Gaslight Revisited: Efficient and Powerful Fuzzing of Digital Forensics Tools

Shravya Paruchuri

Division of Computer Science and Engineering, Louisiana State University

Andrew Case

Volatility Foundation

Golden G. Richard III*

Center for Computation and Technology and Division of Computer Science and Engineering, Louisiana State University

---

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. This rise has coincided with a significant and continued increase in the size, complexity, and number of digital forensic investigations that must be performed. In the past, such 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. Given the central role that such evidence plays in securing organizations and nations against attacks as well as in criminal and civil legal proceedings, it is necessary that such systems are developed in a robust and reliable manner. In this paper, we present our effort to develop a stress testing platform specifically tailored to assess the robustness and reliability of digital forensics tools. For our initial testing, we chose to target The Sleuth Kit framework given its prominence as both as a standalone tool as well as a programming library that is utilized by a large number of open source and commercial filesystem analysis systems. The results of our efforts were the automated discovery of many critical programming errors in The Sleuth Kit framework.

Keywords: memory forensics; digital forensics; fuzzing; software testing;

*Corresponding author

Email addresses: shravyaparuchuri@gmail.com (Shravya Paruchuri), andrew@dfir.org (Andrew Case), golden@cct.lsu.edu (Golden G. Richard III)

incident response

1. Introduction

The prevalence of cybercrime and attacks, including ransomware, insider threats, intellectual property theft, and espionage is well known and well documented [1, 2]. The scale and severity of these attacks has led to a significant increase in the relevance and utilization of the fields of digital forensics and incident response. Practitioners in these fields are responsible for detecting, responding to, and documenting hostile actions taken against an organization's infrastructure and property by both malicious insiders as well as malicious actors anywhere in the world. To achieve these goals, practitioners must collect and analyze all potentially relevant digital evidence to determine if any artifacts that support the investigation can be found. Given the continued increase in the size of digital data, such as hard drive capacity, volatile memory (RAM) sizes, and network activity, it has become impractical to apply traditional, manual approaches to digital forensic analysis [6, 13, 21]. Instead, practitioners now heavily rely on automated tools and frameworks to identify, extract, and present relevant data from digital evidence sources.

While automation allows for scalable and repeatable analysis, it also requires special care in the design and development of digital artifact extraction frameworks as well as between frameworks and user interfaces. Since these frameworks are used in critical situations, such as legal proceedings and analysis after intrusions, they must clearly report accurate results. They must also be as resistant as possible to invalid and corrupt metadata and data streams, whether the malformed data was due to acquisition errors or staged by malware. The most common type of acquisition error in digital forensics is known as smear [9]. Smear occurs when data is acquired from running systems and metadata and its associated data stores change between when acquisition starts and ends. The effect of smear is that, while a majority of the copy of an acquired source is accurate, some portions may be corrupted. This places the burden on the forensic analysis software to recover as much of the intact data as possible while gracefully handling the corrupt data. Depending on the programming language(s) used to develop the forensic framework, improper handling of smear can lead to exploitable programming bugs, infinite loops, memory and filesystem exhaustion, and other critical issues that can cause the program to crash or be vulnerable to control by malware. Mishandling of malformed data can also lead to improper reporting of artifacts, which can mislead investigators and potentially invalidate investigations.

The acquisition of volatile memory is an obvious source of smear as, with the exception of virtual machine environments and hibernation files, the memory sample must be acquired while the system is live [12, 22, 23]. A less obvious source of smear is during disk acquisition as the traditional approach to disk acquisition was to power down the system and then acquire the drives offline in a lab setting. This approach has become untenable in many real-world situations,

however, due to several factors. These factors include the size of the drives to be acquired, an inability to shut down production systems for hours or days while acquisition occurs, and the ubiquitous use of complicated RAID setups that can be trivially acquired live but that are often very difficult to replicate offline without access to the original RAID hardware and software configuration. Also, even in situations where disk images can be acquired offline, smear can still occur due to hardware errors.

The critical importance of the accuracy and robustness of digital forensic tools has led to several efforts to test different frameworks' resiliency to malformed input. A recent effort in this regard was a publication at the 2017 Digital Forensics Research Workshop (DFRWS) that focused on fuzzing popular memory forensic frameworks [10]. This fuzzer, named Gaslight, was able to automatically find and report many programming errors in the tested frameworks' parsing of in-memory artifacts.

