCLUSIONS AND FUTURE WORK .................................... 50
   6.1. Conclusions ............................................................ 50
   6.2. Future Work ........................................................... 51

REFERENCES ................................................................. 53

VITA ................................................................. 59
## List of Tables

3.1. Custom file system overridden system calls ........................................ 26  
3.2. Mutations ................................................................. 29  
3.3. Different types for each mutation .............................................. 30  
3.4. Total number of read system calls invoked by Volatility plugins  
on a 1.07 GB memory sample .................................................. 31  
5.1. Number of tests and resulting errors for Volatility plugins .......... 42  
5.2. Mutation-read combinations causing *makevent* to timeout ........ 44  
5.3. Fuzzing stats for different plugins ....................................... 48
List of Figures

2.1. Fuzzing initialization .......................................................... 15
2.2. Volatility psscan plugin output ........................................... 17
2.3. Volatility command to output DTB ........................................ 17
4.1. High performance fuzzing architecture ................................. 39
5.1. Threads plugin crash .......................................................... 43
5.2. Atoms plugin crash ............................................................. 43
5.3. Atoms plugin crash ............................................................. 44
5.4. Large files produced by the procdump plugin ......................... 45
5.5. ldrmodule warning output .................................................... 46
5.6. Normal exit in drivermodule test ........................................ 47
5.7. Execution times and read numbers of plugins ....................... 49
5.8. Resource usage of plugins .................................................. 49
Abstract

The analysis of the volatile memory (RAM) of a computer system, known as memory forensics, is a critical component of modern digital forensics investigations. Since the evidence provided by memory forensics is vital, it is necessary for there to be automated solutions that implement the analysis. Volatility is the most widely used memory forensics framework and also contains the most functionality of all tools publicly available. Volatility, as well as all other memory forensics frameworks, are extremely complex software systems as they must parse a substantial number of in-memory data structures and their associated values. Given the reliance on memory forensics during digital investigations, robust automation of artifact extraction and presentation is required. In this study, novel methods for scalable fuzz testing were developed and the implementations of these methods were thoroughly evaluated against the Volatility framework. Fuzz testing is a technique in which a target program is intentionally fed faulty data in order to discover whether it enters an unexpected state during processing. The developed fuzzer generates thousands of mutations, each of which is specifically generated to stress test the algorithms of memory forensic frameworks. Since the developed library of mutation is so extensive, complete fuzz testing requires copious amounts of compute time and memory. To handle these requirements in a scalable and flexible manner, the developed system was designed to evenly distribute all resources, even when scaled to hundreds of compute cores. Distributed fuzzing of Volatility using the developed fuzz framework led to the discovery of many issues in Volatility’s analysis engine, including it being vulnerable to resource exhaustion attacks, silent crashes, and unhandled exceptions.
Chapter 1.

Introduction

1.1. Memory Forensics

Memory forensics is the investigation of information stored in volatile memory of a
digital device. Volatile memory of a system serves as an important piece of evidence since
it holds information that cannot be retrieved from the hard disk. Along with traditional
hard disk forensics, memory analysis provides a complete view of system state at the time
memory was acquired. This information includes encryption keys, clipboard data, volatile
registry branches, network connections, memory-only malware, and numerous other arti-
facts. To acquire volatile memory from a running system, specialized software is executed
that is capable of loading into kernel memory in order to read directly from physical mem-
ory addresses. Once a memory capture is obtained, memory forensic algorithms can then
be used to analyze the contained data and extract precise information about the system.

There are currently several commercial and non-commercial tools for memory analysis
and incident response. These tools are programmed in a variety of languages and have
different dependencies. The tools are given memory captures as inputs and generate in-
formation about the RAM contents as their outputs. With the growing size of memory
in all digital devices, the amount of data that a forensic expert needs to analyze is ever-
increasing. This is added to the data retrieved from the hard disk and other devices that
must also be investigated. Therefore, there is a need for automated forensics processes so
that investigators are provided with more refined information to examine.

Achieving automated solutions for memory forensics requires analysis frameworks to
be robust and to not require manual error checking. This is a non-trivial task as capturing
memory often results in partial corruption of acquired data or loss of memory pages. These
issues occur due to acquisition being performed in a non-atomic manner while the system
is still in use. The result of these issues is that the inputs of memory forensics frameworks
can be naturally faulty and the frameworks must carefully handle corrupt data. In order
to have tools that are resilient to corrupt input, comprehensive testing must be performed
to locate and isolate any unexpected behavior.

1.2. Fuzz Testing

Fuzz testing is an effective technique to test a program with flawed inputs and find
unexpected outputs. It can uncover issues that are difficult or impossible to find by other
tests, such as stack and heap buffer overflows and memory overwrites [1][2]. When focused
on memory forensic frameworks, fuzzing suites have historically found critical issues that
would negatively impact real investigations if triggered in the wild [3]. The goal of this
research is to create a scalable and efficient fuzzer for memory forensics frameworks that
can force the frameworks to behave unexpectedly.

Our solution is optimized in terms of both fuzzing and distribution. Fuzzing against
a large memory sample involves millions of testing states. Efficiency of such large scale
fuzzing depends on how lightweight the tests are, as well as the ability to perform tests in
parallel. We distribute the numerous tests on hundreds of cores to execute tests in parallel.

1.3. Contribution of This Thesis

In this thesis we present a high performance, lightweight, portable fuzzer that can tests
any memory forensics framework. The fuzzing technique used in this fuzzer does not depend
on the source code of the program to be tested. We have used our knowledge about memory
samples to design mutations that mimic real world memory acquisition problems. We have
used the HPX library to run hundreds of tests in parallel and achieve high performance
fuzz testing.

This fuzzer is designed with the capability of testing memory forensics tools against
large memory images efficiently. As a result, it is used for finding flaws and vulnerabilities in
memory forensics frameworks that can prevent these tools from being used in automated
forensic approaches. Findings of this fuzzer are used by developers of the programs to correct the errors and create robust forensic tools.

1.4. Outline

Chapter 2 describes fuzz testing methods and explains our approach to fuzz testing. Chapter 3 explores different methods of test case generation and the deciding factors for our mutation method and gives the details of our test case generation. Chapter 4 discusses our approach to high performance fuzz testing. Chapter 5 presents our results. Chapter 6 concludes and outlines our future work.
Chapter 2.

Fuzz Testing

2.1. Background and Literature Review

Fuzz testing is a testing method where a program is run with different random input data to find its unexpected outputs. It has become more widespread lately due to an increased demand for security testing [1]. This technique was first introduced by Barton Miller et al. at the University of Wisconsin. Miller’s students developed programs that tested common UNIX utilities with random inputs with the goal of breaking them. They could crash or hang between 25-33% of the utility programs on the seven Unix variants they tested [4].

Despite the generic nature of those tests, they proved to be very cost-effective. They enhanced the tests and went on to find security flaws in Windows NT applications, too [5]. Ever since, fuzz testing has risen to become essential in application security testing [6] and the most effective solution to finding security bugs automatically [7] [2][8] Fuzzing has become an integral part of several software companies secure development lifecycles, including Microsoft [9], Google [10] and Adobe [11].

The automation of testing a large number of varied inputs results in executing many branch conditions and therefore, covering numerous code paths. Automatic testing also has the advantage of being more time efficient in terms of developing since there is no need to write test cases for every unit of the code [12]. This is essential to software quality assurance considering the fast evolving software applications. Many security related problems or critical defects such as denial of service, degradation of service, memory management bugs, exceptions, and deadlocks can be found by fuzz testing [13].

The purpose of fuzzing is negative testing. Although it has successfully found functional errors in software [7], it is supposed to test software against unexpected or semi-valid inputs instead of the proper data expected by the processing code [2]. In fact, errors seen in
software as results of tests are not considered findings of fuzzing or significant test results as the consequence of invalid input is expected to be erroneous. On the contrary, the tests which do not result in expected errors or those which cause programs to crash are the significant test cases [7].

2.1.1. Fuzzing Methods

Fuzzers can be categorized from different aspects. With regards to generating test cases, which are inputs of the tested program, there are two types of fuzzers. Mutation based fuzzers use existing inputs and modify them to test software. Generation based fuzzing uses the analysis of the program to generate its inputs from scratch. Generation based methods are more time consuming to implement and the existing works on fuzzing mostly focuses on mutation based tests. Although generation based fuzzers achieve higher code coverage in a shorter time, their total test cases are eventually fewer than mutation based fuzzers, and the errors that they find are limited to their design, unlike mutation based fuzzers that apply random changes on every part of the input. On the other hand, since mutation based fuzzing uses existing inputs, there might be features and functionalities of the program that do not even exist in those inputs and therefore are not tested. Generation based fuzzing takes care of this issue by generating inputs according to the execution paths of the program under test [7].

With regards to the knowledge of the target software, a black box fuzzer does not have knowledge about the target program and executes it with different input files. Then based on the program’s outputs, additional mutations are selected. Black box fuzzing is fast, but it has less control over code coverage. It introduces the least overhead for testing compared to white box or gray box fuzzing, since it only needs to run the program without performing any instrumentation or taint analysis [2].

Black box fuzzing also takes minimal initialization since it does not need to preprocess the program in order to design test inputs. But a black box fuzzer’s ability to run good tests is dependent on the quality of the initial inputs provided. Also, the total number of tests
that should be performed in black box fuzzing is generally much larger than gray/white box methods. Therefore, black box fuzzing takes longer to finish one complete test. Another limitation of pure black box testing is the large number of redundant or unnecessary tests. Since the black box fuzzer does not have any knowledge about the program being tested, it cannot decide which changes to the initial input file can reach different results. For example, fuzzing network packets or fuzzing files with grammars or structures can become much more efficient if its mutations consider packet/file structure.

There are works like [14] that increase the efficiency of black box fuzzing by taking analysis of the code into consideration. These extensions to black box fuzzers are sometimes called gray box fuzzing. Although after the popularity of AFL [15], the name gray box fuzzing mostly refers to semi black box fuzzers that use outputs of previous tests to design their next tests. Extending black box fuzzers to gray box also includes coverage guided fuzzing, where the tests are designed by the code paths that are executed. As tests are still not designed according to source code of the program under test, the code coverage in coverage guided fuzzing is shown not to be increased very effectively [16]. The reason is that the fuzzer does not have any knowledge about what type of mutations achieve interesting results or which mutations can test a new execution path. It can only observe the cases where a mutation results in a change of the execution path [2, 17].

