Rochester Institute of Technology

B. Thomas Golisano College

of

Computing and Information Sciences

Master of Science in Computing Security

Fuzzing and Analysis of AV1 Multimedia Codec

Monika McKeown (mam1681@rit.edu)

July 26, 2018

Abstract

AV1 is a new competitive video codec being developed by a consortium of companies under the Alliance for Open Media (AOM) organization. Their goal is to create a competitive open source codec that allows royalty free usage for all its users. This project aims to investigate the reference implementations for this codec and attempt to analyze the source code for potential issues or vulnerabilities.

To do so, a series of fuzzing and static analysis tools were used in the reference encoder and decoder, with automation in place to generate encoded files on a daily basis using the latest version of the library. From these reference files, over 100 million fuzzed versions were created and executed with the decoder providing over 100 unique crashes according to the tools utilized. From these fuzzed files and the static analysis results a number of potential vulnerabilities were discovered. An analysis of these revealed that some were theoretical only, while others have practical concerns and exploitability. The vast majority of these were related to memory management errors, with the worst case causing stack overflows and invalid memory reads.

Contents

1 Introduction **5**

2 Literature Review **5**

2.1 Video Codec Competitive Environment 5

2.2 AV1 Reference Implementation Architecture 6

2.2.1 AV1 Bitstream Format 7

2.3 Techniques 7

2.4 Testing 8

2.5 Common Vulnerabilities 10

2.5.1 Buffer Overflows 10

2.5.2 Null Pointer Errors 10

2.5.3 Denial of Service 11

3 Methodology **11**

3.1 Jenkins CI Server 11

3.2 Test Video Files 12

3.3 Fuzzing 12

3.3.1 ZZUF 12

3.3.2 Radamsa 13

3.3.3 American Fuzzy Lop 13

3.3.4 Clang Static Analysis Tool 14

3.3.5 GNU Project Debugger 15

3.4 AddressSanitizer 15

3.5 Valgrind 15

3.6 Real World Applications 15

4 Findings **16**

4.1 Memory Leak (32KB) on test files 16

4.2 NULL write during error handling of invalid files 16

4.3 Invalid Read / Buffer Overflow Vulnerability 16

4.4 Potential NULL write error during encoding 17

4.5 Potential issues with Matrix operations 17

5 Analysis **17**

5.1 Memory Leak (32kb) on test files 17

5.2 NULL write during error handling of invalid files 18

5.3 Invalid Read / Buffer Overflow Vulnerability 20

5.4 Potential NULL write error during encoding 23

5.5 Potential issues with Matrix operations 25

6 Conclusion **27**

Tables, Charts, Illustrations

List of Figures

1 Example build history on Jenkins CI . . . 12
2 Example of AFL command line interface . . . 14
3 Source of memory leak, args.c . . . 18
4 Source of NULL write, aom/src/aom_codec.c . . . 19
5 Caller that triggers error, av1/decoder/decodeframe.c . . . 19
6 Source of Segmentation Fault, aom_dsp/bitreader_buffer.c . . . 20
7 Calculation of `payload_size`: av1/decoder/obu.c . . . 21
8 Running test file, crashing with a stack overflow error . . . 22
9 Running the same test file, without a application crash, just an error message . . . 22
10 Size decoding function: aom/src/aom_integer.c . . . 22
11 Cause of buffer overflow: av1/av1_dx_iface.c . . . 23
12 Example HTML report for the Potential NULL write error . . . 24
13 Cause of buffer overflow: av1/av1_cx_iface.c . . . 24
14 Example of problematic pattern: av1/encoder/rdopt.c . . . 25
15 Example of problematic pattern: av1/encoder/pickrst.c . . . 26

1 Introduction

This project revolves around the testing and analysis of the AV1 video codec and its reference implementation being developed by the Alliance for Open Media. This reference implementation includes encoding and decoding libraries written in the C programming language. However, media encoding and decoding libraries written in languages like C have been known in the past to contain vulnerabilities especially around array bounds and memory management errors.

The AV1 reference implementation is under active development, but is considered finalized. In fact a number of client applications have begun to integrate the reference implementation for client usage [1], this allows testing of the reference implementation with confidence that no major changes will occur and that any issues found could be potentially exploited on clients today.

The testing done will include both fuzzing and static analysis tools to investigate potential vulnerabilities, with a primary focus on the decoder which at the time of writing was more performant and more likely to be targeted because of its use in web browsers and other applications. Additionally the encoder at the time of writing had unacceptable performance for fuzz testing. In initial tests encoding seconds of video would require hours to complete.

To get a better understanding of the application being tested, the literature review will provide a background on the codec, techniques it uses and the architecture of the decoders, with a focus on the details that are most relevant to the findings. After this background information, the methodology for testing and the tools used are discussed followed by a list of the findings. These findings are followed by an in-depth analysis of each finding, discussing why the finding is important and how the vulnerabilities occur. Finally, the paper concludes with a summary and remarks on the findings. Additional literature and documentation is also available as appendices at the end of the paper.

2 Literature Review

2.1 Video Codec Competitive Environment

When studying the AV1 video codec testing and fuzzing for vulnerabilities it was first important to fully understand the competitive environment of the codec to be examined. When looking into the AV1 competition, there were two major codecs that needed to be evaluated and understood; VP9 and High Efficiency Video Coding format (HEVC).

The first important codec to look at is the VP9 format. VP9 is a traditional block-based transform coding format owned by Google and is the predecessor to the AV1 video codec. The intention of this format was to be an open web media format. VP9 is the direct successor to VP8 with improvements made from generational progression. There was one major factor that contributed to the improved coding efficiency of VP9, the larger prediction block sizes. Otherwise known as super-blocks. These super-blocks encode data in blocks as large as 64x64, down to 4x4. Traditionally the range of these blocks has been small for computational reasons [2]. With AV1 being the successor of VP9, some techniques will be reused. Understanding designers of AV1 previous experience with VP9 is useful to understanding their decisions to use certain features over others in the new AV1 codec, that will be tested and fuzzed.

The second important codec to look at, and the major competitor to AV1 and its predecessor VP9, is the High Efficiency Video Coding (HEVC) format. HEVC is a video compression standard that is known to be very efficient. What makes HEVC efficient is that it benefits from the use of larger coding tree unit (CTU) sizes. It was designed to be more efficient while reducing bitrate requirements by half but keeping the image quality comparable to that of competing formats [3]. Since AV1 and HEVC are in direct competition, it is good to have a basic understanding of the techniques utilized by HEVC to compare against what's done in AV1. Any potential vulnerabilities in shared techniques could affect both formats.

Finally looking at existing comparisons of the two codecs can be useful. In "Comparison of compression efficiency between HEVC/H.265 and VP9 based on subjective assessments" the authors compare the VP9 and HEVC codecs against each other. They go into great detail, explaining the types of tests that both of these two formats went through. These tests were meant to stress the different strengths and weaknesses of each format against each other. In the results the authors state that HEVC offers improvements in compression performance when compared to VP9. They also found that VP9 seems particularly efficient and similar in performance to HEVC for synthetic video content [4]. Knowing and understanding each formats strengths and weaknesses is useful when trying to understand each codec. VP9's strengths should translate to its successor AV1. This understanding might help in the fuzzing and testing steps, or potentially point out areas that the designers might have focused on in order to better compete against HEVC.

2.2 AV1 Reference Implementation Architecture

The AV1 Reference Implementation, written in C [1], is split into two major components, a decoding and encoding library. These two libraries are meant to be usable independent of each other, with a goal of being embedded in other applications. Example encoding and decoding applications are also provided.

Both the decoder and encoder libraries are built to take advantage of multiple processors and exploit parallelism. They do so by creating a number of worker threads. These threads are provided a data structure that contains all the information needed to do the encoding or decoding task for a subset of the video. After these tasks are done, a final thread merges the results. This approach is known by many different names, like Thread Pools or Work Queues [5].

