Effective Fuzzing Framework for the Sleuthkit Tools

Shravya Paruchuri

Louisiana State University and Agricultural and Mechanical College

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

Part of the Computer and Systems Architecture Commons, Data Storage Systems Commons, and the Other Computer Engineering Commons

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_theses/5044

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.
Effective Fuzzing Framework for the Sleuthkit Tools

Shravya Paruchuri

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

Part of the Computer and Systems Architecture Commons, Data Storage Systems Commons, and the Other Computer Engineering Commons
Effective Fuzzing Framework for the Sleuthkit Tools

Shravya Paruchuri

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_theses
Part of the Other Computer Engineering Commons
EFFECTIVE FUZZING FRAMEWORK FOR THE SLEUTHKIT TOOLS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in

Department of Computer Science

by
Shravya Paruchuri
B.Tech., GITAM University, 2016
December 2019
Acknowledgments

First and foremost, I would like to express my profound appreciation to my faculty advisor and mentor, Dr. Golden G. Richard III, for his patient guidance towards my research and continuous support of his funding throughout my graduate years. I am indeed blessed to work under such a wise teacher. I would also like to thank our external collaborator involved in this research, Andrew Case, for his enormous effort spent in assisting me and providing his time and wealth of experience in the field of digital forensics.

I would also like to thank Dr. Konstantin Busch and Dr. Mingxuan Sun for agreeing to serve on my thesis committee and contributing their time.

Finally, I want to take this opportunity to thank my family, friends, and well-wishers, who supported me throughout this journey.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ........................................................................................................... ii  
LIST OF TABLES ......................................................................................................................... iv  
LIST OF FIGURES ........................................................................................................................ v  
ABSTRACT .................................................................................................................................. vi  

CHAPTER 1. INTRODUCTION ................................................................................................... 1  
  1.1. Background ...................................................................................................................... 1  
  1.2. Problem statement .......................................................................................................... 2  
  1.3. Scope ................................................................................................................................ 6  

CHAPTER 2. TECHNICAL BACKGROUND .............................................................................. 7  
  2.1. Fuzzing Overview ........................................................................................................... 7  
  2.2. Digital Evidence ............................................................................................................. 10  
  2.3. Volume system artifacts ............................................................................................... 12  
  2.4. File system artifacts ...................................................................................................... 12  
  2.5. The Sleuthkit Framework .............................................................................................. 15  

Chapter 3. LITERATURE REVIEW .......................................................................................... 19  

Chapter 4. GASLIGHT OVERVIEW ...................................................................................... 26  
  4.1. Design ........................................................................................................................... 26  
  4.2. Mutations in Gaslight .................................................................................................... 27  

Chapter 5. IMPLEMENTATION AND METHODOLOGY ....................................................... 30  
  5.1. Considerations ................................................................................................................ 30  
  5.2. LD_PRELOAD as a Hooking Mechanism ..................................................................... 31  
  5.3. TSK Fuzzer development ............................................................................................... 33  
  5.4. Performance gains ......................................................................................................... 38  

Chapter 6. TESTING ENVIRONMENT .................................................................................... 40  
  6.1. Experimental Results .................................................................................................... 40  

Chapter 7. CONCLUSIONS AND FUTURE WORK ............................................................... 45  

REFERENCES ............................................................................................................................... 47  

VITA .......................................................................................................................................... 50
LIST OF TABLES

Table 1. Sleuthkit tools for Volume and File system analysis................................................................. 16

Table 2. Tested File Systems and results from fuzzer execution............................................................ 40
LIST OF FIGURES

Figure 1. Basic Fuzzing Approach ........................................................................................................ 9
Figure 2. Non-volatile Storage Hierarchy ............................................................................................... 11
Figure 3. mmls command output .......................................................................................................... 16
Figure 4. fsstat command output ......................................................................................................... 17
Figure 5. fls command output ................................................................................................................. 18
Figure 6. icat command output ........................................................................................................ 18
Figure 7. Fuzzing algorithm implemented by FUSE .............................................................................. 27
Figure 8. An Example of LD_PRELOAD mechanism for kernel-level hooking ................................. 32
Figure 9. TSK Fuzzer Design Architecture ............................................................................................ 34
Figure 10. Two-Pass Comprehensive Algorithm for computing and mutate offsets .............................. 35
Figure 11. Overriding the open() system call ..................................................................................... 37
Figure 12. Overriding the read() ......................................................................................................... 38
Figure 13. NTFS image crash for improper sanity check ........................................................................ 41
Figure 14. Another NTFS crash for sanity check .................................................................................. 41
Figure 15. NTFS image crash for boundary check ................................................................................ 42
Figure 16. ExFAT image crash for ils ................................................................................................... 42
Figure 17. NTFS memory allocation issue ............................................................................................. 43
Figure 18. NTFS crash for no size check on malloc () ......................................................................... 43
Figure 19. NTFS malloc () check ........................................................................................................ 44
Figure 20. NTFS crashed for improper memory allocation ................................................................. 44
ABSTRACT

The fields of digital forensics and incident response have seen significant growth over the last decade due to the increasing threats faced by organizations and the continued reliance on digital platforms and devices by criminals. In the past, digital investigations were performed manually by expert investigators, but this approach has become no longer viable given the amount of data that must be processed compared to the relatively small number of trained investigators. These resource constraints have led to the development and reliance on automated processing and analysis systems for digital evidence. In this paper, we present our effort to develop an automated stress testing platform specifically tailored to assess the robustness and reliability of digital forensics tools. For our initial testing, we chose to target The Sleuthkit framework, given its prominence both as a standalone tool as well as a programming library that is utilized by many open source and commercial file system analysis systems. The results of our efforts were the automated discovery of many critical programming errors in the Sleuthkit framework.
CHAPTER 1. INTRODUCTION

1.1. Background

Vulnerabilities are often a critical check for a system to be secure, and vulnerability analysis is considered an important research area in the field of computer security and software development. It is always a challenging task for security experts to identify vulnerabilities in software, as it typically demands a large amount of manual effort and is, therefore, quite time consuming [1]. Previously, Software testers used to trace the vulnerabilities through manual inspection of each line of source code, which consumes their time, effort, and chances of encountering human errors while augmenting the entire source code step-by-step are certainly high [2].

For some years, the testing was also performed by source code auditing, a type of white box testing where the investigator is expected to understand each detail about the programming segments of the software, the debugging environment, and has access to the complete source code [3]. Although auditing procedures are approved to be a part of the testing phase, it led to a gradual downfall of these techniques due to lack of automation, requiring access to the software sources (this may be a problem in case of closed source applications), above all, time and effort spent on dealing with the manual inspection of thousands of lines of code. All these factors made the developers focus their efforts on automating the process of vulnerability discovery towards the goal of reducing manual load and achieving efficiency in fixing software implementation errors.

One of the most commonly used automated testing techniques is fuzzing [4]. Fuzz testing is a stress testing technique that is used to monitor program executions for crashes or potential
memory leaks by generating random or semi-random inputs as testing inputs to the program. Fuzzing is a trending software testing technique that can quickly reproduce programming errors that would go undetected from manual testing and are either fully automated or semi-automated, reducing work overloads on software testers. Also, the errors found by fuzz testing are mostly real unlike the uncertainty observed in software instrumentation where the code needs further testing for false positives.

1.2. Problem statement

Software testing is an essential and complex phase of an application development lifecycle. There are many tools and techniques implemented to test and improve the quality of software. The testing is performed either manually by software testers or automatically with the help of testing tools that generate test cases according to the requirements of the software. Manual testing involves constant monitoring and parsing individual components by applying suitable edge cases for the robust functioning of the source code. However, with an increase in the complexity and the size of the program, manual inspection of thousands of lines of code is error-prone, and it is both tedious and time-consuming. All these resource constraints made investigators switch to automated tools and procedures for high-performance testing [5]. The demand for automated testing marked a rapid increase in the development of forensic tools that are required to perform comprehensive validation checks for measuring the performance and quality of application software.