A limitation of white box fuzzing is that the constraint solver relies heavily on the program domain or the language that the program was written with. Another problem is that symbolic execution may be imprecise due to interactions with system calls and pointer aliasing problems. Thus, white box fuzzing loses one of the best features of black-box fuzzing, namely, you are actually running the program so there are no false positives. The choice between a black box fuzzer versus a white box fuzzer depends on the target program and it cannot be determined without considering the features of the program under test. There have been studies though, that show that black box fuzzing can often perform better than costly white box fuzzing [2, 18, 19].
An example of whitebox fuzzers is SAGE [17]. SAGE uses symbolic execution of an initial input to gather analysis of the program under test. It then uses the analysis and information about the program to change the input in order to change the constraints of the execution of the program. This test changes the input for different constraints of the program in order to test different execution paths. SAGE has had promising results finding security critical vulnerabilities. A potential limitation of SAGE is that its ability to generate good inputs depends on the quality of the initial input. This is the case in black box fuzzing too, and despite adding the analysis overhead, white box fuzzing still carries this issue. Although the number of mutations that are applied on a file is smaller in this approach than black box fuzzing, the difference is shown not to be significant. This is the so called path explosion problem where the number of paths in a program is very large [18, 2, 17].

Many fuzzers are a hybrid of white box and black box. Majumdar and Sen [20] presented a hybrid concolic testing algorithm that interleaves random testing with dynamic symbolic execution to complement the shortcomings of each approach with the other. The use of random search (black box) brings the ability to reach deep program states quickly, and the use of concolic testing (white box) brings the ability to explore states in a neighborhood exhaustively. Their implementation could achieve better results than both black box and white box testing alone and lead to further studies on hybrid approaches.

2.1.2. Security Testing

There has been significant research conducted on fuzzing to find security vulnerabilities. Tsankov et al. [21] performed fuzz testing on security protocols. Their black-box fuzzer on an Internet Key Exchange protocol found use-after-free memory access problems and an unhandled exception vulnerability. Rebert et al. [8] conducted a study on different seed selection strategies in mutational blackbox fuzzing. Their approach was able to find security-critical software bugs. This work was later improved by adding a process of whitebox analysis of the target program [22], which was used in optimizing the mutation ratio...
selection. The overhead of white-box fuzzing in this work is once per program-seed pair as a preprocessing step which is an improvement to former white-box fuzzing solutions.

American Fuzzy Lop is one of the most widely used fuzzers currently available [15], which has been used to find numerous bugs and vulnerabilities in many popular applications. Many fuzzers in the literature have been built on top of AFL because of its effectiveness [23]. AFL and similar fuzzers are not generally applicable to memory forensics tools, however. First, many memory forensics tools are closed source and these fuzzers require access to the source code to instrument programs for analysis. Also, AFL’s mutations are performed on the entire input file and its documentation recommends files under 1 KB in size for performance reasons. Memory dumps which are the inputs for memory forensics tools regularly exceed 1GB in size and do not meet this condition. Furthermore, AFL and the vast number of existing fuzzers target native code and are not directly applicable to many forensics tools that are written in Python. The Python solutions of AFL like python-afl [24] also require significant changes to the source code of the program under test which will not be practical for fuzzing memory forensics frameworks [3].

zzuf is a transparent application input fuzzer that can be used to find bugs for purposes of quality assurance, security, and code coverage analysis [25]. Basic Fuzzing Framework (BFF) is the name of a black box fuzzer written on top of zzuf. It has been used in previous studies on improving black-box mutational fuzzing and has proven effective in finding vulnerabilities in various programs [8, 26]. Zhao et al. [27] also use BFF to analyze the distribution of discovery probability in black-box fuzz testing and achieve a stochastic model of black-box mutational fuzzing. Their results confirm that most vulnerabilities found by fuzz testing are exploitable and also present a model for the process of fuzzing.

Prakash et al. [28] evaluated the output of memory forensics tools to find which members of structures could be mutated while still keeping the machine stable. They use fuzzing not with the purpose of finding vulnerabilities, but to evaluate the efficiency and accuracy of memory analysis tools. This study does not directly test memory forensics frameworks,
but instead the stability of an operating system to remain stable after data is mutated. Also, this approach relies on a virtual machine for mutations which makes the performance of large scale tests unacceptable for our purpose. Also, this approach works on members of data structures and does not achieve comprehensive fuzz testing.

Brubaker et al. [29] test certificate validation logic in SSL/TLS implementations. They implemented random mutations that conform to SSL/TLS to test server authentication. The tests uncovered multiple flaws in popular SSL/TLS libraries and Web browsers, including security vulnerabilities that broke server authentication guarantees and could be exploited for stealthy man-in-the-middle attacks. Newsham et al. [30] tested EnCase and The Sleuth Kit, to find security flaws with simple attack techniques. They performed fuzzing as well as manual, targeted manipulation of data formats to test these forensics tools. They could reveal multiple issues in Encase and the Sleuthkit, including infinite loops, program crashes, and memory allocation errors. Their effort focused on common errors in filesystem related parsing, which is similar to our approach that mimics different memory smears. Their mutations were performed by copying files before processing, which causes an enormous disk space overhead in large scale tests.

Dolan-Gavitt et al. [31] performed fuzz testing on individual members of Windows’ process descriptor data structures (EPROCESS) to find which mutations to these members would cause Windows to become unstable. Using this method, it is possible to generate signatures for kernel data structures that are essential to system’s stability. These signatures can then be used by memory forensics tools as their scanning signatures. As a result, when a memory analysis tool scans memory for certain signature to find hidden objects such as processes or threads, it looks for the members which are critical to the system’s stability, hence can not be manipulated by malware to go undetected. Although the end result of this work is reliability of memory forensics, its fuzz testing is performed on operating system components only and the fuzzing techniques are not useful for fuzzing the
memory forensics tools themselves. Furthermore, using a virtual machine is too costly for fuzzing forensics tools.

Driller [19] takes a new approach for finding security vulnerabilities. It leverages both fuzzing and selective concolic execution (dynamic symbolic execution) in a complementary manner, to find deeper bugs. Inexpensive fuzzing is used to exercise compartments of an application, while concolic execution is used to generate inputs which satisfy the complex checks separating the compartments. Driller uses selective concolic execution to explore only the paths deemed interesting by the fuzzer and to generate inputs for conditions that the fuzzer cannot satisfy. It was also able to find vulnerabilities by the tests. However, they find later [18] that fuzzing tools identify almost three times as many vulnerabilities as dynamic symbolic execution techniques since the costs and limitations of concolic execution are higher than expected. As a result, their studies also confirm that fuzz testing is the prominent method for finding vulnerabilities.

2.1.3. Dynamic Taint Analysis

Similar to fuzzing, dynamic taint analysis [32] is able to find conditions where programs behave erratically. In this method, the parts of the program that handled the unexpected input are marked. These marks are then used in generating next inputs for achieving tests with more code coverage. Since this analysis requires modification to the source code of any memory forensics framework except those written in C or C++, it is not applicable to many memory forensics tools, such as those written in Python. Because otherwise, for programs written in other languages the taint analysis would be performed on the interpreter rather than the program itself. Conti and Russo [33] provided a solution to taint analysis for Python via a library written entirely in Python, and thus avoiding modifications in the interpreter. However, these solutions are not applicable to fuzzing closed source memory forensics tools as they require modification to source code of the tool being tested.

Bekrar et al. [34] studied different approaches to enhance fuzzers and found that adding taint analysis to fuzzing improves the efficiency of finding exploitable bugs. Dynamic taint
analysis engines mark and test untrusted inputs, such as those coming from a user or the network. Trusted inputs are not tested in this method. Because of this design, without modification, all portions of a memory forensics framework driven by the user, such as parsing command line options, reading of environment variables and configuration files, and so on would be tested. This introduces inefficiency in the tests since the tests on portions of memory forensics frameworks that do not work on input do not yield significant results or exploitable bugs [3].

2.1.4. Gaslight

Gaslight [3], the first version of this work, implemented fuzz testing on Volatility and Rekall. It uses FUSE to apply mutations. FUSE (Filesystem in Userspace) is “an interface for userspace programs to export a filesystem to the Linux kernel” [35]. It provides an interface to the filesystem in order to mount or unmount the filesystem, read requests from the kernel, and send responses back. Gaslight tests programs written in Python, so it uses Fusepy [36]. Fusepy is a Python module that provides a simple interface to FUSE. Using Fusepy, Gaslight includes a custom FUSE filesystem that handles mutating inputs by presenting a mutated memory sample to the memory forensics framework under test, without copying the original sample.

FUSE reduces the overhead of creating custom filesystems significantly, since instead of writing kernel drivers, the FUSE interface is entirely contained within userland and filesystem operations can be implemented without kernel modifications. It supports simultaneous mounting of a memory image to efficiently apply mutations to the images dynamically. Gaslight leverages FUSE to mount the memory image that should be mutated, then modifies specific regions of the memory image according to the mutation using its custom filesystem.

Gaslight successfully found errors in a number of Volatility plugins for Linux and Mac. It also tested a Rekall plugin called arp, and it could successfully find errors in that plugin. It fully utilizes all cores of a system where it is running to make tests more efficient, but it
cannot exploit parallelism beyond a single system and as the number of fuzzing states that should be tested for even a single plugin could involve hundreds of thousands or millions of mutations, it does not scale.

Furthermore, the FUSE filesystem that is used as the custom userland filesystem in Gaslight has affected its performance in a harmful manner. Low performance in Gaslight is unacceptable due to the large number of tests that need to be executed. Using a FUSE filesystem can degrade the performance of a program by 5% up to 83%. Also optimizing FUSE does not always alleviate the overhead and in some cases leads to even lower performance [37]. It gets worse in programs with high loads like Gaslight because of the additional round trips between application programs and user filesystem [38]. There are newer frameworks like Direct-FUSE [38] which alleviate overhead of a FUSE filesystem call from crossing the user-kernel boundary but they do not have stable Python wrappers yet, and their improvements to FUSE performance are still not significant enough to solve the performance problems facing the initial version of Gaslight.

