Artifact Evaluated NDSS
Available
Functional

TwinFuzz: Differential Testing of Video Hardware Acceleration Stacks

Matteo Leonelli, Addison Crump, Meng Wang, Florian Bauckholt, Keno Hassler, Ali Abbasi, Thorsten Holz
CISPA Helmholtz Center for Information Security
({*first.last*}@cispa.de)

***Abstract*—Video hardware acceleration stacks, which include multiple complex layers that interact with software and hardware components, are designed to increase the efficiency and performance of demanding tasks such as video decoding, encoding, and transformation. Their implementation raises security concerns due to the lack of operational transparency. The complexity of their multi-layered architecture makes automated testing difficult, especially due to the lack of observability in post-silicon testing. In particular, the tests must consider five different layers, including all interoperation components: the applications, the drivers supporting the user space, the kernel, the firmware of the acceleration peripherals, and the hardware itself. The introspectability and visibility of each layer gradually decrease deeper along the stack.**

In this paper, we introduce our harness design and testing technique based on differential testing of hardware-accelerated video decoding stacks through an *indirect proxy* target. Our key insight is that we can use a white-box software implementation's code coverage as an indirect software proxy to guide the fuzzing of the unobservable black-box hardware acceleration stack under test. We develop a differential oracle to compare software and hardware-accelerated outputs, identifying *observable differences* in video decoding to *indirectly* guide and explore the hardware-accelerated stack's black-box components. We also present a prototypical implementation of our approach in a tool called TwinFuzz. Our prototype implementation focuses on video processing and demonstrates our method's effectiveness in identifying implementation discrepancies and security vulnerabilities across seven bug classes for four different acceleration frameworks. More specifically, we discovered and responsibly disclosed two security vulnerabilities in the application layer and three in the driver layer. We also identified 15 clusters of inputs that trigger observable differences in the four platforms tested, which could be used for fingerprinting hardware-accelerated and software stacks from the device or web browser. On top of that, we identified vulnerabilities in Firefox and VLC media player, leveraging input replay. Our results highlight the need for robust testing mechanisms for secure and correct hardware acceleration implementations and underscore the importance of better fault localization in differential fuzzing.

I. Introduction

Hardware accelerators are used for a wide range of applications, from consumer devices to enterprise servers and supercomputers, to improve performance and efficiency for specific computing tasks or workloads [57]. These specialized hardware components are designed to run certain types of algorithms more effectively and efficiently than general-purpose CPUs by offloading such tasks to an accelerator. Outsourcing tasks to an accelerator presents an interesting challenge from a security perspective, as these tasks are executed with very limited insight into the accelerator's activities [55], and the entire stack consists of several layers, each of which may contain faults. Recent studies and exploits highlight the vulnerabilities in video encoding and decoding, both in software-only and hardware-accelerated implementations, which can lead to information leaks, denial of service attacks, and remote code execution [20], [51], [52], [72]. Even in the presence of hardware-enforced memory safety sanitization [4], bugs in hardware acceleration can allow for unprevented out-of-bounds writes [46].

In recent years, fuzzing has emerged as a key technique for identifying faults in various types of systems. Due to its effectiveness, fuzzing has attracted considerable interest in both academic research and practical applications, as evidenced by a large number of methods and tools developed during this period [6], [10], [17], [22], [36], [41]. Despite the extensive work on fuzzing software, there is limited research on applying fuzzing techniques to hardware acceleration stacks. The complexity of fuzz testing for hardware-accelerated systems arises from their multi-layered architecture. This architecture includes user-level video applications, libraries that abstract kernel interactions, kernel drivers managing user data and hardware interactions, and the hardware components themselves, which may contain firmware to perform specific tasks. A significant hurdle for fuzzer effectiveness is that many of these layers are not introspectable during runtime [14], [25], [37], [60], [79]. This opacity makes it difficult to evaluate the effectiveness of the fuzzing process, e.g., to determine the extent to which the fuzzer directly influences the hardware or merely interacts with the software layers responsible for orchestrating the interactions in the hardware acceleration stack.

In this paper, we address this problem and present a harness design and testing technique for differential testing of hardware-accelerated video decoding stacks. Our approach tests the different layers of the hardware acceleration stack of a video decoder in a black-box manner [7], [53]. At its core, our approach is based on *differential fuzzing*, a technique typically used to identify inconsistencies in software programs [42]. This method often involves running multiple supposedly equivalent

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240526
www.ndss-symposium.org

programs to perform comparable tasks with the same inputs and compare the output or behavior of these programs under similar conditions. If the results or behavior differ, this could indicate an incorrectness in the target program, which could be a security flaw. We propose to use the software-only implementation for differential testing of the hardware acceleration stack in two ways. First, we guide an unmodified fuzzer and measure its effectiveness by inspecting the code coverage of the software-only implementation of the decoder [50], [75]. Since both the software-only and hardware-accelerated implementations perform decoding, we hypothesize that the code coverage of the software stack and the accelerated stack loosely approximate the state space of the decoding process itself and, consequently, they loosely approximate each other. The intuition is that the software-only implementation can act as an *approximation* of the accelerated implementation to guide the testing process. We use the term *proxy* to remain consistent with existing literature regarding guidance approximation by way of a second implementation [38]. Since fuzzing is ultimately a strategy to explore the state space of the system under test, coverage feedback itself already acts as an *indirect* proxy even when testing a single system; utilizing the coverage of one design to explore another can be considered to be a lower-precision approximation of the state space of the general problem and thus of both systems under test.

Second, we compare the frame output of the software-only decoder with that of the accelerated decoder by examining the dimensions and content of each frame. We implement a so-called *differential oracle* [9], [34], [42] to detect inconsistencies between the two. This method allows the fuzzer to identify user-observable differences in the video frame data, revealing both correctness and memory safety errors without requiring additional instrumentation in the acceleration stacks, which are difficult to introspect. This approach enables us to detect potential faults that may indicate memory safety vulnerabilities undetected by existing sanitization methods. However, this method is only feasible when a second implementation exists, meaning that the independent layers of the stack cannot be differentially tested on their own.

We have implemented a prototype of TwinFuzz and evaluated its effectiveness through a series of experiments and extensive analysis on different platforms and applications. TwinFuzz uses FFmpeg, a popular tool consisting of various libraries and programs for processing video, audio, and other multimedia files and streams, as a proxy for differential testing. We use software code coverage as a means to *indirectly* infer coverage of the hardware acceleration stack. TwinFuzz targets both the hardware-accelerated and the software-only code path, which enables it to detect bugs in both domains. We demonstrate the applicability of TwinFuzz in several case studies of video decoders in different hardware acceleration frameworks.

