ReFuzz — Structure Aware Fuzzing of the Resilient File System (ReFS)

Tobias Groß
tobias.gross@cs.fau.de
Friedrich-Alexander University
Erlangen-Nuremberg, Germany

Tobias Schleier
tobias.schleier@fau.de
Friedrich-Alexander University
Erlangen-Nuremberg, Germany

Tilo Müller
tilo.mueller@hof-university.de
University of Applied Sciences
Hof, Germany

ABSTRACT

The Resilient File System (ReFS) from Microsoft promises new features such as increased performance and resilience compared to the New Technology File System (NTFS). On the downside, the ReFS drivers are growing more extensive and more complex, increasing the attack surface of the Windows kernel. Attackers can often use security-critical bugs in file system drivers to escalate privileges by mounting a file system. In this work, we present *ReFuzz*, a structure-aware fuzzer that uses hardware-assisted code coverage to identify bugs in the ReFS driver. The ReFS file system offers several challenges to fuzzing because first, while ReFS is not documented, it exhaustively uses checksums. Second, the minimal size of a ReFS partition is 2GB, notably decreasing the performance of naive fuzzing approaches.

We demonstrate the effectiveness of our fuzzing approach by finding 27 unique payloads that panic the Windows kernel when mounting or accessing ReFS partitions. Furthermore, we find 162 unique payloads that lead to a system hang-up. Microsoft confirmed those bugs and acknowledged ten unique issues which are security-critical, eight of them allowing remote code execution attacks and got assigned with a CVE number.

CCS CONCEPTS

• **Security and privacy → File system security**; **Software security engineering**.

KEYWORDS

ReFS, File Systems, Kernel Driver Fuzzing, Structure-aware Fuzzing

ACM Reference Format:
Tobias Groß, Tobias Schleier, and Tilo Müller. 2021. *ReFuzz* — Structure Aware Fuzzing of the Resilient File System (ReFS). In *AsiaCCS '22: ACM ASIA Conference on Computer and Communications Security, May 30–June 03, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

ACM ASIACCS 2022, May 30–June 03, 2022, Nagasaki, Japan

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Fuzz testing is the prevalent dynamic analysis technique for finding security bugs in software. In general, fuzzing finds actual bugs, compared to static analysis, which often suffers from an extremely high false-positive rate. Also, fuzzing is more practicable for complex software than techniques like symbolic execution, which have to deal with state explosion, among others.

When software is fuzzed, its input interface is tested by supplying randomly mutated input until the target crashes. Fuzzing can appear in different flavors called *blackbox*, *greybox*, and *whitebox* fuzzing. With *blackbox* fuzzing [12], the fuzzer gets no feedback after supplying the input except the information whether the program crashed or not. Conversely, *greybox* fuzzers [3, 6, 11, 16] monitor the execution path for a given input. That way, the fuzzer can mutate the input more effectively to reach new regions in the program code, ultimately increasing the code coverage. *Whitebox* fuzzing [2, 19, 24] describes a class of fuzzers that can calculate the input to reach a particular area of the target program. Symbolic execution, for example, can be donated as a *whitebox* fuzzing approach. In our work, we extend an AFL-like fuzzer [11], implementing the *greybox* strategy.

Software running with kernel privileges is a worthwhile target for fuzzing since bugs in kernel code often lead to a complete system compromise. Once an attacker can hijack the control flow of the kernel, all security measures of modern operating systems, such as process isolation and user rights, are undermined, leading to *privilege escalation attacks*. Especially drivers are a perfect target for fuzzing as they often read data from outside the kernel, potentially being under attackers' control. Such device drivers are, for example, Ethernet and USB, but also file system drivers run in kernel mode for performance reasons and thus, constitute a perfect target for fuzzing.

Modern file systems like Apple File System (APFS), B-tree FS (Btrfs) [17], and ReFS become more and more complex while implementing new features for reliability and effectiveness. Modern file systems are organized in data structures like B-trees and implement new features like copy-on-write [7, 13, 17]. This complexity leads to a significant code base that cannot be bug-free in general. For example, failures when parsing data structure and allocating or de-allocating memory often lead to bugs that can be exploited. New features like checksums hamper the exploitation of code indeed, but eventually cannot prevent it.

We present *ReFuzz* [rɪfʌz], the first fuzzing framework targeting ReFS, a new file system from Microsoft. Three difficulties led us to implement a structure-aware fuzzer using hardware-assisted coverage guidance: (1) ReFS heavily uses checksums that must be recalculated to match mutated data. Since ReFS stays undocumented

by Microsoft, this step was preceded by extensive reverse engineering. (2) The smallest size of a ReFS partition is 2GB, so we focus on mutating metadata. Furthermore (3), the proprietary ReFS driver cannot be instrumented effectively without hardware assistance due to security measures like Windows PatchGuard [20].

1.1 Threat Model

In our scenario, we assume that an attacker has access to a victim's Windows machine and can mount a custom-crafted Virtual Hard Disk (VHD) image. Today, only some versions of Windows, like Windows Server and Windows Enterprise, can create ReFS partitions. However, all Windows 10 versions, including Windows Home, can mount ReFS VHD images, making the found bugs broadly applicable.

Some of the bugs we found were acknowledged by Microsoft as critical, even enabling code execution. Consequently, mounting a crafted ReFS image can be used by attackers to escalate their privileges, e.g., to get administrator rights on a target machine. Either an attacker tricks the victim directly into mounting such a crafted ReFS image, or bugs in ReFS are used to escalate privileges in a two-stage approach where another piece of malicious software runs on the machine, e.g., an EXE file downloaded from the Internet. Also, bugs in ReFS can be used by insider attacks in multi-user environments, where attackers have user rights on a machine but want to gain superuser privileges to spy on the private data of other user accounts.

1.2 Related Work

Schumilo et al. [18] presented kernel-AFL (kAFL), a fuzzer targeting OS kernels and drivers based on the famous American Fuzzy Lop (AFL) fuzzing engine. kAFL uses the Intel PT technology to extract code coverage information without instrumenting the target kernel at runtime. We used kAFL as the basis for our implementation and enhanced it in multiple ways, e.g., by adding a ReFS-specific input generator and a second input dimension.

Xu et al. [23] were the first who proposed a fuzzing technique that uses input mutations of *two dimensions*. In the first dimension, the fuzzer mutates the file system data. In the second dimension, if the mutation of FS data generates no more new paths, the fuzzer mutates the system calls interacting with the target file system. They used a framework called LibOS, which allows running kernel code in user mode. As a drawback, this is only possible if the source code of a targeted kernel component is available, which is not the case for ReFS. Nevertheless, we adapted the idea of a second fuzzing dimension in our work.

Aschermann et al. [1] enhanced kAFL with a so-called input-to-state engine which can speed up the generation of semi-valid inputs tremendously. This engine called Redqueen can find fixed magic bytes and checksummed data in the input. The engine can automatically calculate the checksums correctly by instrumenting the code and extracting the values the target program expects. In contrast to *ReFuzz*, Aschermann et al. do not mind the problem of significant large inputs. Also, in the current implementation, Redqueen can only fuzz Linux software.

Xu et al. [22] developed a fuzzing framework to detect bugs triggered by data races in file system drivers. They found bugs in two modern file systems, namely in the fourth extended file system (ext4) [4] and Btrfs. In contrast to our work, their fuzzer relies on instrumentation inserted in the driver during compile-time, which is impossible for closed source Windows drivers like ReFS because of PatchGuard [20].

Kim et al. [9] proposed a generic and extensible fuzzing framework called Hydra for fuzzing FS drivers. They developed different building blocks needed in the whole fuzzing process, from input mutators to post-processors. With this approach, they were able to find 157 bugs in various Linux file systems. In contrast to our work, they can only test file system drivers which run in user space or ported drivers when the source code is available.

Song et al. [21] developed *Agamotto*, which can fuzz kernel drivers efficiently with the help of virtual machine checkpoints. Their implementation creates VM snapshots in an elaborated manner during testing the fuzzed payloads, which allows them to reuse kernel states and execute repeating system calls efficiently. This approach gave a performance improvement compared to other state-of-the-art fuzzers. In contrast to our work, they used a modified version of Linux as a target system.