2.2. Fuzzing Memory Forensics Frameworks

The fuzzer created in this thesis is an improved version of Gaslight. It is a black box fuzzer that applies mutations dynamically without needing to have knowledge of the source code of the program being tested. Since manipulating the (potential) input of memory forensics frameworks is the most probable attack vector for malicious activity or detection avoidance, and benign memory smearing is common, we focus on finding vulnerabilities that arise as a result of a corrupt memory sample. The fuzzer in this work does not cover execution paths that would not be affected by corrupt input.

Testing memory forensics frameworks to find vulnerabilities that are linked to all execution paths (not just the ones affected by corrupt input files) is also important. It is not possible through fuzz testing though, since fuzzing refers to testing software with a large number of random inputs. Regardless, memory forensics frameworks can become
more robust if they are tested against a larger number of conditions. They can be tested in different operating systems, in different environments, or they can be run with different command line options. Although many of these tests are run regularly before these tools are published, including some of variables such as command line options or environment variables in fuzzing can be a future work for making the fuzzer more comprehensive.

### 2.3. Design and Implementation

Our black box fuzzing approach monitors the portions of the input file that are accessed by the forensics tools during a particular run, and then changes them while they are being accessed in subsequent identical runs. Gaslight is currently configured to test the Volatility framework, one of the most commonly used memory forensics frameworks, but it can be configured to test other tools since it is independent of the code or domain of the program that it tests. In this section, the design and implementation of the fuzzer to test the Volatility framework as it is currently developed is described.

Gaslight is run by the user with the following command line arguments:

- **SUT directory** is the path to the System Under Test (in this case the path to installed Volatility framework). Using this argument, the fuzzer does not depend on a specific path or environment variable to find Volatility, which is the program under test and should be run with different inputs.

- **Sample** is the memory sample file that is the input of tested program and the tests are run on it.

- **Profile** is the string identifier of the sample file’s profile. A profile is “a collection of the VTypes, overlays, and object classes for a specific operating system version and hardware architecture (x86, x64, ARM)”. Volatility needs to know the profile for a memory sample to be able to analyze it [39].
• **Output directory** is the path where the outputs and results of fuzzing should be exported.

• **Plugin** is the Volatility plugin that should be tested. This argument is optional. If no plugin is requested in the command, Gaslight iterates over a default array of plugins and tests them all.

Three arrays of the 10 most used plugins for each operating system supported by Volatility are used in the program. If the command line argument for **plugin** is empty, fuzzer checks the first three letters of the **profile** string, and sets the array of plugins that belongs to the current profile as plugins to test. It then runs its tests for each plugin. At the beginning of testing each plugin, it is necessary to know how many read file operations are made by Volatility when running that plugin on a specific sample file. This *read* number is going to be used by fuzzer as a road map for tests, so the fuzzer calls the **readcounter** module to find the total number of read system calls that are made by Volatility in a single complete execution of a specific plugin. The steps in the initialization phase of fuzzing are shown in figure 2.1..

Gaslight is a Linux application and its fuzzing modules are written in C. Tests run Volatility on Linux, but Volatility is tested against memory samples belonging to any operating system. When an application reads a file, it first gets access to the file by opening it. The programming languages that applications are written in (in this case Python) provide functions for opening files. These functions do their jobs by calling open system calls (syscalls) that are provided by operating system. Specifically, it is the execution of these system calls that result in the opening of files. Therefore, regardless of the language that an app is written in, open system calls are eventually executed when it opens a file. The same applies to reading files in Linux. Linux read functions in all languages call read system calls internally [40]. Therefore, when a program reads a file, it eventually invokes read system calls.
Figure 2.1. Fuzzing initialization

Most functions that different languages provide for reading files add several functionalities such as buffering, indexing, etc., to reading. Therefore, they usually call read system calls several times in order to perform one read. As a result, one function call in the program’s code may be compiled into many read system calls, each of which reads one part of the file [40]. Likewise, when one plugin of Volatility runs, no matter how many reads it performs on the sample file, the actual read system calls are executed multiple times.

Every Volatility plugin works differently to extract different types of information about a memory sample. Based on the information that the plugin extracts, the extent of their “reach” into a sample differs. Some plugins navigate through kernel structures to retrieve
information like the list of network connections, while there are plugins that scan large regions of memory for certain signatures. These scanning plugins perform a significantly larger number of read calls on the memory sample file. Therefore, knowing the total number of read system calls that are made by a plugin gives the fuzzer information about the activity of the plugin on a memory sample, and the fuzzer can use these individual reads to construct fine-grained tests. So, it first performs a complete run of the intended Volatility plugin on the specified sample without mutating anything in order to count the number of read system calls that the tool makes. This step needs to be done once at the start of testing each plugin against a particular memory sample.

Many Windows Volatility plugins internally call another plugin called kdbgscan first to find the debugger data block which contains pointers to the start of the active process and loaded module lists and then walk those active process and loaded module lists. Kdbgscan is one of the scanning types of plugins that scans through the sample file for the kernel debugger data block (\KDDEBUGGER\DATA64). Volatility gives the possibility of providing the kdbgscan value as a command line option when calling other plugins in order to avoid running kdbgscan repeatedly, which saves time [39]. We use this functionality to make tests more efficient. As another step in fuzzer initialization, the kdbgscan plugin is called on the specified memory sample, and the value is stored.

The same functionality is provided for the DTB (DirectoryTableBase). This value is needed for many plugins too, but if it is specified to Volatility via the command line, the plugins avoid scanning for it. Many plugins retrieve the DTB value of a sample file and print it in their output. We call the psscan plugin and get the PDB value for the system process. This is equal to the value of DTB. The output of psscan and the Volatility command to retrieve the DTB are shown in figure 2.2. and figure 2.3. respectively.

These two initialization steps are done once for the entire set of tests against one memory sample. While running the tests, kdbgscan and DTB values are provided when calling Volatility. As a result, the plugins that need these values avoid scanning for them.
and time is saved in every test on those plugins. In most cases the number of tests on each plugin is in order of hundreds of thousands, which means the time saved, hence increased efficiency, is multiplied by this number.

After initialization, fuzz testing is distributed by the distribution module. Volatility is tested under different combinations of read number and mutation and its output is monitored. That is, the mutation module runs the specified Volatility plugin and each time it mutates one of the reads that Volatility performs. After reads from 1 to the total number of reads –which was found in initialization phase– are all tested for with the first mutation, the fuzzer starts over to test Volatility against read numbers from 1 to the total number of reads for the second mutation. This process continues until all tests are exhausted. Implementation of mutations and how they are applied will be discussed in section 3.2.
Fuzzing is controlled by the read numbers that are tested. Since the total number of read calls made by a plugin is very large, it is not optimal to run the plugin for all read numbers and test each of the read numbers with all mutation states. To make this process more efficient, fuzzer is guided by three factors:

- **Exit code returned from testing on last read numbers**
  Fuzzing module starts testing read numbers in steps of 1. After each test, the exit code of Volatility is returned to fuzzing module. If the outputs of consecutive tests all result in no error, it means Volatility exited normally despite the anomaly in its input in that particular read location. So, the fuzzer increases the steps to 2. This step number can increase up to 5 in our current configuration. Whenever an exit with error is returned, fuzzing module decreases the steps according to the error code.

- **Duration of running time of Volatility**
  If the Volatility plugin takes too long to execute, then the mutation in that particular read instance could result in resource exhaustion or an infinite loop. Gaslight has a timeout, which is currently 90 minutes in the current configuration, for each test. If a Volatility plugin takes longer than the timeout, the test is interrupted and execution continues with the next tests. The value of steps in read numbers is decreased so that the execution paths close to the one that was triggered by reading that portion of input are tested.

- **Size of output (dump) file created by Volatility**
  In testing Volatility plugins that create dump files as their outputs, the sizes of dump files are inspected by the fuzzer. If a dump is too large, it means the mutation in that read number caused file system resource exhaustion. In case of creation of large files, since a portion of input file has been found that yields interesting results, the fuzzer decreases its step. Consequently, the next tests are run on read numbers that are closer to the last portion in order to prevent missing other special cases that
might result from mutations in that area. The threshold for dump files in our current configuration is 100 MB.

Since our coverage is guided by exit codes, the fuzzer might skip testing some of the read numbers after an execution with a normal exit. But there are cases where the program under test exits normally despite having an unexpected behavior. These *silent crashes* are detected in the analysis phase and they will be discussed in section 5. Gaslight can be categorized as a black box fuzzer, but its fuzzing is not carried out completely blindly. Its approach is feedback guided fuzzing which is sometimes regarded as gray box fuzzing. The controlling factors for guided fuzzing are generic features that can be used for any program. Therefore, although our current implementation tests Volatility, this fuzzer can be used to test other forensic tools or tools of other categories that work with input files too.
Chapter 3.

Test Case Generation

Generating high-impact mutation sets is very important. To ensure that our fuzzer properly tests memory analysis frameworks, we have designed mutations that are tailored to expose issues commonly encountered in memory acquisition. Investigating a system requires inspection of its running state, i.e. the contents of its RAM. The importance of information that can be revealed from RAM are discussed in section 1.1. In order to perform analysis on the RAM of a computer, the contents of RAM need to be stored on a non-volatile storage. The process of capturing the memory of a running system and creating a dump file out of it is called memory acquisition. This is an important and precarious step in the memory forensics process [39].

Input files analyzed by memory forensics tools are memory samples that are produced by performing memory acquisition. These samples are captured from running systems that should be investigated. Since the system is running as acquisition of volatile memory is performed, the contents of RAM change as it is being acquired. As a result, there might be natural inconsistencies in the captured sample. If the computer is compromised, memory acquisition is even more challenging. The acquisition procedure should not rely on compromised operating systems or unreliable software running on a system. Malware can freely tamper with in memory data and change it. So, the quality of memory acquisition can have a significant impact on memory forensics.