Our results demonstrate how software and hardware-accelerated implementations can differ, leading to functional disparities in the decoding process between the unaccelerated and accelerated components. We consider any bug discovered within interoperation components of the stack under test to be a *hardware acceleration stack* bug. These differences lead to recognizable differences in the content and size of the frame on the different platforms. Additionally, we discovered five security-relevant vulnerabilities in the tested hardware acceleration stacks. The newly found security vulnerabilities represent memory corruption, such as buffer overflows on the stack or heap and wild pointer dereferences. It is important to note that the layer where the *observable differences* are triggered does not directly pinpoint the fault's root cause; the actual fault could be located in an unobservable lower layer and managed by higher layers. On top of that, we uncovered three additional findings through input replay techniques using the original fuzzing campaign corpora: an information leak in Firefox, a VLC issue on Windows related to the interaction with the video driver, and the rediscovery of potential fingerprinting when hardware decoding is enabled. We responsibly disclosed all findings to the affected vendors and received a bug bounty for our efforts.

In summary, we make the following three key contributions:

- We present a new method for testing video hardware acceleration stacks. We derive a differential oracle that may indicate the presence of both correctness and security-relevant faults by differentially testing software implementations against the corresponding full hardware acceleration stack.
- We propose a technique for *indirectly* guiding an unmodified fuzzer that abstracts over any acceleration stack that is otherwise difficult to introspect.
- We implement a prototype of our approach in a tool called TwinFuzz. In the evaluation, we observe seven bug classes in four different hardware acceleration platforms, including five security bugs reported to vendors, and we perform an extensive analysis of input clusters that trigger observable differences.

To foster further research, we release the source code and evaluation artifacts of TwinFuzz at https://github.com/CISPA-SysSec/twinfuzz.

II. Technical Background

In the following, we present several important testing concepts and discuss the interaction between software and hardware acceleration relevant to our work.

A. Differential Testing

Differential testing techniques complement conventional software testing by effectively uncovering semantic or logic bugs that may not manifest as explicit errors, such as crashes or assertion failures [42]. In particular, differential testing aims to identify system errors or inconsistencies by providing the same input to multiple implementations and observing discrepancies in their behavior or output. In other words, differential testing enables each implementation to be an oracle for the correctness of the other(s). Significant differences in the results for the same inputs are analyzed, and deviations are flagged as potential bugs.

B. Fuzzing

Fuzzing is a dynamic testing method for examining software applications or other systems under test by submitting unexpected, random, or erroneous data to a system to induce unusual behavior and uncover hidden faults [6], [10], [11], [17], [41]. This testing strategy has historically been used to detect unusual and hard-to-find bugs that, if ignored, could be exploited by attackers to undermine system integrity and cause system crashes, data leaks, or unauthorized access [44]. The mechanics of fuzzing are multifaceted and entail a structured exploration of the system under test.

In *mutational fuzzing* [80], we typically guide the fuzzer through code coverage. This important metric quantifies how much the fuzzer has explored the program's codebase and functionality. It is used to assess the thoroughness of the testing process and guide the fuzzer to discover new regions by *mutating* (i.e., randomly modifying) an existing input. Comprehensive coverage guidance ensures that the fuzzer traverses diverse execution paths within the software, thereby increasing the likelihood of identifying latent vulnerabilities in less explored parts of the codebase.

1) Differential Fuzzing: Differential testing has been applied in the fuzzing domain ("differential fuzzing") with great success on targets that have multiple alternative implementations, such as cryptographic algorithms [74], JIT engines [9], deep learning systems [24], and blockchain nodes [78].

To perform differential fuzzing, we need to create a so-called *fuzzing harness*, i.e., a piece of code specifically designed to interface the fuzzer with the tested system, which performs the differential testing. To accomplish this, the harness accepts input and executes two different implementations of the same program on that input, transforming the input as needed and storing the output from each. If the outputs are classified to be inconsistent, the harness emits an error to flag the testcase to the fuzzer.

2) Hardware Fuzzing: Both academia and industry have recently shown increased interest in *hardware fuzzing*. This emerging field aims to move beyond the well-established domain of software fuzzing, where tools such as AFL++ [17], AddressSanitizer [62], LibAFL [18], and KernelAddressSANitizer [68] are crucial for detecting and activating faults. This is particularly relevant for hardware fuzzing, where the oracles and feedback metrics inherent in software fuzzing are either unavailable or need to be tailored to the hardware domain.

Hardware fuzzing can be divided into two categories: *pre-silicon* and *post-silicon*. The former identifies bugs before the production of hardware components, and the latter identifies bugs in hardware that has already been distributed to customers. In pre-silicon research, multiple strategies have been presented to port software fuzzing directly to hardware by utilizing a hardware-provided coverage metric [29], [31], [35], [70]. These strategies often employ differential oracles to detect bugs [29], [35]. In contrast, post-silicon fuzzing must implement guidance strategies that work with hardware that was not designed to be introspectable [38] or avoid depending on guidance strategies entirely [54].

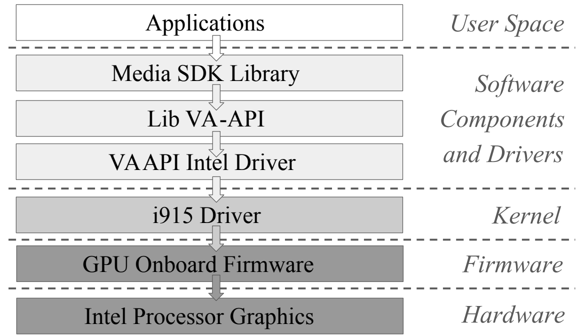

Fig. 1. Overview of Intel's hardware acceleration stack (based on [30]). Diagram entries are shaded according to their introspectability, from most introspectable (white) to least (dark gray).

C. Hardware-Accelerated Video Decoding

Due to the increasing demand for efficient high-resolution video playback and processing, hardware acceleration of video formats is increasingly adopted by both hardware and software vendors. Hardware-accelerated video decoding aims to offload the intensive computational tasks of video decoding from the CPU to the GPU or other specialized hardware acceleration units. The exact implementation of these tasks in software and hardware varies by video format, by hardware manufacturer, and even between generations of video accelerators.

These specialized hardware components are engineered to handle video decoding tasks efficiently; they may be general-purpose parallel computing platforms or tailored to support various video codecs, such as H.264 [32], H.265 (HEVC) [77], VP9 [76], and more. However, developers depend on the user- and kernel-level software to use these hardware components. This software, typically provided by hardware manufacturers such as NVIDIA, AMD, and Intel, facilitates the communication between multimedia applications, the operating system, and the hardware's video decoding capabilities. These software intermediaries offer an abstract interface by which a video decoding library may interact with the hardware.

As an example, Figure 1 provides a high-level overview of the hardware acceleration stack used by Intel. An application interacts with the Media SDK library, but there are four additional software components until the actual hardware is reached, and all of these six components can potentially contain and propagate faults along all the stack layers.

Media libraries, such as *libavcodec* [16], are developed to wrap both a software-only implementation and the corresponding hardware acceleration stack to ensure compatibility across many systems. Should the hardware decoder not support a particular codec, the library will fall back on the slower but semantically equivalent software implementation. Given this need for support across many systems, these libraries offer a consistent interface to end-user applications to perform video decoding. However, this scenario is inherently part of the hardware acceleration stack. Falling back to software-only decoding represents a true negative case in our design since the oracle will not identify any difference in comparing

the software model against itself. This paradigm of *optional* hardware acceleration for operations such as video decoding enables us to perform differential testing between software-only and hardware-accelerated graphics stacks.