Because of massive dependence on automated testing, there comes great responsibility for file forensic tools to become more reliable for digital investigations. The demand for high performance, robust algorithms has been increasing with the growth and complexity of anti-forensic mechanisms [2]. Also, File Forensic tools are challenging to secure as there are always
active chances of being vulnerable to malware exploits and so the need for producing the desired and accurate results is high. When performing hard disk data acquisition by a disk acquisition tool, copying of the file system contents may not always be consistent and appropriate, and so the image acquired by the tool has an active chance of being susceptible to the data corruption in the file system, which is known as file system smearing [6]. Malicious hackers can take advantage of exploring critical code paths, hide important information by pointing metadata file pointers to invalid data units, or use invalid offsets referencing file system critical data structures and manipulate address space of partition tables. All these cases need to be addressed and pass through potential vulnerability assessments before the obtained evidence is claimed to be reliable.

As forensic investigators are extremely dependent on automated analysis for extracting digital evidence, it is crucial for the tools to successfully handle invalid inputs, whether data corruption is intentional or not. Improper handling of file system smearing makes forensic frameworks yield unexpected results, crash, enter infinite loops, or be vulnerable to buffer overflow attacks [7]. So, file system tools need to be designed with optimal and robust algorithms that possess high-end capabilities of knowing the attacker’s mechanisms for withstanding malware exploitation and data smearing. It is also essential to continuously evaluate the accuracy and completeness of these tools with the rapid growth of data complexity and technology advances in recent years. Many different testing strategies have been evolved to assess the effectiveness of automated testing frameworks in the field of storage forensics [8]. In our current research, we focused on developing a stress testing platform for one of the most commonly used file forensic toolkits, called The Sleuthkit framework, which is designed and developed by Brain Carrier [9].
TSK is a widely used forensic framework for reading disk images, recover hidden and deleted files, extract file system, and volume system information from the image files. Despite its importance in vast evidence collection for various file systems like NTFS, FAT, Ext2, Ext3, UFS, there are no significant contributions made to assess its reliability or address any security vulnerabilities. The research is motivated to improve the accuracy of the framework implementation by fuzzing various file system parsing components of the Sleuthkit. The current stress testing approach involves brute-forcing inputs observed during the invocation of each of the command-line tools and discovers any potential programming errors in the command runtime.

1.3. Scope

The problem stated above is addressed by our black box fuzzing architecture with the implementation of an extensive and high-performance algorithm for testing any file forensic framework irrespective of the programming language it is written. The data read by the volume and file system commands of the Sleuthkit are identified as input vectors for the fuzzer as they are a valuable source of essential digital evidence. With the evolution of antiforensics and rootkit attacks, there is a substantial increase in the growth and development of digital forensic analysis procedures in recent years. The file system parsers may contain potential malware exploits that are usually targeted by active attackers. A typical scenario of file parsing exploitation is when an algorithm blindly accepts a file content without specifying its data length values. Malware attackers or outsiders being aware of the defect try to overwrite or embed their offensive code by specifying appropriate data lengths. As the algorithm accepts invalid data, the malware can take control of the target system by overwriting critical sections of memory with offensive data, leading to a typical trail obfuscation attack.
To withstand data manipulation mechanisms, forensics frameworks that primarily analyze the disk image should be designed to treat file system data as an untrusted source. Also, the data that is being extracted by the testing tools need to be evaluated for any faulty evidence being written to the standard output or standard error file as the investigator might not be aware of the attack all the time. The forensic tools are required to discover and analyze the persistent storage evidence without compromising the sufficiency and accuracy of data in case of malware tampering or file system smearing [10]. While our testing platform focuses majorly on those code routines of the toolkit that process the disk images, we do not consider any input data coming from the user, such as network-related information, command line inputs, or any other user-defined variables for applications.

The proposed fuzz testing framework targets one of the most popular toolkits for file forensic evidence, the Sleuthkit framework. For our initial testing, we implemented our fuzzing architecture on a few of the file systems, namely, NTFS, Ext4, and ExFAT as they are the most commonly used file system storage formats. The current fuzzer is uniquely designed to focus only on those code patterns that the Sleuthkit parses, automates the fuzzed inputs, and loads them dynamically during the command runtime. The current testing architecture, while currently aimed at TSK implementations, can also be extended against other file forensic suites like Encase as the fuzzer is not designed to be application-specific and does not require changes to the source code.

The structure of this research is outlined as follows: Chapter 2 provides the necessary background on fuzz testing and types of fuzzers. Chapter 3 discusses previous work in this area of study. Chapter 4 discusses the motivation behind our current research by giving a brief overview of the Gaslight architecture. Chapter 5 describes our approach towards the design and
development of the fuzzer for the Sleuthkit and Chapter 6 presents the results of the tested tools.

Finally, we conclude the research discussion by identifying the limitations and observations for future work in Chapter 7.
CHAPTER 2. TECHNICAL BACKGROUND

2.1. Fuzzing Overview

Fuzzing or fuzz testing is a kind of dynamic execution analysis that involves brute-forcing of inputs to a software application to induce memory leaks, crashes, infinite loops, or other undesirable behavior. There are several ways to provide input vectors, some may input random values, and some may interpret the source code to provide valid and directed inputs, called fuzzing seeds, for the application program. The input vectors are modified with the motivation of exploring different code patterns within the software and are further monitored for any exceptions or programming vulnerabilities.

2.1.1. Stages of Fuzzing

The fuzzing strategy can vary depending on the target source, requirements of the investigator, and the input format required to run the application [12]. No matter what the fuzzer type is, the testing typically follows a basic approach in identifying the code vulnerabilities. The primary phases for fuzzing involve:

1. Identifying a target system: Firstly, the investigator must be aware of the source on which fuzzing is implemented, without which it is impossible to generate proper test cases in an application environment. In cases like security auditing, the target code is known to the tester via feature selection, and he employs a suitable testing strategy to discover new code paths. Analysis of binaries of the application and implementation details can also help in identifying the right target system.
2. Identifying inputs: Most of the vulnerability exploits are found in applications that need user inputs, and when those inputs are being read with no sufficient validation checks. The application becomes vulnerable when they accept random input data without proper input validation checks. The primary goal of the fuzzer is to be aware of the input vectors that are passed to a target application. Identification of the inputs is the most critical attempt for any fuzzer to become successful as there is a possibility of overlooking or missing some of the input coverage, which eventually limits the testing.

3. Generate fuzzing data: The next step to execute after enumerating the input vectors is to generate suitable fuzzing data. The generation of the fuzzed data may include predetermined inputs that are properly tainted and tested with symbolic execution [13]. The fuzzing data, in this case, is an expected input space of vectors. Besides using tainted analysis, the fuzzing data can also be generated by applying mutations on existing user input formats, user input data, or it may also be generated dynamically at runtime depending on the target system or the application.

4. Execute the test using fuzzed data: This step requires automation as an efficient fuzzing strategy necessitates automated testing of the applications. Execution of the test involves testing the data packets that are being sent to the application source (if testing network-related information), running target programs or functions (if testing the application functionality), and so on.

5. Monitor system behavior for exceptions: Fault monitoring or exception handling is the most crucial step to assess the efficiency of the fuzzer. The generation of the fuzzing inputs should be handled by a deeper understanding of the fault possibilities of the target; otherwise, it may fail to trace out abnormal behavior of the code or cannot pinpoint the input vector responsible for system anomalies irrespective of the test cases generated.
6. Determine and log the errors: Once the fuzzer identifies the bugs in the target implementation, the tester needs to record and preserve all those events and determine the exploitability of vulnerable code vectors. Applying suitable security engineering techniques help investigators to determine if the identified error can further be exploited.