The research effort described in this paper was inspired by Gaslight and was focused on specifically testing filesystem forensic tools. In particular, we focused on fuzzing of The Sleuthkit framework (TSK) as it is one of the most commonly used file forensic frameworks [8]. TSK is a set of command line tools that allow for deep parsing of filesystem metadata and contents as well as a library used by numerous commercial products and open source projects to parse filesystem data. This makes it an attractive target for attackers, and a high value target for our fuzzer to locate programming errors and help lead to them being reported and fixed. In this paper, we document our effort to redesign and re-implement the Gaslight architecture as well as add many new features that enhance its usability and usefulness. As will be shown, our new fuzzing engine discovered numerous programming errors in TSK that were efficiently and automatically discovered by our new framework.

2. The Original Gaslight Architecture

As described in the 2017 paper, Gaslight had several key goals:

- Support fuzzing of both open- and closed-source memory forensics tools, without requiring modifications to the framework itself.
- Fuzz memory forensics tools written in any programming language.
- Fuzz as quickly as possible, using all available computing resources.
- Intelligently discover and report a variety of implementation errors for memory forensics tools, including crashes, infinite loops, and resource exhaustion issues.

Gaslight also had a goal of not making any changes to the memory sample being analyzed and instead at runtime dynamically altered the data a framework read from a memory sample. Given the size of modern memory samples, having to make a new copy for each mutation would have been highly impractical.

Furthermore, Gaslight only mutated the bytes accessed by a framework during the particular read operation on which it was accessed. By only mutating the data processed by a framework the entire workflow became highly efficient and targeted.

To accomplish these goals, Gaslight leveraged a custom FUSE [16] filesystem that intercepted filesystem operations by a memory forensic framework to mutate (change) the data in an attempt to trigger programming errors. FUSE allows the implementation of a filesystem in userland, and as such, is much simpler to program than a kernel-level filesystem driver. By storing a memory sample to be analyzed under a Gaslight FUSE mount point, the custom FUSE driver was able to monitor and intercept all filesystem operations by a memory forensic framework.

Gaslight's automation harness was operated by running a tested framework in two phases. First, a particular memory analysis plugin was executed with static command line arguments, and then the order and offsets of *read* operations were recorded. No mutations were applied on the first run. The second phase involved running the same command line invocation once for each read operation as well as each active mutation type. The framework also allowed for a sliding window to fuzz particular offsets inside of buffers read from the memory sample.

The end result of this two-phase approach was that every read performed and every offset processed by the framework would be fuzzed with every active mutation type. As previously mentioned, the use of Gaslight led to the discovery of many programming errors in memory forensic frameworks.

3. Weaknesses of the Original Gaslight

While Gaslight is an innovative framework, the current version suffers from several issues that negatively impact performance, practicability, efficiency, and the ability to detect certain programming errors. We describe these drawbacks in this section, and the remainder of the paper describes our effort to redesign and expand upon Gaslight as well as our results.

3.1. Reliance on FUSE

The most significant drawback in the original Gaslight architecture is the reliance on FUSE as the filesystem operation interception mechanism. FUSE is problematic for several reasons. First, as documented in two recent papers, FUSE causes extreme filesystem performance degradation as well as CPU utilization spikes [7, 25]. In the worst cases, these papers document the use of FUSE causing an extra CPU usage of over 30% and over 80% performance slowdowns in filesystem operations. Considering that Gaslight must run a tested framework many thousands or even millions of times, this performance issue makes relying on using FUSE quite inefficient.

FUSE is also difficult to use in practice as it requires root (or sudo) privileges to mount the custom filesystem. In shared work environments, this is a potential security hazard as well as a non-standard requirement for security testing tools.

Finally, the reliance on FUSE greatly limits the detection capabilities of the fuzzer as it can only monitor filesystem operations. As discussed in Section 5, our newly developed fuzzing architecture monitors other runtime operations, which led to finding several programming errors in TSK that the original Gaslight would be unable to detect.

3.2. Non-Specific Fuzzing Mutations

The next major drawback to Gaslight is that it employed over 25 mutations, many of which had overlapping coverage and that did not expand the detection capability of the framework. Having so many mutations also furthered harmed performance as they all must be tested on each fuzzing operation.