2.2.1 AV1 Bitstream Format

The AV1 codec bitstream format is composed of a series of Open Bitstream Units (OBU). These OBUs are used in one of two ways, first generally as a stream of frames that result in the video image. The second is a series of tiles that allows a decoder to extract only an interesting section in a frame without the need to decompress the entire frame [6].

Each OBU is composed of a header and a content block. The header specifies how the content block should be interpreted and contains additional metadata about the specific image and how it was encoded [7]. Metadata related to the video as a whole is isolated in the container format, like WebM [8].

2.3 Techniques

After an examination of the competitive environment knowing the tools and techniques used by the codec is of great importance to fully understand how the codec actually functions. One such technique is Geometry-Adaptive block partitioning.

Geometry-Adaptive block partitioning was developed to improve issues seen in rate-distortion (R-D) performance of quad-tree partitions that are commonly used in image and video coding. The results of the authors' experiments showed that significantly improved R-D performance is achieved. They achieve this by studying the use of geometry-adaptive intra models, where wedgelet like discontinuities are used in order to define separate coding regions where different statistical/waveform modeling tools can be used [9]. This technique offers some insight into the different operations that will be occurring during the encoding and decoding phases of the AV1 codec. Understanding how this technique works will allow for better testing, allowing me to fuzz data in a specific manner so as to cause potential problems.

Another technique used is called Discrete Cosine Transform quantization matrices. This technique treats each DCT coefficient as an approximation. The DCT quantization errors are adjusted by contrast sensitivity, light adaptation, and contrast masking, and are pooled non-linearly over the blocks of the image for each quantization matrix. A second nonlinear pooling is then done and results in a perceptual error. This model allows for the estimation of the quantization matrix for a particular image in a memory efficient manner. This technique is very useful to achieve a better compression while offering little to no perceptual differences [10]. This technique is not new, and has been used in images successfully over the years, however AV1 is introducing it for video. Specifically, the understanding of how these matrix multiplications work help us look for potential over and underflows in the math. When fuzzing, I could potentially craft special DTC matrices that over or underflow in an attempt to crash or access memory.

Norbert Wiener created and proposed the Wiener filter in the 1940s with its publication in 1942 originally as a classified document. This filter is used for signal processing with its purpose being to reduce noise found in signals, by attempting to minimize the mean square error, a common measurement in the imaging field [11]. The filter while simple can produce acceptable results and is still in use today. In fact, AV1 utilizes it for parts of its encoding and decoding processes [12].

2.4 Testing

One of the most important things and the main purpose of this project is testing for potential security issues. The previous two sections aid with this by providing important information that could point to various things to be tested. But it is also without question necessary to look up actual testing techniques themselves.

A common testing technique is called fuzzing. Haller et. al in their paper "Dowsing for overflows: A guided fuzzer to find buffer boundary violations" define fuzzing as feeding programs invalid, unexpected, or random data to see if they crash or exhibit unexpected behavior[13]. There's a number or ways this can be done, the most simple one is flip bits at random. Others, like their proposed "Dowser" take different approaches. For example, "Dowser" takes a very specific approach looking at over- and under-flows. With their new approach they were able to find problems in a number of open source applications, that other competing testing tools were unable to find or identify. To do so it used a combination of techniques that focus on tight loops that include array accesses, a common pattern in multimedia encoders / decoders.

Another fuzzing technique is "Driller". This technique is similar to that of "Dowser" but adds on to that technique where it falls short. It is a hybrid vulnerability execution tool that combines dynamic fuzzing and selective concolic execution. Fuzzing is fast and cheap but fails when transitioning between compartments. On the other hand selective concolic execution is highly effective when transitioning between compartments. This combination allows for the discovery of deeper buried bugs. When tested on 126 binaries from the DARPA Cyber Grand Challenge Qualifying Event it identified 77 crashes when other fuzzers only found 68 [14]. This paper was helpful for my research as it provided another fuzzing option that digs deeper into the code. I used these as a comparison tool to see the difference in what they found.

There are other techniques that can be used besides actual fuzzing for example the paper "Static Exploration of Taint-Style Vulnerabilities Found by Fuzzing" discusses the technique of statically doing taint analysis, as an extension of using a pure fuzzing technique due to the limits seen within fuzzers for extent of test coverage, and the availability of fuzzable test cases. Taint vulnerabilities make up the majority of fuzzer discovered program faults. Taint vulnerabilities consist of vulnerabilities where untrusted input manages to modify or alter the values of trusted variables or input. These can cause further issues and lead to other vulnerabilities. The results of their experiments in this paper showed that static vulnerability exploration has the potential to weed out flaws at an early stage of software development. The authors recommend that in the future fuzzing tools be complimented by static analysis tools to help uncover more flaws [15].

Godefroid and Kinder also describe another useful technique combining static and dynamic program analysis, their new approach for testing is focused on the memory safety of programs like codecs, image viewers and media players. Traditional analysis tools have found many vulnerabilities in these types of software, however these tools often don't test floating-point arithmetic, which these types of software tend to use a lot. The authors propose an approach to looking at these types of software with consideration of floating-point arithmetic. Their approach combines a lightweight local path-insensitive static analysis of floating point instructions with a high-precision whole program dynamic analysis of non floating point instructions, with an end goal proving the memory safety of the floating point arithmetic [16]. This paper provides good insight that is applicable to fuzzing and testing AV1. It brings a new angle and area of research for media codecs, that extensively use floating point instructions.

Generally there are two main testing techniques used by developers to look for bugs and discrepancies in their code, static and dynamic. While dynamic mostly involves work by the developer, writing unit and integration tests, static tools are mostly automated and generic. However initially, static analysis tools were used for finding and notifying the programmer of simple discrepancies and minor violations. Today these static analysis tools are more advanced and are capable of finding major bugs and errors that include but are not limited to null pointer dereferences, buffer overruns, and resource leaks. The way static analysis works allows the user to analyze the code without additional input. The results from the tools are produced fairly quickly and are usually actionable. However one has to keep in mind these tools are not perfect and do produce some false positives and cannot prove there are no flaws what so ever in a given code [17]. These days static analysis tools are easily integrated and produce results quickly with little to no extra effort. Even with all the abilities and advancements that have made these tools great and useful, they should never completely replace dynamic testing and other traditional techniques.

There are various forms of data analysis for finding vulnerabilities besides the use of fuzzers. There is also data analysis software used specifically for the analysis of media software. The tools mentioned in the paper "Exposing Vulnerabilities in Media Software" focuses on the stream formats of media software that typical data analysis does not, allowing this tool to potentially find more bugs in this specific type of software [18]. This paper highlights potential blind spots in traditional fuzzing algorithms and techniques in the area of multimedia tools.

2.5 Common Vulnerabilities

2.5.1 Buffer Overflows

Buffer overflows are a common vulnerability found in applications of all kinds. The cause of a buffer overflow can vary, for example trusting user data for critical calculations. But in general a buffer overflow exploit allows a potential attacker to send a piece of data that needs to be read by an application, however due to a bug in the application the application is unable to accurately detect the size of the data. This causes the application to read more data than was available, often reading uninitialized data or unrelated data to the operation requested. This extra data read can either be used to trigger other bugs in the application or to exfiltrate information that wouldn't normally be available [19].

A variant of a buffer overflow, is a stack buffer overflow, generally referred to as a stack overflow. This is a specific type of overflow that occurs on the stack portion of memory. This type is unique in that it potentially allows an attacker to modify or alter the return pointers in an application stack. This allows attackers to modify application flow using a programming technique known as Return Oriented Programming (ROP). [20].

ROP allows attackers to modify the application flow in unexpected or normally impossible ways. Shacham, Hovav, et al. describe how ROP is a Turing-Complete system for computations, allowing attackers to essentially craft whole applications from the manipulation of return pointers alone.