1.3 Our Contribution

In this work, we developed a fuzzer called *ReFuzz* to test the proprietary ReFS driver for Windows 10. In short, we contribute the following:

- We optimized the kAFL fuzzer engine by (1) adding a new fuzzing dimension to kAFL with novel mutation strategies, as described below, and (2) implementing a structure-aware mutation engine for ReFS. Due to the closed-source nature of ReFS, this step had to be preceded by substantial reverse engineering, which we published in previous work [14, 15].
- As a result, we found **27 unique** payloads in the ReFS driver leading to a kernel-panic and **162 unique** payloads leading to a system freeze. We reported all bugs to Microsoft in a responsible disclosure process, and Microsoft fixed them during the January 2022 Patch Tuesday. Therefore we got **eight CVEs** assigned.

We make our findings and the implemented fuzzing framework publicly available at https://www.cs1.tf.fau.de/research/system-security-group/refuzz/ to encourage research and transparency.

2 BACKGROUND

Our fuzzer framework builds upon the kAFL fuzzer and targets the ReFS driver. To understand the additions we implemented, we outline the basics of ReFS (Section 2.1) and wrap up the functionality of kAFL (Section 2.2).

2.1 ReFS

ReFS is Microsoft's file system following the NTFS in many use cases. Currently, ReFS is supposed to be used by servers to handle enormous amounts of data. Although only the Windows Server and Enterprise editions can create it, every Windows 10 edition, including Windows Home, can mount ReFS partitions. Due to the novelty of ReFS, as well as its closed-source nature, the research community has not conducted any publicly available security audits yet.

ReFS is organized chiefly as B+ trees. Exceptions are content data of files and three structures called *Superblock*, *Checkpoint*, and the boot sector. ReFS use different chunk sizes to allocate data. There are sectors, clusters, and pages. A sector is the smallest amount of data addressable on a storage device. Typically a sector consists of 512 bytes. In ReFS, the smallest allocatable amount of data is called a cluster and consists of either eight sectors (4 KiB) or 128 sectors (64 KiB). The B+ trees consist of nodes the size of a page, or more precisely, each node is a page. In the most recent version of ReFS, a page consists of 1 to 4 clusters. B+ trees in ReFS can be seen as tables storing key-value pairs on a higher abstraction level.

An essential feature of ReFS regarding data organization is the usage of address translation. Page references in ReFS use virtual addresses with a few exceptions. The driver needs to translate this address first before accessing the physical page on the storage device. This feature allows relocating data on the disk without adjusting all references. Instead, only the translation table has to be changed. The translation is handled in the granularity of so-called containers containing multiple clusters.

When mounting a ReFS partition, the driver needs to read the cluster and sector size of the file system. This information is stored in the boot sector, which is the first sector of the partition. The driver locates one of the three redundant *Superblocks* and parses it with this information. The *Superblock* contains a self-reference and references to the two *Checkpoints*.

The *Checkpoints* contain references to different root B+ trees. At this point, the driver first locates the required translation tables, which are required to parse the remainder of the file system. The root-level B+ trees again can store references to other trees. The trees themselves are built from nodes of a page size that point to other nodes with the help of references that store the target address and a checksum of the target data.

For more details on the reversed engineered structures of ReFS, please have a look at our published technical report [15] and conference paper [14].

2.2 kAFL

ReFuzz builds upon the kAFL fuzzer, which in turn is based on the famous concepts of AFL, implementing three main components: coverage bitmap, input mutators, and input queue. We detail these components in the following.

kAFL is a *greybox* fuzzer. That means it uses coverage information to influence the input generation process of the mutator. The fuzzer serves the input to the target program —e.g., a kernel driver— and records the executed basic blocks. kAFL uses a hardware feature called *Intel PT* to perform this recording efficiently without instrumenting the target and with little to no performance loss. The code path recording is efficiently stored in a bitmap for each input. If an input executed new basic blocks which no other input executed before, the global bitmap gets updated, and a valuable new input has been found.

The bitmap encodes the code branches taken. Every tested payload gets classified according to the code paths the target executed while processing the input. To illustrate the different classifications, we consider three different execution paths shown at an abstract CFG in Figure 1. Consider input **A** with path 1–2–4, **B** with 1–3–4, and **C** with 1–3–5. All three inputs are considered unique because they use at least one different basic-block transition during execution. Every payload with a unique code execution is queued for being a new starting point for further mutations. An example of non-unique payloads is two equal executions, except one executes a loop only once and the other payload multiple times.

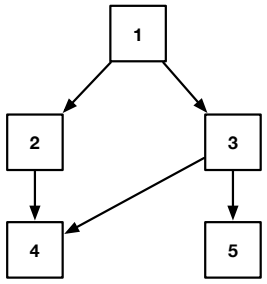


Figure 1: Basic blocks, building an abstract example of a control flow graph.

When starting to fuzz a program with kAFL, the input queue gets filled with at least one provided seed input. This input gets presented to the target program with the help of a helper program called *Agent*, and the returned bitmap is set as the initial global bitmap. After this benchmark, the mutation engine mutates the first seed file. This engine uses different mutators, some of them are deterministic, and others are non-deterministic. For example, the bitflip mutator generates a new input by flipping one bit for every bit present. The splicing mutator is an example of a non-deterministic mutator. It produces new inputs by copping data from one random location of the payload to another random location.

Every tested input will create a bitmap showing the executed code path. If the resulting bitmap shows that new basic blocks get executed from this input, the global bitmap gets updated, and the input is stored in the input queue. All inputs in the input queue will serve as a starting point for the mutation engine to generate new inputs. This way, inputs that trigger the execution of a new basic block will be kept, and others will be discarded.

This feedback-driven input mutation makes the fuzzing process more efficient than mutating the input without coverage information. To use *Intel PT*, kAFL has implemented its custom decoder to interpret created recordings more efficiently, containing the executed jumps. These recordings also allow creating a bitmap from the data. Furthermore, kAFL modified the Quick Emulator (QEMU) and the Kernel-based Virtual Machine (KVM) and implemented different hypercall handlers and the interface to obtain a bitmap. This whole framework allows to fuzz arbitrary kernel components like drivers of closed and open-source operating systems without instrumenting the code. The target OS containing the target component runs in a virtual machine during the fuzzing process.

3 DESIGN CONCEPTS

The goal of *ReFuzz* is to find ReFS image configurations and suitable system calls using the ReFS partition, which triggers unexpected behavior (bugs) that lead to a system crash or freeze. Sometimes these bugs can be exploited to gain kernel privileges as an unauthorized user.

There exist two different kinds of bugs. The first class occurs when mounting the file system, and the bug is triggered without further user interaction. The second class of bugs leaves the driver in a vulnerable state after mounting a malformed ReFS partition, but further user interactions like reading the content of a file or listing a directory's content are required to trigger the bug. Our concept to find these two types of bugs is to mount a mutated ReFS image and access it by a variable set of actions in every fuzzing loop. Therefore, in our concept, two dimensions can be mutated: First, a mutation of the file system, and second, a sequence of file operations that are executed on malformed file systems to trigger kernel bugs eventually.

On an abstract view, *ReFuzz* mutates the ReFS image until the bitmap does not change anymore, i.e., the fuzzer can not find any image that executes new basic blocks. After this stage is reached, the second mutation engine starts, which mutates the system calls that operate on the malformed file system files. With this combination, we can find both types of bug classes.

The main barriers to performing such a fuzzing loop are that a ReFS partition requires a minimal size of around 2GB and that ReFS uses checksums heavily to guarantee data integrity. 2GB of input data constitutes a massive search space for the fuzzer. We tackle this problem by extracting the organizational and metadata from the ReFS image, which decreases the input space to a factor of around 2000 and let the mutators work on this extracted data. Before testing the driver with an extracted and mutated input, it gets expanded over the original image.