![Basic Fuzzing Approach Diagram](image)

Figure 1. Basic Fuzzing Approach

2.1.2. Types of Fuzzing

Fuzz testing is classified into various categories based on the awareness of the input structure, program structure, and input generation [4], [12]. One of them is termed as white-box fuzzing, and the other is known as black-box fuzzing. White box fuzzing requires access to source code for instrumentation and generates valid test data according to the input format of the original program. On the other hand, black-box testing does not need source code access, but the compiled software is presented to perform fuzzing. Black box fuzzing incorporates both static and dynamic analyses depending on the requirements needed for testing. The fuzzing techniques are broadly classified into two types.

1. Generation-based
2. Mutation-based fuzzers.
Generation-based fuzzing involves creating test cases depending on the target protocol or input model of the tested application[4]. The fuzzer designs inputs from scratch, and so the testing is limited in scope for more in-depth analysis. Because this approach comprehends all of the input vectors before implementation, it is a challenging task for the fuzzer to generate suitable test cases for new and unknown input formats and also to frame the inputs that can pass maximum code coverage.

On the other hand, mutation-based fuzzers need not be aware of the input models of software to perform fuzzing. The mutation-based fuzzers refer to black-box fuzzing, where the test cases are generated from the existing samples, and therefore, not much initial upfront is needed to start the testing. However, the testing is time-consuming as it produces a large number of mutated samples for a single file, and testing all these mutated inputs is an exhaustive process compared to a generation-based strategy. But this fuzzing approach is effective in terms of maximizing the input space for discovering new code paths.

2.2. Digital Evidence

The proposed fuzz testing platform targets the Sleuthkit framework, which is predominantly known for the identification and extraction of filesystem forensic evidence. In our current research, we focus on the areas that the Sleuthkit extracts the data from and presents its digital evidence. The digital forensic evidence includes the content from storage devices such as hard disks, removable media, or CD-ROM. In other words, we are looking for evidence in non-volatile memory or physical storage media. Analysis in persistent storage involves structured or layered analysis as it deals with different types of data starting from the sectors of a disk at lower levels of storage called physical device analysis to the data being mapped at a higher-level called application analysis [14].
Physical storage analysis includes reading the data from magnetic media, which are typically 0s and 1s written to the device. Data from the physical media is organized in sectors of disk storage, and these sectors are analyzed at the volume layer by addressing partitions. Each volume contains different file systems or similar, and the layout of each file system can be identified based on the partition structures. The filesystem drivers produce necessary metadata for every file created on the operating system and are further analyzed by understanding the file format, data units, and relevant metadata structures. Given our research scope, we discuss only the volume and file system artifacts that the Sleuthkit identifies and extracts from a target system.

![Non-volatile Storage Hierarchy](image)

Figure 2. Non-volatile Storage Hierarchy
2.3. Volume system artifacts

Volume analysis involves the inspection of data structures related to partitions, allocated and unallocated storage, and other data present within a volume. Digital investigation tools usually perform the acquisition of an entire disk image and break the image into partitions for analysis. To perform file system analysis, investigators first need to process the partition tables to understand the type and partition layout of the volume before they can look for file system data structures. The essential data to look for in a partition table is the start and end location of each of the partitions and see if they are corrupted [15]. In cases where partition data structures are not identified or are missing, the type of partition and its boundaries can still be known by understanding the data stored in sectors of the partition.

Consistency checks in volume analysis play an essential role in proper addressing of file system partitions [16]. As the partitions are laid out, filesystem forensic tools sort them in a sequential order according to their start and end offsets, which makes it easier for the investigators to analyze any of the hidden data within the volume or any data that is corrupted or has overlapping boundaries, and also to identify data that may exist between partitions due to previous file system structures and recover any modified or deleted partition tables.

2.4. File system artifacts

File system analysis is an integration of procedures based on the type of evidence needed by a forensic examiner during an investigation. Though file systems have their specific structure, there is an abstract reference model developed for understanding the file system internals in general by Brain Carrier [14]. Briefly, data analysis inside the file system is performed under five different categories: file system category, content category, metadata category, file name category, and application category [17]. For the file system category, the analysis is performed
on the layout of files and the associated metadata. Analysis in the file system category involves a general understanding of the filesystem that provides a roadmap to figure out the location of the data structures in different data categories. Since the file system category provides a layout for other data categories, it is crucial to preserve this section of data because any corruption or loss of the file system data would affect further analysis. In many cases, the data in the file system category lies in the first sectors of the file system. In a FAT filesystem, this information is present in the boot sector data structure.

In the content category, data is organized in groups called data units, and these units are assigned whenever a new file is created by an application, or an existing file is modified [18]. Analysis of the content category involves recovering deleted files, hidden and allocated content. If a user deletes a file, the operating system makes sure that the corresponding data unit is unallocated, erases the filesystem-specific entries (like logical file system address, offsets), and makes them available for newly created or existing files. The details of data unit allocations are stored in a data structure of the associated filesystem.

The third phase of analysis involves the extraction of metadata associated with file-specific descriptions such as the time of creation, last access, and user access information. Similar to the data unit layer that holds actual file system data, the metadata layer contains data about the data units. Analysis in the metadata category involves the extraction of file-specific data and helps to recover the deleted files by metadata lookup [16] in most of the cases. It is important to carefully look for the evidence, especially when the files are deleted and data units are unallocated even before the metadata was. For example, when a file is deleted, the corresponding data unit is unallocated, but the metadata entry of the file may still have the data unit address. So, when the metadata entry is reallocated to a different data unit by a newly created file, it would be
challenging to know if the previous data unit has the content of the deleted file or the contents of the new file that is created after the metadata entry is reallocated. In cases where metadata is lost or wiped out, it is impossible to restore the file through metadata-based recovery, and the analysis depends on the application that created the file.

The data in this category also comprise of non-essential data like access controls and timestamps, date of creation, deleted or modified file details, and so on. The data is termed as non-essential evidence because the metadata entries keep changing whenever there are modified timestamps for a file or access change for a user. So, additional evidence in the form of application-based techniques is always needed to support the non-essential data in the metadata analysis.

The next phase of file system analysis includes the file name category. Analysis in this category involves recovering the deleted files by names. In some cases, locating the root directory is needed to identify the file when its full path is known. Though the file name category recovers deleted files similar to the metadata category, it is still necessary to retrieve files by names. In some cases where metadata might contain information about files that no longer exist by name; the file can quickly be recovered using file name date structures. Conversely, for a deleted file that is traced by file name analysis, the assigned metadata entries are still helpful in the extraction of corresponding data unit contents. Finally, the application category does not contribute much to the actual file system analysis, but it is still useful to refer to the unique features of a file system, like Journaling. In cases when a computer is crashed during the process of creating or deleting a file by a user, the journaling feature of the application category helps to restore the incompleted file operation. The operating system runs the journal program and locates missing entries to maintain the data consistency of the file system.
2.5. The Sleuthkit Framework

The Sleuthkit is a set of command-line tools written in C, used to extract volume and file system related to forensic evidence. The Autopsy browser is the graphical interface to the Sleuthkit [9]. The TSK framework developed by Brain Carrier [6] is used to analyze volume and file systems associated with the target system, retrieve files and directories, and perform hash-based searches for unknown or suspected files. As the tools are independent of the type of computer architecture or the type of operating system being used, they extract the digital evidence, including hidden and deleted content that is critical to an investigation.

