TREATING YOUR FILESYSTEM AS A WEB BROWSER USING FUSE

Travis Near
Computer Science
Northwestern University
Evanston, IL

June 9th, 2023

ABSTRACT

One challenge of working with Linux's command line tools is that their interface for network data differs greatly from that of viewing local files. For local files, one typically uses commands such as `cat`, `ls`, and `less` to browse the file. For Internet content, there exist separate commands `wget` or `curl` to download it. And for S3 data, Amazon recommends using its SDK. Instead of a user memorizing multiple command line interfaces, WebBrowserFileSystem unifies them all: users can now `cat example.com` or `grep s3://<bucket>/<file>`! WebBrowserFileSystem utilizes FUSE to intercept Linux system calls to make network requests for supported protocols. This provides a seamless experience for users who no longer need to concern themselves with the location of the data they are viewing.

Keywords *networking · file system · fuse · kernel · curl · sqlite · s3 · fuzzing*

1 Introduction

Can your file system be treated as a web browser? The WebBrowserFileSystem repository enables using UNIX's core utilities such as `cat`, `ls`, vi, and more to make network requests to download content from the Internet. The inspiration for this project came from MATLAB's I/O functions which support multiple data locations. For example, calling `readlines` on a local file, HTTP document, or S3 bucket all share the same interface [14]. Engineers using MATLAB can supply any path to their data and the `readlines` function knows how to find it.

There is an industry-wide push to cloud storage. Often times this is convenient: users can access their data across different devices plus gain revision control which allows diffing files. However, engineers following scripted workflows must learn new command line interfaces to access their remote data. Core UNIX utilities such as `cat` and `grep` do not support networking. Instead, users must first use a different command such as `wget` or `curl` to download the data then process it as a local file. Although this is functional, users must learn new commands. Compounding the challenge are remote files such as Amazon AWS S3 data. Amazon's SDK is radically different from classical UNIX commands [1]. This paper introduces a new library, WebBrowserFileSystem, which allows using standard UNIX utilities to access remote data using FUSE.

2 What is FUSE?

2.1 Filesystem in Userspace

Filesystem in Userspace (FUSE) is a software library that enables developers to create custom file systems in user space, without requiring modifications to the operating system kernel [9]. FUSE allows developers to implement custom file system operations by implementing FUSE's interface which translates these operations into standard system calls. This approach provides flexibility and ease of development, as file systems can be made to do things they weren't

designed to do. This unique ability allows FUSE to perform functionality such as networking which is needed for WebBrowserFileSystem.

2.2 How FUSE bridges the OS kernel

FUSE serves as a bridge between the user space and the OS kernel. After registering with the kernel, FUSE initializes the filesystem and establishes a communication channel between the user space and kernel through a special mounted directory. When file system operations are performed in user space, FUSE invokes the corresponding callback functions implemented by the developer. This translation is shown in Figure 1.

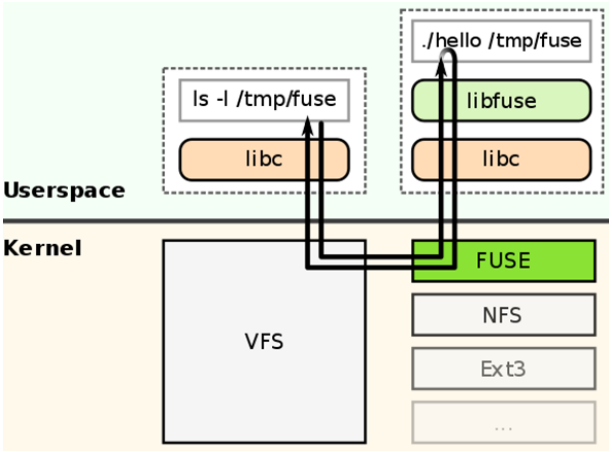

Figure 1: FUSE bridge: *https://en.wikipedia.org/wiki/Filesystem_in_Userspace*

These functions translate the operations into requests that are sent to the kernel, where they are executed by the appropriate file system driver. The kernel then communicates the results back to FUSE, which processes the responses and returns the final results to the user space application. This is a flexible approach which allows custom behavior such as the networking done by WebBrowserFileSystem. FUSE callbacks typically wrap system calls, thus allowing your implementation to work with nearly all Linux commands.

2.3 How FUSE operates

Here is a concrete example using the FUSE callback `read`. To implement custom file system behavior using FUSE, there are several steps to follow. First, set up the `read()` function pointer in the `fuse_operations` struct to define the behavior for reading data from the file system as shown in Figure 2.

```
struct fuse_operations urlfsOperations =
{
    .getattr = urlfs_getattr,
    .read    = urlfs_read,
    .readdir = urlfs_readdir
};
```

Figure 2: FUSE callbacks

Next, implement this `urlfs_read` function with the predefined signature as shown in Figure 3.

```
// Read data from an open file
int urlfs_read(const char *path, char *buf,
               size_t size, off_t offset)
```

Figure 3: FUSE open() parameters

When a `read()` system call is invoked, FUSE will relay the execution to your `urlfs_read` function, allowing you to handle the read operation according to your requirements. Table 1 explains the usage of each function argument.