Fuzzer needs to test not only general vulnerabilities in memory forensics tools, but also their robustness against corrupt acquired memory images. Therefore, mutations are designed not only to cover the code paths that process input files, but also to test code against memory samples that are compromised in similar ways as acquired memory.
3.1. Background and Literature Review

There has been an abundant of research on effective memory acquisition on different operating systems. Many operating systems provide access to physical memory. In Linux, /dev/mem holds the image of the main memory of the computer. Byte addresses in /dev/mem are interpreted as physical memory addresses. Since Linux 2.6.26, access to memory regions through /dev/mem has changed. For example, RAM access is not allowed on x86 but accessing memory-mapped PCI regions is. The file /dev/kmem contains the kernel virtual memory rather than physical memory. Since Linux 2.6.26, this file is available only if the CONFIG_DEVKMEM kernel configuration option is enabled [41].

Acquisition procedures provided by operating systems are convenient and easy to use, but they do not satisfy reliability and usefulness requirements. These acquisitions rely on the local operating system to supply the memory contents despite the ability of malware to freely tamper with in memory data and manipulate the memory image. Also, modern malware is no longer limited to the userland of an operating system and kernel level rootkits now impact the reliability of acquisitions performed by kernel. These acquisition procedures depend on correct functioning of operating systems that might be compromised [42]. Acquisition procedures also require the memory capture process to run. Running the capture process might need running other programs like shell or network transport. This results in changing the contents of memory and changing memory allocations [43].

Carrier et. al [43] propose a procedure for acquiring volatile memory to address the shortcomings of said procedures using a hardware expansion card that can copy memory to an external storage device. Although their hardware card needs to be installed into a PCI bus slot before an incident occurs, this solution can be more reliable than a software-based solution as it does not rely on software or operating system to acquire memory. Their work also highlights the issues that occur in memory acquisition.

A commonly used Linux memory acquisition tool is fmem [44]. fmem is a loadable kernel module that creates a /dev/fmem device supporting memory capture. It supports
multiple architectures such as ARM, i386, and x86_64. Schatz [42] proposed an approach to snatch control of the host hardware from the running operating system. The approach named bodysnatcher first injects an independent, task specific minimalist OS into the kernel space of the running kernel. It then saves the state of the running operating system, and then boots the acquisition OS in an independent and restricted subset of the machines memory. The acquisition OS then employs a small and common subset of the hosts hardware as an output channel for dumping an image of the physical memory of the host. Although this approach addresses some acquisition problems in theory, it requires completely new operating systems. It cannot solve problems of acquisition on current operating systems.

Goyal et al. [45] proposed kdump to tackle the acquisition problem of corrupted memory due to software. Kdump is based on kexec. Kexec is a system call that enables loading and booting into another kernel from the currently running kernel. It runs the boot loader from within the kernel without the need for hardware initialization that is done by the BIOS or firmware during a normal boot. Therefore, it is capable of booting a new kernel inside a kernel with a faster speed than a normal boot [46]. Kdump modified kexec to enable it to boot a new kernel even in an event of a system crash or panic. It boots the capture kernel from a reserved region of memory rather than a default location. Therefore, there is a smaller chance that the captured memory is corrupted by direct memory access in host kernel [45].

Much recent work on memory acquisition is focused on smartphones. Sylve et al. [47] discuss memory acquisition on Android and present a kernel module for dumping memory named dmd. dmd parses the kernels iomem_resource structure first to find the physical memory address ranges of system RAM. It then performs physical to virtual address translation for each page of memory, and finally reads all pages in each range and writes them to either a file (typically on the devices SD card) or a TCP socket.

sun et al. [48] designed a hardware-assisted memory acquisition mechanism named TrustDump to acquire the RAM and CPU registers of the OS on smartphones, even if the
OS has crashed or has been compromised. It divides the mobile platform into two isolated execution environments, normal domain and secure domain. The OS running in the normal domain is usually called Rich OS, and the one running in the secure domain is called Secure OS. Its memory acquisition module called TrustDumper is installed in the secure domain to perform memory dump and malware analysis of the Rich OS.

Case and Richard [49] explored page swapping and demand paging as obstacles to complete acquisition of memory. Page swapping is operating system’s solution to use more memory than is available in RAM. It stores least accessed memory pages in disk storage. It results in RAM not including all the contents of the actual memory of the system. Demand paging is when the operating system does not load data from files on RAM until they are explicitly needed by a read or write operation on the portion of file where they reside. Using this method by the operating system results in RAM including parts of files or processes rather than their complete layouts.

The Computer Forensic Tool Testing (CFTT) program at the National Institute of Standards and Technology (NIST) developed specifications for disk imaging tools [50]. The specifications define requirements for digital media acquisition tools used in computer forensics investigations [51], and test assertions and a test methodology based on these requirements [52]. These tool specifications and test requirements became bases for many forensic disk acquisition products. Although specifications for memory acquisition are to be published by NIST too [53], some memory acquisition solutions use similar standards for memory imaging [43].

3.1.1. Page Smearing

Libster and Kornblum [54] proposed an approach to integrate the memory imaging software in the operating system to achieve a more reliable memory snapshot. One of their stated problems in memory acquisition that they attempted to tackle was getting a fuzzy snapshot state of the system. They explained that when memory captures takes a long time, as other processes are modifying memory while the capture is proceeding, the
captured data becomes inconsistent. It can even result in inadvertently crashing the system when reading certain address ranges [54]. This problem was then observed and studied in other research works [55, 42, 56].

Cohen [57] discussed that pagetable smear can have much larger consequences to memory analysis than kernel or process memory page smear. He found a new type of smear that he called pagetable smear which depended on the order of acquisition. He found that since the page tables are usually located at low memory addresses, these page tables will be the first to be acquired from memory and written to an image. But as the acquisition runs, the kernels working set trimmer might decide to trim the processes working set resulting in the change in processes’ page table entries. So, by the time the acquisition tool gets to higher addresses pointed by page tables, those pages are re-purposed. As a result, memory forensics tools might interpret invalid pages pointed to by page table entries without detecting that those pages no longer belong to the process working set.

Case and Richard [49] explore current issues and research challenges in memory forensics. They discuss problems and the path forward in areas of memory acquisition and memory analysis. They call page smearing one of the most pressing issues and discuss it. Page smearing is said to be commonly encountered on systems that have 8 gigabytes or more of RAM installed as well as systems that are under heavy load. There are not any solutions to automatically detect smear in captured memory. The current workaround used is leveraging hypervisor capabilities for taking memory snapshots. But this is not practical in real world operating systems that do not run on virtual machines and need to be investigated.

Gruhn and Freiling [56] evaluated 12 memory forensics acquisition tools and methods and compared them in terms of atomicity and integrity. Their study reveals different issues that exist in memory acquisition performed by different tools. They report observations of smear that happened in acquisition processes of several tools such as FTK Imager, DumpIt, win64dd, and WinPmem which are kernel level software acquisition tools [56]. Smear was
an issue in acquisition procedures that took longer as well. In the current era of increasing performance needs, the size of RAM installed on systems is increasing. This leads to an inevitable increase in the duration of memory capture even by fast acquisition tools. As a result, page smearing becomes even more common in acquired memory images of current systems with larger RAM sizes.

3.2. Design and Implementation

Mutations are designed so that they can reveal forensics tools’ vulnerabilities against corrupt input. The corruption can happen in different ways as detailed in the previous section. Also, applying mutations should not need copying the original memory samples, as it would otherwise lead to huge resource consumption making the process impractical. In this section, our design of mutations and method of applying them is explained.

Our mutations are applied by leveraging the functionality of intercepting system calls that Linux provides. As explained in section 2.3., all file operations in Linux are handled through file system calls. So if we intercept system calls, we can have control over all file access in a program. LD_PRELOAD is an environment variable in Linux. If you set LD_PRELOAD to any shared object library, that library will be loaded before all other libraries, even the C runtime library (libc). So, any system call to file system can be overridden and intercepted by userspace programs.

Using this environment variable, we have created a custom file system. This file system is in userspace and does not need any privilege escalations. The custom file system consists of overloaded system calls. The list of overloaded system calls are presented in table 3.1.. LD_PRELOAD is set to the custom file system library that overrides these calls before Volatility is called for testing. So, the custom file system is loaded before any other system library, and therefore the overloaded system calls in this library are invoked when any of these read or open system calls are called.
Table 3.1. Custom file system overridden system calls

<table>
<thead>
<tr>
<th>Read</th>
<th>Open Stream</th>
<th>Open File</th>
<th>Close File</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>fopen</td>
<td>open</td>
<td>close</td>
</tr>
<tr>
<td>pread</td>
<td>fopen64</td>
<td>open64</td>
<td>fclose</td>
</tr>
<tr>
<td>pread64</td>
<td>freopen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fread</td>
<td>freopen64</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

By preloading our custom file system library, its overloaded functions are called instead of all read and open calls that are made by the program under test. This means all the read and open calls that the program makes on files other than memory samples are intercepted too. The fuzzer avoids intercepting access to any file that is located outside its testpath. The directory where input files of the program under test are stored, is passed to the preloaded library by an environment variable. The fuzzer sets this environment variable to the directory of input files. When an overloaded function is called, it checks the path of the file that this function is called on. If the file resides in a directory other than testpath, the original system call is invoked and its result is returned. But if the file is in the specified directory, the fuzzer monitors and intercepts all of these system calls made and performs mutations through these functions.

In order to find original functions, dlsym() is used. dlsym can find the address of a given symbol in a shared object or executable. RTLD_NEXT finds the next occurrence of the function in the search order after the current library. Fuzzer calls dlsym with the name of the function and glibc library and uses the returned address to call original functions. The original functions are needed not only for files that should not be mutated, but also for the tested file itself. All the overridden functions first execute the original call to get the values needed by the program under test, and then take required actions.