The volume system analysis tools are used to examine the disk and partitions layout, and the evidence obtained can further be analyzed with file system tools. TSK can be installed on both Windows and UNIX environments. There are about 20 tools currently in TSK, designed, and developed for various analysis purposes. Since each tool is designed to perform analysis for a specific data category, the TSK framework makes the analysis scalable and flexible by integrating different analysis modules into one common platform and allow access to relevant files for the application layer modules to operate, without any implementation issues [19].

The reason for having this toolkit on hand by many investigators is because of its extensive coverage of the data layers starting from disk-level storage to application-level storage. The TSK framework includes many of the tools implemented at the disk level, volume layer, and file system layer. It takes a raw disk image as input (acquired using dd (or a similar mechanism), or raw AFF disk images. It supports various file systems, predominantly NTFS, ExFAT, Ext4FS, Ext3/Ext2, UFS1/UFS2, and HFS file systems [9].
Below is a brief overview of some of the commonly used tools in the Sleuthkit for extracting digital evidence.

Table 1. Sleuthkit tools for Volume and File system analysis

<table>
<thead>
<tr>
<th>Data analysis category</th>
<th>TSK tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk image</td>
<td>disk_stat</td>
</tr>
<tr>
<td>Volume category</td>
<td>mmls, mmstat, mmcat</td>
</tr>
<tr>
<td>Data unit layer category</td>
<td>blkcat, blkls, blkstat, blkcalc</td>
</tr>
<tr>
<td>File system category</td>
<td>fsstat</td>
</tr>
<tr>
<td>File name category</td>
<td>ffind, fls</td>
</tr>
<tr>
<td>Metadata category</td>
<td>icat, ifind, ils, istat</td>
</tr>
</tbody>
</table>

Recovering a deleted file:

One of the interesting things about the Sleuthkit analysis is that it uses different tools for multiple layers of non-volatile memory extraction. Below is an example to show how various commands from the TSK framework are used to recover deleted file contents. In the first step, we use the mmls command on the USB device that is used for analysis. The command identifies different partitions with start and end addresses, the size of each partition, what filesystems are present. Here, we see three partitions, and there is an NTFS partition table identified at sector 2168.

![mmls command output](image)

Figure 3. mmls command output
After identifying the target partition, we now look for the details of the NTFS partition. We use fsstat command with the corresponding offset 2168. The output gives us the general layout and statistics of the NTFS file system, metadata structures, data units and their ranges, and other categories present in the file system.

![Figure 4. fsstat command output](image)

We then use the fls command from the file system tools to view the NTFS structure. The command lists all the active files and directories along with the deleted ones. The directories are indicated as d/d, files as r/r, and the numbers next to them are inodes. We see there is a deleted file called deleted.txt, and it is marked by the Sleuthkit using an asterisk symbol before its inode notation.
After the file system analysis, we now want to recover the deleted file contents. The `icat` command from the metadata layer is used to retrieve the data from the deleted file. The command is invoked with the file system offset 2168 and inode 36. Similarly, the visible file content is extracted with the help of corresponding inode number 35 and offset 2168.
Chapter 3. LITERATURE REVIEW

Software is considered to be highly secure when companies regularly conduct periodic code reviews and perform unit testing from time to time. All the software bugs should ideally be addressed before the shipment of the code to vendors, and it is always safe to continuously monitor the code as the developers create updated versions of the source code.

Most of the companies today tend to use fuzz testing as part of their software testing [20]. The common aim in the development of any fuzzing framework is that the fuzzer should be able to locate potential programming errors or anomalous system behavior because of vulnerable code execution. The first person to introduce us to the world of fuzzing was Barton Miller from the University of Wisconsin, Madison [20]. In the late 1980s, he was researching random data injection when his computer programs crashed while connecting the modem to remote computers. The system protocol could not detect the junk characters that were inserted periodically during the program's execution and eventually resulted in the program crashing. The concept of random data injection interested him, and he assigned his students the task of testing the standard UNIX utility programs. The results from one of his students showed that the utilities lacked necessary command-line validation checks.

The injection of random data as command-line arguments made the utilities throw segmentation faults and eventually caused the programs to crash because of insufficient validation of bounds checking. The research motivated the development of more efficient fuzzers with time. Until the late 1990s, fuzzing was a random, blind injection of mutations and was relatively shallow, which made it unaware of more hidden code paths that needed proper testing. Random or dumb fuzzing could produce millions of test samples, and all those mutated inputs might not lead to fault detections. The scope of dumb fuzzing is limited to traverse all the code
paths of the program, and so it is not an effective strategy. The evolution of genetic fuzzers then started with the creation of American Fuzzy Lop (AFL) by Michal Zalewski [21]. The fuzzer typically implements genetic algorithms for broad coverage of source code. The fuzzing program is directed to take the inputs from the user and break it to the smallest possible input vector for the application as it reaches the end of the execution. In the first run, the fuzzer creates inputs based on the input format of the source code and see if there is any change in the functional behavior. If the fuzzer is successful in passing the input checks, it proceeds with breaking down the input file further until the fuzzer detects any change in the implementation behavior caused by the nested inputs. All those input files responsible for the program crashing or system state anomalies are preserved as a test corpus for vulnerability inspections. It uses code coverage as the feedback for obtaining new inputs to generate unknown or unidentified paths and run the application with newly created input files for further brute-forcing.

The AFL fuzzer was also implemented as a file system specific fuzzer by starting a fuzzing instance with a predetermined set of file system images [22]. The file system images are the initial stack of input files, called as ‘seeds’ for the fuzzer to start the process. The initial process of the fuzzer drives it forward by changing extended attributes in the images to traverse along with new and hidden filesystem data. Though AFL is useful in testing many applications for program vulnerabilities, it is not suitable for our current research for two main reasons. First, AFL requires access to the source code of the application to perform branch instrumentation. Second, the fuzzer cannot instrument files of size more than 1MB. It does not relate to our current fuzzer goals as the testing framework accepts inputs with no limitation on the file size, and also it avoids copying the entire input file each time the application executes.
One of the most known fuzzing frameworks is Mangle, a mutation-based file fuzzer developed by Ilja van Sprundel in 2005. Mangle became prominent as one of very few file system fuzzing techniques, and it targets the header sections of all input files to target metadata. The fuzzing of the header sections and metadata information helps to detect any incorrectly processed data by a file specific parser and identify insufficient and inefficient boundary validation checks for the parsing code. The algorithm takes filename and header size as inputs, and it changes contents in the header section with randomly generated bytes. This fuzzer discovered unrevealed programming bugs in many applications. Though Mangle targeted filesystem parsers, the testing is limited to only metadata specific while our current fuzzing approach is suitable to test all the data in the filesystems.

There were also efforts presented in file system forensics to identify vulnerabilities in two commonly used file forensic tools, namely the Sleuthkit and Encase. Their work includes random fault injection testing against the two frameworks, targeted on NTFS, MS-DOS, and Ext2FS [23]. The research contributed significantly to the implementation of a random fuzzing algorithm targeting the metadata and file parsing routines. They fuzzed individual filesystem structures by mutating the entire file system images. Each of the mutated images is individually stored in disk partitions, and the entire disk is analyzed. The faulted filesystem image is isolated from the rest, to identify the partitions. The faulted partition is isolated to analyze the filesystem. The corrupted file system is compared to the original image for errors. The fuzzing efforts resulted in uncovering some vulnerabilities in both Sleuthkit and Encase as they failed to validate the fuzzer inputs. Their research was motivated to target the file system-specific parsers of the forensic software, yet the input space provided was mostly huge. The fuzzer consumes a maximum of disk space by copying the entire disk image file with only a few bytes smeared, causing time and
performance overheads. There was an attempt to test the file systems using model checking. They developed a model checker that generates new inputs based on the required input format and systematically run the filesystems with modified inputs for programming errors [24]. With each new state generated, the model checker intercepts the disk writes by the file system. After every disk write, the model compares the checked file system against a stable file system state that it is expected to be in after crash recovery. The approach mainly targets the UNIX utility fsck (file system consistency check) and its disk access patterns as fsck tries to recover the disk from the intercepted writes applied in its new test state.