III. Design

We now present the design of TwinFuzz, a novel approach to testing hardware acceleration stacks. Our method is based on a harness designed to differentially test video hardware accelerator stacks using an indirect proxy with a white-box software reference. This means that the software implementation serves as a reference model (or *proxy*) for the black-box hardware acceleration stack. The underlying intuition is that the software-only implementation behaves similarly to the hardware-accelerated implementation in terms of processing inputs and generating outputs, as they accomplish the same task and thus must perform semantically similar subtasks. Note that this *indirect* method does not require a model of the hardware or, in fact, any knowledge of its internals. There is no need to collect hardware coverage, which, without access to the hardware design, is challenging and hardly practical. Next, we discuss three key challenges related to fuzzing hardware acceleration stacks and then present the actual design of TwinFuzz.

A. Challenges

We explore the challenges we identified and present our strategies to overcome each of them effectively.

1) Fuzzing Oracle Availability: Crafting effective oracles for systems that cannot be introspected is often challenging because there is no effective way to determine if unexpected behavior occurred, as highlighted in other related works [13], [47]. This is in stark contrast to classical software fuzzing, where tools like *AddressSanitizer* [62] and other sanitizers [64] serve as robust bug oracles. The complexity of developing oracles that accurately detect behavioral anomalies lies in the need to define a reliable ground truth against which the test results can be verified.

Proposed Solution: Our solution involves creating a *differential* oracle design [23], [63]. In our design, we propose comparing decoded frames from two different sources: a hardware-accelerated stack and a software-based decoding stack. The consensus between the two stacks serves as an oracle for validation, as described in Section II-A. Importantly, this oracle not only identifies situations where one implementation is merely incorrect but also situations in which an unobserved memory safety violation occurred that affected the result. Memory safety violations encountered in unobserved drivers or unobservable hardware that affect the computation of the result (e.g., overread of the input buffer, use of uninitialized memory, overwrites that taint neighboring data, etc.) will manifest as *observable differences*. This strategy ensures that we detect issues incurred by the hardware acceleration stack without the need for additional introspection (i.e., sanitizers) in the platform-specific libraries, drivers, and hardware.

2) Black-Box Hardware Acceleration Stack Coverage: For the same reasons that we cannot easily determine whether unexpected behavior occurred, exploring the coverage of the hardware acceleration stack is infeasible in practice. Some components, such as userspace libraries interacting with the constituent kernel drivers that manage hardware acceleration, may be relatively simple to instrument for measuring code coverage. Conversely, collecting code coverage may be harder or even infeasible in other components, such as the kernel driver, the firmware running in the hardware, or the hardware logic itself. As previous studies suggest, even if we collect hardware code coverage, these metrics are unlikely to be suitable for guidance of input generation in post-silicon [65] and in pre-silicon [28] scenarios. Hence, we need a new coverage strategy or an alternative to the classical code coverage to understand how effectively the hardware acceleration stack has been covered.

Proposed Solution: To overcome this challenge, we propose measuring code coverage metrics from the white-box software-only instrumented implementation and using them as indicators for black-box acceleration stack exploration. Intuitively, the coverage of the software-only implementation provides an approximation of the state space of the implemented algorithm. Transitively, as the hardware-accelerated implementation realizes the same algorithm, its coverage must both approximate the same state space and, therefore, be approximated by the software-only implementation's coverage. In the case of hardware acceleration testing, we abstract away not only the hardware components but also the entire uninstrumentable software stack that manages the acceleration.

3) Acceleration Implementation Differences: The testing process for hardware acceleration stacks is considerably more obscure and complex than software testing due to the large number of hardware and driver implementations, which often come from different vendors and require specific ad-hoc testcases and tools for such hard-to-test components. The variability of driver implementations manifests itself in differences in interfaces, functionality, permissions, and security measures. This diversity is a significant obstacle to the consistency and effectiveness of the evaluation process.

Proposed Solution: While implementations of hardware acceleration stacks are highly diverse, many libraries already offer APIs that provide an abstraction over software-only and hardware-accelerated implementations to make hardware acceleration more accessible. Consider widely available libraries like OpenSSL [69], ALSA [3], and FFmpeg [15], which offer abstractions over cryptographic routines, audio processing, and media processing, respectively. We may use the common interface offered by such programs to test the constituent acceleration platforms they use. This insight serves as a general approach to expanding the testing to a broader set of devices and vendors with little effort and many options for optimization based on the chosen system under test. Now, enabling and disabling hardware acceleration is as simple as throwing a proverbial switch, and comparing the outputs is trivial, as the

program's interface must be consistent to ensure compatibility across platforms.

B. High-Level Overview

We hypothesize that there is a correlation between the execution of code at the software-only level and when hardware acceleration is used. This assumption is based on the insight that hardware accelerators are designed to accelerate specific compute-intensive tasks compared to a software implementation running on a general-purpose CPU. The underlying *goal* is the same, so the computation result must be consistent. We assume that certain functions, such as parsing and frame extraction, will directly match between the software- and hardware-accelerated stack. However, other parts, especially related to decoding and optimization, may have different implementations in the two stacks. Based on the intuition that the decoding processes are deterministic and consistent (i.e., the results should be the same), we also hypothesize that despite the lack of a direct feedback guidance mechanism with the hardware, we can still efficiently explore its state space. This indirect proxy technique, in turn, allows us to meaningfully test the hardware acceleration stack and to identify bugs that will occur in practice by exploring code regions and logic edge-cases that would otherwise be hard to reach in a pure black-box hardware testing.

Guided by these rationales, our approach is:

- We stimulate the components of the hardware acceleration stack through their corresponding software libraries and drivers (see also Figure 2 for a high-level overview). This process is driven by code coverage metrics derived from the software-only implementation, which are used to *guide and monitor progress* during the testing phase.
- Within this framework, we use *differential fuzzing* to systematically compare the hardware-accelerated output with the behavior and output of the software-only implementation. This comparison helps us to identify inconsistencies that could indicate unexpected behavior across the entire hardware acceleration stack. Note that we do not attempt to discover bugs within hardware alone but in the entire acceleration stack, including all interoperation components.

The core function of TwinFuzz is to enable the creation of valid video streams by generating and modifying inputs—either from an empty seed or a predefined input—via a generic mutational fuzzer. In this position, TwinFuzz is completely fuzzer-agnostic and does not optimize for a specific engine [61]. Hence, we do not elaborate on its performance in conjunction with different fuzzers; instead, we compare it to other input generation techniques for the same targets. The range of video formats that TwinFuzz can handle is not limited to any specific video codec. It can support all formats supported by the available hardware acceleration devices. When mutational fuzzing is used, providing an initial set of valid inputs to the fuzzer can streamline the exploration of difficult-to-reach code regions because the fuzzer does not need to produce valid inputs from thin air. Hence, we use minimized, valid raw video snippets as seed inputs for our evaluation.