Argument	Purpose
`path`	The path to the file in the mounted directory.
`buf`	The valid buffer with to which to write data.
`size`	The size of the buffer.
`offset`	How far into the file to seek before reading.

Table 1: FUSE read function call arguments

To implement other FUSE callbacks such as `write`, `unlink`, and `getattr`, repeat this process of implementing FUSE callbacks. FUSE will call the appropriate function at the appropriate time, providing a seamless integration of your custom file system logic into the FUSE framework.

3 WebBrowserFileSystem introduction

WebBrowserFileSystem is an open-source repository written in C which allows using core UNIX utilities to make network requests. The source code is available on GitHub: https://github.com/tnear/WebBrowserFileSystem

3.1 Motivation

In traditional computing environments, commands like `cat`, `head`, and `less` are limited to operating on files stored on a file system. However, with WebBrowserFileSystem, these commands now seamlessly interact with network content, enabling users to access remote files and data within their familiar command line environment. This capability helps to bridge the gap between local and remote content. Users no longer need to run separate commands based on where their data is located.

3.2 WebBrowserFileSystem setup

Run the WebBrowserFileSystem executable while supplying the directory to mount, ex:

```
$ ./WebBrowserFileSystem -f -s mnt/
```

Next, in a separate terminal, navigate to your mounted directory. This directory (`mnt/`) is associated with WebBrowserFileSystem, therefore all commands which use supported protocols will automatically download the content you are trying to view. For example, typing `cat example.com` will intercept the `read()` system call using FUSE, detect that this is an HTTP request, download the content, then display the page's HTML data to you. This will also create a file in the directory named "example.com".

4 Using WebBrowserFileSystem

Now that WebBrowserFileSystem is awaiting requests for downloading content, here are a few examples which showcase its capabilities.

4.1 Making network requests using cURL

WebBrowserFileSystem utilizes cURL [7], an open-source command-line tool for making network requests in various protocols, such as HTTP, FTP, and DICT [2] (but not S3 as described in section 4.5). cURL's ease of use, wide-ranging protocols, and open-source nature make it a natural choice for C networking needs. cURL also permits requesting header information and performing partial downloads. These are used by WebBrowserFileSystem to support paging, described in section 4.6.

4.2 HTTP and HTTPS

HTTP (Hypertext Transfer Protocol) is a widely-used protocol used for transmitting and retrieving web content. It facilitates communication between web servers and clients.

HTTPS (Hypertext Transfer Protocol Secure) is a secure version of HTTP that employs encryption mechanisms to ensure secure data transmission. HTTPS provides privacy and integrity for sensitive information exchanged between web servers and clients.

To read HTTP and HTTPS content using WebBrowserFileSystem, use any UNIX command for reading local data but pass a URL instead of a local path:

```
$ head -n1 example.com
<!doctype html>
```

The `head` command shows the beginning of files. By using the `-n1` flag, it outputs the first line of `example.com`. Because there is no file called 'example.com' in the present working directory, this uses cURL to download its content then caches the result. This does not create a "true" file. Instead, it creates an object which *looks* like a file due to WebBrowserFileSystem implementing all the appropriate system calls through FUSE to make it appear like a file. This phantom file has also attributes which must be set through FUSE's `getattr` callback:

```
$ ls -l
-r--r--r-- 0 kali kali 1256 Jun  3 15:37 example.com
```

WebBrowserFileSystem provides a read-only view to all data, hence the R/R/R file permissions. The file size, 1256 bytes, is size of the downloaded content. The timestamp is when the response for the request was first received. The filename is represents a descriptive, valid Linux filename. To illustrate the importance of filename, consider this example:

```
$ cat example.com\\index.html
$ ls -l
-r--r--r-- 0 kali kali 1256 Jun  3 15:44 example.com
-r--r--r-- 0 kali kali 1256 Jun  3 15:45 index.html
```

The canonical URL, "example.com/index.html", uses a slash character, which poses a problem for UNIX file systems. This limitation is explained in more detail in section 10.1. For now, it suffices to say that all slashes must be replaced. WebBrowserFileSystem has opted to use the backslash character instead. Backslash must be escaped, hence the '\\' in the command above. Also note that only the trailing portion of the URL ("index.html") becomes the file name. The complete path is `"example.com\\index.html"`. However URLs are often long and users are most interested in the right-most portion of the URL. This repository applies heuristics to set the file name to be the most descriptive portion of the URL. It also ensures there are no slash characters in the URL.

4.3 FTP

FTP (File Transfer Protocol) is a network protocol designed for transferring files between a client and a server over a computer network. WebBrowserFileSystem uses cURL in its implementation for downloading resources. This means that the FTP protocol is automatically supported. This example uses the `vi` text editor to download using the FTP protocol:

```
$ vi ftp:\\\\ftp.slackware.com\\welcome.msg
```

Upon successful downloading of the FTP data, `vi` will open file's contents in a read-only file.