Research work on identifying Windows' critical processes and the associated data structures was presented to test the stability of the operating system and eventually developed scanning signatures for each of the members in the tested data structures [25]. The analysis targets the EPROCESS data structure of the operating system to identify the members that are critical for the stability. Each member present in the data structure is mutated at different virtual machine instances of Windows. The data members that are affecting the operating system are selected to set up the constraints in the dynamic detection stage, and these are used to build scanning signatures. Though the testing involves fuzzing data present in memory, it is not aligned with our fuzzer architecture. The mutations are implemented to target specific members of the EPROCESS data structure, and focused only on the operating system behavior rather than testing any specific application. An approach for finding security vulnerabilities on executable files by performing static code analysis [26]. In their research, a standard methodology is designed for statically analyzing a PE file to identify vulnerabilities based on the file header, information located in sections, symbol tables, optional headers, and other file relevant data without having to disassemble the code.
The static analysis provides a file fact summary for the PE file. The analysis is performed manually to identify anomalies and discover hidden or compressed data in the file by mapping the contents after sorting the section boundaries. The methodology is further deployed in a software application \textit{findssv}, and this application is executed to monitor and parse different file formats of executables for sanitization checks, buffer overflow attacks, hidden sections and compressed data, and random filling of bytes in the data allocation spaces.

Using symbolic execution, there was a proposal made to perform white-box fuzzing on binary files. They presented an automated fuzzing technique for binary executables that accepts structured inputs \cite{27}. The research was intended to use both black-box and white-box fuzzing strategies by considering model file format inputs from the black box fuzzer. The inputs were marked as constraints before the symbolic execution engine can explore the code further deeper. The testing environment captures the data fields and adds a new valid chunk by removing the referenced chunk from the input model. All the newly formed inputs explore new execution paths and perform integrity and vulnerability checks. Model-based white-box fuzzing found significant vulnerabilities when tested on file formats of PDF, PNG, FLV, and WAV files and proved its testing efficiency compared to the traditional testing approach.

Another attempt for testing the reliability of UNIX utilities is made using black-box testing \cite{28}. The research targeted GUI applications on MacOS X and identified crashes for bad error handling, insufficient bounds checking, incorrect return values, and pointers with no sanity checks. There was a research contribution made to propose an effective symbolic execution strategy for automatically generating test cases in real programs \cite{29}. The symbolic executor \textit{EXE} marks the conditional derivatives as inputs. Before linking, \textit{EXE} derives true and false expressions of code conditionals as two individual constraints and runs them by forking the
execution on corresponding symbolic code branches. A constraint solver further solves these constraints by identifying concrete inputs and automatically generates test cases when either of the paths encounters an error or when it terminates. The test cases are implemented against the uninstrumented program for execution, and it was successful in identifying the program vulnerabilities.

Similar contributions were made to automatically find file system errors using EXE, a symbolic execution system developed in their previous work [29]. The system generated the symbolic inputs on three widely used Linux systems, Ext3/Ext2, and JFS solved the constraints for actual values using a constraint solver and identified critical bugs related to kernel memory allocation and insufficient boundary checks [30]. There were efforts made on identifying integer overflow vulnerabilities by applying symbolic execution and data flow analysis on code patterns that lack required input validations [31]. The approach involves running the binary file on a symbolic execution engine. The symbolic executor taints only those code paths that might be relevant to integer overflow vulnerabilities. On all the tainted paths, symbolic constraints are generated to verify if either of those paths is missing sanity checks for integer overflows caused by any of the tainted data. The proposal of path sensitive symbolic execution was successful in identifying 0-day vulnerabilities in binary files.

There was an attempt made to implement smart fuzzing, which performs both static and dynamic analysis techniques on input binaries [32]. Firstly, the static analysis of the executable is performed to identify different code patterns and control loops throughout the program. The program is then allowed to run on monitored executions to analyze the sensitive control flows and targeted pointers for memory accesses. Finally, the target code is marked for a constraint solver to solve the constraints for definite inputs to flow through the code paths and detect any security vulnerabilities as the code breaks or terminates execution.
Recent works on fuzzing techniques have been significantly improving and are more focused on maximizing the input coverage and so the evolution and rapid growth of symbolic execution and taint analysis is remarkably high in recent times [33]. One such attempt is “T-FUZZ: fuzzing by program transformation” by Peng et al., focused on improving the coverage of fuzzer inputs by transformational fuzzing [34]. They implemented a coverage-guided fuzzer, which is initiated when the fuzzer inputs are no longer leading to new code paths for vulnerability checks. The dynamic tracing technique identifies all those input checks for which fuzzer failed to proceed and disables them automatically to guide the fuzzer to get through complex sanity checks. Further, the bugs are reproduced in the original program by filtering false positives through symbolic execution. The proposed fuzzer was successful in automatically identifying potential security flaws and increased the code coverage by implementing mutations on both the inputs and the program itself.
Chapter 4. GASLIGHT OVERVIEW

4.1. Design

Gaslight is a fuzzer developed by Case and Richard [35] to identify implementation errors in memory forensics frameworks, such as the Volatility framework. Volatility framework is one of the most commonly used forensic frameworks for analyzing in-memory evidence [36]. The testing framework targeted only the critical code functions of the memory forensics framework that are used to extract the evidence from the memory image.

Gaslight introduced the dynamic fuzzing of inputs to memory forensics tools by incorporating a custom FUSE filesystem. There are three core components of Gaslight: Fuzzing harness, mutations queue, and custom FUSE filesystem. Fuzzing harness is responsible for creating fuzzing instances to the memory forensics framework. The harness exposes the mutated versions of the tested sample to the forensics framework. The mutated memory image is an instance of the original sample with random bytes changed. The mutations implemented by Gaslight are stored in mutations queue, and they are implemented as the framework executes the memory sample. The custom FUSE provides the mutated memory samples as inputs to the framework.

At first, the forensics framework is allowed to read the memory image, and the custom FUSE counts and records all the read operations of the framework in a results directory. Second, the fuzzing harness implements an algorithm for creating fuzzing instances each time the framework reads the memory sample. The fuzzing algorithm implemented is a two-pass comprehensive algorithm that helps the fuzzer to decide if it can apply mutations to the memory sample during all the reads of an application plugin, i.e., number_of_reads present on line 2.
When fuzz_past is set to 0, the fuzzer attempts to apply mutations in one of the plugin reads to simulate memory smearing or malware tampering. When fuzz_past is set to 1, the fuzzer starts applying mutations to the sample for each plugin read starting from the read_number. Mutations are the functions that perform data manipulation in the read buffer of the framework and return the buffer with mutations applied. The mutation index tells us the type of mutation that is being executed from the mutations queue.

4.2. Mutations in Gaslight

Gaslight implemented the following patterns of mutations that targeted the data on memory [35].