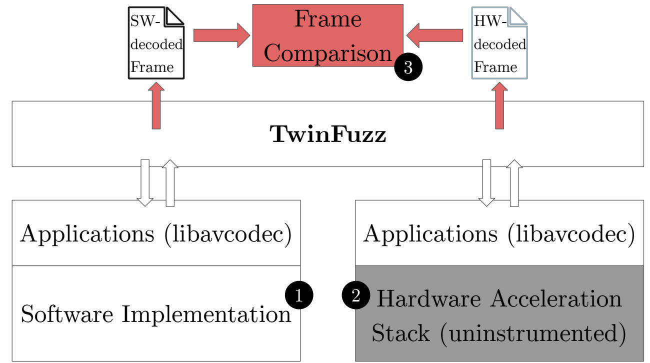

Fig. 2. High-level overview of the design of TwinFuzz. The input is processed with both the software-only and hardware-accelerated stacks, and the output is compared on a per-frame basis. We use the coverage of the software implementation as the feedback loop to guide the testing process.

Upon receiving an input video, TwinFuzz first identifies the appropriate codec. TwinFuzz then ensures that the hardware is correctly configured and supports decoding the respective codec. The video is then segmented into individual frames for processing, and each frame is decoded in parallel by both the white-box instrumented software implementation (❶) and the black-box uninstrumented hardware acceleration stack (❷). After decoding, the frames output by the software-only and hardware-accelerated implementations are analyzed side-by-side to identify discrepancies in size or content (❸). This comparison is crucial to detect possible differences or anomalies in the decoding process. Note that this approach enables the detection of correctness issues and vulnerabilities in APIs, drivers, the kernel, and possibly the hardware itself through using oracles defined only in software. However, additional effort is required to pinpoint the specific cause of an observable difference.

IV. Implementation

In the following, we explain how the TwinFuzz strategy implemented within our harness validates frames and detects faults via *indirect proxy* using differential fuzzing. After that, we discuss several design decisions.

A. Fuzzer Overview

We chose *FFmpeg* as a software proxy to investigate differences in the decoding behavior of hardware-based video accelerators since it is an open-source project, widely used in practice and integrated into multimedia process applications such as Chromium [66] and Firefox [49]. While we use an unmodified *libFuzzer* [40] for our prototype, the implementation is easily portable to other fuzzing engines, such as AFL++.

We compile the userspace software stacks (i.e., FFmpeg and the hardware acceleration *libraries*, but not the drivers) with libFuzzer to report the coverage data and to explore the states of both software and hardware-accelerated decoder, as described in Section III-A2. Internally, the mutations, input selection strategy, and other feedback mechanisms used in our implementation are those of libFuzzer. The harness enables the

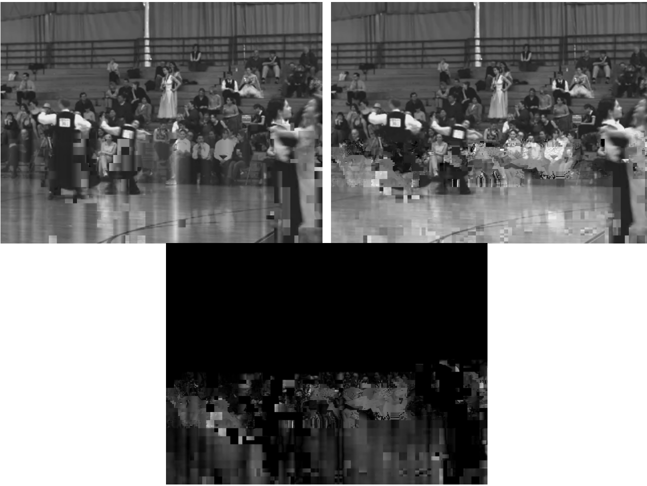

Fig. 3. An example of an observable difference present in a testcase generated by TwinFuzz. Images of the frame as decoded by software (top left), the frame as decoded by hardware (top right), and the difference of these frames (bottom).

fuzzer to detect logical bugs in the decoders by comparing the software and hardware-accelerated decoded outputs frame by frame, starting from the same input. If the two output frames provided are the same, then the decoding process for such input is correct for both implementations. In contrast, if the two output frames are different (see Figure 3 for an example), then at least one of the two implementations is incorrect or incomplete and needs further investigation. Such a scenario will raise a SIGABRT that we label as an *observable difference*.

B. Setup and Fuzzing Harness

OSS-Fuzz [22] is a collaborative project spearheaded by Google and aimed at improving the security of open-source software through automated and continuous fuzz testing, including the FFmpeg project. As noted above, FFmpeg is a free, open-source software project that provides multimedia tools and libraries for handling multimedia data, including audio and video. Our harness is based on the FFmpeg harness provided by the OSS-Fuzz project.

We built this project for multiple devices, as explained in Section V-A. To this end, we adapted the build script for FFmpeg provided in OSS-Fuzz, building the specific and needed dependencies to run the hardware acceleration offered by each platform. Reusing the build script from OSS-Fuzz ensures that the software coverage of FFmpeg is collected by libFuzzer, implementing the coverage guidance mechanism described in Section III-A2.

We extended the OSS-Fuzz harness to leverage FFmpeg's hardware acceleration decoding, retaining as much of the existing harness as possible to stay as close as possible to FFmpeg's provided testing strategy. The harness expects raw video input in different formats, such as H.264 [32], AV1 [2], and so on. It first identifies the video format present in the provided input, discarding the input if it cannot determine a specific format. Once a format is identified, the harness checks if hardware acceleration support is available for the listed format on this platform, discarding the input if it is not. Next, the video is sent to the respective hardware and software decoders in parallel. Afterward, a subroutine walks through the frames of one decoded video stream and compares them to the frames of the other. Concretely, this entails checking the x and y dimensions of the two frames, emitting SIGABRT if they are not equal. Then, the frame's content is checked pixel by pixel, emitting SIGABRT if any two pixels are different. For debugging and illustration purposes, the harness emits both the software and hardware-accelerated frames to disk, as well as the pixel difference of their content, allowing for visual analysis (see Figure 3 as an example). This implements the differential oracle described in Section III-A1.

V. Evaluation

We evaluated TwinFuzz on four hardware platforms in two main aspects: first, we wanted to understand how effective our strategy is compared to existing bug-finding approaches. Second, we examined the bugs found in our cross-platform evaluation and coarsely clustered them into seven classes. As outlined above, TwinFuzz is fully agnostic of the fuzzing engine chosen; hence, we fixed one fuzzer for this evaluation and referred to existing work for comparing the effectiveness of different fuzzers [12], [27], [43].

With regard to effectiveness, we started by running both the software and the hardware-accelerated side of TwinFuzz separately, without our *differential oracle*, to verify that it is indeed the *combination* of both that makes our approach effective. Additionally, we performed a comparative analysis of which code regions of the software-only implementation are covered by TwinFuzz and which are covered by H26Forge [72], the current state-of-the-art in testing hardware video accelerators. This allows us to understand the relationship between TwinFuzz and H26Forge with regard to their corresponding abilities to exercise video decoders. Note that we have to base this comparison on the software-only implementation because it is infeasible to get coverage information from the hardware.