As mentioned before, ReFS stores the organizational and metadata in B+ tree nodes. The nodes are linked with references, each consisting of the target address and a checksum of the target data. In the end, ReFS confronts us with chains of checksums that will prevent the driver from mounting the file system when not correctly adjusted to the mutated data. We address this by analyzing the locations of all present checksums on the ReFS image before the fuzzer starts and by correcting the checksums after the mutators altered the input data.

Our second dimension —the file system actions— gets initialized with at minimum one seed. This seed contains the information whichever system calls and parameters should be called after mounting the ReFS image. These calls get mutated after no more new basic block can be reached by only mutating the ReFS image. The system call mutation engine can alter the list of actions by removing or adding calls with random parameters. Additionally, the engine can alter the parameters of existing calls.

In every iteration of the fuzzing loop, two concrete inputs get tested as a tuple. That means a helper program receives altered ReFS metadata and a serialized sequence of actions. The helper mounts the modified ReFS image and executes the specified actions.

4 IMPLEMENTATION

This section details the implemented components and how they interact as a whole framework. In general, there are two main types of systems: (1) A host system that spawns (2) guest VMs as targets for our mutated ReFS images. Figure 2 shows the components present on the host machine and an exemplary guest system. The host executes the master process, which coordinates the mutation

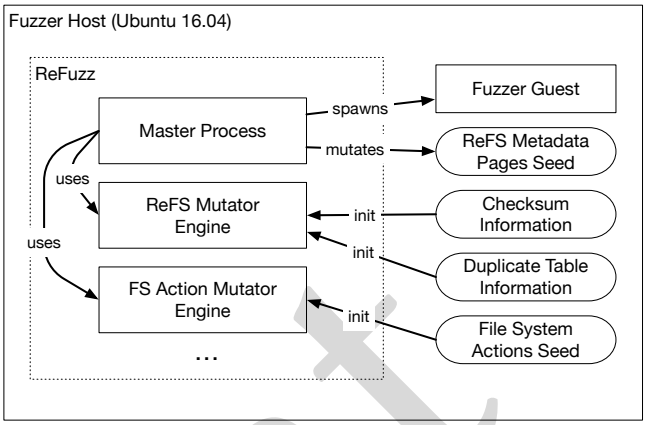


Figure 2: Excerpt of the different components involved in fuzzing the ReFS driver and residing on the host. Stock kAFL components are not shown.

of inputs and the distribution of mutated inputs to the guest VMs. The master process also uses the stock AFL mutators to mutate the "compressed" ReFS image and corrects the checksums with the ReFS mutator engine. For the structure-aware data correction, the engine needs the *Checksum Information* and *Duplicate Table Information*. The "compressed" image is named ReFS *Metadata Pages Seed* in Figure 2. The last component shown is the *File System Actions Seed* file which contains the initial action list executed by the agent after mounting the ReFS image.

The VM executes a specially prepared Windows system based on Version 21H1 (OS Build 19043.1052). It contains preinstalled components for the fuzzing process and will load and create additional components during the fuzzing process. Figure 3 shows an overview of all components. The preinstalled components are drawn with a solid line. Components that get transferred from the host to the guest during the fuzzing or components created during the fuzzing loop are drawn with a dotted line. The preinstalled *Loader* executes the *ReFS Fuzzer Agent* (1). At the first start of the *ReFS Fuzzer Agent*, it initializes the metadata translation table (2). In every fuzzing loop, it injects the *Mutated Metadata* with the help of the translation information (3) and creates an *Overlay Image* (4) to protect the original image from alterations. Last, the *Overlay Image* gets mounted, and the actions defined in the *File System Actions* data get executed (5).

We adapted the *Loader* from the kAFL framework. It patches the panic handler of the kernel to perform a hypercall when executed, allowing a quick response in case of a panic. Additionally, the *Loader* submits the memory address where the host should transfer the two payloads (compressed image and file system action list). In the end, it requests the *ReFS Fuzzer Agent* program and executes it.

The preinstalled *Translation Table* gets used together with the transferred *Mutated Metadata* from the agent to expand the metadata into the preinstalled *ReFS Image*. A newly created *Overlay Image* protects the *ReFS Image* from unwanted changes from the mounting process and the file system actions.

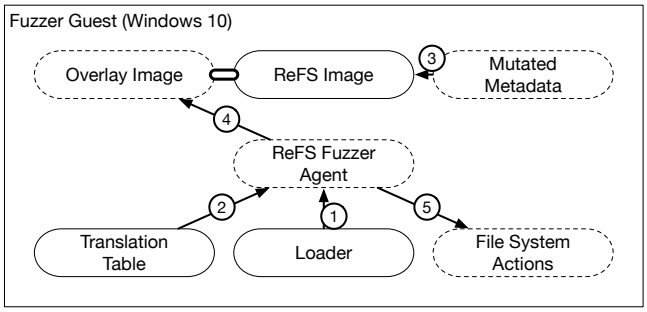


Figure 3: Software and data components inside the guest system. Solid drawn components are preinstalled. Dotted components are injected or created during runtime.

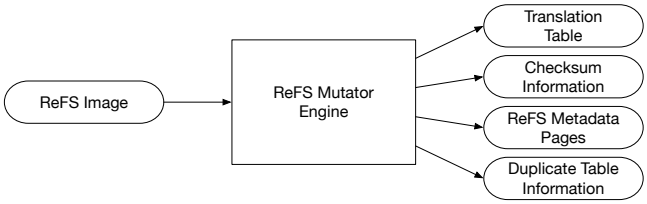


Figure 4: Showing the output of the ReFS Mutator Engine after the analysis of a ReFS image.

The *ReFS Fuzzer Agent* coordinates the actual fuzzing loop in consultation with the *ReFuzz* running on the host machine. The agent is detailed in the following subsection.

4.1 ReFS Mutator

The *ReFS Mutator Engine* is used at two different points while fuzzing the ReFS driver. First, the offline component analyzes a given ReFS image and outputs several artifacts needed later. The created artifacts are shown in Figure 4. The main goal of this step is to reduce the amount of data that the mutators will later alter. The relevant data, i.e., the metadata pages, gets collected and outputted in a file called *ReFS Metadata Pages*. The second goal is to collect all present checksums. After a mutation, this information is used to correct the checksums to create a semi-valid ReFS image, which will force the driver to pass the initial integrity checks successfully.

This offline component collects all pages present on the file system by traversing the whole metadata tree starting at both checkpoints. Since every reference to a node contains checksum information, these also get collected during the traversing process. Additionally, the boot sector and the three *superblocks* get extracted.

From our reverse engineering of ReFS, we know that some metadata is stored twice on the file system. These duplicate tables also get identified during the traversing process, and the location information of duplicates gets stored in the D*uplicate Table Information* file. The revisor can use this information later to copy mutated data to the corresponding duplicate table optionally.

Every page the engine encounters gets written sequentially to the metadata pages output file. To copy the pages back to the correct location in the original image, we save the offset in the *ReFS*

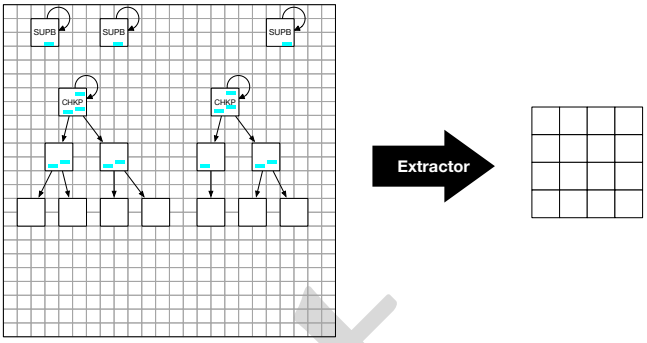


Figure 5: Data view showing a complete ReFS image and the extracted metadata after the extractor process. (SUPB = Superblock, CHKP = Checkpoint)

Figure 6: Data view of the revisor, showing the mutated metadata before and the checksum corrected data afterward.

Metadata Pages file together with the actual offset in the ReFS image. This data allows translating the location of checksums in the ReFS image to the location in the pages file. An abstract data view is shown in Figure 5. The whole partition with lots of unallocated data is on the lefthand side, and the extracted pages as a compact block on the righthand side.