4.4 DICT

The DICT (Dictionary) protocol is a network protocol that enables clients to query and retrieve word definitions from dictionary databases scattered across the Internet. cURL natively supports DICT thus making this protocol easy to use in WebBrowserFileSystem:

```
$ cat dict:\\\\dict.org\\m:curl
220 dict.dict.org dictd 1.12.1/rf on Linux 4.19.0-10-amd64 <auth.mime> <192175844.20718.1685827425@
dict.dict.org>
250 ok
152 18 matches found
gcide "cul"
gcide "Cur"
gcide "Churl"
<truncated>
```

4.5 Amazon S3

Amazon S3 (Simple Storage Service) is a scalable and reliable cloud storage service provided by Amazon Web Services (AWS), designed to store and retrieve any amount of data from anywhere on the web. It provides its own SDK and offers convenient interoperability to other AWS resources.

cURL does not support the Amazon S3 protocol [6]. WebBrowserFileSystem supports viewing public S3 data using the "s3://" protocol:

```
$ grep 'hello' s3:\\\\my-bucket\\index.html
<html><body><h1>hello, world</h1></body><html>
```

As is the case with other supported protocols, the syntax is the same. This is one of the selling points of WebBrowserFileSystem. Users wrap regular Linux commands around a path to a resource they want to process.

WebBrowserFileSystem supports public S3 data by replacing the S3 protocol URL with the canonical Amazon AWS URL. For example, "s3://<bucket>/<file>" is converted into "https://<bucket>.s3.amazonaws.com/<file>" before beginning the download.

Private S3 data, however, requires access keys and tokens (i.e., `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN`). Although not yet implemented, WebBrowserFileSystem should be able to support private S3 by setting environment variables to these keys and tokens. The implementation reads these environment variables, signs the access key, combines this signature along with the access key, then passes to cURL this authorization header information.

4.6 Paging

For large files, operating systems leverage paging to avoid loading the entire file into memory at once. The `less` and `more` UNIX commands are vital tools for viewing and navigating through large text files. Paging enables users to analyze and search content efficiently without overwhelming system resources. These commands provide a convenient way to scroll, search, and page through large volumes of text.

WebBrowserFileSystem implements the `offset` parameter in FUSE's `read()` callback to support paging through data. The example below demonstrates an example using `less` to page through a large data page (100,000 digits of pi):

```
$ less https:\\\\math.tools\\numbers\\pi\\100000
<!DOCTYPE html>
<truncated>
```

The 'd' key pages down in `less` mode. Doing this until reaching the end of the file shows output in the WebBrowserFileSystem process terminal indicating the offset parameter used to display the file:

```
Downloaded: https://math.tools/numbers/pi/100000
Offset: 16384
Offset: 49152
Offset: 114688
```

Note that there are multiple calls to `open()`, each with a different offset. FUSE provides a buffer and WebBrowserFileSystem fills it honoring the size and offset arguments relayed by FUSE from the kernel.

4.7 Byte-dumping mode

In addition to paging, there is a "byte dump" mode if the user tries to display an entire file larger than a threshold (50,000 bytes). The idea behind this mode is to prevent users from downloading a file larger than intended. WebBrowserFileSystem first requests header data and reads the Content-Length parameter to avoid downloading any large files without express user authorization. Here is an example of dumping the entire minified jQuery JavaScript code:

```
$ cat https:\\\\code.jquery.com\\jquery-3.5.0.min.js
```

The other terminal now contains the prompt, "This URL has a large content length (90112). Using a paging command such as 'more' or 'less' is preferred. How many of these bytes do you want? [1-90112] (0 to exit):". The prompts continue until either the user types '0' or reads the entire file. This happens because, as far as the system call is concerned, the action isn't completed until the entire file has been `seek`ed. The current threshold (50,000 bytes) is much too low for practical purposes. It was set to a low number for easy testing and demoing.

5 SQLite

SQLite is a lightweight, self-contained, and serverless relational database that is widely adopted due to its ease of use, flexibility, and exhaustive testing [17]. SQLite's compact size and minimal dependenies make it a natural choice for a simple applications such as WebBrowserFileSystem. SQLite allows powerful SQL queries, transactions, and has ACID compliance. This power can be harnessed by a wide range of applications from simple data storage to complex data management.

5.1 Caching in WebBrowserFileSystem using SQLite

Networking can be slow and unreliable due to factors such as limited bandwidth, network congestion, high latency, packet loss, and packet corruption. These issues cause delays, interruptions, and data errors during communication. Therefore, networking should be kept to a minimum.

One way to achieve this is to avoid downloading the same remote content multiple times. WebBrowserFileSystem employs a caching mechanism using SQLite. Once a URL is visited, its content and metadata is saved in a local SQLite database. On subsequent commands, before making any network requests, WebBrowserFileSystem checks the indexed URL field. If it finds the content, it returns the cached from the database, thus bypassing all networking. Otherwise, it downloads the data for the first time then stores the result in SQLite.

5.2 WebBrowserFileSystem schema

A database schema defines the structure, organization, and relationships of a database. The schema defines the tables, fields, constraints, and other attributes that define the data model.

WebBrowserFileSystem only requires one table with four fields: URL, PATH, HTML, and TIME. Table 2 shows the schema structure and example records.