For the second aspect, we clustered the correctness bugs according to the frame difference we observed and the platforms they were present on. Looking at the resulting bug classes, we deducted some underlying properties. Afterward, to determine the bug-finding effectiveness of TwinFuzz, we performed a "time-to-bug" experiment for each of the bugs uncovered in our work. This allowed us to indicate how difficult certain bugs were to find (i.e., how quickly does such a strategy uncover bugs?) and to investigate the overall performance of our tool. Finally, we replayed the corpora generated from our fuzzing campaign to compare our strategy with that of H26Forge for targeting the same applications and demonstrate that we can identify similar bugs.

A. Evaluation Setup

We ran ten trials for each experiment sequentially on a single CPU core for 24 hours each. For each trial, we used three corpus entries (initial seeds). Two are regular H.264 video files, and one is a dummy file consisting of four ASCII characters (`AAAA`)

to reduce the bias induced by the codec of the two other files. We executed the trials sequentially to avoid contention between the CPU and the peripherals responsible for hardware-based acceleration. We consistently applied this evaluation method to all experiments described below to ensure a standardized approach to evaluating the performance and effectiveness of our fuzzer in different scenarios. The harness described in Section IV-B is used for each experiment unless otherwise specified. Where applicable, we followed the guidance by Klees et al. [36] and Schloegel et al. [61] on how to evaluate fuzzers. As explained in Section IV-A, we utilized the libFuzzer [40] runtime. We did not evaluate with other fuzzer runtimes (e.g., AFL++ [17]), as we are evaluating the effectiveness of the general approach, not the specific performance of each runtime with our approach.

Hardware Specifications: We tested our approach on various hardware platforms. Based on the hardware specifications, we compile FFmpeg to support different hardware acceleration drivers based on the target machines. A summary of the platforms under test is provided in Table I. We use the platform labels therein consistently in the following sections and in Table II.

B. Comparison without Proxy or Differential Testing

In our initial fuzzing campaign and exploration, we ran an experiment to test the software-only and hardware-acceleration-only implementations independently. We evaluated this in the same way as the other experiments: ten libFuzzer trials for 24 hours. We instrumented FFmpeg and its software dependencies for fuzzer guidance and enabled ASAN. This experiment did not use a differential oracle, and only invoked one of the software-only or hardware-acceleration-only implementations. In the latter, the only coverage feedback was for the decoding process up until the point at which it was offloaded to the driver or accelerator.

In both experiments, *none* of the fuzzing trials trigger any bug later discovered by TwinFuzz. Since the bugs that we discover using TwinFuzz are either differential issues or memory violations in the software components of the hardware acceleration stack, this suggests both that the software-only parts of FFmpeg are extensively being fuzzed and that a naive harnessing strategy for the hardware-acceleration components is not enough.

C. Comparison of Mutation and Generation Approaches

H26Forge [72] is a framework for analyzing, generating, and manipulating H.264 video files that are valid according to the specification (but potentially not semantically meaningful). When used for fuzzing, this framework allows for a deep exploration of the regions of the acceleration stack that handle valid input files. Such an approach is inherently limited since it only *generates* systematically correct H.264 inputs, which may not exercise regions where *specification-invalid* data is accepted from the target. Unlike the generative fuzzing strategy employed by H26Forge, mutational fuzzing is aware of coverage and thus guides itself toward new regions without knowledge of

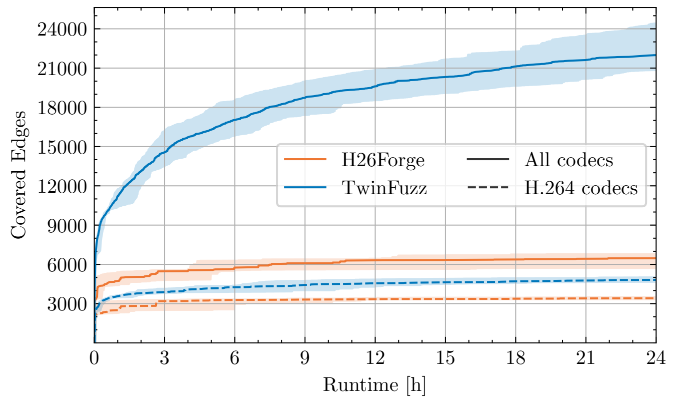

Fig. 4. Coverage over time. Both H26Forge and TwinFuzz are indicated, with solid lines and dashed lines indicating the median coverage over time of 10 runs considering corpus entries of all codecs and only those of the H.264 codec family, respectively. The shading around each median denotes range.

the input format. However, mutational fuzzing is unaware of the specification of any video codec. Evaluating the respective ability of mutational fuzzing and grammar-based generation to explore the state space of video processing tasks is necessary to understand their respective strengths and weaknesses.

To understand this relationship, we set up a fuzzer that solely invoked H26Forge using the same harness as TwinFuzz. In this way, we separately exercised the target platform by traditional mutational fuzzing in the TwinFuzz trials and by H26Forge in the H26Forge trials. Note that this evaluation does not provide insight into the degree to which hardware is tested, which, as described in Section III-A2, cannot be measured. Rather, we used software coverage as a proxy for the state space of the video processing *task*, which is the standard mechanism by which fuzzers are compared to evaluate their ability to explore a program under test [36], [61]. In effect, this evaluation compares how libFuzzer (the fuzzer runtime used in our prototype) and H26Forge respectively explore the state space of the video processing tasks.

Standard fuzzing evaluations consider the aggregate code coverage and compare the coverage as a single value over time, which fails to capture details about *what* code regions are actually covered. Aggregate measurement can be misleading in that coverage is not a single value but rather represents a set of points within the code; for fuzzers of significantly different design, like TwinFuzz and H26Forge, aggregate coverage offers no insight into when one fuzzer can reach certain code regions that another cannot, or vice versa. We provide the aggregate coverage over time plots in Figure 4 to demonstrate the coverage, in keeping with recommendations from Klees et al. [36]. However, we *strongly* emphasize that comparison by aggregate code coverage does not suggest that one strategy is inherently "better" than the other. To understand how these strategies actually compare, we define a *relative coverage* metric based on the Tversky index[1] [71], a measure for asymmetric similarity between two sample sets, that allows

[1]Specifically, this corresponds to the Tversky index with $\alpha = 0, \beta = 1$.

TABLE I
SPECIFICATIONS OF THE PLATFORMS UNDER TEST.