The second use case of the *ReFS Mutator Engine* is the online usage which we integrated into the basis fuzzing framework. This part gets called after every mutation of the pages file. It corrects the checksums present in the pages data. Optionally, it also copies the mutated data to a duplicate table if it exists. This process is shown in Figure 6. The blue rectangle symbolizes mutated data, and the red rectangles the corrected checksums and duplicated information.

Figure 7 shows the whole lifecycle of the *ReFuzz* mutation engine, starting from a valid ReFS image to ending with a mutated semi-valid image. It shows the steps performed by the offline component *Extractor* and the two components integrated into the fuzzer host: *Mutators* and *Revisor*. These mutate the extracted metadata and correct the affected checksums. The Agent implements the last component. It injects the semi-valid mutated metadata into the full valid image to receive a complete semi-valid ReFS image.

4.2 Fuzzer Agent

The agent gets executed by the *Loader* and executes the different steps of the fuzzing loop. On startup, it initializes the translation data from the preinstalled *Translation Table* file. After that, it performs the following six steps in every fuzzing loop iteration:

- **Request Payload** via hypercall from the *ReFuzz* host.
- **Expand Image** with the help of the ReFS injector. This process is shown in Figure 8. The *Translation Table* defines where the injector must place the ”compressed” image data in the

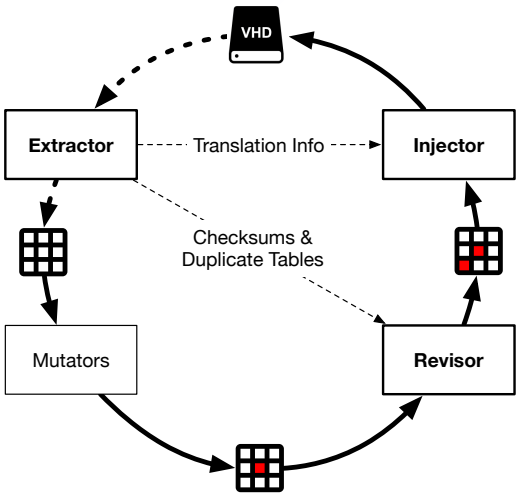


Figure 7: Lifecycle of the *ReFuzz* mutation engine.

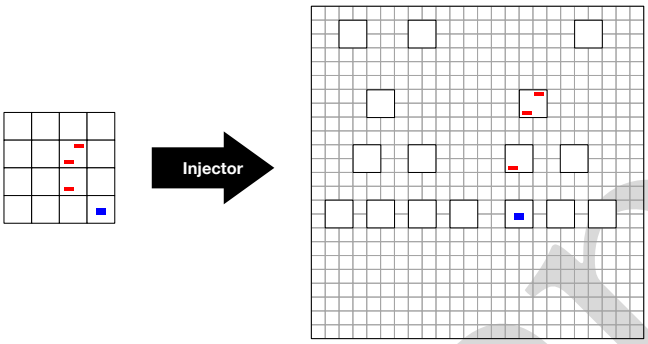


Figure 8: Data view before and after the injector takes the metadata and overwrites parts of the complete ReFS image.

full image. On the lefthand side, the mutated "compressed" image is shown, and on the righthand side, the resulting mutated complete ReFS image.

- **Create Overlay** protecting the mutated image from unwanted write modifications during the mounting process and the executed actions.
- **Mount Image** by using the overlay image.
- **Execute FS Actions** by interpreting the second payload containing a serialized list of remote procedure calls.
- **Unmount Image** and delete the overlay image.

If no failure occurs during the fuzzing loop, i.e., no bug in the ReFS driver was triggered, a new loop iteration starts. If a failure occurs, the VM gets reset to a snapshot right before the *Loader* requests and executes the ReFS Fuzzer Agent. The agent then starts with initializing the translation data as described before.

4.3 File System Action Mutator

As stated before, we added a second fuzzing dimension to kAFL. This second dimension reflects the actions performed on the data present on the ReFS image. We implemented a specific mutator for this dimension, which will alter the actions when the mutation of the ReFS image does not find any new input that executes unseen code paths. In other words, the mutation of the ReFS metadata will create a tree of interesting metadata payloads. Every time a payload gets mutated, it will be added as a new leaf node to the node representing the original data when it executes code paths never executed before. If the mutation engine has processed all leaf nodes and no new paths are found, the process starts over at the root node, and every node gets mutated again.

We modified this behavior by mutating our file system actions when no new code paths are found by mutating the file system metadata exclusively anymore. The action mutator operates on a given sequence of actions, the starting payload. Overall, our framework can handle 26 different actions, such as creating directories, creating files, creating symbolic links, or getting compressed file size.

Every mutation has the chance to modify the parameters of an existing action, adding a new action or removing a present action. To alter the parameters meaningfully, we have implemented different parameter types that get treated differently: string, path, pointer, int, and long. All but the path parameter get mutated so that their type constraints are respected. For the path parameter, we implemented a file pool. This pool is initialized with the paths of files and folders present on the ReFS image. The file pool gets accordingly updated while iterating over the sequential list of actions. If we encounter a delete action, the respective target gets removed from the pool. An action that adds new objects to the file system (e.g., create a file) will add a new path to the pool. This technique allows mutating paths parameter meaningful.

When the mutator finishes, the sequence of actions gets serialized. It can be transferred to the agent component running on the target VM in this form.

4.4 Other Modifications

The most significant changes we implemented in the kAFL framework are the usage of a second fuzzing dimension. We had to adjust the kAFL QEMU and KVM with a new hypercall to transmit the memory address of the second payload. Additionally, we extended the kAFL QEMU device to set the file for sharing the second payload. Several other kAFL components had to be adjusted to implement a second fuzzing dimension: e.g., *MapserverProcess*, *KaflTree*, and *benchmark*.

Our long-running fuzzing loop revealed a bug in the synchronization between the fuzzing master and the agent running in the target VM. Therefore we adjusted the locking in QEMU (*hypercall.c*). Additionally, we fixed the handshake protocol between these two components to fit our needs.

When the kAFL component on the host detects a time- out of a target VM —it does not send a request for a certain amount of time— it will reload the VM, which can be in two flavors: soft and hard reload. Soft reload uses a function in QEMU to reset the VM to the starting snapshot. This kind of reload is a relatively fast process. A Hard reload will kill the QEMU process and start a new one, where the snapshot has to be loaded entirely from the hard disk. This hard reload is slower than the soft reload, but this reload mechanism is needed if the QEMU process hangs up. We had to fix the mechanism of choosing the reload method. Before

our modification, the framework performed every reload in the hard mode.

We also made some slight adjustments to the kAFL mutators. In general, a fixed amount of deterministic mutations dependent on the size are performed on the payload (i.e., ReFS metadata). The number of non-deterministic mutations (called *havoc* and *splicing*) applied to a payload depends on the fuzzing loop's performance, which is determined in a benchmark beforehand. If a certain threshold of VM reloads is reached, the mutation process gets stopped, and another payload in the queue is taken. This solution will prevent wasting time on a payload that seems to be a blind end.

There are 376 million deterministic mutations to perform on a single payload for our payload size. If we theoretically assume the test duration of one payload as 10ms, the deterministic phase needs 43 days to finish for a single payload. A second observation was that fuzzing the ReFS driver with our setup was sometimes unstable, resulting in VM timeouts with no causal relationship to the tested payload. This misbehavior sums up, and the payload mutation is aborted in the early deterministic phase, meaning that only the first view kilobytes of our payload get muted with the bitflip mutators. We solve these problems by introducing a chance that a specific mutation gets tested. With this option, a user can set the chance, which will lead to a reasonable period for fuzzing one specific payload. The benefit is that this way, more new code paths can be found because other found payloads will gets mutated much more quickly.

5 EVALUATION

The evaluation of *ReFuzz* consists of three perspectives. First, we want to investigate the performance of our framework in respect of the fuzzing loop and the found new code paths. The second evaluation treats the bugs we could find with our framework. Last we want to compare our framework with other competitors, which are publicly available and can be used to fuzz closed source kernel drivers or can handle checksummed data.