This type of vulnerability is especially problematic to detect, but very hard to exploit. However in certain scenarios, like media players, the damage can be extensive allowing attackers the ability to remotely execute code in clients or servers alike. In the last year there's been at least 5 vulnerabilities of this type in FFmpeg, a popular open source video decoding library: CVE-2017-7866 [21], CVE-2017-7865 [22], CVE-2017-7863 [23], CVE-2017-7862 [24], and CVE-2017-7859 [25]. All these examples and their frequency shows how common of a vulnerability this can be, despite the difficulty it presents to attackers.

2.5.2 Null Pointer Errors

In programming languages that allow access to references, a NULL pointer reference generally refers to the memory at location 0. It is generally considered to be an invalid memory location used to represent a state where data is not present or available. In the C programming language, there are a couple of important notes about this special value. For example, dereferenceing a NULL pointer is considered undefined behavior, triggering a segmentation fault or memory access violation. However, since the behavior is undefined this is just some of the common behaviors. Other behaviors are also possible, and malicious actors could leverage those to perform attacks on a specific piece of software. For example FFmpeg had such a vulnerability

[26]. Additionally this construct has been the source of countless other bugs, errors and vulnerabilities. Its creator, Sir Charles Antony Richard Hoare, calls it his "Billion Dollar Mistake" [27] due to the amount of damage he estimates it has caused.

2.5.3 Denial of Service

While not strictly a vulnerability type, Denial of Service (DoS) is a consequence of vulnerabilities like the ones described above. Denial of Service, refers to an attackers ability to exploit flaws, vulnerabilities, or resources to deny legitimate users access to resources. For example, a denial of service could be the flooding of network resources leading to the inability of normal network services to operate. Attackers could also exploit vulnerabilities like the above to cause additional resources to be utilized, or to become unavailable due to crashes [28].

3 Methodology

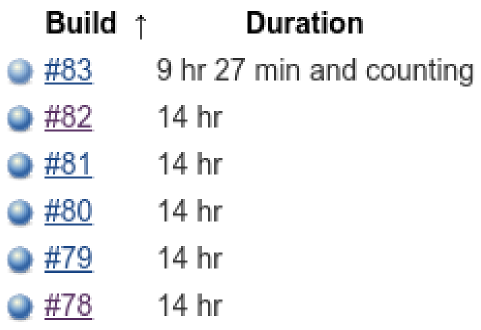
In order to execute on this project, a variety of tools had to be collected, setup, and joined together to effectively analyze the AV1 Reference Encoder and Decoder applications. In this section we describe each tool, their purpose, why it was selected and how it was used in the analysis.

3.1 Jenkins CI Server

At the start this project a Jenkins Server was setup on an Ubuntu 17.10 Virtual Machine (VM) hosted by Digital Ocean for the purpose of compiling the latest versions of the AV1 Encoder and Decoder binaries. Jenkins is a continuous integration server software meant to allow developers and IT personnel to primarily run code related tasks on a schedule or on a trigger, like a source code change [29]. The large number of configuration options and its ubiquity made it an easy choice for this project.

The standard configuration provides a basic interface of shell scripts to execute the commands needed to build a given piece of software. An important feature besides building a piece of software is the ability to archive "artifacts" these can be either the result of the build process or test artifacts. These are then archived and accessible if you wanted to navigate back to previous versions.

For this project, the Jenkins server is accessible via a web interface, or via ssh. The server was configured to build the binary versions of the encoder and decoder every night in two varieties a debug build and a release build with AddressSanitizer enabled. This allowed for the continuous generation of the latest binary versions of the encoder and decoder as active development on the code continued. These would also be automatically archived to make searching for issues between versions easier.



Build ↑	Duration
● #83	9 hr 27 min and counting
● #82	14 hr
● #81	14 hr
● #80	14 hr
● #79	14 hr
● #78	14 hr

Figure 1: Example build history on Jenkins CI

However, early on in this investigation the need to generate test files on a daily basis became necessary due to the forward incompatibility of encoded files and the amount of time that was required to encode test files. The original test files were 30 seconds in duration, and took approximately two days to encode into a final file [1]. To address this time constraint the test file was broken down into multiple smaller parts that could be encoded separately and faster. These one second clips took between two to two and a half hours instead, allowing for the generation of more test files for testing. An example of the time taken for each day can be seen in Figure 1.

3.2 Test Video Files

In order to get a variety of test files to test the encoder with, this project looks at the existing test repository for the MJPEG project [30] [2]. This repository includes a large number of decoded test files for use in testing video encoding technologies. These vary from SD resolution video clips, to full length 4K films.

3.3 Fuzzing

3.3.1 ZZUF

ZZUF is an open source fuzzer that was created by Sam Hocevar (samhocevar) and available on Github. The fuzzer according to the documentation page is a multipurpose fuzzer and a transparent application input fuzzer [31]. ZZUF was created with the intended purpose of, like all fuzzers, finding bugs in an application by way of corrupting user-contributed data in one way or another. Specifically in the case of ZZUF its targets are primarily media players, image viewers and web browsers, which make it perfect for this project.

[1]A number of different configurations were attempted to try and improve this performance, but none made a significant difference.

[2]The AV1 Project does not include these, but make reference to them in their documentation.

To begin fuzzing and testing applications, files and network operations are intercepted by the fuzzer, where then random bits are changed. What makes this fuzzer unique is its behavior. Unlike some other fuzzers, ZZUF is deterministic in its fuzzing. This was done on purpose in order to make reproducing bugs an easier task.

Diving in a little more to the options of running ZZUF, there are two major ones, the fuzzing ratio, and the random seed. The fuzzing ratio option tells the fuzzer what portion of the bits is to be changed. The random seed is used to seed the random number generator used by ZZUF. This option allows for fuzzing in different ways while keeping the fuzzing ratio the same if so desired and generate a different output.

With those options in mind, a series of scripts were written to quickly test thousands of possible combinations of fuzzed test files. The script takes a folder of valid encoded files, and for each test file it will be fuzzed with six different frequency levels, from less frequent than the default to fifty percent of all bits.

The script then captures the output of the decoder, and makes note of any crashes. To help look for potential non-crash situations the decoder used has AddressSanitizer enabled to inform of any memory related issues during the fuzzing step.

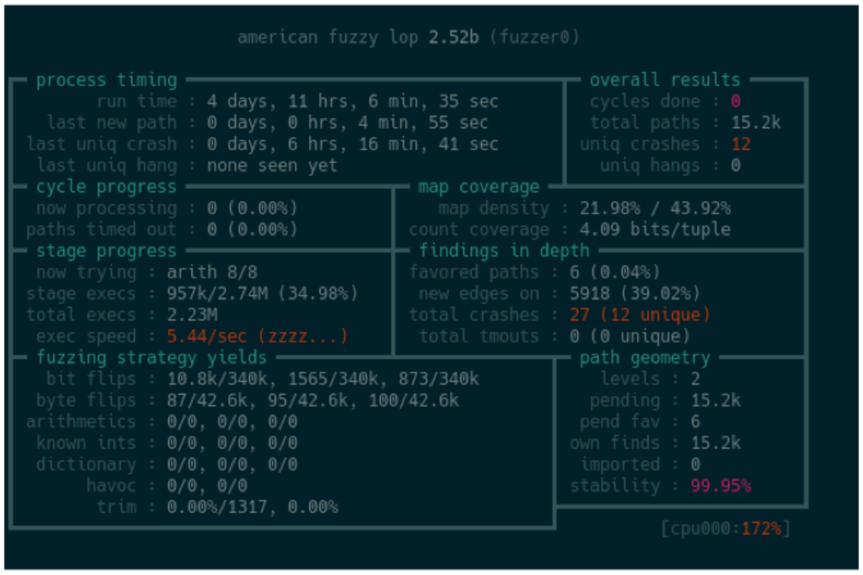
3.3.2 Radamsa

Radamsa is another general purpose fuzzing tool used for this project. It was created by Aki Helin and like ZZUF is available via Github [32]. Like all fuzzers, its purpose is to see how a program will react or withstand malformed or malicious user input. However, it differs from ZZUF in how it determines which bits to flip. It relies and is guided by a series of test files, and not totally random. It uses information from the valid test files and iterates on patterns it detects, fuzzing them for a greater chance of a valid file with flipped bits. Leveraging this information, this tool was scripted to run over 10k iterations looking for potential crashes or issues.