1. Fill the read buffer with 0x00 values
2. Fill the read buffer with 0xff values
3. Fill the read buffer with randomly generated bytes
4. On each boundary of 2, 4, 8 and 128 bytes, fill the boundary byte with 0x00, 0xff or a random byte
5. On each boundary of 2, 4, 8 and 128 bytes, fill the entire boundary with 0x00, 0xff or a random byte
6. On each boundary of 2, 4, 8 and 128 bytes, fill the boundary byte with current boundary byte value plus or minus the value in 2, 4, 8, 128 or 4096 bytes
7. On each boundary of 2, 4, 8 and 128 bytes, fill the entire boundary with current byte value plus or minus the value in 2, 4, 8, 128 or 4096 bytes

Each plugin in the framework is run through all versions of mutated memory samples. The variables fuzz_past, mutation_index, and read_number from the fuzzing algorithm are incorporated as environment variables to the custom FUSE implementation during the plugin runtime. The custom open handler of FUSE is called when the plugin opens the sample. When the open system call is invoked, the file descriptor is matched with the environment variables set by the harness. The use of environment variables allows the harness to send inputs to the custom read handler inside of the mutation library. As the plugin reads data from the memory sample on the custom open file descriptor, the read handler applies mutations on plugin reads configured within the environment variables.

The fuzzer successfully revealed the implementation errors in memory forensics frameworks like Volatility and Rekall, targeting Windows, Linux, and macOS. We considered extending the implementation of Gaslight in our current fuzzer for testing file forensics frameworks. The goals of our current TSK fuzzer closely align with the Gaslight approach of fuzzing for the following reasons:

□ The fuzzer is capable of testing forensics analysis frameworks without requiring modification to the source code.

□ The fuzzer invokes mutations dynamically at the plugin runtime to avoid memory overhead. As the framework reads the sample, the harness exposes only the instances of the mutated samples without having to copy the entire image file onto the disk each time.

□ Gaslight applies mutations to the memory sample by targeting only the data that the memory forensics framework reads from the sample. By doing so, the fuzzer performance
can be improved and reduces the time taken to complete the test runs. Mutating the data which is not read by the framework would not be useful as it only causes file exhaustion issues.

The fuzzing is targeted to test only the code patterns of the framework that processes the memory image.
Chapter 5. IMPLEMENTATION AND METHODOLOGY

5.1. Considerations

The motivation for our fuzzer came from the Gaslight implementation of fuzzing [35]. The fuzzing technique we implement is broadly scoped to any file forensic framework irrespective of the programming language it is written. Similar to Gaslight [35], the fuzzer neither requires access to the source code nor any modification to the application framework. The proposed testing environment do not target all of the application code like command-line parsing, user interrupts, configuring any of user-defined variables, or any of the TSK function calls that are not involved in analyzing a file system image. This approach of the fuzzer is solely intended to perform extensive stress testing on all those critical code paths of the TSK libraries that are used to process the image samples and hence improve performance over time.

Another important consideration is replacing the FUSE implementation [35] with custom shared libraries. The goal of our fuzzer is to possess the capability of kernel data manipulation by overriding the system calls. The custom functions defined by the fuzzer are compiled as shared libraries, and it is made to execute only those functions from the shared object file instead of executing the standard library functions. System calls are wrapped within the custom functions and are called to execute the standard functions but are not actually allowed to return the original values. Instead, the custom functions are preloaded for the fuzzer to execute and return the desired outputs. The detailed description of library preloading and shared libraries are discussed in section 5.2.

In Gaslight, the mutations are applied to the memory sample by the custom FUSE during the application runtime. The current fuzzer implements dynamic loading of the shared library –
Here it is our mutations library. This approach allows us to pass the mutations as fuzzing inputs to the command read at Sleuthkit runtime. Again, we are not intended to copy the original filesystem image each time the Sleuthkit command reads the mutated sample. The multiple copies of the image files that contain only a few changed bytes are avoided to prevent disk space exhaustion and focused on improving the maximum utilization of computing resources of CPU cores.

5.2. LD_PRELOAD as a Hooking Mechanism

Dynamic loading is a mechanism of a computer program that can load a library at runtime, i.e. loading the required functions, variables with their addresses onto the memory, execute those functions and unload them from memory after completing the execution of the program [37]. Preloading is a feature of the dynamic linker in most of the UNIX systems, and it is used to link the user-specified, shared library before any other shared library is linked to the executable. During the application runtime, the preloaded library implements those functions that a programmer wants to override. Library preloading is commonly used when a programmer wants to customize functions in the library to be called. This is possible only when the application is using any of the dynamically loaded libraries, as no other functions can be overridden if they are called from a static library or if an application itself implements it.

LD_PRELOAD is an environment variable that is used by the dynamic linker of the operating system, and it is used to preload the shared libraries that are needed by a program to execute [38]. It allows us to bind symbols (such as functions, variables, or structures) from the user-specified shared library before functions from standard C library are linked. Shared libraries are files with .so extension and are referenced by programs at runtime [39]. LD_PRELOAD
functionality is not just for completely replacing the system calls but also used to keep the original symbols within our custom function (known as system call wrapping) and make the malicious code injections look transparent to the program. Consider a sample of code below where the original function is wrapped within the custom read function.

```c
#define GNU_SOURCE
#include <dlfcn.h>

typedef ssize_t (*real_read_t)(int, void *, size_t);

ssize_t real_read(int fd, void *data, size_t size) {
    return ((real_read_t)dlsym(RTLD_NEXT, "read")(fd, data, size);
}

ssize_t read(int, void *, size_t size) {
    ssize_t amount_read;

    // Perform the actual system call
    amount_read = real_read(fd, data, size);

    // Our malicious code
    fwrite(data, sizeof(char), amount_read, stdout);

    // Behave just like the regular syscall would
    return amount_read;
}
```

Figure 8. An Example of LD_PRELOAD mechanism for kernel-level hooking

The dlfcn library is a standard C library that provides an API for loading the shared libraries dynamically at the program runtime. The dlsym function in dlfcn library is used to load the symbols (functions and variables) of the shared object or the executable at runtime. In the above code, we first retrieve the original read function “read” using the dlsym function by
loading the corresponding symbols of read(). The RTLD_NEXT macro is also added to the call, which helps to find the requested function stored next to the default one in the order of the linker’s load stack. In this case, RTLD_NEXT points to the original definition of the read() from standard libc. Once the original function is retrieved, it is called within our custom read(). Upon execution, we expect the system call to return the data in the amount_read. But the program returns the values of custom read function even before returning the original data. This is because the code is compiled to a shared library and is preloaded by setting the path of LD_PRELOAD to the path of the object file, and so whenever the read system call is invoked, it is being hooked by the dynamic linker to overwrite the system call implementation.

5.3. TSK Fuzzer development

5.3.1. Design overview

The fuzzing harness of the TSK fuzzer simulates the fuzzing harness in Gaslight [3]. The harness generates fuzzing instances to the Sleuthkit framework. The fuzzing harness loads mutations from the mutations queue and applies them to different instances of the TSK command. The mutations program is a shared library that is activated by setting the LD_PRELOAD environment variable to the path of the library. Once loaded, the custom implementation of the system calls are used.
5.3.2. Fuzzer Functionality

The fuzzer works in two phases. First, in order to properly mutate specific reads, the fuzzer keeps track of the offsets from where and for how many bytes a specific Sleuthkit invocation reads inside the image file when it is not being fuzzed. This data is recorded to a log file “record_file.” In the second phase, the fuzzer can operate in one of two modes. The first mode attempts to apply mutations during one of the command reads to simulate filesystem smearing or malicious tampering. In the second mode, the mutations are applied on all the command reads starting from the read_offset until it reaches the last byte of the current offset in the read call. The window size is the byte length for which data is mutated upon each read operation of the command, and the fuzzer applies mutations across all the offsets for given window size and for a given read number. This allows an extensive coverage of fuzzing operations as the mutations are enforced for every possible byte sequence of total bytes over a read operation.
Figure 10. Two-Pass Comprehensive Algorithm for computing and mutate offsets

We generated the following combinations of mutations targeting the data in the file system and metadata structures.