We created the data by running *ReFuzz* for 52 days with four target-VMs in parallel. As hardware, we used a notebook with an Intel® Core™ i5-10210U CPU @ 1.60GHz with four cores (8 threads) supporting Intel PT and 16GB of RAM. The same 2GB ReFS formatted image was used as a seed for all tests. This image contains two folders, 24 files in the root directory, nine files in the first folder, and 19 files in the second folder. All files are plain text files containing random ASCII characters. Also, the same file system action seed was used for all evaluations defining the following actions: list directory one, list directory two, and read the content of file one.

5.1 Performance

First, we want to present our results in terms of performance. Thereby we can consider two different aspects: (1) We evaluate the temporal performance of our fuzzing agent from start to end during testing a tuple of payloads (i.e., ReFS metadata and actions). (2) We want to look at the development of our payloads.

We measured the duration of several operations during the test of payloads in a fuzzing loop iteration. The results can be seen in Figure 9. We evaluated the performance for the following operations: startup —which the loader only performs once if the VM gets reloaded—, transfer of the two payloads, expansion of the mutated ReFS metadata into the full VHD image, creating an overlay image and mounting it, using the file system as defined in the second payload and unmounting and deleting the overlay VHD image.

The diagram shows a separate candlestick for every operation. It consists of a box that represents the first and the third quartile. The black line defines the calculated mean value over all measured durations. The lowest and highest line shows the observed min and max values. We measured each operation at least 17,000 times except the startup operation, which occurred only 1,100 times. Note that the scale on the y-axis is logarithmic.

We see that the startup and transfer operation (in the mean) is very efficient compared to the others. The other operations directly use persistent data (i.e., the ReFS VHD image) stored on the target VM system image, which is again stored on the solid-state disk of the host system.

The expand operation writes the mutated metadata from RAM to the persistently stored VHD image. The mounting operation reads data from the VHD image through an overlay image. The use of the file system also needs to load persistent data from the disk, and the unmounting operation writes uncommitted data to the overlay image.

The different types of payloads have to be defined first to understand the evaluation of code paths. Every payload which was once found to execute new parts of the code is stored in the findings tree.

One type of payload is called *favorite*. A *favorite* is a payload that executes a specific node transition (i.e., edge) and is the fastest payload to do so relating to the fuzzing loop duration. That means that over time some payloads can lose their favorite status if a new one could be found, which is more performant.

Another type of payload in this evaluation is the pending type. A new mutated payload that executes a unique code path is appended to a queue for further mutations. Every item in that list is typed as pending because they have not been mutated yet.

Figure 10 shows the development of found payloads with unique paths during our run. After the first 24 hours, we had twelve findings, six pending, and five favorites. The overall amounts constantly grew in all three categories. In the end, the pending payloads seem to get fewer.

Two exceptions in the steady growth can be observed at the start of day seven and during day ten, where almost every pending payload where processed. After 52 days, we enlarged the exploration of the path to 260 findings, 49 pending, and 168 favorites. The number of pending payloads shows a great potential to find more bugs after we stopped our evaluation.

5.2 Found Bugs

During our longest run of 52 days, *ReFuzz* was able to find 29 payloads (i.e., altered ReFS metadata) which caused the Windows kernel to panic, meaning that the system crashed and rebooted. We verified that 27 are true positives, and only two are false positives that did not cause a kernel crash while we tested the corresponding mutated ReFS.

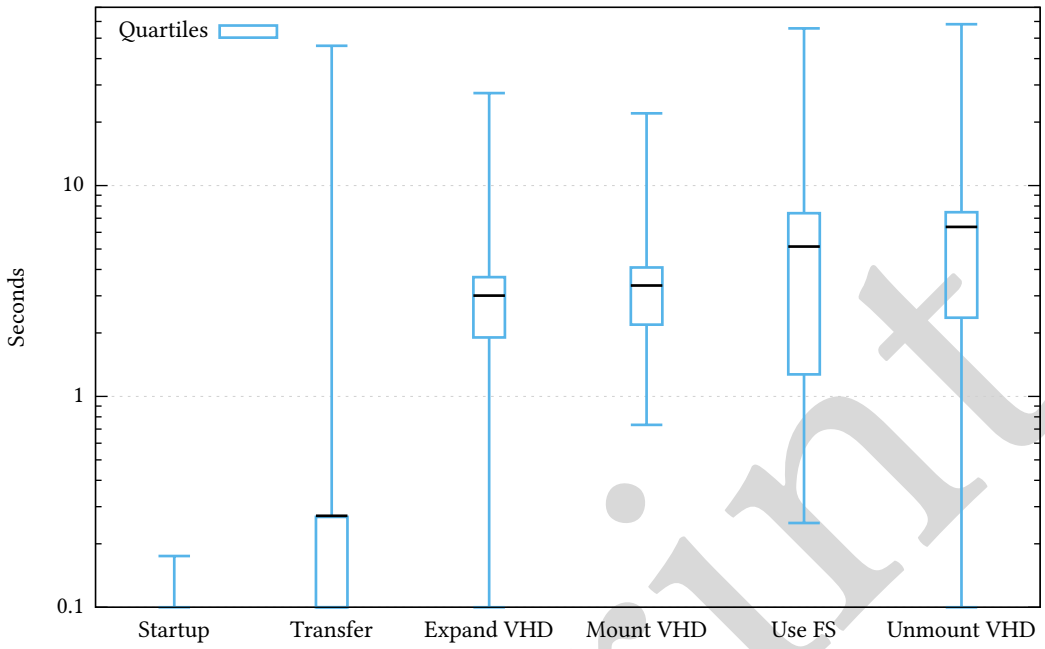


Figure 9: Amount of time passed during a fuzzing loop iteration partitioned by individual operations.

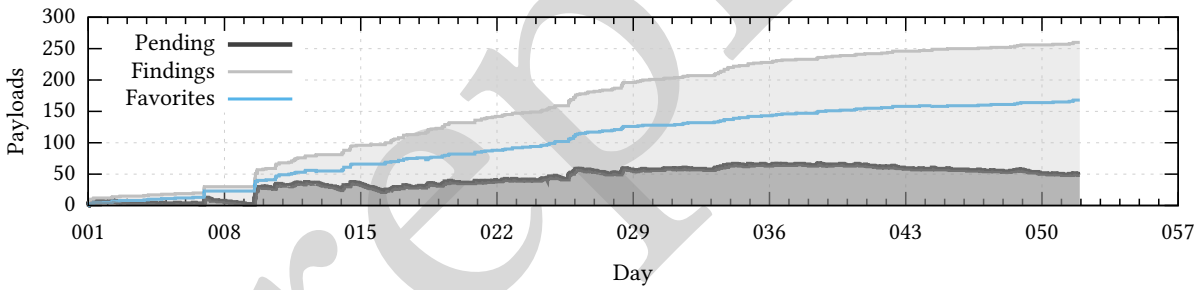


Figure 10: Development of the number of payloads while fuzzing the ReFS driver.

We reported our findings to Microsoft in a responsible disclosure process. Microsoft acknowledged that these payloads include ten unique issues. Eight are classified with critical severity as they allow for remote code execution[1]. The other remaining issues allow a moderate denial-of-service attack. These issues are fixed by eight patches and rolled out at the January 2022 Patch Tuesday. The table in the appendix A, shows details of our findings. More details are available at https://www.cs1.tf.fau.de/research/system-security-group/refuzz/. For example, the table shows the technical reason for the crash in the column *Crash Code*, a rough description of the action needed to trigger the crash, and the assigned CVE.

The number of different bug check codes that occurred while testing our findings shows that we found several unique bugs in the ReFS driver, not necessarily related to each other. The following list of codes[2] occurred while we have tested mutated payloads:

- **1x** KMODE EXCEPTION NOT HANDLED **(0x1E)**: This indicates that a kernel-mode program generated an exception that the error handler did not catch.
- **11x** PAGE FAULT IN NONPAGED AREA **(0x50)**: This indicates that invalid system memory has been referenced. Typically the memory address is wrong, or the memory address is pointing at freed memory.