At the first step of tests, readcounter module obtains the total number of read calls that the Volatility plugin under test makes on the tested file. The fuzzer preloads readcounter library and runs the specific Volatility plugin once. The library monitors opened files in
test directory and increments a counter each time a read call is performed on each file. After there are no more open instances of the file, it writes the total number of reads in an output file. This is the reason for overloading close file functions. close and fclose are intercepted only in the readnumber module to find when the program under test is done with a file and the total read number is final.

Create file system calls are not intercepted by the fuzzer since they are not of any use for fuzzing purposes. Memory forensics frameworks which are the subjects of our tests analyze existing memory samples. Fuzzing tests whether those frameworks behave unexpectedly given a corrupt input. Therefore, access to created files by these frameworks can be of no use to the fuzzer. The fuzzer does check the size of memory dumps created by forensics frameworks however. But doing that using interception of create function would be inefficient. In order to check the sizes of dump files created by Volatility, the fuzzer reads the output of Volatility after it is done creating dumps. The output includes the names of dump files created. The fuzzer then reads the sizes of those files and acts accordingly if they are not in the ordinary size range. Using this method, the fuzzer deals with the dump files only after Volatility is done running, and it does not need to do any processing other than reading their names from the text output and getting their sizes.

Another open function not overridden by the fuzzer is *fdopen*. *fdopen* returns a new stream for an already open file. The fuzzer does not need to know how many streams are created for a file. It has access to read bytes of an open file by intercepting read system calls and does not need streams to access a file. The streams also do not make a difference as long as the fuzzer knows if the file is opened or closed. So monitoring the streams on a file that is already open will not add any functionalities.

After retrieving the total number of reads that take place in one full run of a Volatility plugin, the fuzzer starts the testing process. It loops through different combinations of read numbers and mutations. In each loop, it sets two specific environment variables to these values, then preloads the custom file system module and executes the Volatility
plugin. Consequently, its overloaded read and open functions are called whenever Volatility performs an open or read. The fuzzer keeps track of opened files in the specified test path. A read counter stores the number of reads that Volatility performs while running. When this counter reaches the number in the specific environment variable, it mutates the data which was read at that particular read.

Overloaded read functions call the original function first to get the buffer that is the actual result of reading the memory sample. If the current read counter is not equal to the value of the specific environment variable, resulting buffer from the original function call is returned. Otherwise, the buffer is mutated first and the mutated result is returned. As a result, Volatility calls its intended read function. But instead of getting the actual buffer of the memory sample, a mutated version of the buffer is returned to it.

As described earlier, memory smearing occurs when an acquisition tool fails to acquire a page correctly and instead reads random data or data from the wrong offset in physical memory. To robustly test memory analysis tools, we introduce smear by leveraging our knowledge of the location of key data structures in the sample and then mutating them based on the specific data type.

We have designed 38 bit level mutations shown in table 3.2. These mutations are similar to different benign or malicious memory corruptions and memory smears. They change bytes of the read data returned from original read system calls. One mutation zeros out all bytes in that particular buffer returned by read function. Another one sets all bytes to 0xFF. 4 mutations add the current bytes in buffer to 2, 4, 8, 128, 4096 values. 4 mutations subtract 2, 4, 8, 128, 4096 from buffer values. For each of the above operations, there are mutations that make those changes on each boundary of 2, 4, 8 and 128 bytes of returned buffer. 4 mutations change bytes on each boundary of 2, 4, 8, and 128 bytes to random values.

There are two options for each mutation. These options and the bytes that they affect are shown in table 3.3.
<table>
<thead>
<tr>
<th>all_zero</th>
<th>mutate_every_128_all_random</th>
<th>mutate_current_plus_128_fill</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_ffs</td>
<td>mutate_every_2_all_random_fill</td>
<td>mutate_current_plus_4096_fill</td>
</tr>
<tr>
<td>mutate_every_2_all_zero</td>
<td>mutate_every_4_all_random_fill</td>
<td>mutate_current_sub_2_fill</td>
</tr>
<tr>
<td>mutate_every_4_all_zero</td>
<td>mutate_every_8_all_random_fill</td>
<td>mutate_current_sub_4_fill</td>
</tr>
<tr>
<td>mutate_every_8_all_zero</td>
<td>mutate_every_128_all_random_fill</td>
<td>mutate_current_sub_8_fill</td>
</tr>
<tr>
<td>mutate_every_128_all_zero</td>
<td>mutate_every_2_current_plus_2</td>
<td>mutate_current_sub_128_fill</td>
</tr>
<tr>
<td>mutate_every_2_all_ffs</td>
<td>mutate_every_2_current_plus_4</td>
<td>mutate_current_sub_4096_fill</td>
</tr>
<tr>
<td>mutate_every_4_all_ffs</td>
<td>mutate_every_2_current_plus_8</td>
<td>mutate_every_2_current_sub_2</td>
</tr>
<tr>
<td>mutate_every_8_all_ffs</td>
<td>mutate_every_2_current_plus_128</td>
<td>mutate_every_2_current_sub_4</td>
</tr>
<tr>
<td>mutate_every_128_all_ffs</td>
<td>mutate_every_2_current_plus_4096</td>
<td>mutate_every_2_current_sub_8</td>
</tr>
<tr>
<td>mutate_every_2_all_random</td>
<td>mutate_current_plus_2_fill</td>
<td>mutate_every_2_current_sub_128</td>
</tr>
<tr>
<td>mutate_every_4_all_random</td>
<td>mutate_current_plus_4_fill</td>
<td>mutate_every_2_current_sub_4096</td>
</tr>
<tr>
<td>mutate_every_8_all_random</td>
<td>mutate_current_plus_8_fill</td>
<td></td>
</tr>
</tbody>
</table>

- If `fuzz_past` is set, the specified mutation is applied on bytes read at the particular read counter and all further reads. So, it affects the bytes returned from the read system call at the particular read counter, as well as bytes returned from all read system calls that are invoked afterwards.

- if `consistent` is set, when read counter reaches the particular read number, the fuzzer stores the current cursor position of the sample file and mutates the read buffer. Then in all later reads, the fuzzer checks the position of the cursor in the file and mutates the read buffer if it is read from the same position. This option is created to mimic static file manipulation. Volatility might read a specific chunk of file multiple times, and if the sample was actually corrupt all these reads would result in the same value. Therefore, this `consistent` option is implemented to prevent conflicts and possible disruption of analysis.
Since the number of read system calls that execute on a sample file is generally larger than the actual read calls that are made on a high level language, intercepting system calls gives Gaslight the advantage of lightweight precision. Instead of mutating specific parts of a large file that would normally happen in the absence of intercepting system calls, the fuzzer mutates a specific number of smaller chunks that are returned from read system calls. Not only mutating these smaller chunks is less costly, but also the division of files into different parts is readily available for the fuzzer for creating diverse mutations. The total number of read calls made by Volatility while running some of its plugins on a 1.07 GB memory sample is shown in table 3.4.

It is worth noting that the input files of the program under test are not opened by the fuzzer separately. In other words, the input file instances only exist in the tested program and they are loaded on memory as the program loads them. The fuzzer gets access to the already opened sample files by intercepting system calls without reloading them. Thus, our method manipulates the data that is already on RAM and does not load any additional data on memory.

Furthermore, the source code of the program tested is not manipulated and it is not passed any different parameters when being tested. Execution of the program is done in the normal way and the chunks of file that it reads are dynamically manipulated. Also, the parts of file that are not read by the tested application are not mutated since no read function is called on them. Therefore, despite the low level architecture of the fuzzer that has no knowledge about the source code of the program being tested, it has a high grip
Table 3.4. Total number of read system calls invoked by Volatility plugins on a 1.07 GB memory sample

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>atoms</td>
<td>28490</td>
</tr>
<tr>
<td>drivermodule</td>
<td>15746</td>
</tr>
<tr>
<td>impscan</td>
<td>3266</td>
</tr>
<tr>
<td>threads</td>
<td>72216</td>
</tr>
<tr>
<td>procdump</td>
<td>121020</td>
</tr>
<tr>
<td>sockscan</td>
<td>541</td>
</tr>
<tr>
<td>pslist</td>
<td>7677</td>
</tr>
<tr>
<td>dlllist</td>
<td>271319</td>
</tr>
<tr>
<td>ldrmodule</td>
<td>806512</td>
</tr>
<tr>
<td>timers</td>
<td>87851</td>
</tr>
<tr>
<td>deskscan</td>
<td>10923</td>
</tr>
</tbody>
</table>

over testing different execution paths with high precision. Especially with programs like memory forensics frameworks that consume large memory samples as inputs, this fuzzing technique saves greatly on memory and overhead by avoiding duplication of the file.
Chapter 4.

High Performance Fuzzing

Fuzzing memory forensics frameworks involves running the program under test repeatedly and applying mutations on different read calls. The number of read system calls invoked by an average plugin on a 1.07 GB memory sample is 167469. Plugins should be tested by mutating each of these read calls. Since our fuzzing algorithm skips reads that are unlikely to lead to significant findings, the number of tested read calls on average is 55820. These instances are tested against 128 mutation states, which requires running Volatility 7144960 times to cover all conditions. Running these many tests is impossible without an efficient parallel solution.

Scalability is an important factor in the performance of a fuzzer. Our fuzzer needs to achieve the highest level of parallel execution while maintaining reliability with minimal overhead. Running tests with the best time and energy efficiency highly depends on the approach to distributed and parallel computation. The appropriate method corresponds to program’s features and conditions. The tests that the fuzzer needs to run in parallel involve running memory forensics frameworks that are written in different languages. These tools get access to files on disk, create new files, and write on disk several times during a single execution. Furthermore, these tools have a diverse set of functionalities each of which performs totally different operations. Therefore, there are significant disparities in the processing load, file system access, and memory load of various individual executions. In this chapter, we explore different distribution methods and explain our approach to high performance fuzzing.

4.1. Background and Literature Review

The growing demand for high performance computing has lead to a significant advancement in this field. As programs become more computation heavy and data intensive, their large scale computing solutions evolve and become more specialized for different function-
alities. Currently, there exist a large number of high level programming paradigms that can be utilized in different parallel applications. A basic point of difference in programming paradigms is the memory access policies of multiprocessors. They fall into two groups of shared-memory and message passing paradigms. In shared-memory model, all processors work with a common shared memory, and this shared memory is the means for all communication among processors. While in message passing, all communication among processors is handled by transferring send and receive commands [58].