4. The Redesigned Gaslight Architecture

Our goal with the redesign of Gaslight was to keep the portions that were successful, redesign and replace the insufficient components, and to develop completely new capabilities that enhance detection of programming errors as well as aiding framework developers with ongoing testing. Our research effort began with addressing the issues previously described in the original Gaslight architecture. To start, we designed a system that could replace FUSE with *LD_PRELOAD*, and then we developed specific mutations that targeted digital forensic tools in an efficient manner.

4.1. Leveraging LD_PRELOAD

4.1.1. Background

LD_PRELOAD is a debugging feature of the dynamic linker implemented on Linux systems that allows a shared library to intercept library function calls at runtime [14]. In practice, this means that a shared library can monitor and change the implementation and return values of functions in other libraries. A common debug use is to print the arguments sent to functions to determine what parameters they are being sent and how they are processing the data. Given its power over the system runtime, this feature has also been heavily abused by real-world malware starting in the mid-1990s [15] through the present [11, 20].

4.1.2. Supporting Fuzzing Operations

Our new Gaslight architecture uses *LD_PRELOAD* to hook filesystem and memory allocation operations. Unlike FUSE, *LD_PRELOAD* has no discernible overhead since it adds just one function call, the one implemented by the hooking library, into the existing function call stack.

To monitor filesystem operations, Gaslight hooks the *open*, *read*, and *close* system calls. This allows it to monitor when a disk image is accessed by a forensic framework as well as count the number of reads performed. To monitor memory allocations, the *malloc*, *realloc* and *calloc* functions are monitored. As described later, this allows us to find a wide class of programming errors in

```
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 1: Gaslight's open hook

frameworks. Figure 1 shows the *open* hook used by Gaslight to track when the disk image is accessed.

Each Gaslight function starts by locating the address of the original implementation through the call to *dlsym*. This is a standard approach for all applications that leverage *LD_PRELOAD*. After locating the original *open* address, the real implementation is called. The path and return value are then sent to Gaslight's *start_fuzz_tracking* function, which ensures that only read operations on the configured disk image will be monitored and not those of other files, such as configuration files. The file descriptor returned by the real open function is then returned to the calling code.

Figure 2 shows the high-level design of our new framework that leverages *LD_PRELOAD* as its interception mechanism. Our newly designed architecture removes the need for users to be root, significantly improves performance, and allows a wide range of operations to be monitored.

4.2. Tuning Gaslight's Mutations

After studying the original Gaslight's set of mutations and mapping them out to the effects such changes would have on frameworks programmed in a variety of languages, we decided to significantly reduce the number of mutations used by our framework. By using a smaller number of highly effective mutations, we were able to find programming errors in a more efficient and performant manner. The final set of mutations that were chosen filled mutated buffers with the following:

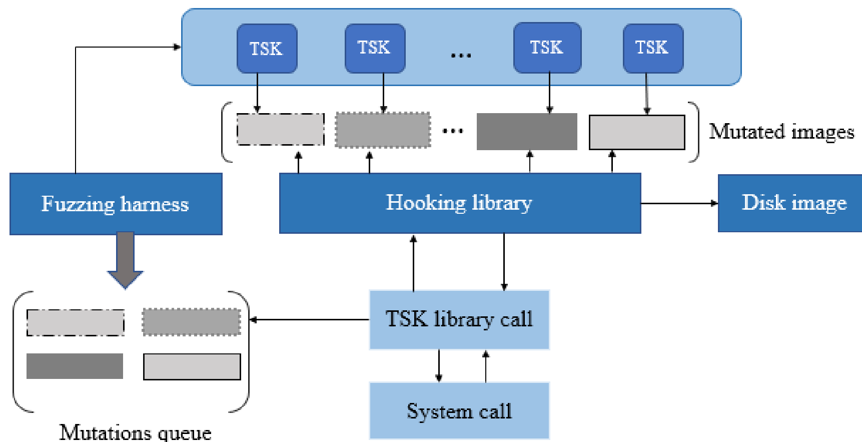

Figure 2: Gaslight Redesigned

1. All NULL (0x00) values
2. All hex 0xff values
3. Randomly generated values
4. The current byte value from the disk image shifted right by 2
5. The current byte value from the disk image shifted left by 2
6. The current byte value from the disk image logical XOR'd with 0x80

Mutations 1-3 were kept from the original Gaslight architecture due to their effectiveness and close mirroring of the real-world effects of smear. Many acquisition tools fill unreadable buffers with either NULL or 0xff bytes as padding. Randomly generated values provide good coverage of a tool's robustness and also mimic how smear or malware can place any value in any location desired.

Mutations 4 and 5 were developed by our research team as they help to trigger serious errors that the original Gaslight could have only potentially triggered through its random generation mutation. Mutation 4 and 5 mimic metadata, such as bitmasks and size fields, being slightly changed during acquistion. It also breaks brittle filesystem parsing code that does not properly handle unknown values in bitmasks. Mutation 6 flips the sign of the integer value being examined. This can trigger a variety of programming errors as large values become small and small values become extremely large. It also helps to find signedness mishandling issues in C and C++ applications.

5. Enhancing Operations and Usability