[1]https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-21892
https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-21928
https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-21958
https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-21959
https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-21960
https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-21961
https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-21962
https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-21963

[2]https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/bug-check-code-reference2

- **2x** KERNEL SECURITY CHECK FAILURE **(0x139)**: This bug check indicates that the kernel has detected the corruption of a critical data structure.
- **20x** REFS FILE SYSTEM **(0x149)**: This indicates that a file system error has occurred.

Another interesting fact is the progress of findings of timeout and panic bugs. The amount of findings over time is shown in Figure 11, separated into panic and timeout payloads. Bugs that force Windows to freeze are not as critical as bugs that crash the system but can also be used for denial-of-service attacks.

After the whole evaluation phase of 52 days, we have found 162 unique timeout payloads. As shown in Figure 11, the finding of timeout payloads is distributed evenly over time, whereas we found most panic payloads in the first seven days. If we look at the number of findings after 24 hours, we can see that *ReFuzz* has found nine unique panic and one timeout payload.

5.3 Comparison with other Fuzzers

As the related work section states, some other research targets fuzz testing of file system drivers or drivers in general. However, only some of them are suitable for fuzzing the ReFS driver of Windows because we need the ability to test closed source software with a feedback channel.

Janus [23], Hydra [9], and *Agamotto* [21] are all state-of-the-art fuzzers proposed as of late, targeting (file system) kernel drivers. They all have in common that they require the sources of the target drivers and kernels to either modify them (*Agamotto*) or compile them to run as a userspace application (Janus and Hydra) to retrieve coverage information. Both are impossible for the ReFS Windows driver because this software is closed source.

To the best of our knowledge, the only fuzzing frameworks allowing us to retrieve coverage information from closed source software via hardware assistance are kAFL based fuzzers. These allow running the target code without modifications. So we decided to compare the performance of *ReFuzz* to the base kAFL implementation and the Redqueen fuzzer, which is also based on kAFL, in the following sections.

5.3.1 *Base kAFL*

. Since *ReFuzz* is based on kAFL, we want to compare if our implementation performs better than the base fuzzer. Because some modifications of kAFL are necessary to target the ReFS driver (e.g., increase the payload size and implement an *Agent* running on the target **VM!** (**VM!**)), we decided not to use the pure kAFL implementation for this evaluation.

Because our *ReFuzz* builds upon the kAFL fuzzer, we only disabled the checksum and duplicate table correction of our framework together with the second fuzzing dimension to keep it straightforward. We kept the extraction and insertion mechanism of our *ReFuzz* implementation. This feature is advantageous over the pure kAFL because our mutation engine can precisely target the used **FS!** (**FS!**) data without wasting time mutating unused parts of the **FS!** image. The untouched kAFL version would perform worse than the version used in our evaluation.

After a 44 hour run of the slightly modified base kAFL fuzzer where 37,000 payloads were tested, we found zero panic and timeout payloads. That shows that checksum correction is essential in fuzzing ReFS and other modern **FS!** drivers with excessive checksums to guarantee data integrity. Without a checksum correction to generate semi-valid **FS!** images, checksum validation methods in the driver will always safely abort the code execution before hitting potential bugs. Moreover, base kAFL would perform worse than our evaluation because it has to handle the full-sized ReFS images with 2 GB of data.

To evaluate the differences of kAFL and *ReFuzz* even further with a statistical significance, we ran both fuzzers ten times for 24 hours and evaluated the bitmap development over time. The bitmap of an AFL-based fuzzer tracks the executed code path, i.e., the code coverage. kAFL and *ReFuzz* calculate a hash for every executed jump in the target from the source and destination address of the jump. The hash defines where the jump gets logged in the coverage bitmap, but the hash calculation can collide. Resulting in two different jumps can be logged in the same bitmap's position.

We use the bitmap as an approximation of the actual code coverage. The accurate coverage of the executed payloads is higher than the logged bitmap, but it is adequate to see the differences in reached code coverage of kAFL and *ReFuzz*.

Figure 12 shows the development of code coverage over 24 hours. The solid lines represent the mean coverage of both fuzzers. There are no significant differences present. Between hour six and hour 16, *ReFuzz* performs slightly better, but in the end, kAFL seems to outperform *ReFuzz* in code coverage. The minimal and maximal code coverages (drawn as dashed lines) show that *ReFuzz* has a more significant variance than kAFL. One *ReFuzz* run produced a much higher code coverage than the other kAFL and *ReFuzz* runs. In conclusion, if we look only quantitatively at the coverage over time, no statistically significant differences of both fuzzers can be experienced. This result is unexpected because, in the long 44-hour run, kAFL has found no bugs in contrast to *ReFuzz*.

In Figure 13, the merged bitmaps of all ten runs of *ReFuzz* (13b) and kAFL (13a) are visualized qualitatively. Jumps executed once are drawn as black dots. Green dots are jumps executed twice, and red dots are executed more than twice. *ReFuzz* was able to execute more different code paths than kAFL, but kAFL executed the same jumps more often.

Figure 14 compares the bitmaps of kAFL and *ReFuzz*. The figure reveals a significant difference regarding the coverage of the fuzzers. Subfigure 14a shows the code path only found by kAFL, Subfigure 14b shows the code path only found by *ReFuzz*, and Figure 15 shows the code path executed by both fuzzers. With this comparison, we can draw the following conclusions. Both fuzzers can execute different code paths, but *ReFuzz* can execute more code exclusively. Both fuzzers can reach some identical code parts. Together with the outcomes of the long runs of *ReFuzz* and kAFL, it seems that the code paths executed exclusively by *ReFuzz* are more promising in finding bugs than the ones executed by kAFL.

This result is very interesting because it shows that code coverage is not only interesting as a quantitative measurement as Klees et al. [10] propose in their work, but there is also a qualitative part in the code coverage metric, as we have shown in our evaluation.

5.3.2 *Redqueen*

. The Redqueen fuzzer proposes to overcome checksums and magic

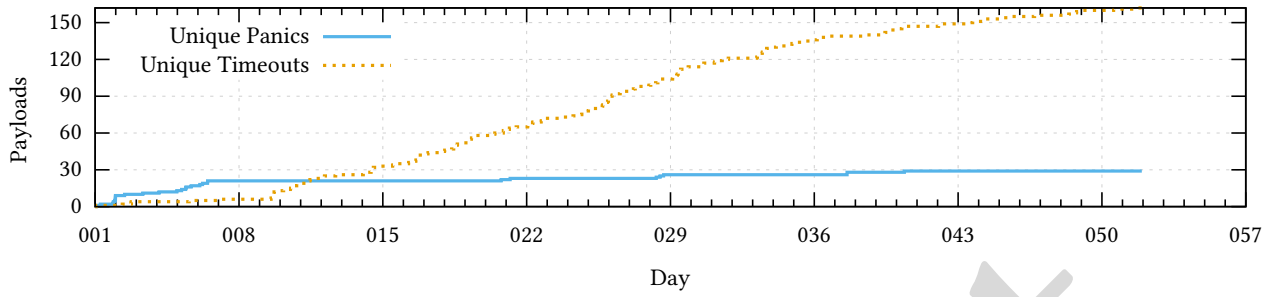


Figure 11: Temporal progression of payload findings leading to a kernel panic or timeout.