3.3.3 American Fuzzy Lop

American Fuzzy Lop (AFL) is a security-oriented fuzzer, and has an impressive list of both security and non-security bugs identified and improvements to its name. This fuzzer like ZZUF has some unique characteristics and features. According to its documentation, it “employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary.” On top of this it was designed to be practical, and easy to use. It also uses a variety of fuzzing techniques known to be highly effective [33].

In practice AFL works by instrumenting the executable with some instructions to track the path each input file takes within the application. The fuzzer then attempts to use some basic machine learning to



```
                  american fuzzy lop 2.52b (fuzzer0)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 4 days, 11 hrs, 6 min, 35 sec      │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 4 min, 55 sec       │  total paths : 15.2k │
│ last uniq crash : 0 days, 6 hrs, 16 min, 41 sec      │ uniq crashes : 12    │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 21.98% / 43.92%      │
│ paths timed out : 0 (0.00%)         │ count coverage : 4.09 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : arith 8/8             │ favored paths : 6 (0.04%)             │
│ stage execs : 957k/2.74M (34.98%)   │  new edges on : 5918 (39.02%)         │
│ total execs : 2.23M                 │ total crashes : 27 (12 unique)        │
│  exec speed : 5.44/sec (zzzz...)    │  total tmouts : 0 (0 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 10.8k/340k, 1565/340k, 873/340k       │    levels : 2          │
│  byte flips : 87/42.6k, 95/42.6k, 100/42.6k         │   pending : 15.2k      │
│ arithmetics : 0/0, 0/0, 0/0                         │  pend fav : 6          │
│  known ints : 0/0, 0/0, 0/0                         │ own finds : 15.2k      │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : 0          │
│       havoc : 0/0, 0/0                              │ stability : 99.95%     │
│        trim : 0.00%/1317, 0.00%                     ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000:172%]
```

Figure 2: Example of AFL command line interface

determine how to create test files that take different paths with the goal of creating test files that cover as many paths as possible looking for potential crashes or problematic behavior.

The interface AFL presents is simple and easy to follow, an example of it can be seen in Figure 2. The interface is a terminal based UI that displays the current fuzzing progress of the application. You can see the number of paths taken, crashes and "stalls" the fuzzer has found in the application. You can also see general statistics on the different techniques it's attempted to use for fuzzing the test files. This information is very useful to take a quick look at what the fuzzer has achieved.

3.3.4 Clang Static Analysis Tool

Clang Static Analyzer, as the name suggests is a static analysis tool that happens to be open source. This tool was built to analyze code written in C and its many variants like C++ and Objective-C. The documentation about Clang Static Analyzer points out four important things to keep in mind when using this tool. The tool is a continuous work in progress, that is open-source so people can help continuously improve it, static analysis tools are typically slower and take more time than the time needed to compile the application and this tool is no different, static analysis tools are prone to false positives where a "bug" is flagged in the code where the code actually runs without issue, finally, static analysis tools can only identify issues they are programmed to find. Since this project is open source the authors state that they are expanding on the bugs that can be detected and are always willing to take suggestions on new bugs or issues to detect [34].

3.3.5 GNU Project Debugger

GNU Project debugger (GDB), is a general purpose debugger that supports several languages including C, C++, and Objective-C. It provides the ability to see what goes on in a program during run time or after the fact with the assistance of a code dump file. GDB has four main things it can do to help assist in debugging an application.

- On program start-up, specify any parameters that might affect its behavior
- Allows for the placement of breakpoints, to pause execution in a particular location
- Examine the values of variables and program state at any time
- Modify program state to experiment to better understand an applications logic flow

These four features allow you to get a better understanding of an application and find the root cause of a crash or bug easier.

3.4 AddressSanitizer

AddressSanitizer (ASan) is an open source tool written by Google to detect memory errors in C and C++ applications. This tool works well for detecting several types of buffer overflows like stack and heap as well as many other errors. AddressSanitizer works by making changes on how an application allocates and reads memory, inserting markers between memory locations to detect when memory areas overflow. AddressSanitizer can also detect memory leaks and invalid memory accesses via other mechanisms. It also includes integrations with the GNU Project Debugger (GDB) [35].

3.5 Valgrind

Valgrind is an open source dynamic analysis tool with its main focus on memory issues for applications on Linux. Valgrind is aimed at helping automatically detect different memory management bugs, threading issues related to data races and performance profiling. This set of tools can be used in conjunction with other debugging and analysis tools to help track hard to find bugs and issues that other tools might miss [36].

3.6 Real World Applications

While the codec is still in development stages, a number of browsers and media player applications have already included initial support for the codec. Among those are Google Chrome, Firefox, and VLC. For each potentially malicious file, or file that causes a crash or misbehaviour these applications will be used to observe what kind of effect it might have on the client application.

4 Findings

During the analysis done by the tools mentioned above a number of potential issues where found. This section helps understand what the issues are and why they are important. The next section helps us understand why these issues occur and what could be done to mitigate them.

4.1 Memory Leak (32KB) on test files

As part of the testing done with ZZUF and Radamsa, AddressSanitizer detected a memory leak of 32KB. This would occur with fuzzed files that were considered invalid by the OBU header decoding process. This leak however, cannot be exploited further because the error is in the command line interface's argument parsing library.

4.2 NULL write during error handling of invalid files

As part of the testing done with American Fuzzy Lop (AFL), a series of interesting crashes were discovered. These crashes were reported by GDB and AddressSanitizer as invalid write attempts on a null pointer. This invalid write appeared to be occurring in the error handling function for an invalid frame header. In this function, the data structure to store error information for the error is set to null and attempting to assign a new value results in a crash.

This error allows an attacker to potentially craft a malicious file that crashes the decoding application, a browser or another application. Due to the number of protections in place today around null pointer errors, this problem is limited to denial of service. However, if this implementation bug is present in hardware implementations this could have unforeseen consequences.

4.3 Invalid Read / Buffer Overflow Vulnerability

As part of the testing done with American Fuzzy Lop (AFL) a series of interesting crashes were discovered. These crashes were reported by GDB and AddressSanitizer as invalid reads on a given address. According to the first pass with debugging tools, the issue was in the bit reading function for reading OBU headers. However further analysis discussed in the next section revealed this to be a symptom of a potential buffer overflow vulnerability.

This vulnerability allows an attacker to specify a relative memory location from the start of the header. In the context of a standalone decoder this might not seem like a large problem, but once embedded as a library into another application this could give the decoder access to the application memory, leading to potential issues like modifying the application behavior or information disclosure. This is especially problematic for something like a video sharing platform that accepts potentially malicious input.

4.4 Potential NULL write error during encoding

As part of doing static analysis on the full codebase, an interesting set of potential issues were discovered with the assistance of the Clang static analysis tool. The first of which is a potential logic error in `static aom_codec_err_t encoder_encode()`. This potential logic error detected by the Clang static analysis tool can lead to a potential error where `memmove()` is called with a NULL pointer as one of its arguments. This error can lead to a potential denial of service vulnerability when dealing with untrusted or potentially malicious code.

4.5 Potential issues with Matrix operations

Another set of interesting results from static analysis of the codebase utilizing the Clang static analysis tool was the appearance of possible issues with some matrix operations. A number of entries in the results table indicated a potential issue, "Result of operation is garbage or undefined". Upon investigating this message a pattern emerged from some of the code examples seen. A number of the cases highlighted followed the same pattern, initialize a matrix data structure with some initial values in a double for loop, the bounds of which are passed as function arguments. A secondary loop then, uses those bounds for additional calculations without checking the width bounds to be correct. This can lead to uninitialized memory to be read and used for calculations. At best this can result in a bad user result, at worse it might trigger unexpected or undefined behavior in subsequent steps or during the decoding process.