URL	PATH	HTML	TIME
www.example.com	www.example.com	<!doctype html>...	1685370500
https://www.northwestern.edu	www.northwestern.edu	<!DOCTYPE html>...	1685370516
s3://new-bucket-travis-near-1/index.html	index.html	<html><body>...	1685370520
ftp://ftp.slackware.com/welcome.msg	welcome.msg	Oregon State University	1685370527

Table 2: WebBrowserFileSystem schema and example records.

The meanings of the record names are as follows:

- URL: (text, primary key, indexed) the visited URL (or URI) by the user. This is the path to the resource which was downloaded and has backslashes replaced with the proper slash character.
- PATH: (text) a shortened, descriptive version of the URL used as the file name in the mounted directory. The PATH applies heuristics to remove slash characters and shorten the URL to a meaningful name.
- HTML: (text) the textual content of the downloaded resource.
- TIME: (integer) the timestamp of when the content was saved to the database. This field is used to set the timestamp attribute of the file.

5.3 WebBrowserFileSystem benchmarking

Caching Internet content is crucial in enhancing performance and reducing network congestion. Caching content improves response times, reduces bandwidth usage, and enhances overall user experience by delivering content quickly and efficiently.

Using the schema above, SQLite caches the response of all its queries. On subsequent usages, it queries against the indexed primary key *URL* to see if the URL already exists. If it doesn't, it must fetch the data from a web server. This introduces a delay where the CPU is mostly idle.

The example below shows the timing metrics of downloading three URLs the first time compared (not cached) to the second time (cached):

```
$ cat /tmp/cmds.txt
```

```
cat example.com/path 2> /dev/null
cat https:\\\\www.amazon.com 2> /dev/null
cat https:\\\\www.northwestern.edu 2> /dev/null
```

The `source` command will run each line. This is wrapped within the `time` command to measure the duration:

```
#1: Time to download the three URLs above:
$ time (source /tmp/cmds.txt) > /dev/null
real    0.64s
user    0.01s
sys     0.00s
cpu     1%

#2: The time of returning cached content:
$ time (source /tmp/cmds.txt) > /dev/null
real    0.01s
user    0.01s
sys     0.00s
cpu     82%
```

The first run shows a comparatively long duration (0.64 seconds) and minimal CPU usage. This slow time and low CPU usage are an artifact of networking: most of the processing is done on other computers while your CPU waits for data to return.

The second run shows a remarkable decrease to 0.01 seconds with higher CPU utilization. This indicates that the subsequent request never left the local machine. This machine's CPU is now doing all the work of querying the database and retrieving the cached content to display to the terminal.

6 Chaining commands

Chaining operators, such as pipe ('|') and redirect operators ('>', '>>', '<') are essential in Linux scripting as they allow powerful data manipulation in a convenient and readable syntax. The pipe operator allows the output of one command to serve as the input for another. This single character allows the versatile chaining of commands to allow complex operations. Redirect operators control where the output of a command is directed. They allow redirecting output to files, streams, or other programs.

These operators, in tandem with Linux commands, are natively supported in WebBrowserFileSystem. Consider this example which downloads all the Northwestern University computer science professor names amongst the sea of HTML markup containing their biographies:

```
$ awk 'match($0, /\/(profiles|affiliated)\/.*?\.html">([^<]+)</, arr) { print arr[2] }'
https:\\\\www.mccormick.northwestern.edu\\computer-science\\people\\faculty.html |
sed 's/ / /g' > /tmp/nu.txt > /tmp/nu.txt | less
```

This command matches all computer science professors (plus professors in other departments who are affiliated with computer science). It does this by pattern matching the professor name within the HTML markup using `awk`. It uses a capture group [15] to capture and print just the professor name. Then it uses `sed` to replace non-breaking spaces with regular spaces to make the plaintext output prettier. Finally, it dumps the output in a local `/tmp` file and pipes it to `less` to allow users to page through this growing list of NU CS faculty.

All this works in WebBrowserFileSystem without any special casing. This means that nowhere does it need to detect which command was run (`awk`, `grep`, `cat`, `ls`, etc). because it intercepts *system calls*. All these Linux commands leverage many of the same system calls. Therefore, every Linux command works. This example demonstrates the power of using FUSE over other designs such as modifying libc.

7 Testing

Memory testing low-level software is essential for ensuring its reliability and stability by identifying and diagnosing memory-related issues, such as memory leaks, file handle leaks, buffer overflows, and uninitialized memory accesses.

These issues can lead to crashes, data corruption, and security vulnerabilities. Rigorous testing of low-level software is vital for building robust and secure applications which are depended on daily by millions of users.