Beyond redesigning Gaslight to be more efficient and scalable, we also added three significant features that were not present in the original Gaslight architec-

ture and whose absence greatly reduced usability and error detection coverage.

5.1. Monitoring Memory Allocations

Frameworks implemented in C and C++ must manage their own memory allocations and deallocations. This complex task has historically led to many exploitable programming errors in a wide variety of programs and frameworks. TSK, along with other open source and commercial filesystem parsers, are implemented in these languages and are potentially vulnerable to the same implementation issues. Parsing filesystem data requires allocating memory based on size and length fields encoded within the filesystem's metadata. This means that values are fully controllable by malware as well as being potentially affected by smear. Forensic frameworks, such as TSK, must take extreme care when using such values for allocations and subsequent operations as there are number of programming errors that can lead to heap-based buffer overflows, integer underflows and overflows, memory exhaustion attacks, and more. Beyond exploitable vulnerabilities, crashes in forensic frameworks also have other negative affects such as program crashes, automation engines breaking, and partial results being returned from analysis, which is often unnoticed by the analyst.

The original Gaslight architecture is only capable of detecting a limited set of memory allocation issues, specifically, those that cause the application to crash upon using too much memory or a failed allocation. This set of circumstances is very limited, as these typically occur on modern systems only when a system completely exhausts available RAM and swap space. One exception is if applications have been configured to use only a subset of system RAM, but this is not a common configuration for forensic tools in real-world labs. Similarly, fuzzing engines are generally run on powerful systems with large amount of RAM and CPU cores so that as much coverage can be completed as quickly as possible. This masks the fact that applications may be using very significant amounts of memory without crashing due to the system being used for fuzzing being extremely powerful.

To detect potential memory exhaustion vulnerabilities in forensic frameworks, we implemented *LD_PRELOAD* hooks for the *malloc*, *realloc* and *calloc* memory allocation functions. Before calling the real allocation function, our hooking code checks to see if the memory size requested by the framework is more than 64MB. If so, an error log entry is generated and the program is aborted. This maximum allowable size is configurable, but we found 64MB to be the ideal size based on our testing. By enforcing a maximum allocation well beyond what filesystem parsers would normally make, our framework is able to detect potential memory allocation and exhaustion vulnerabilities on its own, independent of how much RAM the system it is being executed on has installed.

5.2. Reusable Crashes

The original Gaslight architecture provided no mechanism to "replay" crashes nor to use the configuration that caused a crash for future testing purposes. To alleviate these issues, we added several features to Gaslight's fuzzing harness

that not only recorded the exact configuration variables used for each invocation, but that also recorded the exact mutated data stream. This was particularly important for the random generation mutation as otherwise there was no information to recover the original values substituted by the fuzzer engine.

We then developed a script that would read a saved state file and only run the fuzzer engine with the exact crash state reproduced, including the original mutation data for the affected read operation. This allows developers to reproduce crashes in their frameworks, including inside a debugging environment where the application state can be assessed.

5.3. Replayable Crashes

A top priority of our research effort was to remove the need for developers to keep an entire disk image around to re-trigger a previous crash in the future. Instead, we wanted to capture the entire data stream that caused a crash and allow it to be usable (replayable) without the original disk image present. To accomplish this, we developed a special operation mode of our redesigned Gaslight that will read a saved state file and then produce a second file which includes the contents of the data read from disk for each read operation before the one that caused the crash. For the read operation that caused the crash, the mutated buffer inside the saved state file is used to populate the new file. The end result of this special operation mode is that at any point in the future the fuzzing engine can recreate the exact state and set of data that caused a framework to crash in the past. This mode will be useful to developers who wish to implement regression testing or who will be testing a wide variety of frameworks for correctness and robustness.