Label	Platform	OS (Kernel)	Graphics Card	Driver (Version)
a) *linux-intel*	Intel GPU	Ubuntu 22.04 (6.8.0-40-generic)	Integrated Intel Iris Xe	Intel Media Drivers (24.1.5)
b) *linux-nvidia*	NVIDIA GPU	Ubuntu 22.04 (6.5.0-41-generic)	NVIDIA RTX 3070 Mobile	NVIDIA's CUDA (535.183.01)
c) *mac-intel*	Intel MacBook	macOS 12.6.8 (Darwin 21.6.0)	Integrated Intel Iris Pro	VideoToolBox (OS-bundled)
d) *mac-arm*	M1 Mac Studio	macOS 14.1.1 (Darwin 23.1.0)	Integrated Media Engine	VideoToolBox (OS-bundled)

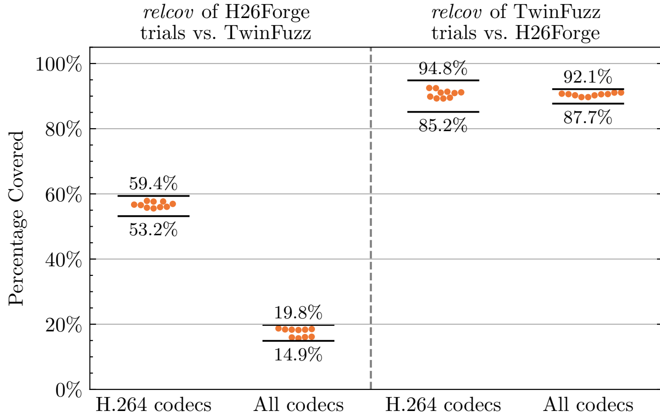

Fig. 5. Relative coverage computed by intersecting each trial with the theoretical "best case" run of the other tool as determined by $\mathrm{relcov}(\mathrm{cov}(t), f_{\mathrm{other}})$. The black bars indicate the theoretical "worst case" $\mathrm{relcov}(\mathrm{lower}(f_1), f_2)$ and "best case" $\mathrm{relcov}(\mathrm{upper}(f_1), f_2)$ results.

us to investigate what code regions one fuzzer can reach compared to another.

We define the fuzzers F, the trials T, and the coverage points[2] P. Additionally, we define functions for the coverage for each trial and the trials conducted for each fuzzer as:

$$\begin{aligned} \mathrm{cov} &: T \to P^* \\ \mathrm{trials} &: F \to T^* \end{aligned} \tag{1}$$

To compute the *relative coverage* of a given trial to the *potential* of a given fuzzer, we define upper, lower, and relcov:

$$\begin{aligned} \mathrm{upper}(f) &= \bigcup \mathrm{cov}(t) \qquad \forall t \in \mathrm{trials}(f) \\ \mathrm{lower}(f) &= \bigcap \mathrm{cov}(t) \qquad \forall t \in \mathrm{trials}(f) \end{aligned} \tag{2}$$

$$\mathrm{relcov}(c, f) = \frac{|c \cap \mathrm{upper}(f)|}{|\mathrm{upper}(f)|} \tag{3}$$

Relcov can be utilized to compute the proportion of coverage points discovered across all trials of a fuzzer f_1 that are reached by a given trial of a second fuzzer f_2. By computing relcov for each trial, we gain insight into how many code points reached by f_1 are reached by f_2 and vice versa. The results of the relcov analysis are provided in Figure 5 and are discussed below.

[2]This is determined specifically by LLVM's PC Guard coverage of edges [67].

To ensure a meaningful comparison, we compare code coverage results once for all inputs and once only for inputs that are supposed to represent video data of the H.264 codec family. Since H26Forge is only designed to generate input files of the H.264 codec family, this allows us to judge relative coverage compared to H26Forge's *intended target code surface* as well as code regions that H26Forge was not designed to explore. Notably, the inputs from the H26Forge trials are either detected as H.264 codecs for 33.2% of the 9,324 input files or rejected as unknown streams (66.8%). TWINFUZZ, on the other hand, is not constrained to any specific codec type. The final corpora for TWINFUZZ (158,475 files[3]) consist of 7.8% H.264-type streams, 25.1% other recognized codecs, and 67.1% invalid files.

We find that a majority of the regions explored by H26Forge are also explored by TWINFUZZ, as demonstrated in the right half of Figure 5. This is true in both cases where we consider all codecs generated by both fuzzers and in the case of only FFmpeg-recognized H.264 inputs. Notably, even in the theoretical worst case formed by lower(TWINFUZZ), we cover 87.7% of the coverage observed by the theoretical best case of H26Forge when considering all codecs and 85.2% when considering only H.264 inputs. This indicates that approximately 10% and 15% of the coverage discovered by H26Forge is *not* discovered by TWINFUZZ in a 24-hour period, respectively. This, in turn, indicates that while most of the coverage discovered by H26Forge is also discoverable by TWINFUZZ, there are some regions that may not be reachable by standard feedback-guided byte mutation alone and require the specialized grammar-based input generation provided by H26Forge. Despite this, even when considering only codecs for which H26Forge was designed, TWINFUZZ still covers the vast majority of the regions H26Forge is capable of covering. This indicates that mutational fuzzing is capable of reaching a majority—but *not* all—of the code regions reachable by grammar-based generation.

In the left half of Figure 5, we assess which regions covered by TWINFUZZ are covered by H26Forge. That is, we identify an upper bound for TWINFUZZ's coverage over 24 hours and determine the intersection of this upper bound with every H26Forge trial, as well as their intersections and union. We find that for each of our trials, less than 20% of the coverage

[3]Since H26Forge generates testcases according to a grammar, it is more likely to cover *many* regions with a single testcase, as compared to TWINFUZZ, which may only make incremental progress. As a result, H26Forge generates fewer files while covering the set of reachable edges, and the produced files are generally more representative.

discovered by TwinFuzz is also discovered by H26Forge. When filtered only for inputs of the H.264 family of codecs, H26Forge still only covers around 60% of the code regions explored by TwinFuzz. This result shows that TwinFuzz's codec-agnostic strategy allows it to cover significantly more of the FFmpeg implementation compared to specification-tailored solutions like H26Forge. Indeed, even for the codecs for which H26Forge was designed, we find that TwinFuzz uncovers a considerable portion of edges undiscovered by H26Forge.

Together, these results indicate that neither the strategy provided by our prototype, TwinFuzz (coverage-guided fuzzing with simple mutations), nor the strategy of H26Forge alone (grammar-based generation) is sufficient to explore the software-only implementation entirely on their own. We infer from this that, while we *do* discover bugs in the hardware acceleration stack across several codecs, it is likely that there are additional latent bugs in each codec that were not triggerable with either our prototype implementation or H26Forge. Simply put, there are bugs of all categories: some discoverable only with TwinFuzz, some discoverable only with H26Forge, some discoverable with both, and certainly some discoverable with neither. We discuss the implications of this result further in Section VI-C.

D. Bug Analysis

Next, we provide an overview of several security vulnerabilities that were uncovered by our approach (see also Table II). These vulnerabilities, identified during the fuzzing campaigns described above, highlight the effectiveness of our technique in detecting security and correctness bugs. In several case studies, we briefly describe the nature of each flaw, its potential impact, and the context in which it was found.

1) Bug 1.x: Observable differences in frame content: The first and most prominent issue we observed during the fuzzing campaigns was observable differences in frame data. These occurred when the software implementation produced a different frame than the hardware-accelerated implementation. In line with the recommendation of Klees et al. [36], we define this issue as a single "bug" for ground truth evaluation, as it has a single, common point of program termination indicated by the emitted SIGABRT when a difference is detected. This makes each "bug" distinguishable for the purposes of our evaluation. Unfortunately, while differential fuzzing is effective in discovering inconsistencies, it differs from the conventional implied oracles offered by sanitizers [64] in terms of pinpointing the exact location of a bug. While sanitizers induce a crash the moment a fault occurs, differential oracles only provide us with an indication of failure after the entire subroutine has returned (e.g., "frames are different"), and multiple factors may contribute to a single observable difference.