WebBrowserFileSystem contains about 2800 lines of code, half of which are tests (separated in the `/test` and `/src` directories):

```
$ find . -name '*.[ch]' | xargs wc
  130   398  3280 ./test/fuzz/fuzzTarget.c
 1389  3601 34952 ./test/test.c
   18    26   247 ./src/fuseData.h
   37   149  1356 ./src/operations.h
   59   155  1418 ./src/WebBrowserFileSystem.c
   22    68   572 ./src/linkedList.h
   21    37   384 ./src/fuseData.c
   40    87   852 ./src/website.c
   98   233  1973 ./src/linkedList.c
   25    63   626 ./src/sqlite.h
   18    38   306 ./src/website.h
  255   821  6479 ./src/sqlite.c
  369  1215 10474 ./src/operations.c
   28    91   803 ./src/util.h
  304   895  7582 ./src/util.c
 2813  7877 71304 total
```

7.1 Gcov

GNU's Gcov library is a code coverage analysis tool provided by the GNU project that helps measure the extent to which a program's source code is exercised during testing [3]. Gcov collects metrics about the executed code and produces detailed coverage reports. This tool enables developers to assess the effectiveness of their test suites and identify areas of the code that require additional testing.

WebBrowserFileSystem achieves 100% branch and conditional coverage through its comprehensive test suite and uses Gcov to collect the metrics:

```
                        GCC Code Coverage Report
File                                       Lines    Exec  Cover   Missing
------------------------------------------------------------------------
fuseData.c                                     9       9   100%
linkedList.c                                  45      45   100%
operations.c                                 137     137   100%
sqlite.c                                     130     130   100%
util.c                                       141     141   100%
website.c                                     25      25   100%
------------------------------------------------------------------------
TOTAL                                        487     487   100%
```

7.2 Valgrind

Valgrind is a powerful dynamic analysis tool that helps detect and diagnose various memory-related issues in C and C++ such as memory leaks, invalid memory accesses, and uninitialized variables. Fixing these memory errors improves the stability, reliability, and security of software [20]. Although Valgrind is powerful, it also comes with an order of magnitude overhead. This prevents its direct usage on large applications. Instead, developers typically run Valgrind on their tests or a stub executable.

Languages such as C and C++ do not have garbage collection. Developers must manage memory allocation and deallocation on the heap. Forgetting to free memory (or freeing multiple times) is not something the compiler or runtime environment is equipped to detect. Valgrind fills this gap by analyzing your binary while it is being run to find memory issues.

Running Valgrind against WebBrowserFileSystem's test suite revealed numerous examples of leaked memory, leaked file handles, forgotten initialization, and undefined behavior. Memory is particularly easy to leak in C compared to C++

because the latter language supports destructors. The destructor is called automatically when a variable goes out of scope. Imagine a class holding onto a dynamic resource. When the class instance goes out of scope, it calls the destructor to free the memory. This is particularly powerful when there exist multiple code paths. In WebBrowserFileSystem's C code, there were many such instances of an early return statement added later to handle an edge-case bug gracefully. This early return then prevented memory from being freed later in the function.

The C code below illustrates this idea (albeit in a contrived way). The dynamically-allocated memory returned by getData is owned by the data pointer. Assume this memory is not passed to any other functions. Therefore, the memory gets freed for every code path (N=1):

```
char *data = getData(); // returns dynamically-allocated memory
// <use data>
free(data);
return 0;
```

There are no memory leaks above. Now assume an engineer finds a crash which needs to be fixed. In this hypothetical scenario, the developer opts to check for an error condition and return early from the function with the error code:

```
char *data = getData();
if (error)
    return -1; // <-- obvious memory leak!

// <use data>
free(data);
return 0;
```

The memory leak is obvious in this small example (free is not called during the early return), but these issues are incredibly hard to detect in larger code bases as the number of bug fixes and enhancements accumulate. Valgrind is almost mandatory to detect memory leaks at scale.

Does a clean Valgrind log with 100% testing coverage indicate no memory issues? Not even close! Valgrind can only test what is being run. We need a new tool to synthesize new tests to explore more deeply the application being run. Enter fuzz testing.

7.3 Fuzz testing

Fuzz testing (or fuzzing) is a software testing technique that involves providing invalid, unexpected, or random inputs to a program to discover vulnerabilities and bugs [18]. By subjecting a program to a barrage of inputs generated with a focus on edge cases, fuzz testing helps identify security vulnerabilities, crashes, and unexpected behaviors that traditional testing approaches frequently miss. Its ability to simulate real-world scenarios and uncover hidden bugs makes fuzz testing an essential component of the low-level software development process. Figure 4 shows a coverage-guided fuzz testing feedback loop. Inputs are crafted with an intention to explore more code paths. Upon exercising a new code path, that input is fed back to the input generator which applies techniques such as bit flips, permuting, known integer attacks, and trimming to discover bugs [5].

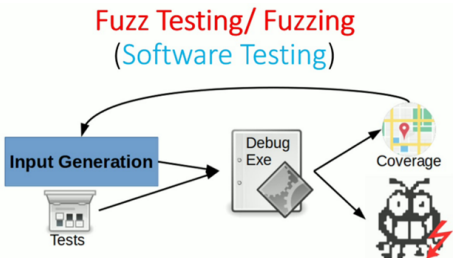

Figure 4: Fuzz testing process. *Image from Cyber Security Entertainment*

7.4 LibFuzzer

WebBrowserFileSystem uses LibFuzzer, an in-process, coverage-guided, evolutionary fuzzing engine [12]. LibFuzzer works by repeatedly executing a target function with mutated inputs, intelligently tracking code coverage to drive the

fuzzing process. LibFuzzer automatically generates new inputs to maximize code coverage and uncover potential bugs in the target program. Its instrumentation-based approach and mutation strategy make it effective in finding software defects.