Besides being useful for regression testing, replayable crashes also conserve disk space. In our testing, the majority of TSK commands required less than one hundred (100) read operations. The worst case commands, such as parsing the metadata of all files in the filesystem, required around four hundred (400) read operations on average. TSK uses a default read size of 65,535 (65KB). Taken together, this means that even in the worst case scenario of a crash occurring on the 400th read of 65KB blocks, the resulting crash file would 26MB whereas the full filesystem partition might be very large, with even single consumer grade hard drives exceeding 12TB as of late 2019. For several of the programming errors triggered by our framework, the crashes occurred within the first ten reads and the associated replayable crashes are less than 1MB in size.

We envision publishing replayable files alongside our code library once all the errors that we found in TSK are patched and included in a stable release. We also envision that this library of crash streams will significantly grow over time and can be used by all open source and commercial vendors for robustness testing of their software, which will aid the entire digital forensics community.

6. Fuzzing Environment

6.1. Hardware Setup

Our fuzzing framework does not have a discernible impact on the amount of memory, CPU, or disk space used beyond that of the framework being tested. As discussed previously, the size of the generated crash state logs and other logging information is generally in the tens of megabytes. The memory footprint is only a few data structures used to track the read operations of the framework being tested and which mutations the fuzzer is currently configured to generate.

During the main testing periods, the fuzzer ran on a bare-metal system configured with 64GB of RAM and a 6 core/12 thread Intel CPU.

6.2. Software Environment

The 64-bit version of Ubuntu 18.04 TLS was used as the operating system for all tests. All core system runtime libraries and the kernel used were those provided by the Ubuntu software manager.

6.3. The Sleuthkit Configuration

Testing was performed against the latest version of TSK from its official GitHub repository. TSK is actively developed, and as such, relying on periodic releases would potentially lead us to triggering bugs fixed since release, which does not align with our goal of finding undiscovered bugs to help aid the project. Furthermore, by compiling TSK ourselves, we were able to compile with source code debugging enabled, which greatly sped up crash triage. During the testing phase, we updated to the latest code base once a week and recompiled the library and tools. This kept us fully up to date with the project.

6.4. Test Image Generation

To test TSK with our fuzzing architecture, we created test images of several filesystems, including NTFS, HFS+, ExFat, and Ext4. Each image file was 256MB in size and created using *dd* with an input source of */dev/zero*. For our architecture, the size of the disk image does not affect the total time needed to fuzz a forensic framework task. Instead, the main factor related to time needed to run an entire fuzzing operation is the particular task being performed by the forensic framework. As an example, operations that read the metadata of a single file, such as through TSK's *istat* command, only require reading the metadata of a single file and do not require a significant number of read operations. On the other hand, gathering the metadata of every file in the filesystem through TSK's *fls* can require a significant number of read operations if the filesystem contains many files and directories.

After creating each image, we then mounted it and instructed a script we developed to create a number of files, directories, and sub-directories inside the image. It also creates symbolic links for filesystems that support it and alternate data streams for NTFS images. The goal of the script is to produce varying types of files and metadata throughout the filesystem to trigger as many code paths as possible in the filesystem parser.

Function	Programming Error
ntfs_load_bmap	Insufficient bounds checking of attribute before processing leads to crash
tsk_fs_attrlist_get	Insufficient bounds checking of attribute list before processing leads to crash
ntfs_proc_compunit	Insufficient bounds checking of compression unit before processing leads to crash
ntfs_dir_open_meta	The 64-bit size of the index allocation attribute is not checked before using the size directly in a call to malloc
ntfs_proc_attrlist	The 32-bit size of MFT entries is not checked before using the size directly in a call to malloc

Table 1: Discovered NTFS Parser Errors

Function	Programming Error
fatfs_inode_walk	malloc is called with unchecked size when allocating memory to hold the directory bitmap

Table 2: Discovered ExFat Parser Errors

Function	Programming Error
ext2fs_block_getflags	Use of unchecked mutated offset leads to invalid memory access and program crash