To estimate a more precise lower bound of how many unique *actual* bugs were identified that could cause an observable difference, we re-executed testcases discovered during the experiment described in Section V-A from each platform on every platform. This approach allows us to cluster testcases

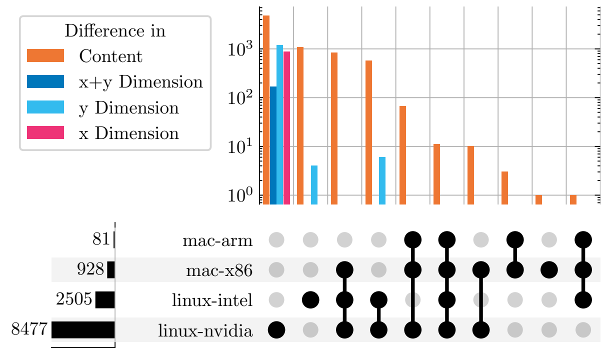

Fig. 6. Distribution of testcases that trigger observable differences by the combinations of platforms that they affect. The horizontal bar chart bottom-left indicates the total number of testcases that triggered observable differences identified per platform denoted on the right. The vertical bar chart indicates the total number of these testcases that occurred on the exact combination of platforms denoted by filled nodes in the diagram below it, categorized by observed difference.

based on the type of observed output differences and the specific combination of platforms where each testcase triggers these differences. Figure 6 summarizes these results in a categorized UpSet diagram [39]. We discuss the results in the following. Overall, we find 15 clusters of inputs based on these measurable constraints. Of particular note is that of the approximately 12,000 testcases gathered, *all* exhibited consistent types of observable differences across the platforms in which a difference was detected. While we are working with the respective vendors and developers to resolve these issues, we do not yet have a ground truth for the accuracy of these clusters, as we cannot yet attribute root cause(s) to testcases that trigger them. Among these clusters, we find a few interesting trends that we discuss next.

Observable differences on only one platform: A stark majority of the discovered testcases capable of triggering observable differences were those that only occurred on a single platform. We interpret this to mean that there are likely several bugs associated specifically with the hardware acceleration provided on these platforms. *Seven* of the clusters identified are in this category, a majority of which are located on the *linux-nvidia* platform.

Observable differences across several platforms: We find no testcases that are unique to the *mac-arm* platform; instead, each of the testcases affecting this platform also affects the *mac-intel* platform, which indicates that these bugs are more likely to be associated with macOS or VideoToolbox than with the components specific to these platforms. Similarly, of the 12,000 discovered testcases, 800 only occurred on x86 platforms. This is likely caused by incorrect x86-specific optimizations in libavcodec or its dependencies. Similar reasoning can be applied to testcases that only trigger observable differences on the Linux-based hosts, where several platforms with an obvious connection between them reproduce the observable differences.

Some of the clusters do not have obvious common components, such as the clusters across *mac-arm*, *mac-intel*, and either of the Linux-based platforms. This may indicate that

TABLE II
BUGS DISCOVERED BY TWINFUZZ DURING LONG-TERM (>24HR) FUZZING CAMPAIGNS AND MANUAL INSPECTION.

Bug ID	Platform	Description	CWE IDs [45]	Layer	Discovery Method	Status
1.a-d	All platforms	Observable difference	204, 474	Undetermined	Fuzzing	Unconfirmed
2	*linux-intel*	Timing-dependent observable difference	204, 362, 474	Undetermined	Fuzzing	Unconfirmed
3	*linux-intel*	Global buffer overflow	126	Application	Fuzzing	Patched
4	*linux-intel*	Heap buffer overflow	122, 787	Driver	Fuzzing	Patched w/ bounty
5	*linux-intel*	Wild pointer dereference	824	Driver	Fuzzing	Disputed
6	*linux-nvidia*	Invalid pointer free	415	Application	Fuzzing	Patched before report
7	*linux-nvidia*	Near-null pointer dereference	824	Application/Driver	Fuzzing	Patched
8	*windows-nvidia*	Information leak on Firefox	908	Application/Driver	Input replay	Confirmed
9	*windows-nvidia*	Windows driver interaction with VLC	476	Application/Driver	Input replay	Reported
10	*linux-intel*	Hardware fingerprinting	497	Hardware	Input replay	Confirmed

the hardware acceleration stacks are consistently incorrectly handling this testcase across multiple platforms or that the software-only implementation is incorrect and merely happens to be incorrect in the same way as the platforms that do not trigger observable differences. It could also indicate a scenario where the software implementation was incorrect, and the hardware acceleration stack fell back to software decoding, thus leading to no observable difference for the testcase. Finally, it is possible that a testcase was generated that triggered more than one root cause of incorrectness across multiple platforms. *Three* of the clusters identified are in this category, and only approximately 100 testcases.

Observable differences across all platforms: Of 12,000 total discovered testcases that produced observable differences in output on any platform, only 11 do so on all platforms. We suspect that the software implementation or a hardware acceleration subroutine present in libavcodec itself likely exhibits incorrect behavior for this set of inputs.

2) Bug 2: Race condition in iHD driver: Beyond analyzing differential behaviors, we also wanted to gain a better understanding of the actual root cause of the observable differences. While inconsistencies between different machines can stem from numerical instabilities or differing feature support, decoding on the same platform should always return the same result. We assume that the existence of inconsistencies present on a single platform could indicate issues resulting from the use of uninitialized memory, data races, and similar error conditions.

We discovered one such testcase on the *linux-intel* platform, which exhibited a curious behavior: the decoding results were inconsistent between two executions on the same software and hardware setup in the way that some runs returned two decoded frames, while others returned only one. We investigated further by inserting an artificial delay before fetching the decoding results using FFmpeg's API and identified the frequency at which each decoding outcome appeared. These measurements are presented in Figure 7. Given that the number of frames available differs according to the timing of the polling for results, we assume that this discrepancy is the result of an improper synchronization issue within the hardware acceleration stack.

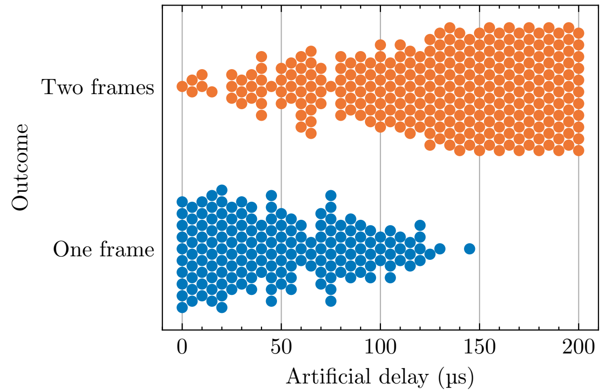

Fig. 7. Distribution of nondeterministic decoding results leading to bug 2 according to the artificial delay inserted before polling. 10 trials per delay increment.