LibFuzzer's generated input is a pointer to an `uint8_t` byte array. Users call the function to be fuzzed with this byte array, which typically requires a cast due to `uint8_t` being an uncommon data type in practice:

```
// Fuzzing entry point
int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)
{
    // Call the function to be fuzzed
    myFunction((const char *) data, size);
    return 0;
}
```

7.5 Fuzzer corpus

Providing a corpus to a fuzzer is important because it serves as a set of initial inputs that guide the fuzzing process. The corpus consists of valid inputs or test cases that help the fuzzer understand the structure and behavior of the target program. By using a diverse and representative corpus, the fuzzer can explore different execution paths, detect edge cases, and increase the chances of finding bugs or vulnerabilities in a shorter amount of time.

7.6 What makes a function a worthy fuzz target?

The best functions to target for fuzz testing are ones which directly process user-provided input. Imagine this from an attacker's perspective: they will try to craft an input which produces a crash or undefined behavior to gain control of a machine. The easiest route for an attacker is by providing a program an input of their choosing.

7.7 Undefined behavior detected by fuzzing WebBrowserFileSystem

WebBrowserFileSystem contains a function called `convertS3intoURL` which converts s3 URLs into a canonical URL. For example, the input string `"s3://example-com/file.html"` is converted to `"https://example-com.s3.amazonaws.com/file.html"` before being processed by cURL. The function which does takes as input a user-provided string and tries to pattern match and convert. As noted above, this means it is a worthwhile fuzzing target. In addition, developers often only valid inputs or slightly-invalid inputs. Permuted strings, bit flips, and unprintable characters are often forgotten.

LibFuzzer was provided a corpus of valid s3 URLs along with a few which reproduced earlier bugs. Using this corpus as a starting point, LibFuzzer quickly generated this string to reproduce undefined behavior:

`"s3://ample[\\\\-com"`.

You can see the ingenuity behind this generated string. Inside the corpus was the sample string, "s3://example-com/". The generated string (`"s3://ample[\\\\-com"`) also begins with the s3 protocol then contains a rotation and partial deletion of "example-com". The fuzzer added the bracket ('[') character and backslashes. Most importantly, it deleted the trailing slash character.

WebBrowserFileSystem's function `convertS3intoURL` was never tested against strings which contained zero slashes. Due to this testing gap, the original implementation of `convertS3intoURL` overstepped the string's memory boundary thus causing undefined behavior. The fix was simple: use pointer arithmetic and `strncpy` to avoid reading past a string's buffer.

7.8 Dynamic memory testing and fuzz testing are complementary

Tools such as LibFuzzer and Valgrind are complementary. Valgrind only detects issues with the inputs that you supply. Valgrind missed this undefined behavior bug because it was not covered anywhere in the test suite. Fuzzers, however, can intelligently create new inputs to be tested against. The combination of memory tools outlined here have detected countless issues in open-source software ranging from SQLite [16] to QEMU [5] to Chromium [19].

8 mmap and its limitations with FUSE

The mmap system call allows a process to map a file or device into its virtual memory space [10]. For files, it provides a way to access data efficiently by establishing a connection between the process's address space and the file's contents. By using mmap, the process can treat the mapped region as if it were part of its own memory, enabling efficient reading and writing operations without the need for explicit I/O calls. Figure 5 shows mmap's mapping of virtual memory to a file descriptor.

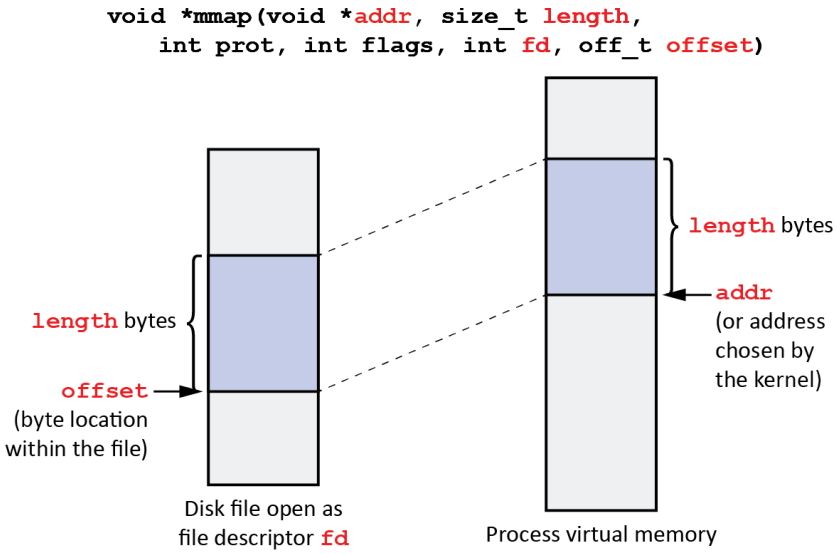

Figure 5: mmap memory mapping. *https://www.clear.rice.edu/comp321*

Using mmap has multiple advantages over traditional file reading methods. First, mmap allows for efficient memory mapping, enabling direct access to file data without the need for explicit read/write calls. This eliminates the overhead of frequent system calls, resulting in improved performance. Second, mmap provides a convenient way to share memory between processes, enabling efficient inter-process communication. Third, mmap allows for memory-mapped I/O, which means that modifications made to the mapped memory are automatically reflected in the underlying file, eliminating the need for explicit write operations.