ext2fs_dinode_copy	Use of unchecked mutated offset leads to invalid memory access and program crash

Table 3: Discovered Linux Ext Parser Errors

7. Fuzzing Results

We tested our developed fuzzing architecture against the previously mentioned disk images and found many critical programming errors in TSK's filesystem parsers. Table 1 describes these errors found in the NTFS parsers, while Table 2 describes the errors found in the ExFat parser, and Table 3 describes the errors found in the Linux Ext parsers.

As can be seen, a variety of programming errors were uncovered during

our testing. Combined, these errors allow for crashing any program using the affected TSK parsers, memory exhaustion attacks, and potentially arbitrary code execution inside of a victim application.

8. Related Work

8.1. Fuzzing for Security Vulnerabilities

Fuzzing for security vulnerabilities has a long history going back to 1988 when Bart Miller assigned his students the task of fuzzing UNIX utilities [17]. Since then, there have been significant research efforts to develop smart and efficient fuzzing techniques to find software vulnerabilities. Below, we briefly discuss only the most closely related work.

One of the most powerful and commonly used fuzzing tools is American Fuzzy Lop, normally refereed to as AFL [26]. Unfortunately, AFL, along with other similar tools, did not meet the requirements and goals of the original Gaslight architecture nor does it meet our new requirements. In particular, AFL and associated tools require access to the source code of an application to fuzz it. Gaslight was designed to be tool and programming language agnostic and to not require any inspection or access to a fuzzed framework's source code. Since the majority of digital forensic practitioners use commercial forensic tools to which they do not have source code access, they are unable to leverage AFL to test the robustness of the tools on which their investigations rely. Another drawback of AFL as it relates to testing forensics tools is that it mutates entire files that are tested. To help with performance in mutating entire files, the AFL documentation recommends that input files be 1KB or smaller, which makes it completely unsuitable for fuzzing software that processes disk images, memory samples, and most other forensic artifacts. Gaslight supports efficient fuzzing of input files of any size by only targeting the portions of the input file processed by a framework in a particular invocation. Similar drawbacks and violations of Gaslight's requirements affect other automated testing techniques, such as dynamic taint analysis systems.

8.2. Filesystem Fuzzing

There have been a few efforts to fuzz filesystem parsers. Mangle, a mutation-based file fuzzer developed by Ilja van Sprundel in 2005, was one of the earliest [19]. It worked by mutating portions of a filesystem's header with random bytes. This managed to break several forensic tools at the time, but fuzzing such a small amount of metadata does little to break modern parsers and only provides a fraction of the coverage of Gaslight.

In 2007, Newsham, Palmer, and Stamos presented a fuzzing effort that led to the discovery of several programming errors in TSK and Encase at the time [18]. This work focused on fuzzing partitions tables, NTFS data structures, and several common file formats. Like AFL, this fuzzing framework mutated entire files during its processing and led to untargeted and inefficient operation.

A more recent effort, named Janus, by a team at Georgia Tech led to the discovery of many filesystem driver bugs in the Linux kernel [24]. This project works by leveraging the Linux Kernel Library, which allows encapsulation of the Linux kernel inside a userland process [5, 4]. While Janus, and the more recently published Hydra project [3] by the same team, are highly efficient at finding bugs in the Linux kernel, they do not substantially overlap with Gaslight's goals or functionality. In particular, these projects require source code access to the kernel drivers being examined. Access to the source code not only violates Gaslight's requirement of not relying on a framework's source code, but it also prevents these projects from fuzzing a filesystem not directly supported by the Linux kernel. Gaslight has no such limitation and can effectively fuzz any filesystem type supported by the fuzzing framework being tested. Furthermore, to avoid mutating entire disk images, these projects only mutate the metadata of supported filesystems. This constraint requires that the fuzzer framework's developers write their own filesystem parsers to locate the metadata locations. This adds a significant layer of complexity and requires that the fuzzer developer's code accurately finds all metadata for a particular filesystem. Gaslight again has no such requirements or limitations.

9. Conclusions