5 Analysis

5.1 Memory Leak (32kb) on test files

As mentioned in the previous section, certain files generated by zzuf and radamsa appeared to leak 32kb of memory according to AddressSanitizer. This was found by running the AddressSanitizer enabled version of the example encoder and decoder applications over the test files generated by zzuf and radamsa.

This leak appears to be of a fixed amount regardless of how a file was fuzzed. The output from the decoder was consistent across the board, an error decoding an invalid file. Enabling additional logging and options in AddressSanitizer configuration revealed that the error originates in one of the decoder's argument parsing functions, seen in Figure 3, specifically in the `malloc()` call made in line 143.

```
142 char **argv_dup(int argc, const char **argv) {
143   char **new_argv = malloc((argc + 1) * sizeof(*argv));
144
145   memcpy(new_argv, argv, argc * sizeof(*argv));
146   new_argv[argc] = NULL;
147   return new_argv;
148 }
```

Figure 3: Source of memory leak, args.c

This indicates that in certain error handling scenarios, the decoding application fails to cleanup all memory allocated. For the purposes of this investigation this is not a concern and can be easily ignored. The decoder binary is provided by the authors as an example on how to consume the underlying library.

5.2 NULL write during error handling of invalid files

The NULL write error during error handling described above occurs when handling invalid files. These specific invalid files were generated by AFL as part of extensive fuzzing sessions. A significant number of the fuzzed files that were reported as unique crashes were due to type of error.

The specific source of the NULL pointer is `void aom_internal_error` whose full source code is available in Figure 4. This function is simply a helper that takes information about an internal decoding error and places it in the data structure used to propagate errors. This procedure is necessary because of the decoder's design.

The decoder utilizes a worker structure to delegate work to different threads. Each thread contains a data structure with all the information needed to decode frames and communicate any data or errors back to the main thread. Other traditional error handling mechanisms wouldn't necessarily be available with this application structure. In this particular case however, the function assumes that the pointer for error information is allocated and points to a valid memory address (a normally correct assumption). This assumption is broken and causes the attempt to write to a NULL pointer and an application crash. In the particular cases encountered the caller that exhibited this error was in the `static int read_uncompressed_header` function, specifically line 3187 – 3188 seen in Figure 5. This error appears to be due to invalid dimensions specified in the header.

This function is utilized throughout the application and works correctly in certain scenarios tested, leading to a clean exit and complete error message. The initial theories before investigating were either a use after

```c
void aom_internal_error(struct aom_internal_error_info *info,
                        aom_codec_err_t error, const char *fmt, ...) {
  va_list ap;

  info->error_code = error; // <- info is NULL
  info->has_detail = 0;

  if (fmt) {
    size_t sz = sizeof(info->detail);

    info->has_detail = 1;
    va_start(ap, fmt);
    vsnprintf(info->detail, sz - 1, fmt, ap);
    va_end(ap);
    info->detail[sz - 1] = '\0';
  }

  if (info->setjmp) longjmp(info->jmp, info->error_code);
}
```

Figure 4: Source of NULL write, aom/src/aom_codec.c

```c
 if ((!av1_is_valid_scale(&ref_buf->sf)))
    aom_internal_error(xd->error_info, AOM_CODEC_UNSUP_BITSTREAM,
                       "Reference frame has invalid dimensions");
```

Figure 5: Caller that triggers error, av1/decoder/decodeframe.c

free or a failure to allocate the memory in the first place. The use after free seemed more likely, but failure to allocate the memory in an edge case could also be reasonable.

Using GDB to step through the execution of the application reveals that the pointer has a valid memory address as its value at the beginning of the decoding step. Stepping through the application reveals that the pointer retains a valid value in the beginning. However due to the complexity of the application the location of the error in the application remains to to be found. Stepping carefully through the application was an extremely fragile and time consuming task. Other tools like Valgrind or AddressSanitizer did provide additional information to assist in this task.

While the source of the NULL pointer has not been identified, an attacker could create test files using Fuzzing tools as done here to trigger crashes in clients, causing a denial of service vulnerability. However, with the information collected it is hard to determine if this is the extent of this error. Depending on how the affected pointer was turned into NULL, different attacks might be theoretically possible.

5.3 Invalid Read / Buffer Overflow Vulnerability

The Invalid Read / Buffer Overflow Vulnerability involves the libraries attempt to read a memory location that is invalid and out of bounds. This generally results in a segmentation fault signal being thrown by the underlying operating system. However, as noted before it is possible that due to how memory is allocated, this out of bounds read might be a valid address leading to a possible information disclosure or further errors.

This specific vulnerability was found with fuzzed files created by AFL. A small number of the unique errors and "stalls" generated by AFL presented the symptoms of this error. Some only caused the invalid read error, while others lead to the buffer overflow vulnerability.

To better understand this vulnerability we must first understand how the decoder code itself is structured and where exactly this vulnerability lies. In Figure 6 you can read the full source code for the offending function that causes the segmentation fault. There is no documentation for this function, but based on the naming, its structure and how it is used / called, you can infer that its purpose is to read a single bit from a buffer structure and return the value.

```c
int aom_rb_read_bit(struct aom_read_bit_buffer *rb) {
  const uint32_t off = rb->bit_offset;
  const uint32_t p = off >> 3;
  const int q = 7 - (int)(off & 0x7);
  if (rb->bit_buffer + p < rb->bit_buffer_end) {
    const int bit = (rb->bit_buffer[p] >> q) & 1; // <-- Segmentation Fault
    rb->bit_offset = off + 1;
    return bit;
  } else {
    if (rb->error_handler) rb->error_handler(rb->error_handler_data);
    return 0;
  }
}
```

Figure 6: Source of Segmentation Fault, aom_dsp/bitreader_buffer.c

There's a number of potentially confusing operations going on but the pseudo code for the function can be summarized as follows:

- Read the current bit offset
- Calculate which byte needs to be read, by doing a bit shift operation
- Calculate the position of the bit within the byte by using a bit mask
- Calculate the memory address for the byte needed to be read, by taking the memory address of the buffer and adding the byte offset calculated above. This is compared with the pre-calculated end of the buffer.

- If the memory address would be larger than the end, this would be considered out of bounds and an error should be triggered

- If the memory address is within the buffer, continue and calculate the bit using the data calculated above for byte and bit offsets with a bit mask applied.

However the above code does not consider the following scenarios: A buffer of size 0 or a buffer of already invalid memory.

In the test files produced you can see both examples fail and lead to a segmentation fault. In the case of an buffer size of 0, the result would be a crash. However, the case where the address is already invalid a malicious actor might be able to craft a file to read specific memory locations leading to other vulnerabilities.

Triggering this second scenario is actually done by exploiting a header type named in code: `OBU_TEMPORAL_TYPE`. This special type seems to inform the decoder that the previously read header should be skipped at this time. Per the official documentation, a Temporal OBU type provides "An indication that the following OBUs will have a different presentation/decoding time stamp from the one of the last frame prior to the temporal delimiter"[37]. However, this previous header read will have modified an important variable for this process `payload_size`.

This size variable is used for determining the buffer for the next header. This value isn't validated and simply trusted to be valid. Figure 7 shows how the value is determined for the first case (Annex B) and how it can miss the above two scenarios of a size 0 or a possibly invalid scenario.