1. Fill the buffer with 0x00 address values
2. Fill the buffer with 0xff address values
3. Fill the buffer with randomly generated byte values
4. Fill the buffer with the current byte value plus right shift by 2

The mutations set stored in queue mimic metadata overwrites and file system smearing. First, second and third patterns of mutations write all zeros, 0xff values, and randomly generated bytes that mimic filesystem pointers referring to invalid data units as when a file is deleted, but the data unit might still have the file contents. These mutations also simulate a suspected wiping tool that creates duplicates of data units by randomly writing to the disk or may fill up all zeros
before they are even allocated. The last set of mutations buffer simulates a complex scenario in
the metadata category where malware manipulates the metadata pointers pointing to a valid
logical file system address but the content in the data unit might be different due to file system
smearing. Filesystem smearing is the data inconsistency which usually during the disk data
acquisition and so filesystem metadata appears to be different from that of original disk image
data.

5.3.3. Implementation

The fuzzing harness monitors the forensic framework being analyzed by first monitoring
the number, offset, and size of each read operation performed by a Sleuthkit invocation while it
is not being fuzzed. It then runs a fuzz operation that mutates each read of file system related data
on the boundary assigned as the window size. For each fuzzing operation generated, the
following options are passed as environment variables by the fuzzing harness to make the fuzzer
execute the custom functions defined in our shared library. The fuzzer looks for these
environment values on each read and performs mutations for each fuzzing instance of the TSK
framework.

FUZZER_WHICH_OPERATION, set to 1 or 0
FUZZER_WHICH_MUTATION, mutation index from the queue
FUZZER_WHICH_READ, read number of the total reads
FUZZER_WHICH_OFFSET, set to the current offset
FUZZER_WHAT_LENGTH, length of the bytes to mutate

As a TSK invocation begins to parse the disk image, it will first start by calling the open
function used by the Sleuthkit. Below is the code executed by our custom open function during
the command runtime. We retrieve the original definition of open from the standard library using
The `dlsym` function. The `dlsym` locates the runtime address of the native open and once it is retrieved, it is wrapped within the custom implementation. The native open function returns the file descriptor relating to the path of the filesystem image. Our `start_fuzz_tracking` compares the pathname of the current open call to that of the memory sample, and if they match, an internal tracking array is populated to monitor future read() calls.

```c
int open (const char *pathname, int flags, ...) {
    va_list args;
    mode_t mode;
    int fd;

    // original syscall
    if (!func_open)
        func_open = (int (*) (const char *, int, mode_t)) dlsym (REAL_LIBC, "open");

    va_start (args, flags);
    mode = va_arg (args, int);
    va_end (args);

    fd = func_open (pathname, flags, mode);

    printf("MUTATIONS: open() file '%s' (fd=%d)\n", pathname, fd);
    start_fuzz_tracking(fd, pathname);

    return fd;
}
```

Figure 11. Overriding the open() system call.

The read function is made to function in a similar way to that of the open function call.

The read function is retrieved using the `dlsym` function. The original read() is then called within our custom read function. When the read system call is invoked, it is expected to return the data read from the image with no mutations applied. Since the mutation library is compiled as a shared library and loaded by LD_PRELOAD before any other library, the dynamic linker hooks the original read function to return the values of our custom read(). By doing so, the custom read function can successfully perform mutations by
calling fuzzer_handle_buffer() during each read operation of the TSK command. The 
fuzzer_handle_buffer() returns the read buffer with proper mutations applied.

```c
ssize_t read(int fd, void *buf, size_t count) {
    ssize_t offset;
    ssize_t read_bytes;

    if (!func_read)
        func_read = (ssize_t (*)(int, const void*, size_t)) dlsym (REAL_LIBC, "read");

    read_bytes = func_read(fd, buf, count);

    if (read_bytes > 0) {
        offset = lseek(fd, 0, SEEK_CUR);

        fuzzer_handle_buffer(fd, buf, count, offset - read_bytes);
    }

    return read_bytes;
}
```

Figure 12. Overriding the read()

5.4. Performance gains

The replacement of FUSE [3] with LD_PRELOAD in our current fuzzing architecture is
the key performance boost for two main reasons. First, the removal of FUSE allows for all fuzzer
operations to be performed in userland and without filesystem overhead. The data returned from
the system calls are effectively manipulated by hooking the standard Glibc functions with the
help of the dynamic linker. In doing so, the fuzzer avoids additional processing overhead
involved in switching from kernel to userspace whenever the fuzzer is executed. The other
advantage of LD_PRELOAD is that it does not require root permissions to read or write data,
and thus, can be utilized in standard operating environments where the use of the root user (or
sudo) for common tasks is discouraged and/or disallowed. In contrast, FUSE requires root privileges in order to mount the pseudo-file system.
Chapter 6. TESTING ENVIRONMENT

We tested the current fuzzer on iMac with 32 GB of RAM and 500 GB SSD, and Intel Core i7 processor. The Sleuthkit tools we used for testing are file system tools as they are the most commonly used tools by investigators and are the easy target for the attackers to exploit. File system images were acquired using dd with approximately 128 MB of image size.

6.1. Experimental Results

We implemented fuzz testing against the most commonly used forensic software, called the Sleuthkit framework. The Sleuthkit supports multiple file systems like NTFS, Ext2/Ext3/Ext4, HFS, ExFAT, FAT, and UFS 1/UFS 2. In our current research, we considered to run the fuzzer on some of the file systems as a proof of concept and successfully revealed many implementation errors in NTFS, ExFAT, and Ext4 systems. We acquired one disk image per file system for initial testing to demonstrate the effectiveness and correctness of our fuzzing framework. The errors are further explored in detail by source code debugging with GDB.

Table 2. Tested File Systems and execution results

<table>
<thead>
<tr>
<th>Image type</th>
<th>size</th>
<th>No. of Errors detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTFS</td>
<td>128 MB</td>
<td>7</td>
</tr>
<tr>
<td>ExFAT</td>
<td>128 MB</td>
<td>1</td>
</tr>
<tr>
<td>Ext4</td>
<td>128 MB</td>
<td>1</td>
</tr>
</tbody>
</table>
1. NTFS image for fls: Crashed at line 3029 due to insufficient bounds checking on the size of MFT in ntfs.c.

```c
3029:      attr = (ntfs_attr *) ((uintptr_t) mft +
3030:         tsk_getu16(fs->endian, mft->attr_off));
3031:      data_attr = NULL;
3032:      
3033:      /* cycle through them */
3034:      while ((uintptr_t) attr + sizeof (ntfs_attr) <=
3035:              ((uintptr_t) mft + (uintptr_t) ntfs->mft_rsize_b)) {
3036:          if ((tsk_getu32(fs->endian, attr+len) == 0) ||
3037:              (tsk_getu32(fs->endian, attr->type) == 0xffffffff)) {
3038:              break;
3039:          }
3040:          if (tsk_getu32(fs->endian, attr->type) == NTFS_ATYPE_DATA) {
3041:              data_attr = attr;
3042:              break;
3043:          }
```

Figure 13. NTFS image crash for an improper sanity check

2. NTFS image tested for ils: Crashed when dereferencing fs_attr_cur at line 325 in fs_attrlist.c.

```c
322:         for (fs_attr_cur = a_fs_attrlist->head; fs_attr_cur; fs_attr_cur =
323:             fs_attr_cur->next) {
324:             if ((fs_attr_cur->flags & TSK_FS_ATTR_INUSE) &&
325:                 (fs_attr_cur->type == a_type)) {
326:                 if (((name == NULL) && (fs_attr_cur->name == NULL)) ||
327:                     ((name) && (fs_attr_cur->name) && (!strcmp(fs_attr_cur->name, name)))) {
328:                     /* If we are looking for NTFS $Data;
329:                      * then return default when we see it */
330:                     if ((fs_attr_cur->type == TSK_FS_ATTR_TYPE_NTFS_DATA) &&
331:                         (fs_attr_cur->name == NULL)) {
332:                         return fs_attr_cur;
333:                     }
```