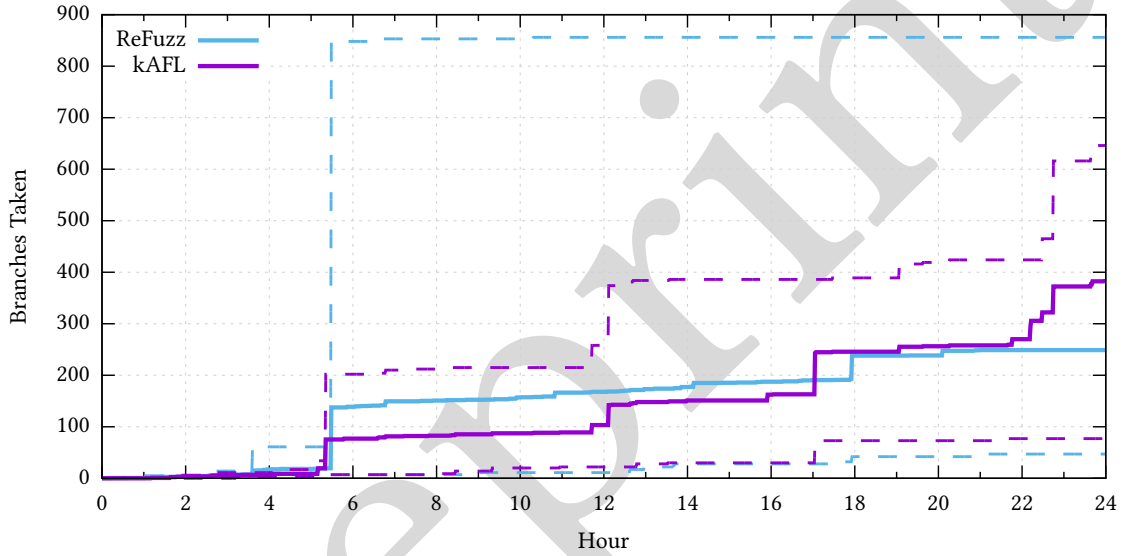


Figure 12: Amount of code path executed from kAFL and *ReFuzz* over time. The values are extracted from the bitmap.

bytes automatically with no further data format knowledge. Although, in theory, this fuzzer should be performant in fuzzing the ReFS driver, we decided not to test the driver with Redqueen. The reason is that we were not able to utilize Redqueen without any or only minor modifications. Since Redqueen lacks the ability of metadata extraction and injection, we would need to increase the payload size to 2GB which exceeds the default 128kB of Redqueen.

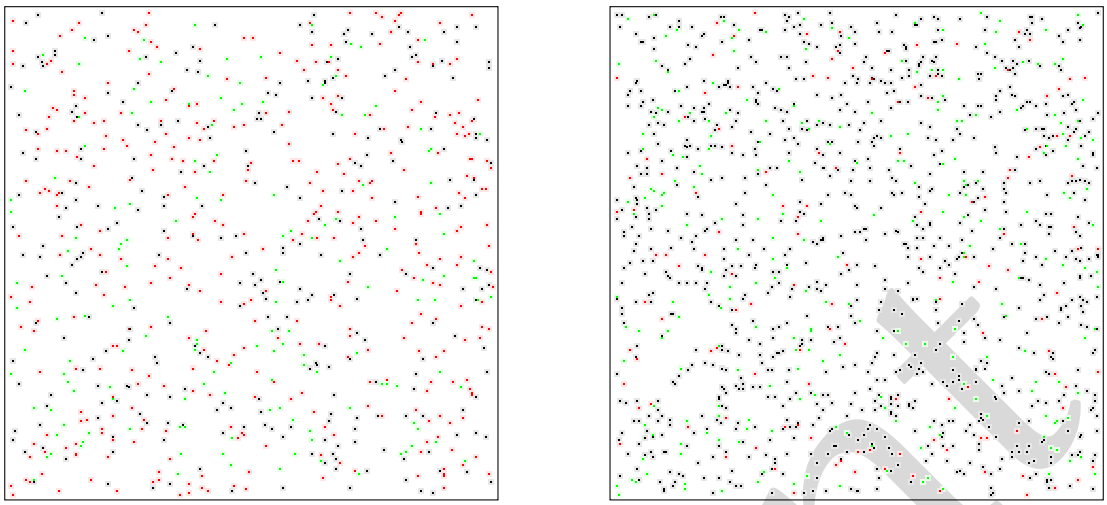
We doubt that a slightly modified version of Redqueen would outcompete our implemented solution because of the bigger input space —2GB vs. 916kB— that needs to be explored and the chains of checksummed data. Also, Redqueen has to run the target code multiple times iteratively with a slightly modified version of the input to figure the correct checksums out. Because of the relatively long-running fuzz loop of kAFL, this process would be costly performance-wise.

Another constraint of Redqueen is that, in theory, it can be used with any target OS. However, the current implementation only focuses on special packed user applications combined with a Linux environment. Redqueen needs to instrument the target code to observe compare-instructions, which is not easily possible for Windows because of PatchGuard [20].

To summarize, we doubt that Redqueen can outcompete our highly specialized approach for ReFS because of the much bigger input space and the additional expensive fuzzing runs required to figure out the corrected checksums. Moreover, it would need notable changes to target the Windows kernel with Redqueen.

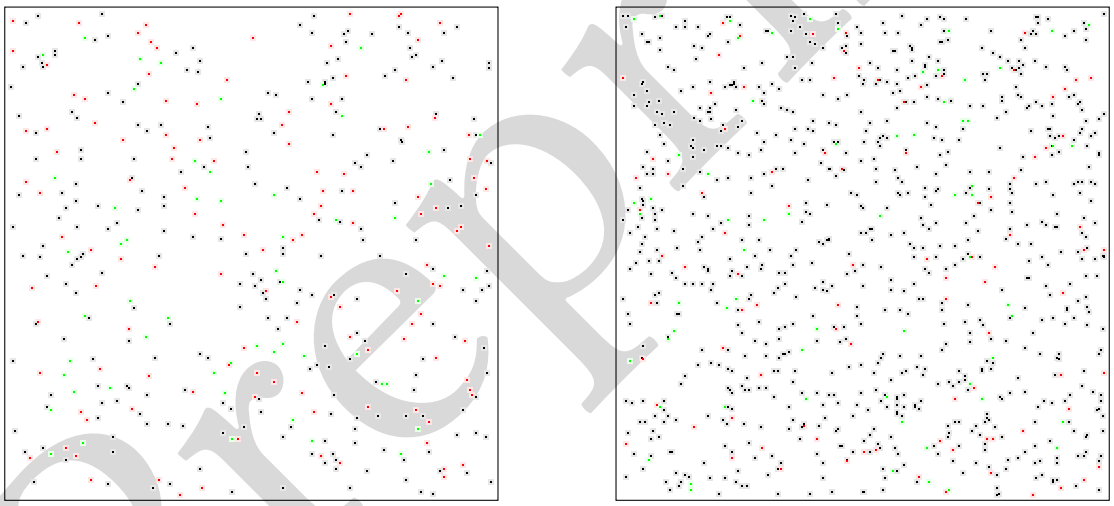
6 CONCLUSION

To the best of our knowledge, we are the first who systematically fuzzed the ReFS driver and found a massive amount of previously unknown bugs. During our evaluation, we have shown that with pre-existing solutions, such as *base kAFL*, no bugs can be found in ReFS. While the performance of *ReFuzz* in terms of iterations per unit time cannot compete with other solutions, we outperform other solutions in terms of effectiveness. That means while the performance of *ReFuzz* suffers from the ineffective agent component, which has to deal with a large amount of persistent data, we were



(a) kAFL (b) ReFuzz

Figure 13: Coverage bitmaps produced by base kAFL and *ReFuzz* visualized as a heatmap. Black: Jumps executed once. Green: Jumps executed twice. Red: Jumps executed at least three times. (Part 1)



(a) Exclusive kAFL (b) Exclusive ReFuzz

Figure 14: Exclusive coverage reached by kAFL and *ReFuzz* visualized as a heatmap. Black: Jumps executed once. Green: Jumps executed twice. Red: Jumps executed at least three times. (Part 2)

able to compensate for that and provide generally better fuzzing results for ReFS than any other solution before.

Although works like REDQUEEN exist, that in theory, can find correct checksums while generically mutating input [1], we think that a specialized mutation engine always outcompetes generic processes. Additionally, we have the hurdle of a massive fuzzing input with modern file systems, which we had to circumvent.

In the future, *ReFuzz* could be extended to use a lightweight snapshot mechanism similar to *Agamotto*[21] to allow tuning its speed by skipping the unmounting process and loading a VM snapshot instead. We unsuccessfully tried many ways to speed up the fuzzing loop during our research so far. We had the idea to keep the ReFS VHD image in RAM instead of storing it on persistent data. However, the VHD interface from Microsoft cannot use files stored in RAM. Our second approach of keeping the whole Windows system in RAM did not bring improvements either.