Message passing is the most common programming paradigm used in parallel systems. High level programming paradigms that are based on message passing can help make efficient parallel programs. A popular message passing library which is also an environment is Parallel Virtual Machine (PVM) [59]. PVM can be used to run parallel applications on systems ranging from high-end supercomputers to clusters of workstations. The most popular high level message-passing system for scientific and engineering application is Message Passing Interface (MPI) [60]. It is a specification for message passing, designed to be standard for distributed memory parallel computing using explicit message passing. This interface attempts to establish a practical, portable, efficient, and flexible standard for message passing [61].

Different implementations of the MPI standard have been in use for decades. MPICH [62] is a high-performance and widely portable implementation of MPI. MPICH contains many implementations of device specific functions called Abstract Device Interfaces (ADI) to achieve high portability and ease of implementation. In general, message passing architectures provide simpler design than shared memory models, while their programming is more complicated. MPICH uses ADIs to isolate device dependent specifications from other parts of MPI implementation that can be shared among many devices [62].

Another implementation of MPI which is open source is Open MPI [63]. It uses a component architecture called the Modular Component Architecture (MCA) which is flexible and enables convenient extending of the implementation. It also has improved point-to-
point communications performance for a wide variety of interconnects. In addition, there are MPI implementations created by different vendors such as HP [64], Microsoft [65], and Intel [66] which provide MPI libraries for Microsoft Azure, Windows, and Linux.

Since these existing approaches do not satisfy present-day performance and scalability needs [67, 68], other parallel programming models like OpenMP [69], which is a shared memory programming model, have become more common. There have also emerged new language extensions like OpenCL [70] which is an open, royalty-free standard for cross-platform, parallel programming of diverse processors.

4.1.1. Parallex Execution Model

Improvement in computing technology is not limited to software. Semiconductors and computer hardware has evolved in the past decades besides software solutions. Kaiser et al. [71] proposed a new execution model to address the new challenges introduced by the evolving technology, and to fully benefit from it. This execution model called Parallex enables a new computing dynamic through the application of message-driven computation in a global address space context with lightweight synchronization to address critical bottlenecks of starvation, overhead, latency, and contention in effective use of new generation HPC systems.

Parallex dynamically schedules multiple threads using a message-driven work-queue methodology of applying user tasks to physical processing resources. This results in the advantage of efficiency and latency hiding over message passing paradigm as it separates the work from the resources and processors continue to do useful work even in the presence of remote service requests and data accesses. Parallex tackles the problem of load balancing by a fully dynamic adaptive resource management enabled by Active Global Address Space (AGAS). It handles parallel processes differently than conventional practices. It allows application modules to be defined with a shared name space and to exploit many layers of parallelism within the same context. It also takes advantage of Local Control Objects to organize flow control and achieve variable granularity. Parallex’s capability of variable
granularity results in scalability for many applications. Since fine grained granularity does not always result in better performance [72, 71].

4.1.2. High Performance Parallex

HPX (High Performance Parallex) is a runtime system implementation of the ParalleX execution model. It is an open source runtime system that delivers the mechanisms required to support the parallel execution, synchronization, resource allocation, and name space management. It supports static pre-binding at link time while having the capability of dynamically extending the functionality of application specific modules. Its modular architecture allows for easy compile time customization, minimizing the runtime footprint. It strictly adheres to Standard-C++ [73] providing a homogeneous API and utilizes Boost [74], enabling it to combine powerful compile time optimization techniques and optimal code generation with excellent portability [72].

HPX has shown considerable improvement in scalability and efficiency compared to conventional alternatives [75, 76]. When testing weak scaling of an application from 1 node to 1024 nodes (each 16 cores), HPX outperformed the equivalent MPI code by a factor of 1.4. Unlike other methods that have several bottlenecks due to implicit and explicit global barriers to parallel code, HPX unifies remote and local operations, which results in eliminating almost all serial portions of code. Its per-task overhead is very small and it does not grow with an increase in the number of cores [75].

4.2. Design and Implementation

We use HPX runtime system for scaling and parallelizing our fuzzer’s tests. Since various tests that the fuzzer runs do not have equal loads, the load balancing ability of the distribution framework plays an important part in the scalability of our fuzzer. HPX provides An Active Global Address Space (AGAS) that supports load balancing through object migration. In our tests, the difference in loads is due to the difference in operations of plugins’ analysis. Our current tested tool, Volatility, has about 117 Windows plugins, 69
Linux plugins and 71 Mac plugins. Each plugin reads certain information from a memory sample and performs certain operations on fetched data. Therefore, running different plugins imposes completely different workloads.

The computationally intensive part in each test is running Volatility plugins. The plugin needs to run once for every test. Volatility is written in python and each plugin takes up a certain amount of workload to finish. We parallelize the execution of plugins using HPX. After the initialization phase, fuzzer gets the number of HPX localities assigned to the current job, and divides the tests among localities. In HPX, a locality is a contiguous physical domain, managing intra-locality latencies, while guaranteeing compound atomic operations on local state. Different localities may expose entirely different temporal locality properties. It is equivalent to a node in a cluster [72]. Dividing all tests among localities at the beginning, allows the fuzzer to avoid starting too many tasks and overflowing the head locality with tasks.

Divided tasks are wrapped in HPX futures. A future refers to an object that acts as a proxy for a result that is initially not known, usually because the computation of its value has not yet completed. The future synchronizes the access to this value by optionally suspending the requesting thread until the value is available. This allows the computation to proceed unblocked until the actual value is required to produce a result. Futures also permit anonymous producer-consumer computation when neither the producer of a value, nor its consumer are known at compile time. In addition, the future construct allows a trade-off between eager and lazy evaluation by postponing the calculation of a value until it is actually required [72]. For each locality assigned to the fuzzer’s job, one future is created and the read numbers that are allocated to that locality are passed along to its future. Creating a future for each set of tests, takes care of transferring every set to one locality, performing the jobs on that locality, while keeping the future’s status available in the head locality. The fuzzer takes advantage of this feature to keep track of assigned localities.
After dividing tasks among localities, i.e. nodes, the job is distributed in all nodes rather than executing on the head node only. On each locality, the number of processors is fetched using HPX and the read numbers that were assigned to that locality are further divided among all processors. Again, to keep track of the jobs running on each processor and to know when they are finished, HPX futures are used. One future per processor, hence one thread per processor, is created, and the read numbers that need to be tested by that thread are passed along to the future. Then each thread is in charge of testing the specified read numbers.

In this step, the fuzzer could have taken two other approaches which would not yield the current efficiency. One is handling the tests in locality level rather than thread level. Meaning, running as many tests as the number of cores at a time on a locality, and waiting for them to finish in a loop until all read numbers are tested. This can seem as an intuitive solution since in the locality level, HPX can handle distributing the jobs on cores without explicit directives, and the main point of parallelization is distributing the jobs on nodes with proper communication. This aspect of high performance testing has already been handled in the above step completely. So, regardless of the method of handling threads, fuzzing tests would run on all nodes simultaneously hence speeding up testing. But if this approach was taken, every thread would have to finish before the next tests on all threads could execute. If one of the cores was done with its test and was free for the next, it would have to wait for all other threads in the same locality to finish their tests. Therefore, this would be an inefficient approach that does not make use of available resources properly.

Another method that could come to mind is creating as many futures as required reads per node (usually in order of hundreds of thousands) on each locality and have HPX handle running all of those futures and return the results. Again in this case since the distribution on all nodes have been already achieved, and as HPX handles threads on a locality automatically, parallelization could seem complete. But this is also an inefficient
approach that can even cause nodes to go offline. The reason is, creating each future spawns a new thread, causing the overflow of threads on the operating system.

the number of futures created on each locality is equal to the number of cores (threads returned by HPX) of that locality, because running more threads than the number of processors would only result in worse performance. The maximum number of threads that a node runs concurrently is equal to the number of its cores. So, increasing the number of futures, hence threads, will only cause the operating system to schedule multiple threads on the same core and switch between them in order to run them all. This slows down the cores since the overhead of scheduling is also added to the processor’s workload. Therefore, the optimum performance is achieved when there is one thread running on each processor of a locality.

After all cores of all nodes are designated a certain range of read numbers to test independently, each core starts the fuzzing process for designated read numbers combined with all mutations. The fuzzer module is called using HPX Process, and the required read and mutation that should be tested are passed to it via this HPX component. Among the parameters passed to the fuzzer module, is the path to log file. The path name of log file for each thread is made up of the name of tested plugin, locality ID, and thread ID. Therefore, although all localities and threads on the cluster have shared access to log files directory, they do not write their logs to the same file. This prevents any distortion in the output that can be caused by two threads writing to a file at the same time. It also makes analyzing the outputs and tracking logs much easier. On the other hand, the memory sample file that memory forensics frameworks running on all threads read and analyze, is a single file in a single directory. Accessing the same file for opening and reading by several threads on different threads and localities does not pose any harm and therefore, there is only one sample file for all the tests that mutate it.

The Process component provides the ability to know when the execution of an outside process (fuzzer module in this case) is complete. This feature is provided by wait_for_exit
function, which blocks the caller thread until the called process has finished executing. HPX also provides wait functions for futures, so that the completion of futures can be visible to the caller. There are wait functions that wait for one future to finish and return, and there are wait functions that wait for a set of futures and return after all futures of the set have completed. Wait functions are either callback or non-callback. Callback wait functions take an extra argument that specifies a method to be called when each future returns. The fuzzer in this work uses wait functions to block for all threads in each locality to finish, and to block until all localities are done with their designated tests. Figure 4.1 shows the current architecture of our distributed fuzzer.

![Figure 4.1. High performance fuzzing architecture](image)

Our current environment is a homogeneous system where all processors that we use are the same. Nevertheless, 360 of the compute nodes of the supercomputer that we are currently using have 2 Intel Xeon Phi 7120P coprocessors. 20 of these compute nodes have
1 Intel Xeon Phi 7120P coprocessor and 1 NVIDIA Tesla K20X. These coprocessors can be taken advantage of using HPX especially since it is capable of handling portability of heterogeneous systems very well.
Chapter 5.