FUSE currently does offer an mmap or munmap callback in its documented interface [8]. Although FUSE works great when it provides a callback for a functionality you need, FUSE's utility drops once you step outside its domain.

8.1 Trying mmap in WebBrowserFileSystem

WebBrowserFileSystem has a flag (useMmap) to use mmap() but the manner with which it uses mmap is unnatural and not recommended. The flag is currently turned off and requires recompilation to enable. Once enabled, it joins with the FUSE's read() callback, urlfs_read(). This function populates the buffer provided by FUSE with the data the user is reading. When the flag is enabled, WebBrowserFileSystem creates a temporary file with the page's data then mmaps that data. This buffer is then copied to FUSE's buffer followed by an munmap() call to free the memory. Lastly, WebBrowserFileSystem deletes the temporary file. This data flow is shown in Figure 6.

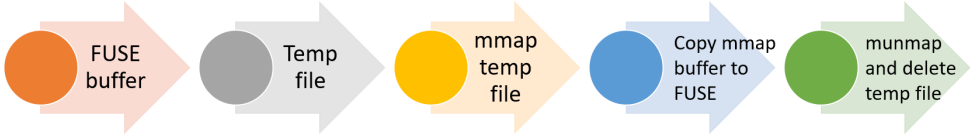

Figure 6: Data flow of mmap with FUSE

This begs the question, why use mmap to create a buffer then throw it away before the system call completes?

8.2 An idealized mmap in FUSE

A more natural usage for `mmap`, which doesn't yet exist, would require changing FUSE's source code. If FUSE offered a callback to return an `mmap` buffer, users could fill in this buffer during the `read` system call. The pseudocode for telling FUSE what buffer to use might look like this:

```
void* urlfs_mmap(const char *path, struct fuse_file_info *fi)
{
    return mmap(NULL, statbuf->size, prot, MAP_PRIVATE, fd, 0);
}
```

Inside this callback, users have complete control over the buffer to use, which file backs it, and the permission attributes. There must also be a corresponding `munmap` callback which frees this buffer when it is no longer needed. This approach would allow developers fine-grained control over the buffer used in the `read` callback. This would simplify the read/write interfaces and allow other processes to access the buffer.

9 `swapon` and its limitations with FUSE

The `swapon` command in UNIX is used to enable the swapping space on an operating system. It allows a user to specify a file to be used as the swap space when the OS needs to page out RAM. `swapon` initializes the swap space, making it available for the system to use for storing pages of memory that are not currently in use, helping to manage memory resources effectively.

FUSE has the same limitation with `swapon` as it does with `mmap`: there is no native support for it. Proper `swapon` would require changes to FUSE's source code create a `swapon()` callback for clients to hook into. Similarly, a `swapoff()` callback must also be added to disable swapping when done. With these callbacks, users could enable and disable files on disk to be used as swap areas by the OS when when physical RAM is full.

10 Challenges faced

10.1 The slash ('/') character

The slash character ('/') is disallowed on UNIX file names. The Linux kernel reserves this character to separate directories when performing a link path walk [11]. FUSE acts as a bridge between user space and the kernel. Because FUSE's data crosses into and out of the kernel, the slash character gets removed by the kernel (plus everything after it) before the user callback is called. Therefore, WebBrowserFileSystem cannot see any user data after the slash character in the terminal.

This poses a problem for WebBrowserFileSystem because nearly all URLs use the slash character. The documented workaround is for the user to use the backslash character ('\') instead of slash ('/'). The backslash itself must be escaped ('\\'). Therefore, supported URLs are of the form:

```
https:\\\\www.example.com\\data.txt
```

instead of:

```
https://www.example.com/data.txt
```

This unfortunate limitation prevents users from directly copy pasting URLs from their web browser to the terminal. Users must take extra care to replace slash with backslash. However, the slash character is hard restriction imposed by the kernel and must be avoided.

10.2 Where to cache data

All cached data is stored in the SQLite database which consists of a single file called "websites.db". This file is located in the same directory as the WebBrowserFileSystem executable (`src/`).

Initially, WebBrowserFileSystem tried to save the database inside the mounted directory. However, writing and reading to the database from inside the mounted directory triggers many system calls with WebBrowserFileSystem listens to. This recursive callback action completely locked up the entire operating system upon application start. Trial debugging to learn the cause of the hang required restarting the OS after each debug session. This was a painful lesson learned.

10.3 Memory management

Memory management in C is a challenge. Before running Valgrind and LibFuzzer, WebBrowserFileSystem had many memory leaks, file handle leaks, and undefined behaviors. It is crucial to develop test suites from the start so that these tools can efficiently run all relevant tests to identify memory issues. Even something as innocuous as an early-return-bug-fix can introduce a large memory leak which basic regression tests will not catch. Although challenging, coding in the C language helps one to appreciate how the OS manages memory and how the CPU operates. Understanding these operations is necessary for writing efficient code which emits the exact machine instructions you desire.