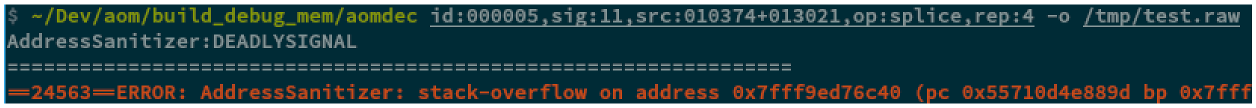
```
// Derive the payload size from the data we've already read
if (obu_size < obu_header->size) return AOM_CODEC_CORRUPT_FRAME;

*payload_size = obu_size - obu_header->size;
```

Figure 7: Calculation of `payload_size`: av1/decoder/obu.c

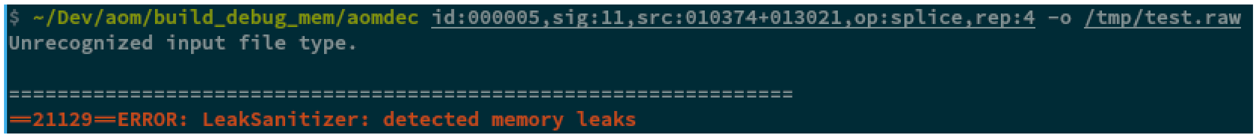
For the second case, Annex A, the calculation relies on the `int aom_uleb_decode()` function, whose full source can be seen in Figure 10. This function looks at the existing buffer, and the number of bytes available, looking to determine the size of the payload. The code then searches the buffer for a value with a leading 0, setting the length to match the number of iterations plus one. However this length value is unused, the true size used is the result of the `*value` calculations. Which based on the logic seen in the code, can explode in value due to the exponential growth due to the use of a bitshift, leading to invalid values.

These invalid values can then either trigger an invalid read immediately or cause the decoder to enter an invalid state and cause a buffer overflow. However, some scenarios might not end up in a buffer overflow, in fact some test files would result in either one of these scenarios or a regular error message about an invalid



```
$ ~/Dev/aom/build_debug_mem/aomdec id:000005,sig:11,src:010374+013021,op:splice,rep:4 -o /tmp/test.raw
AddressSanitizer:DEADLYSIGNAL
=================================================================
==24563==ERROR: AddressSanitizer: stack-overflow on address 0x7fff9ed76c40 (pc 0x55710d4e889d bp 0x7fff
```

Figure 8: Running test file, crashing with a stack overflow error



```
$ ~/Dev/aom/build_debug_mem/aomdec id:000005,sig:11,src:010374+013021,op:splice,rep:4 -o /tmp/test.raw
Unrecognized input file type.

=================================================================
==21129==ERROR: LeakSanitizer: detected memory leaks
```

Figure 9: Running the same test file, without a application crash, just an error message

file. The results can be unpredictable, you can see in Figures 8 and 9 the same file causes two separate errors.

```
int aom_uleb_decode(const uint8_t *buffer, size_t available, uint64_t *value,
                    size_t *length) {
  if (buffer && value) {
    *value = 0;
    for (size_t i = 0; i < kMaximumLeb128Size && i < available; ++i) {
      const uint8_t decoded_byte = *(buffer + i) & kLeb128ByteMask;
      *value |= ((uint64_t)decoded_byte) << (i * 7);
      if ((*(buffer + i) >> 7) == 0) {
        if (length) {
          *length = i + 1;
        }
        return 0;
      }
    }
  }

  // If we get here, either the buffer/value pointers were invalid,
  // or we ran over the available space
  return -1;
}
```

Figure 10: Size decoding function: aom/src/aom_integer.c

In trying to understand why the implementation is complex, the specification document was searched for any potential clues. This leads to the specification of the `leb128()` function. Looking at the pseudocode there it is clear that this is almost a direct translation from the document. Additionally the documentation notes: "It is a requirement of bitstream conformance that the most significant bit of leb128_byte is equal to 0 if i is equal to 7. (This ensures that this syntax descriptor never uses more than 8 bytes.)" [38]. This leads me to believe that this particular sequence has not been hardened against potential malicious input and is susceptible to the types of issues seen above. If this had been hardened against potentially malicious input, you'd see common techniques like boundary checks for "sane" values and validation of the decoded numbers.

This implicit trust of untrusted data is made worse by the fact that for this special OBU header type, this variable is used to determine how many bytes ahead to skip. The code for this logic is in Figure 11. In that code block the data pointer is incremented beyond the allocated memory buffer. `data_sz` is declared as `unsigned int data_sz`, causing an underflow in the subtraction operation.

```
// If the first OBU is a temporal delimiter, skip over it and look at the next
// OBU in the bitstream
if (obu_header.type == OBU_TEMPORAL_DELIMITER) {
  // Skip any associated payload (there shouldn't be one, but just in case)
  data += bytes_read + payload_size;    // <- Problematic line
  data_sz -= bytes_read + payload_size; // <- Problematic line

  status = aom_read_obu_header_and_size(
      data, data_sz, si->is_annexb, &obu_header, &payload_size, &bytes_read);
  if (status != AOM_CODEC_OK) return status;
}
```

Figure 11: Cause of buffer overflow: av1/av1_dx_iface.c

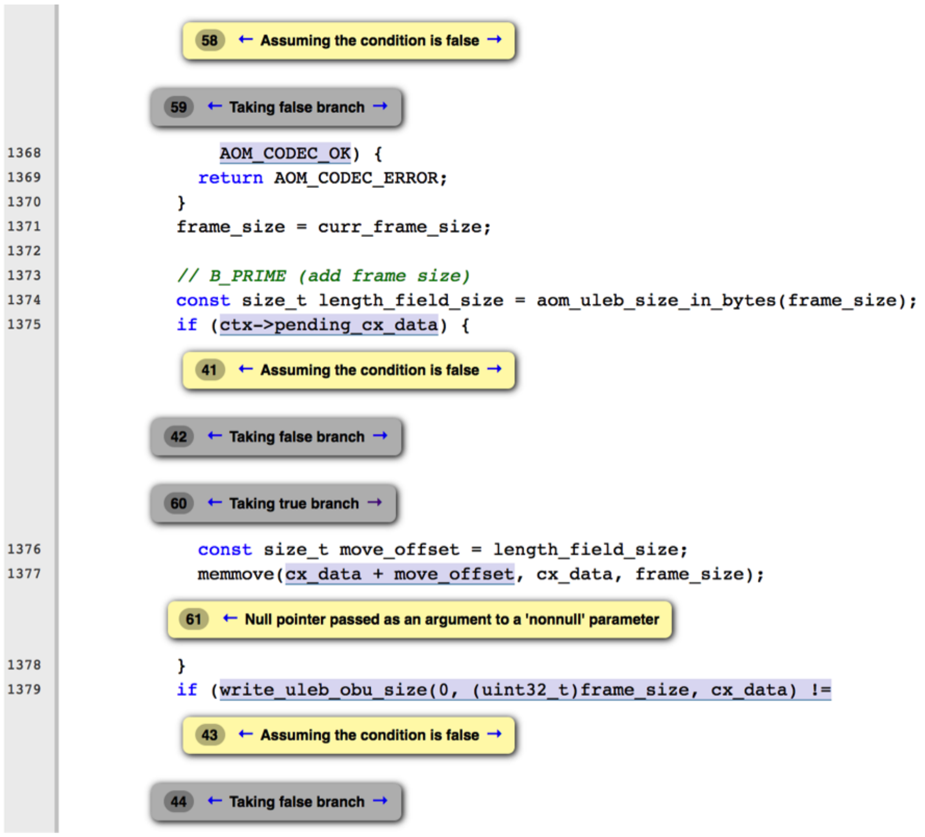
In theory this problem can be caught in this location, by checking that the original `data_sz` is not larger than the `payload_size`. Based on the declaration that it is an unsigned integer, this should be a safe assumption to make, excluding all other values as invalid. However, the decoding specification does not make any concrete statements about it and the existing code provides poor documentation around these internal methods.

5.4 Potential NULL write error during encoding

As mentioned in the previous section, a potential NULL write error was found thanks to the clang static analysis tool. Specifically, the tool pointed out a potential logic error inside of the `static aom_codec_err_t encoder_encode()` function. This potential logic error could lead to a crash in the application, leading to a denial of service. However it is important to note that this is a theoretical problem and not something that is realistically possible.

To better understand why this is a theoretical issue and not a true one we need to look at a limitation in static analysis tools. Static analysis tools have to make certain assumptions about code execution that might not be correct in practice, leading to false positives [17]. Additionally, it's important to highlight that this finding can't be reproduced outside of this theoretical case due to computational requirements.