Figure 14. Another NTFS crash for the sanity check
3. NTFS image tested for icat: Crashed at line 202 as the ‘buf’ parameter is too small to accommodate ‘len’ bytes.

```c
200 }  
201  
202 cnt = read(cimg->f0, buf, len);  
203 if (cnt < 0) {  
204     tsk_error_reset();  
205     tsk_error_set_errno(TSK_ERR_IMG_READ);  
206     tsk_error_set_errstr("raw_read: file \"%s\" PRIttocTSK \"\" offset: \%
207    PriOff " read len: \%" PRIusSIZE " - \%", raw_info->img_info.images[idx],
208     rel_offset, len, strerror(errno));  
209     return -1;  
210 }  
211 } endif  
212 cimg->seek_pos += cnt;  
213  
214 return cnt;  
215 }
```

Figure 15. NTFS image crash for boundary check

4. ExFat tested for ils: Crashed as there is no size check on block count. We fuzzed the block count with a vast amount of random data and so the bitmap pointer points to invalid data units.

```c
/* Allocate a bitmap to keep track of which sectors are allocated to * directories. */  
if ((dir_sectors_bitmap =  
    (uint8_t*)tsk_malloc((size_t)((a_fs->block_count + 7) / 8))) == NULL) {  
    tsk_fs_file_close(fs_file);  
    return 1;  
}
```

Figure 16. ExFAT image crash for ils
5. NTFS tested for fls: Crashed due to insufficient sanitation check for idxalloc_len parameter when tsk_malloc function is called at line 1009.

```c
/* Copy the index allocation run into a big buffer */
idxalloc_len = fs_attr_idx->nr_d.allocsize;
if ((idxalloc = (char *) tsk_malloc((size_t) idxalloc_len)) == NULL) {
    return TSK_ERR;
}

/* Fill in the loading data structure */
load_file.total = load_file.left = (size_t) idxalloc_len;
load_file.cur = load_file.base = idxalloc;
if (tsk_verbose)
    tsk_fprintf(stderr,
        "ntfs_dir_open_meta: Copying $IDXALLOC into buffer\n");
```

Figure 17. NTFS memory allocation issue

6. NTFS tested for fls: Crashed at line 2824 as tsk_malloc is called without the boundary check on the size of MFT. So, the malloc function allocated large amounts of memory for all the mutated bytes generated for fuzzing the MFT size.

```c
// see if they are looking for the special "orphans" directory
if (mftnum == TSK_FS_ORPHANDIR_INUM(fs)) {
    if (tsk_fs_dir_make_orphan_dir_meta(fs, a_fs_file->meta))
        return 1;
    else
        return 0;
}

if ((mft = (char *) tsk_malloc(ntfs->mft_size)) == NULL) {
    return 1;
}

/* Lookup inode and store it in the nfts structure */
if (nfts_dinode_lookup(ntfs, mft, mftnum) != TSK_OK) {
    free(mft);
    return 1;
}
```

Figure 18. NTFS crash for no size check on malloc ()
7. NTFS malloc crash: Crashed due to lack of sanity check missed at line 2915. The code does not check the size on fs_attr->size before calling malloc.

    /* Get a copy of the attribute list stream using the above action */
    load_file.left = load_file.total = (size_t) fs_attr->size;
    load_file.base = load_file.cur = tsk_malloc((size_t) fs_attr->size);

    if (load_file.cur == NULL) {
        tsk_fs_file_close(fs_file);
        return 1;
    }

    ntfs->attrdef = (ntfs_attrdef *) load_file.base;

    if (tsk_fs_attr_walk(fs_attr, 0, tsk_fs_load_file_action, (void *) &load_file)) {
        tsk_error_errstr2_concat(" - load_attrdef");

        Figure 19. NTFS malloc () check

8. NTFS image crash for ffind: Crashed again as there is no proper sanitation check at line 2407. The size of fs_attr_attrlist->size is not checked before allocating memory, and so the malloc function is broken as the mutated file is too large to allocate.

    /* Get a copy of the attribute list stream using the above action */
    load_file.left = load_file.total = (size_t) fs_attr_attrlist->size;
    load_file.base = load_file.cur = buf =
    tsk_malloc((size_t) fs_attr_attrlist->size);

    if (buf == NULL) {
        free(mft);
        free(map);
        return TSK_ERR;
    }

    endaddr = (uintptr_t) buf + (uintptr_t) fs_attr_attrlist->size;
    if (tsk_fs_attr_walk(fs_attr_attrlist, 0, tsk_fs_load_file_action,
            (void *) &load_file)) {
        tsk_error_errstr2_concat(" -- processing attrlist");
        free(mft);
        free(map);
        return TSK_ERR;
    }

    Figure 20. NTFS crashed for improper memory allocation
Chapter 7. CONCLUSIONS AND FUTURE WORK

The current research made an initial attempt from the antiforensics perspective to make the forensic tools more reliable and possess high-end defensive mechanisms to overcome the challenges put forward by automated forensic tools and techniques. As of today, most of the forensic tools used by the analysts are highly automated. The tools must require sufficient validation and continuous testing to improve their robustness in the collection of forensic evidence. When conducting digital forensic investigations, the evidence extracted by the tools is expected to be reliable with the increasing complexity of antiforensic techniques.

We designed and developed the dynamic fuzzing framework for the Sleuthkit tools in a way that it can be used for any file forensics framework. One of the primary goals of our fuzzer implementation is to perform kernel-level hooking of functions for dynamic fuzzing of the inputs. The fuzzer successfully incorporated the library preloading mechanism by exploiting the functionality of LD_PRELOAD to load our fuzzer injected code as a shared library. In doing so, the fuzzer has overcome the processing overheads involved in switching from kernel to user boundaries, and hence, the goal of effectively performing millions of test runs for each instance of the forensic framework is achieved. The fuzzer is useful in exposing millions of copies of mutated filesystem samples to each instance of the TSK framework without copying the entire original sample each time it is induced with mutations. The testing framework revealed several implementation issues for NTFS, Ext4, and ExFAT file system parsers of TSK. Most of the issues we encountered are of improper memory allocations without sufficient bounds checking.

The testing framework can further extend its scope of testing on other file systems that Sleuthkit supports besides the three currently targeted file systems. We chose to perform our initial testing on NTFS, ExFAT, and Ext4 as these are the most commonly used file systems. We
also plan to test other command-line tools in the Sleuthkit thoroughly as we currently targeted only the tools available for the filesystem analysis. As the fuzzer architecture is designed to test any file forensics framework without having to modify the source code, we plan to test other forensic tools besides Sleuthkit as we focus on tuning our current set of mutations more effectively to apply for various filesystem image samples. Once we test the fuzzer for extensive coverage of file systems and other forensic software, we are looking forward to seeing this as a distributed implementation of fuzzing to perform continuous testing on any forensic software instead of testing it for a specified time.
REFERENCES

[1] “The digital age can be characterized as the application of computer technology as a tool that enhances traditional methodologies,” vol. 1, no. 3, p. 12, 2002.


VITA

Shravya Paruchuri was born on May 3, 1996, in Nizamabad, India. She then moved to Hyderabad, India, for schooling and received her high school education in Vignan Vidyalayam. She received her bachelor’s degree in Computer Science and Engineering from GITAM University, India, in April 2016. She later joined Louisiana State University, Baton Rouge, to pursue her master’s degree in Computer Science, taking Applied Cybersecurity as her specialization. She started her research on Digital Forensics and fuzz testing under the guidance of Dr. Golden G. Richard III and expects to graduate in December 2019.