11 Future enhancement ideas

11.1 Writable data

The data in WebBrowserFileSystem is all read-only. Pushing to the Internet or cloud requires extra configuration and credentials. Although writing is doable, it requires significant design, requirements gathering, and understanding of the underlying protocols and web server architecture. Unfortunately, it was beyond the scope for this project.

11.2 Reading private Amazon S3 data

Please see section 4.5 for a detailed explanation of how this would work.

11.3 Distributed file systems

There are two important efforts from Carnegie Mellon University relevant to this project regarding distributed file systems.

The first is the the Andrew File System (AFS). AFS is a distributed file system designed to provide scalable access to files across a network. It enables users to access their files from any location seamlessly, regardless of the physical location of the data. AFS's distributed nature and caching make it suitable for environments with a large number of users across a network [4].

The second is the Coda filesystem. Coda is a distributed file system which strives for seamless access to shared files even when faced with network failures and disconnections. Coda achieves this by allowing clients to continue accessing and modifying files locally when disconnected, then later synchronize changes with the server upon reconnection. Coda utilizes server replication and caching to ensure data consistency and availability [13].

WebBrowserFileSystem (v1) is scoped to a single, local user using a mounted drive. One of the motivations behind this project is for accessing files located anywhere using the same interface. AFS presents ideas in how to achieve concurrency in such an environment. Additionally, Coda addresses some of the challenges faced when confronted with unreliable networks and users who wish to collaborate. This research would be a natural fit for scaling WebBrowser-FileSystem to company-wide collaboration. The caching WebBrowserFileSystem already uses should fit into the APIs behind these two file systems.

12 Hardware and Software

WebBrowserFileSystem has been tested on Debian (Kali) Linux and Ubuntu Linux virtual machines.

13 Acknowledgements

I would like to thank Professor Dinda for his mentorship throughout this quarter. His limitless of ideas and approaches were instrumental in making WebBrowerFileSystem what it is. Suggesting FUSE yielded more Linux command support than I ever imagined I would accomplish.

14 Conclusion

Cloud data is pervasive in today's collaborative environments. Some of the conveniences of the cloud are weighed down by burden of learning all their different programmatic interfaces. Interfacing with remote data is quite different from that of local files. Classical (and more familiar) UNIX commands such as `cat` and `ls` do not natively work on

remote data. Instead, developers must also learn networking protocols such as `wget` and Amazon S3. To unify viewing both remote data and local data, WebBrowserFileSystem introduces a capability which automatically intercepts system calls and makes network requests automatically for the user so that core Linux commands now succeed for remote data. Users can finally run, for example, `$ cat example.com` in their terminal and see the page's output. With this convenience, developers can use their favorite UNIX commands without having to think about the location of their data.

References

[1] Amazon. Developing with Amazon S3 using the AWS SDKs, and explorers. https://docs.aws.amazon.com/AmazonS3/latest/userguide/UsingAWSSDK.html, 2023.

[2] cURL. curl - Tutorial. https://curl.se/docs/manual.html, 2023.

[3] GNU. Gcov Intro. https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html, 2023.

[4] John H Howard. An Overview of the Andrew File System. http://reports-archive.adm.cs.cmu.edu/anon/itc/CMU-ITC-062.pdf, 1988.

[5] lcamtuf. american fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2023.

[6] libcurl. curl - How To Use. https://curl.se/docs/manpage.html, 2023.

[7] libcurl. libcurl - the multiprotocol file transfer library. https://curl.se/libcurl/, 2023.

[8] libfuse. fuse_operations Struct Reference. https://libfuse.github.io/doxygen/structfuse_operations.html, 2023.

[9] libfuse. The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. https://github.com/libfuse/libfuse, 2023.

[10] Linux. mmap(2) — Linux manual page. https://man7.org/linux/man-pages/man2/mmap.2.html, 2023.

[11] Linux Kernel. namei.c - fs/namei.c - Linux source code (v6.3.6) - Bootlin. https://elixir.bootlin.com/linux/latest/source/fs/namei.c#L2239, 2023.

[12] LLVM. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2023.

[13] M. Satyanarayanan. Coda File System. http://www.coda.cs.cmu.edu/, 2023.

[14] MathWorks. Read lines of file as string array. https://www.mathworks.com/help/matlab/ref/readlines.html, 2023.

[15] regular-expressions.info. Regular Expression Reference: Capturing Groups and Backreferences. https://www.regular-expressions.info/refcapture.html, 2023.

[16] SQLite. How SQLite Is Tested. https://www.sqlite.org/testing.html, 2023.

[17] SQLite. SQLite Home Page. https://www.sqlite.org/index.html, 2023.

[18] Synopsys. What Is Fuzz Testing and How Does It Work. https://www.synopsys.com/glossary/what-is-fuzz-testing.html/, 2023.

[19] The Chromium Projects. Security Bugs–. https://www.chromium.org/Home/chromium-security/bugs/, 2023.

[20] Valgrind. Valgrind Home. https://valgrind.org/, 2023.