The encoding application where this issue lies is extremely slow, so careful iterative testing of this issue


58 ← Assuming the condition is false →

59 ← Taking false branch →

```
        AOM_CODEC_OK) {
      return AOM_CODEC_ERROR;
    }
    frame_size = curr_frame_size;

    // B_PRIME (add frame size)
    const size_t length_field_size = aom_uleb_size_in_bytes(frame_size);
    if (ctx->pending_cx_data) {
```

41 ← Assuming the condition is false →

42 ← Taking false branch →

60 ← Taking true branch →

```
      const size_t move_offset = length_field_size;
      memmove(cx_data + move_offset, cx_data, frame_size);
```

61 ← Null pointer passed as an argument to a 'nonnull' parameter

```
    }
    if (write_uleb_obu_size(0, (uint32_t)frame_size, cx_data) !=
```

43 ← Assuming the condition is false →

44 ← Taking false branch →

Figure 12: Example HTML report for the Potential NULL write error

is not practical at this time [3]. However, the clang static analysis tool does provide a nice visual explaination of the steps that would be required to trigger the condition. The tool's HTML report for this vulnerability can be seen in part in Figure 12. The HTML report spells out every condition and step required to reach the descibed error condition.

```
if (ctx->oxcf.save_as_annexb) {
    //  B_PRIME (add TU size)
    size_t tu_size = ctx->pending_cx_data_sz;
    const size_t length_field_size = aom_uleb_size_in_bytes(tu_size);
    if (ctx->pending_cx_data) {
      const size_t move_offset = length_field_size;
      memmove(ctx->pending_cx_data + move_offset, ctx->pending_cx_data,
              tu_size);
    }
```

Figure 13: Cause of buffer overflow: av1/av1_cx_iface.c

With an understanding of the limits of this finding, we can get to understanding the issue in `static aom_codec_err_t enc`
The method in question appears to be an entry point for the encoding process, being provided a frame and

[3]As mentioned in the methodology creating test files took hours, running fuzzers and other tools to target this code path would be non-trivial and time expensive.

some associated metadata data structures. However, during this process it is theoretically possible for a `memmove()` call to be provided a NULL pointer. The problematic `memmove()` call can be seen in Figure 13. This potential error is possible only due to a complex path in the encoder detected by the static analysis tool. This path would require the application to run into an issue in the process of encoding invisible frames. During this process there's a series of 60 conditionals that must happen in a particular way to hit this case. However, working through these steps reveal that there might be multiple conflicting cases relating to AV1 Annex B related checks. If it is possible to trigger these contradicting checks, the issue might be possible. However, from a superficial analysis these do not seem possible rendering this issue impossible in a practical sense, but theoretically possible.

5.5 Potential issues with Matrix operations

As alluded to in the previous section, during the process of running the clang static analysis tool over the source code a number of potential issues around matrix operations were found. In particular in a number of scenarios, the code assumed that the 2 dimensional array being operated on was always of a square shape while having 2 variables for the height and width. Two examples of this can be seen in Figure 14 and Figure 15.

```
for (i = 0; i < bsize_h; i++) {
  for (j = 0; j < bsize_w; j++) {
    e[i * bsize_w + j] = x[i * bsize_w + j] - y[i * bsize_w + j];
  }
}
int mid = OD_DIST_LP_MID;
for (i = 0; i < bsize_h; i++) {
  tmp[i * bsize_w] = mid * e[i * bsize_w] + 2 * e[i * bsize_w + 1];
```

Figure 14: Example of problematic pattern: av1/encoder/rdopt.c

These two functions attempt to do different kinds of matrix operations (the former a statistical distribution function for the $dist_8x8$ experiment, the latter a Weiner Filter implementation) without verification that the array bounds are equal (square). The case highlighted by the static analysis tools in both these scenarios assumes that `bsize_h = 2` and `bsize_w = 1` in this scenario the first set of loops would initialize the values in the result matrix, while the second set would do an operation over this result matrix. However, the second loop relies only on checking or bounding on one of the values but using both in its calculations leading to an overflow into uninitialized memory. This does not lead to an out of bounds memory access because the results matrix is being allocated correctly in all cases observed.

It is interesting to note however that the Weiner filter implementation seen in Figure 15 actually acknowledges the fact that matrices are expected to be of equal length / width and even includes a debug time assertion. While this might prevent developer errors in calling the function, it will not prevent malicious actors from potentially causing the issue here.

```
for (i = v_start; i < v_end; i++) {
  for (j = h_start; j < h_end; j++) {
    const double X = (double)src[i * src_stride + j] - avg;
    int idx = 0;
    for (k = -wiener_halfwin; k <= wiener_halfwin; k++) {
      for (l = -wiener_halfwin; l <= wiener_halfwin; l++) {
        Y[idx] = (double)dgd[(i + l) * dgd_stride + (j + k)] - avg;
        idx++;
      }
    }
    assert(idx == wiener_win2);
    for (k = 0; k < wiener_win2; ++k) {
      double Yk = Y[k];
      M[k] += Yk * X;
      double *H2 = &H[k * wiener_win2];
      H2[k] += Yk * Yk;
      for (l = k + 1; l < wiener_win2; ++l) {
        // H is a symmetric matrix, so we only need to fill out the upper
        // triangle here. We can copy it down to the lower triangle outside
        // the (i, j) loops.
        H2[l] += Yk * Y[l];
      }
    }
  }
}
```

Figure 15: Example of problematic pattern: av1/encoder/pickrst.c

However, the impact of including garbage data in these equations is questionable. At best, a user might see artifacts in their final encoded video or receive a corrupted file. At worst, the incorrect or unexpected values might trigger other unknown bugs in the encoding application leading to further vulnerabilities or exploits, however based on the analysis done here this seems unlikely.

Solving or mitigating this issue can be achieved in two ways, depending on the developers requirements. The first potential solution would be to change all functions operating on matrices to accept a single dimension parameter, ensuring that the height and width are equal and never change. This would require changes to the interface these functions use, but would be less error prone and harder to break. The second alternative would be to include run-time checks against these variables diverging from each other. A simple equality check would be sufficient. These run-time checks should be enabled for all build levels, not just a simple assertion like what's done in Figure 15.

6 Conclusion

In summary, this project discovered a number of vulnerabilities and errors in the AV1 reference library. A number of them, are potentially problematic, and deal with NULL pointers in the error handling and frame size calculations. The latter specifically can potentially lead to arbitrary memory reads and stack buffer overflows. The other errors or vulnerabilities are minor, from a memory leak in the sample decoder to potential logic errors in the encoder found by static analysis.

In the process of finding these, a number of issues and roadblocks where encountered and overcome. The largest one was the lack of performance optimization in the encoder, this lead to the need for automation for test cases and the reduction of scope in test files. Other small issues were, the lack of reproducibility and instability of the files produced by the reference implementation, and the lack of clear documentation on how it works internally.

The findings could have potential security implications for users of the standard library. While in the initial tests, the existing implementations either guarded against the errors or failed in unspecified way that did not affect the parent application. However, if these vulnerabilities continue to go unpatched they could be used to launch other attacks, specifically the stack buffer overflow vulnerability discovered.

As the reference implementation begins to receive additional performance improvements, additional fuzzing could be done on both the encoder and decoder. Due to the effectiveness of American Fuzzy Loop (AFL), encoder fuzzing would likely provide additional possible findings. In leiu of waiting for performance improvements, access to additional computing resources could also allow fuzzing of the encoder. For additional decoder testing, some of the techniques described in the literature review could be applied beyond the guided fuzzing provided by AFL. A implementation of the Driller [14] techniques could prove to be very effective here.