In this paper, we have documented our research efforts to design and develop a new fuzzing architecture that is highly effective at finding real-world programming errors in filesystem parsers. Given the reliance on digital forensic tools to solve significant cybercrime investigations, it is of the utmost importance that these tools be reliable and robust. Using our developed architecture, framework developers and users can perform extensive stress testing of a framework's resiliency in the face of corrupt filesystem data. Such corruption is highly common in real-world investigations where systems must be imaged live, where malware is present, and where hardware errors in source media affect the acquisition process.

Our new fuzzing architecture builds on a previous architecture, known as Gaslight, that was highly effective at finding programming errors in memory forensic frameworks. Our research effort began by fixing several deficiencies in Gaslight, such as its reliance on FUSE and its unrefined set of mutations. We then developed several innovative features that allow discovery of memory allocations bugs and that support repeatable testing using previously triggered crashes. The end result of our work is a framework that meets the needs of real-world developers and investigators and which can be used to discover critical programming errors in any tool that ingests filesystem data. To show the efficacy of the work, we targeted the Sleuthkit framework, which is a widely used library and set of associated tools to perform disk image forensics. During testing, our fuzzing architecture was able to automatically find many critical programming errors in the Sleuthkit.

10. References

[1] 300+ terrifying cybercrime and cybersecurity statistics trends. https://www.comparitech.com/vpn/cybersecurity-cyber-crime-statistics-facts-trends/, 2019.

[2] 50 noteworthy cybercrime statistics in 2019. https://learn.g2.com/cybercrime-statistics, 2019.

[3] Hydra: an Extensible Fuzzing Framework for Finding Semantic Bugs in File Systems. https://github.com/sslab-gatech/hydra, 2019.

[4] Linux kernel library. https://github.com/lkl, 2019.

[5] Linux Kernel Library [LWN.net]. https://lwn.net/Articles/662953/, 2019.

[6] J.Grier andGolden G. Richard III. Rapid forensic imaging of large disks with sifting collectors. *Digital Forensics Research Conference (DFRWS).*

[7] E. Zadok B. Vangoor, V. Tarasov. To fuse or not to fuse: Performance of user-space file systems. *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, 2017.

[8] Brian Carrier. The Sleuthkit. https://www.sleuthkit.org, 2019.

[9] Harlan Carvey. Page smear. http://seclists.org/incidents/2005/Jun/22, 2005.

[10] Andrew Case, A. Das, S-J Park, R. Ramanujam, and Golden G. Richard III. Gaslight: A Comprehensive Fuzzing Architecture for Memory Forensics Frameworks. *Proceedings of the 2017 Digital Forensics Research Conference (DFRWS)*, 2017.

[11] chokepoint. Azazel is a userland rootkit based off of the original LD_PRELOAD technique from Jynx rootkit. https://github.com/chokepoint/azazel, 2019.

[12] Volatility Foundation. VMware Snapshot File. https://github.com/volatilityfoundation/volatility/wiki/VMware-Snapshot-File, 2014.

[13] S. Garfinkel. Digital forensics research: The next 10 years. *Digital Forensics Research Conference (DFRWS)*, 2010.

[14] GNU. ld.so. http://man7.org/linux/man-pages/man8/ld.so.8.html, 2019.

[15] halflife. Shared Library Redirection Techniques. http://phrack.org/issues/51/8.html, 1997.

[16] libfuse. The reference implementation of the linux fuse (filesystem in userspace) interface. https://github.com/libfuse/libfuse, 2019.

[17] Bart Miller. Fuzzing Creation Assignment. https://fuzzinginfo.files.wordpress.com/2012/05/cs736-projects-f1988.pdf, 1988.

[18] Tim Newsham, Chris Palmer, and Alex Stamos. Breaking Forensics Software: Weaknesses in Critical Evidence Collection. https://www.defcon.org/images/defcon-15/dc15-presentations/Palmer_and_Stamos/Whitepaper/dc-15-palmer_stamos-WP.pdf, 2007.

[19] OWASP. Fuzzing. https://www.owasp.org/index.php/Fuzzing#File_format_fuzzing, 2019.

[20] Ignacio Sanmillan. HiddenWasp Malware Stings Targeted Linux Systems. https://www.intezer.com/blog-hiddenwasp-malware-targeting-linux-systems/, 2019.

[21] C. Stelly and V. Roussev. Scarf: A container-based approach to cloud-scale digital forensic processing. *Digital Forensics Research Conference (DFRWS)*, 2017.

[22] Johannes Stüttgen and Michael Cohen. Robust linux memory acquisition with minimal target impact. *Digital Investigation*, 11:S112–S119, 2014.

[23] Joe Sylve, Lodovico Marziale, and Golden G. Richard III. Modern Windows Hibernation File Analysis. *Digital Investigation*, 2017.

[24] S. Kashyap P. Tseng T. Kim W. Xu, H. Moon. Janus: a state-of-the-art file system fuzzer on Linux. https://github.com/sslab-gatech/janus, 2019.

[25] A. Moody K. Sato M. Kahn W. Yu Y. Zhu, K. Mohror. Direct-fuse: Removing the middleman for high-performance fuse file system support. *ROSS'18: Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers*, 2018.

[26] Michael Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2016.