Results

Results of this research are presented in terms of performance of fuzzing and the errors that the fuzzer has discovered. So far we have tested 11 Windows plugins, 2 mac plugins and 1 Linux plugin of Volatility framework. Because of the difference in the analysis done by different plugins, their operations and hence the numbers of read calls made by them are different. Our tests are performed on read system calls of the program under test. So, the final number of tests that we run on each plugin depends on the number of read calls performed by that plugin.

A total number of 2858752 tests were run on 14 plugins. Out of these executions, 88325 in total crashed with errors, 454 resulted in timeout, 128 resulted in creation of too large output files, and 7 resulted in too long output results. There were also numerous cases of anomalous outputs with normal exit codes almost all of which were not significant since Volatility detected the wrong input and exited just without producing an error. A summary of the number of tests and errors and their results is given in table 5.1.

These results are fetched automatically after tests from log files. The fuzzer saves the output printed by Volatility, the exit code, read number and mutation of each test in log files. These log files then go through text analysis and the errors and stats of tests are retrieved from the analysis. After getting all the errors that occurred, a series of semi-manual analyses are performed on the outputs to find anomalous cases that did not crash Volatility, but may show strange activity.

The exit with error cases include errors thrown by Volatility as a result of detecting corrupt input, and unhandled exceptions in Volatility. The first type of errors has the desired behavior and is in fact the most accurate output for corrupt files. An example of these cases is shown in figure 5.1, which occurred by applying the 1st mutation (change all bytes to 0x00) on 16478th read of threads plugin. Many mutations result in changed,
Table 5.1. Number of tests and resulting errors for Volatility plugins

<table>
<thead>
<tr>
<th>Plugin</th>
<th>executions</th>
<th>exit with error</th>
<th>Timeout crashes</th>
<th>large output file</th>
</tr>
</thead>
<tbody>
<tr>
<td>atoms</td>
<td>184608</td>
<td>1286</td>
<td>17</td>
<td>N/A</td>
</tr>
<tr>
<td>drivermodule</td>
<td>102912</td>
<td>104</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>impscan</td>
<td>20992</td>
<td>20913</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>threads</td>
<td>44797</td>
<td>562</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>procdump</td>
<td>221158</td>
<td>12711</td>
<td>32</td>
<td>128</td>
</tr>
<tr>
<td>sockscan</td>
<td>17280</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>pslist</td>
<td>245664</td>
<td>421</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>dlllist</td>
<td>120913</td>
<td>9</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>ldrmodule</td>
<td>79963</td>
<td>3147</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>timers</td>
<td>349184</td>
<td>4626</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>deskscan</td>
<td>348935</td>
<td>1079</td>
<td>436</td>
<td>N/A</td>
</tr>
<tr>
<td>mac_kevents</td>
<td>622272</td>
<td>631</td>
<td>60</td>
<td>N/A</td>
</tr>
<tr>
<td>linux_psscan</td>
<td>342272</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>mac_ldrmodules</td>
<td>157802</td>
<td>42836</td>
<td>0</td>
<td>N/A</td>
</tr>
</tbody>
</table>

yet valid input files where for example only one field has taken a different value. These mutations cannot and do not need to be detected by a memory analysis tool, neither is it necessary for the fuzzer to expect a meaningful error message for every mutation. The fuzzer rather tries to avoid these mutations by jumping over a couple of subsequent reads to get to reads whose mutations result in “interesting outputs“.

Crashes of importance can be found among the cases in exit with error. They are reported by the operating systems as unhandled exceptions and their stack traces are stored in the log files. One example is shown in figure 5.2, where the 1\textsuperscript{st} mutation is applied on 26787\textsuperscript{th} read and the mutated buffer resulted in incorrect string format which was not checked by Volatility leading to its crash.
Another example can be seen in image 5.3, where mutation 26 (mutate_current_plus_4096_fill) – which means add every byte in the read buffer to 0x1000 – was applied on 524th read while the atoms plugin of Volatility was running.

Another set of significant findings of the fuzzer are resource exhaustion cases. Four of the tested plugins entered a near infinite loop in some read-mutation combinations which caused the fuzzer to time out on them and terminate the execution. The timeout configuration was 3600 seconds for these executions, while the response time of Volatility was significantly less. For plugin mac_kevents as an example, the average response time was 108.92 seconds. Table 5.2. shows the read number and mutation combinations when the
timeout cases happened. As can be seen, all 60 time out crashes happen while mutating 11 reads. This means the data that is accessed during those reads can affect the performance of Volatility critically.

Table 5.2. Mutation-read combinations causing `makevent` to timeout

<table>
<thead>
<tr>
<th>Mutation</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>14716</td>
</tr>
<tr>
<td>12,13</td>
<td>5979</td>
</tr>
<tr>
<td>8,9,12,13</td>
<td>6018</td>
</tr>
<tr>
<td>12,13</td>
<td>7536</td>
</tr>
<tr>
<td>12,13</td>
<td>10158</td>
</tr>
<tr>
<td>8,9,12,13</td>
<td>10197</td>
</tr>
<tr>
<td>8,9,12,13</td>
<td>16203</td>
</tr>
<tr>
<td>22,27</td>
<td>16205</td>
</tr>
<tr>
<td>6,10,14,15,16,17,18,19,20,21,23,24,25,28,29,30,31</td>
<td>16206</td>
</tr>
<tr>
<td>8,9</td>
<td>16229</td>
</tr>
<tr>
<td>8,9,12,13</td>
<td>16268</td>
</tr>
<tr>
<td>6,10,14,15,16,17,18,19,20,21,23,24,25,28,29,30,31</td>
<td>18389</td>
</tr>
</tbody>
</table>

In addition to timed out cases, there were executions that finished within less time than the time-out, but resulted in very long outputs. These resource exhaustion problems
usually happen when the mutated buffer is supposed to store the size of an element to be read and it is mutated to a large value. Such cases were found in plugins **atoms** and **threads**.

One of the plugins that we have tested so far creates dump files. plugin **procdump** is a Windows plugin that finds the mapped processes on memory and dumps them as executable files on disk. The process id of a specific process can be passed as a parameter to **procdump** for it to dump that process only, or it can be called without parameters to dump all processes as different executable files in a directory. It prints the list of files that it dumps as its output. In our test, **procdump** was called without parameters to store the dumps of all processes. After each completion of running the plugin, the sizes of dumped files are obtained and if they are larger than 100 MB, the test environments are stored. Out of 221158 tests on **procdump**, 128 created dump files larger than the threshold in our current configuration. These are cases of file system exhaustion and are usually a result of a pointer being mutated to address invalid data resulting in returning a wide address range as the process’s memory space. a sample of reported large files is shown in figure 5.4.

![Table](https://via.placeholder.com/150)

File too large:="/work/arian/hxfuzzer/fuzzerresults/procdump/dumps12th15/executable.344.exe" size: 1577106096 locality: 2 thread: 15 read: 26682 mutation: 27
File too large:="/work/arian/hxfuzzer/fuzzerresults/procdump/dumps14th1/executable.1320.exe" size: 60483936 locality: 4 thread: 1 read: 46612 mutation: 27

**Figure 5.4. Large files produced by the procdump plugin**

There were numerous runs that exited normally but had some warning messages printed in their outputs. These are example of Volatility handling invalid data and reporting
it, while extracting all other information from the memory sample. These cases can be easily used in forensics automated solutions since Volatility’s output is comprehensive. An example of such cases is shown in figure 5.5, where 1st mutation is applied on 561670th read when executing plugin ldrmodule. The plugin first prints as much information as can be extracted (the long output is trimmed from top) and then outputs warning about the corrupt input and exits normally.

<table>
<thead>
<tr>
<th>1312 juscved.exe</th>
<th>0x7c9c0000 True True True \WINDOWS\system32\shell32.dll</th>
</tr>
</thead>
<tbody>
<tr>
<td>1312 juscved.exe</td>
<td>0x77e70000 True True True \WINDOWS\system32\rpcrt4.dll</td>
</tr>
<tr>
<td>1312 juscved.exe</td>
<td>0x78130000 True True True \WINDOWS\system32\ur\mon.dll</td>
</tr>
<tr>
<td>1312 juscved.exe</td>
<td>0x76300000 True True True \WINDOWS\system32\imm32.dll</td>
</tr>
<tr>
<td>1312 juscved.exe</td>
<td>0x71200000 True True True \WINDOWS\system32\oleaut32.dll</td>
</tr>
<tr>
<td>1312 juscved.exe</td>
<td>0x77e60000 True True True \WINDOWS\system32\secur32.dll</td>
</tr>
<tr>
<td>1312 juscved.exe</td>
<td>0x78000000 True True True \WINDOWS\system32\kernel32.dll</td>
</tr>
<tr>
<td>1312 juscved.exe</td>
<td>0x38000000 True True True \WINDOWS\system32\iertutil.dll</td>
</tr>
<tr>
<td>1312 juscved.exe</td>
<td>0x50000000 True True True \WINDOWS\system32\comctl32.dll</td>
</tr>
<tr>
<td>1312 juscved.exe</td>
<td>0x77e60000 True True True \WINDOWS\system32\ole32.dll</td>
</tr>
<tr>
<td>1312 juscved.exe</td>
<td>0x7e100000 True True True \WINDOWS\system32\user32.dll</td>
</tr>
<tr>
<td>1312 juscved.exe</td>
<td>0x7c900000 True True True \WINDOWS\system32\ntdll.dll</td>
</tr>
<tr>
<td>1312 juscved.exe</td>
<td>0x7c100000 True True True \WINDOWS\system32\msvcr7.dll</td>
</tr>
<tr>
<td>1312 juscved.exe</td>
<td>0x7ddf0000 True True True \WINDOWS\system32\msvcr7.dll</td>
</tr>
</tbody>
</table>

Invalid Address 0x86E77F30, instantiating Control Area
mutation: 0 read: 561670 plugin: ldrmodules

Figure 5.5. ldrmodule warning output

There were also a large number of cases with strange outputs that exited with normal exit codes, cases that printed only a few lines of output and exited normally, and cases that did not print any outputs and exited normally. Many of these cases were analyzed and were deemed insignificant findings since they resulted in inputs that Volatility found invalid and terminated its execution as a result. Although Volatility did not output any errors for these tests, it did detect they were invalid. Volatility did not crash or run into any other problems either. Therefore, they are not test cases that exhibit any vulnerabilities. Figure 5.6. shows an example of such cases.