Appendices

Appendix A: Index of Supplemental Storage Media

- Papers
  - AV1 Bitstream & Decoding Process Specification.pdf
  - dowsing for overflows.pdf
  - driller_augmenting-fuzzing-through-selective-symbolic-execution-ndss2016.pdf
  - dtc quantization matrices visually.pdf

  - geometry-adaptive block partitioning.pdf
  - hevc vs vp9.pdf
  - HEVC_Overview.pdf
  - iSEC_Thiel_Exposing_Vulnerabilities_Media_Software_0.pdf
  - memory safety of floating-point computations.pdf
  - static-exploration-of-taint-style.pdf
  - use and limitations of static-analysis tools.pdf
  - vp9-an overview and preliminary results.pdf

- Source Code
  - afl-latest.tgz
  - aom.tar.gz
  - fuzzed_files.tar.gz
  - radamsa-master.tar.gz
  - zzuf-master.zip

- Websites
  - AddressSanitizer · google_sanitizers Wiki · GitHub.pdf
  - Buffer Overflow - OWASP.pdf
  - Clang Static Analyzer.pdf
  - jenkins docs.pdf
  - mozilla-dash-playback-of-av1-video.pdf
  - NVD - CVE-2017-7859.pdf
  - NVD - CVE-2017-7862.pdf
  - NVD - CVE-2017-7863.pdf
  - NVD - CVE-2017-7865.pdf
  - NVD - CVE-2017-7866.pdf
  - NVD - CVE-2017-14225.pdf
  - Understanding Denial-of-Service Attacks _ US-CERT.pdf

- Valgrind Home.pdf
- valgrind_manual.pdf
- WebM Container Guidelines.pdf
- Work Queue - CodeProject.pdf
- Xiph.org Test Media Collection.pdf
- zzuf – Caca Labs.pdf

References

[1] R. Giles and M. Smole. (2017, November) Dash playback of av1 video in firefox. Retrieved May 14, 2018. [Online]. Available: https://hacks.mozilla.org/2017/11/dash-playback-of-av1-video/

[2] D. Mukherjee, J. Bankoski, A. Grange, J. Han, J. Koleszar, P. Wilkins, Y. Xu, and R. Bultje, "The latest open-source video codec vp9-an overview and preliminary results," in *Picture Coding Symposium (PCS), 2013*. IEEE, 2013, pp. 390–393.

[3] G. J. Sullivan, J. Ohm, W.-J. Han, and T. Wiegand, "Overview of the high efficiency video coding (hevc) standard," *IEEE Transactions on circuits and systems for video technology*, vol. 22, no. 12, pp. 1649–1668, 2012.

[4] M. Rerabek and T. Ebrahimi, "Comparison of compression efficiency between hevc/h. 265 and vp9 based on subjective assessments," in *Applications Of Digital Image Processing Xxxvii*, vol. 9217, no. EPFL-CONF-200925. Spie-Int Soc Optical Engineering, 2014.

[5] U. Twig. (2003, February) Work queue - codeproject. Retrieved May 14, 2018. [Online]. Available: https://www.codeproject.com/Articles/3607/Work-Queue

[6] P. de Rivaz and J. Haughton, "Av1 bitstream & decoding process specification," *The Alliance for Open Media*, p. 182, 2018, retrieved April 29, 2018. [Online]. Available: https://aomediacodec.github.io/av1-spec/av1-spec.pdf

[7] ——, "Av1 bitstream & decoding process specification," *The Alliance for Open Media*, pp. 111–181, 2018, retrieved April 29, 2018. [Online]. Available: https://aomediacodec.github.io/av1-spec/av1-spec.pdf

[8] T. W. Project. (2017, January) Webm container guidelines. Retrieved May 14, 2018. [Online]. Available: https://www.webmproject.org/docs/container/

[9] C. Dai, O. D. Escoda, P. Yin, X. Li, and C. Gomila, "Geometry-adaptive block partitioning for intra prediction in image & video coding," in *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, vol. 6. IEEE, 2007, pp. VI–85.

[10] A. B. Watson, "Dct quantization matrices visually optimized for individual images," in *proc. SPIE*, vol. 1913, no. 14, 1993.

[11] R. E. Kalman, "New methods in wiener filtering theory," in *Proceedings of the First Symposium on Engineering Applications of Random Function Theory and Probability, edited by JL Bogdanoff and F. Kozin, John Wiley & Sons, New York*, 1963.

[12] P. de Rivaz and J. Haughton, "Av1 bitstream & decoding process specification," *The Alliance for Open Media*, p. 59, 2018, retrieved April 29, 2018. [Online]. Available: https://aomediacodec.github.io/av1-spec/av1-spec.pdf

[13] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations." in *USENIX Security Symposium*, 2013, pp. 49–64.

[14] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[15] B. Shastry, F. Maggi, F. Yamaguchi, K. Rieck, and J.-P. Seifert, "Static exploration of taint-style vulnerabilities found by fuzzing," *arXiv preprint arXiv:1706.00206*, 2017.

[16] P. Godefroid and J. Kinder, "Proving memory safety of floating-point computations by combining static and dynamic program analysis," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 1–12.

[17] P. Anderson, "The use and limitations of static-analysis tools to improve software quality," *CrossTalk: The Journal of Defense Software Engineering*, vol. 21, no. 6, pp. 18–21, 2008, retrieved May 5, 2018.

[18] D. Thiel, "Exposing vulnerabilities in media software," in *Black Hat conference presentation, BlackHat EU*, 2008.

[19] OWASP. (2016, June) Buffer overflow. Retrieved May 14, 2018. [Online]. Available: https://www.owasp.org/index.php/Buffer_Overflow

[20] H. Shacham, E. Buchanan, R. Roemer, and S. Savage, "Return-oriented programming: Exploits without code injection," *Black Hat USA Briefings (August 2008)*, 2008.

[21] MITRE. (2017, April) Cve-2017-7866 detail. Retrieved May 14, 2018. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-7866

[22] ——. (2017, April) Cve-2017-7865 detail. Retrieved May 14, 2018. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-7865

[23] ——. (2017, April) Cve-2017-7863 detail. Retrieved May 14, 2018. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-7863

[24] ——. (2017, April) Cve-2017-7862 detail. Retrieved May 14, 2018. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-7862

[25] ——. (2017, April) Cve-2017-7859 detail. Retrieved May 14, 2018. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-7859

[26] ——. (2017, April) Cve-2017-14225 detail. Retrieved May 14, 2018. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-14225

[27] T. Hoare, "Null references: The billion dollar mistake," *Presentation at QCon London*, vol. 298, 2009.

[28] "Understanding denial-of-service attacks," November 2009, retrieved June 6, 2018. [Online]. Available: https://www.us-cert.gov/ncas/tips/ST04-015

[29] J. Developers. (2018) Jenkins user documentation. Retrieved April 29, 2018. [Online]. Available: https://jenkins.io/doc/

[30] (2018) Xiph.org video test media. Retrieved April 29, 2018. [Online]. Available: https://media.xiph.org/video/derf/

[31] S. Hocevar. (2015, June) zzuf - multi-purpose fuzzer. Retrieved April 29, 2018. [Online]. Available: http://caca.zoy.org/wiki/zzuf

[32] A. Helin. (2015) Radamsa fuzzer. [Online]. Available: https://github.com/aoh/radamsa

[33] M. Zalewski, "American fuzzy lop (afl) fuzzer," 2017.

[34] Clang. (2016, Nov) Clang static analyzer. Retrieved May 14, 2018. [Online]. Available: https://clang-analyzer.llvm.org/

[35] Google. (2018) Address sanitizer. Retrieved May 18, 2018. [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizer

[36] V. Developers. (2017) Valgrind: About. Retrieved May 18, 2018. [Online]. Available: http://valgrind.org/info/about.html

[37] P. de Rivaz and J. Haughton, "Av1 bitstream & decoding process specification," *The Alliance for Open Media*, p. 5, 2018, retrieved April 29, 2018. [Online]. Available: https://aomediacodec.github.io/av1-spec/av1-spec.pdf

[38] ——, "Av1 bitstream & decoding process specification," *The Alliance for Open Media*, p. 24, 2018, retrieved April 29, 2018. [Online]. Available: https://aomediacodec.github.io/av1-spec/av1-spec.pdf