To summarize, we introduced a structure-aware fuzzing engine called *ReFuzz*, which targets the Windows ReFS driver. Our approach circumvents two major problems when fuzzing: (1) Correction of checksums present on a ReFS image to execute code after the checks are performed. (2) The massive amount of input data being fuzzed —at least 2GB— which no current greybox fuzzer can handle in a performant manner. As a result, we were able to find 27 unique panic payloads triggered from bugs in the ReFS driver.

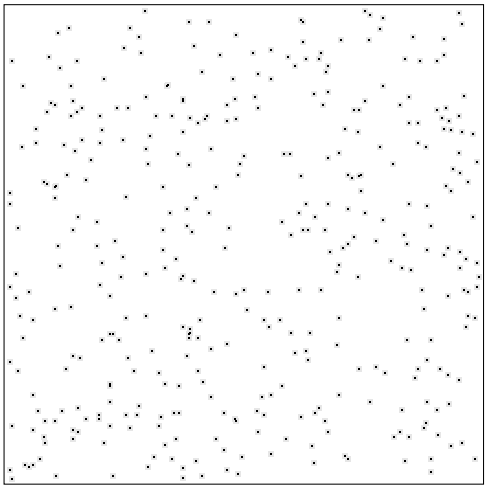


Figure 15: Overlapping coverage bitmap from kAFL and *ReFuzz* visualized as a heatmap. Black: Jumps executed once. Green: Jumps executed twice. Red: Jumps executed at least three times. (Part 3)

We reported these bugs to Microsoft in a responsible disclosure process, which led to eight CVEs assigned to us, and those flaws got patched in the January 2022 Patch Tuesday.

To conclude, we think our approach for ReFS can be adapted to fuzz other modern file system drivers in the future. Any open or closed source drivers running in kernel mode could be fuzzed. For example, fuzzing Apple's APFS could also benefit from our metadata extraction and checksum correction method. Plum and Dewald [13][5] analyzed that every file system data in APFS is wrapped inside an *object* which contains a self-checksum and heavily uses B-Trees to structure the data. Mutating only the file system data and skipping content data and unused space could also speed up the fuzzing process of APFS, although the smallest possible APFS image is much smaller than ReFS [8].

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF) as part of the FIDI project.

REFERENCES

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*.

[2] Domagoj Babic, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-directed dynamic automated test generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, Matthew B. Dwyer and Frank Tip (Eds.). ACM, 12–22. https://doi.org/10.1145/2001420.2001423

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2329–2344. https://doi.org/10.1145/3133956.3134020

[4] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. 2007. Ext4: The Next Generation of Ext2/3 Filesystem. In *2007 Linux Storage & Filesystem Workshop, LSF 2007, San Jose, CA, USA, February 12-13, 2007*, Ric Wheeler (Ed.). USENIX Association. https://www.usenix.org/conference/2007-linux-storage-filesystem-workshop/ext4-next-generation-ext23-filesystem

[5] Andreas Dewald and Jonas Plum. 2018. *APFS INTERNALS FOR FORENSIC ANALYSIS.* https://static.ernw.de/whitepaper/ERNW_Whitepaper65_APFS-forensics_signed.pdf Accessed: 16.02.2022.

[6] Google. [n.d.]. *syzkaller - kernel fuzzer.* https://github.com/google/syzkaller Accessed: 10.11.2021.

[7] Kurt H. Hansen and Fergus Toolan. 2017. Decoding the APFS file system. *Digit. Investig.* 22 (2017), 107–132. https://doi.org/10.1016/j.diin.2017.07.003

[8] hoakley. 2019. *Hitting the limits of APFS is both easy and confusing.* https://eclecticlight.co/2019/08/12/hitting-the-limits-of-apfs-is-both-easy-and-confusing/ Accessed: 16.02.2022.

[9] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2020. Finding Bugs in File Systems with an Extensible Fuzzing Framework. *ACM Trans. Storage* 16, 2 (2020), 10:1–10:35. https://doi.org/10.1145/3391202

[10] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 2123–2138. https://doi.org/10.1145/3243734.3243804

[11] Michal Zalewski (lcamtuf). [n.d.]. *american fuzzy lop (2.52b).* https://lcamtuf.coredump.cx/afl/ Accessed: 11.10.2021.

[12] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 33–50. https://www.usenix.org/conference/osdi18/presentation/mohan

[13] Jonas Plum and Andreas Dewald. 2018. Forensic APFS File Recovery. In *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018, Hamburg, Germany, August 27-30, 2018*, Sebastian Doerr, Mathias Fischer, Sebastian Schrittwieser, and Dominik Herrmann (Eds.). ACM, 47:1–47:10. https://doi.org/10.1145/3230833.3232808

[14] Paul Prade, Tobias Groß, and Andreas Dewald. 2020. Forensic Analysis of the Resilient File System (ReFS) Version 3.4, In DFRWS 2020 EU, Oxford, United Kingdom, June 3-5, 2020. *Digital Investigation* 32, Supplement. https://doi.org/10.1016/j.fsidi.2020.300915

[15] Paul Prade, Tobias Groß, and Andreas Dewald. 2019. *Forensic Analysis of the Resilient File System (ReFS) Version 3.4.* Technical Report CS-2019-05. Department Informatik. https://doi.org/10.25593/issn.2191-5008/CS-2019-05

[16] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[17] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage* 9, 3 (2013), 9:1–9:32. https://doi.org/10.1145/2501620.2501623

[18] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[19] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[20] Skywing. 2007. *PatchGuard Reloaded - A Brief Analysis of PatchGuard Version 3.* http://www.uninformed.org/?v=8&a=5 Accessed: 07.10.2021.

[21] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. 2020. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2541–2557. https://www.usenix.org/conference/usenixsecurity20/presentation/song

[22] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1643–1660. https://doi.org/10.1109/SP40000.2020.00078

[23] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 818–834. https://doi.org/10.1109/SP.2019.00035

[24] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson R. Engler. 2006. Automatically Generating Malicious Disks using Symbolic Execution. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. IEEE Computer Society, 243–257. https://doi.org/10.1109/SP.2006.7

A RESULTING DATASET

The following table shows a list of payloads our *ReFuzz* found during evaluation, which causes the Windows kernel to panic. Column # indicates the VHD image number in the dataset. *Crash Code* defines the crash code returned by the kernel after mounting and using the image. The third column defines the action to be executed after mounting the image to provoke the crash. The last column defines the concrete flaw detected by the image by assigning a CVE. All images can be downloaded at our project website: https://www.cs1.tf.fau.de/research/system-security-group/refuzz/. The image number is used to identify the image name with the pattern `panic_{}.vhd`.

#	Crash Code	Action to Crash	CVE
3	0x149	open folder	CVE-2022-21892
5	0x149	open folder	CVE-2022-21892
13	0x149	open folder	CVE-2022-21892
17	0x139	open folder	CVE-2022-21892
20	0x149	open folder	CVE-2022-21892
21	0x149	open folder	CVE-2022-21892
1	0x50, 0x149	open two folders and a file	CVE-2022-21928
4	0x149	open folder	CVE-2022-21958
6	0x50	mount file system	CVE-2022-21958
7	0x149	open folder	CVE-2022-21958
8	0x149	open folder	CVE-2022-21958
9	0x149	open folder	CVE-2022-21958
12	0x149	open folder	CVE-2022-21958
15	0x50, 0x149	open folder	CVE-2022-21958
16	0x50, 0x149	open folder or mount file system	CVE-2022-21958
19	0x149	open folder	CVE-2022-21958
23	0x50, 0x149	mount file system	CVE-2022-21958
24	0x149	mount file system	CVE-2022-21958
10	0x1E	open folder	CVE-2022-21959
11	0x149	open folder	CVE-2022-21960
14	0x50, 0x149	open folder	CVE-2022-21961
27	0x50	open file	CVE-2022-21961
29	0x50	open file	CVE-2022-21961
2	0x139, 0x149	open file	CVE-2022-21962
18	0x50, 0x139, 0x149	open file	CVE-2022-21962
22	0x50	open folder	CVE-2022-21963
25	0x50	mount file system	CVE-2022-21963