The fuzzer in this work was run on one of Louisiana State University’s supercomputers called SuperMIC. SuperMIC contains a total number of 382 nodes running Red Hat Enterprise 6.9, each with two 10-core 2.8GHz Intel Ivy Bridge-EP processors. 380 compute nodes each have 64 GB of memory and 500 GB of local HDD storage. 360 of the compute nodes
have 2 Intel Xeon Phi 7120P coprocessors. 20 of the compute nodes have 1 Intel Xeon Phi 7120P coprocessor and 1 NVIDIA Tesla K20X. The maximum total time of running a job allowed on SuperMIC is 72 hours and the maximum number of nodes per job is 128. Table 5.3. shows the tests that were run on a set of plugins.

As can be seen in Table 5.3., some plugins like threads, ldrmodule, and mac_ldrmodules are much slower than other plugins and fewer number of executions could be completed on them along all their assigned nodes in 72 hours. Tests on these three plugins did not even get to finish one round of all designated reads mutated with the first mutation to reach the second mutation. The reason for this difference in plugins’ speeds is the diversity in plugins’ operations.

The differences in plugins’ performances can be seen in Figure 5.7. Here, the average number of executions on all cores per hour is calculated for each plugin. A correspondence can be seen between the total number of reads that a plugin performs, and the amount of time that takes it to executes. The intuitive explanation is, plugins that perform fewer numbers of total read calls, also do less processing. Therefore, they take less time to finish execution. But this is not a definitive factor since plugins’ time consuming operations are not limited to reading files. Besides, it should be kept in mind that four of the plugins have had multiple timeouts in their executions. Each timeout takes one hour (can be found.
Table 5.3. Fuzzing stats for different plugins

<table>
<thead>
<tr>
<th>Plugin</th>
<th>executions</th>
<th>reads</th>
<th>mutations</th>
<th>nodes</th>
<th>cores</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>atoms</td>
<td>184608</td>
<td>28490</td>
<td>32</td>
<td>10</td>
<td>200</td>
<td>250:55:26</td>
</tr>
<tr>
<td>drivermodule</td>
<td>102912</td>
<td>15746</td>
<td>32</td>
<td>9</td>
<td>180</td>
<td>38:16:29</td>
</tr>
<tr>
<td>impscan</td>
<td>20992</td>
<td>3266</td>
<td>32</td>
<td>9</td>
<td>180</td>
<td>22:27:25</td>
</tr>
<tr>
<td>threads</td>
<td>44797</td>
<td>722216</td>
<td>1</td>
<td>10</td>
<td>200</td>
<td>376:45:16</td>
</tr>
<tr>
<td>procdump</td>
<td>221158</td>
<td>121020</td>
<td>12</td>
<td>9</td>
<td>180</td>
<td>781:11:46</td>
</tr>
<tr>
<td>sockscan</td>
<td>17280</td>
<td>541</td>
<td>32</td>
<td>10</td>
<td>200</td>
<td>02:07:24</td>
</tr>
<tr>
<td>pplist</td>
<td>245664</td>
<td>7677</td>
<td>32</td>
<td>5</td>
<td>100</td>
<td>268:43:11</td>
</tr>
<tr>
<td>dllist</td>
<td>120913</td>
<td>271319</td>
<td>2</td>
<td>14</td>
<td>280</td>
<td>412:26:28</td>
</tr>
<tr>
<td>ldrmodule</td>
<td>79963</td>
<td>806512</td>
<td>1</td>
<td>15</td>
<td>300</td>
<td>358:50:36</td>
</tr>
<tr>
<td>mac_kevents</td>
<td>622272</td>
<td>19456</td>
<td>32</td>
<td>15</td>
<td>300</td>
<td>353:19:00</td>
</tr>
<tr>
<td>linux_psscans</td>
<td>342272</td>
<td>10700</td>
<td>32</td>
<td>9</td>
<td>180</td>
<td>309:42:20</td>
</tr>
<tr>
<td>mac_ldrmmoduls</td>
<td>157802</td>
<td>238855</td>
<td>1</td>
<td>9</td>
<td>180</td>
<td>400:09:01</td>
</tr>
<tr>
<td>timers</td>
<td>349184</td>
<td>87851</td>
<td>32</td>
<td>17</td>
<td>340</td>
<td>205:40:06</td>
</tr>
<tr>
<td>deskscan</td>
<td>348935</td>
<td>10923</td>
<td>32</td>
<td>10</td>
<td>200</td>
<td>366:23:33</td>
</tr>
</tbody>
</table>

in table 5.1.). So taking deskscan as an example, 436 hours of the total time of running this plugin over all nodes were spent for 436 of the executions, and the rest of the time was allotted to all the other executions. Therefore, the average execution time does not necessarily give a precise scale of plugins’ speed.

Figure 5.8. shows the resources used for testing different plugins as reported by TORQUE. The plugins with heavy computation can be noticed here too. Average load shows the number of running processes per node. Maintaining a load equal to the number of cores gives the ideal outcome in terms of the amount of jobs done. Because the maximum number of processes that can run in parallel is equal to the number of cores in a node. So, having fewer running processes than the number of cores means not using all the resources hence taking a longer time for the same amount of work. Also having more running processes
Figure 5.7. Execution times and read numbers of plugins

than the number of cores means imposing overhead on the processors, leading to jobs taking a longer time for the same amount of work.

Figure 5.8. Resource usage of plugins
Chapter 6.

Conclusions and Future Work

6.1. Conclusions

Fuzz testing memory forensics frameworks is of vital importance both in terms of software security assurance and reliability of forensic investigations. Our tests on Volatility were able to find cases that crash the framework, cause it to enter infinite loops, or exhaust file system resources. These findings of our fuzzer were cases that could not be found by other manual or automatic tests. They were caused by changes in pointer references, data structure layouts, and mapping information of memory samples. But inability to find these crashes through manual or automatic tests does not mean that they cannot happen. These errors are likely to exist especially in memory forensics where the analysis is performed on memory images. As explained in section 3.1., memory images are prone to these changes and tools that analyze these images must be able to handle such abnormalities securely.

Furthermore, much needed automated forensics relies on robust forensic frameworks which cannot be guaranteed without comprehensive tests. Found crashes and errors confirm the necessity of testing more plugins and analysis frameworks in order to ensure their security. As more plugins and tools are tested and more vulnerabilities are discovered and tackled, we will have more robust memory forensics frameworks used in investigations. The results of these analysis frameworks can be affected if their vulnerabilities are exploited and that can affect the results of investigations. So it is imperative that we perform these tests on tools with different inputs and make sure any vulnerability is covered by us before it is exploited.

Most plugins tested by far were Windows plugins, but we need to test plugins of other operating systems too as they are completely different than their Windows versions due to substantial differences among operating systems. We have been tuning the configuration variables in tests based on the results as we proceeded in tests. More effective configurations
will be achieved as we analyze and evaluate more test outcomes. Additionally, analysis of discovered vulnerabilities can further guide the design and architecture of mutations. Consequently, this lightweight and portable fuzzer which has a high coverage with fine grained precision, has configuration variables that enable more guided tests and better efficiency.

We have performed our analysis on memory forensics frameworks, and our drive to develop this fuzzer was the urgency of comprehensive fuzz testing for these tools. But our fuzzer can also test other tools, either closed source or open source, written in different languages. It can test any tool that takes files as their inputs. Especially for tools that work with large files, such as photo and video editors, this is an efficient solution to fuzz testing and finding vulnerabilities.

Our high performance computing approach enhances the fuzzing performance greatly. The latency of parallel executions did not encounter any increase over time and all cores continued running jobs throughout the tests. HPX has several features that can be leveraged to further boost the fuzzer. Its advantages of low overhead for computational intensive programs and dynamic load balancing support scalability and enhancement of our parallel implementation.

6.2. Future Work

Running more tests on other plugins, other memory forensics frameworks, and with more memory samples as initial inputs, is the primary outcome and future of this work. As the current findings prove, there are important vulnerabilities that can be found by fuzzing and we need to continue fuzz testing to cover all possible flaws. In order to make this possible, the whole process of fuzz testing needs to become automated. That is, performing tests, auto tuning more configuration variables as test results change, and automatically analyzing crashes and exporting them. Adapting configurations to the results as the fuzzer continues to test different pages of memory or different plugins has been an ongoing work,
and more customization is going to be added to the program as we go on. The process of extracting crashes is also automated for each log file. But the fuzzer needs to add the automatic analysis of crashes and integrate all these procedures into one. The next step in this procedure is exporting the issues and automatically ticketing them, so that the developers are made aware of discovered issues as soon as they are found.

Another major future work is on high performance fuzzing. We are currently running distributed tests and for almost the whole duration of tests all nodes are running with full load and without slowdown. But we need to scale the distribution up and make it run as fast as possible. There are many HPX features that can be taken advantage of to improve our tests.

We are going to take advantage of the work queue based architecture of HPX thread managers to migrate threads to localities where execution resources are available. This is needed because we assign static numbers of executions to localities at the initialization phase. But as the tests run, fuzzer might decide to skip a number of tests according to the output of the program it is testing. So the number of executions to be done on localities change while tests run and they will not remain equal among all nodes. This can result in one node finishing all assigned jobs while there is still remaining work on other nodes. Solving this problem is possible using HPX and it will be worked on in future.
References


Vita

Arian Shahmirza was born in Tehran, Iran. She graduated with her B.Sc. in Electrical Engineering from Shiraz University in 2011. She has 5 years of work experience as full stack software developer. In 2017, she started her master's studies in Computer Science at the Division of Computer Science and Engineering at Louisiana State University. She anticipates graduating with her M.Sc. in August 2019.