3) Bugs 3-7: Various memory-safety violations: In addition to the testcases that triggered observable differences, we also uncovered testcases that triggered memory-safety bugs in the userspace portions of the hardware acceleration stack. Given that these stacks had not been previously fuzzed, this was somewhat expected. TWINFUZZ was built with *AddressSanitizer* [62] enabled, which allowed us to observe the crash site and cause. We have reported each of these five, found during our fuzzing campaign, potential vulnerabilities to the relevant vendor and we offer a brief description of each in Table II. However, we received acknowledgment and a bug bounty from Intel for Bug 4. We offer a detailed analysis of bugs 3-6 in Appendix A.

Bug 7 is of particular note. This bug was originally reported to NVIDIA by way of their PSIRT reporting portal and, though confirming the presence of the bug, they indicated that the root cause was within FFmpeg but declined to provide further details. When reported to FFmpeg, the developers noted that the sample had caused a frame to end without starting it, leading to a degenerative state in which FFmpeg improperly resets a pointer for a buffer but not the length. As a result, the NVIDIA driver attempts to write to the null page due to lacking a null check. The FFmpeg developers note that, while this was an issue with FFmpeg improperly resetting the pointer/length pair,

they believe that there is a deeper issue within FFmpeg that causes them to signal to the NVIDIA media drivers to end the frame without starting it. Such an issue could only be detected within the context of the full stack and thus highlights the need to test the stack in its entirety, not just individual components.

4) Bugs 8-10: Security violations discovered with input replay: After our fuzzing campaign, we identified three additional potential vulnerabilities by replaying our corpus entries: an information leak in Firefox on the *windows-nvidia* platform, a bug in VLC media player when interacting with Windows video drivers on the *windows-nvidia* platform, and a rediscovered hardware fingerprinting issue related to the hardware-accelerated decoding process on the *linux-intel* platform. These vulnerabilities are further detailed in Table II.

5) Kernel, Firmware, and Hardware bugs: In Table II, we highlight that none of the bugs listed are under kernel or firmware, and only bug 10 is directly associated with the hardware. At this time, though we are confident that correctness and safety bugs in these components were identified as observable differences, the investigation of these issues in greater depth is beyond the scope of this paper, and we label them "Undetermined" in Table II. We further discuss how we attempted to diagnose and identify root causes for these issues in Section VI-B.

E. Time-to-Bug Analysis

As discussed in Section V-D (and detailed further in Appendix A), we encountered multiple bugs during our fuzzing campaigns. We consider any bug discovered within interoperation, including *all* components, such as an acceleration stack bug (i.e., software, libraries, drivers, and firmware interacting with the hardware). Table II provides a high-level overview of the identified bugs. We manually triaged and disclosed the bugs to the vendors in a coordinated way.

In summary, we identified five memory safety violations in the hardware acceleration stack of different platforms. In addition, we identified many scenarios in which the hardware decoder produces visually different results compared to the software decoder. We have grouped these bugs under *bug ID* $1.x$ in Table II, with x denoting the platform on which it was discovered, as reported in Table I. As a matter of terminology, we refer to a *bug* as a single point at which fuzzer execution is terminated (as per the recommendation of Klees et al. [36]) and a *crash* as a testcase which triggered a bug, i.e., there are zero-to-many crashes associated with a single bug. Note that bugs 1.a-d are each considered a different "bug", as they occur on different platforms, which we consider a different location for the purposes of this evaluation.

To understand how effectively TwinFuzz uncovers bugs in a hardware acceleration stack, we ran the 24-hour experiment described in Section V-A and measured the time until each bug was first uncovered. The results are shown in Figure 8. As bug 2 is not observable with this fuzzing configuration, it is not included in this experiment. Bug 3 was not found by any trial during a 24-hour period and is thus excluded from the diagram. Moreover, we excluded bugs 8–10 from the time-to-bug analysis, as they were discovered through input replay technique and not found during our original fuzzing campaign. We observe that the distribution of crashes for bugs 3-7 is centered towards the end of each trial or that no crashes were generated that were caused by these bugs. We interpret this to indicate that inputs that would trigger these bugs were difficult for the fuzzer to find. As the initial seeds for this experiment were H.264 files, this is somewhat expected; none of the bugs 3-7 were associated directly with the processing of H.264 videos. Conversely, bugs 1.a-d were detected typically early in their campaigns or not at all, which is particularly evident in bug 1.c. In these cases, we expect that the fuzzer began exploring H.264 regions, where some of the observable differences were identified, but quickly began exploring other regions, as new codecs were synthesized by mutation. As a result, either the fuzzer identified the bug early when it was prioritizing inputs of H.264 format, or it had difficulty in discovering it later when the corpus was saturated with other types of video files. We expect that, in longer campaigns, the fuzzer would eventually re-prioritize these inputs and discover the associated observable differences.

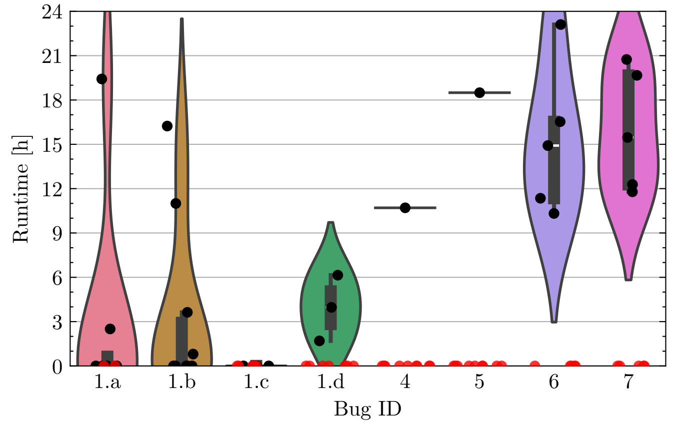

Fig. 8. Time-to-bug information. Individual points represent a single trial's time-to-bug for the bug category listed on the x-axis. The trials that did not discover a bug within the 24-hour time limit are indicated in red as a point at the bottom of the chart. The figure is truncated to 24 hours, as we cannot speak to the distribution of bugs *after* the end of the trial.

We ran the same experimental setup with H26Forge on the *linux-intel* platform, where it could have discovered bugs 1.a, 3, 4, and 5, but it did not discover any of these issues during a 24-hour period across ten trials. This is somewhat expected, as bugs 3-5 are not related to the processing of H.264 video files and bugs 1.a-d are likely a consequence of underspecified, invalid, or otherwise non-H.264 video format features. We were unable to run the H26Forge trials on the other platforms but expect similar results, as the requirements to trigger these bugs are the same.

F. Comparison with H26Forge Findings

In addition to the relative coverage comparison (see Section V-C), we investigated whether TwinFuzz can rediscover

the same (or a similar set of) bugs that H26Forge [72] found. To this end, we performed three small experiments similar to H26Forge, in which we replayed the corpora accumulated from previous experiments (≈150 000 inputs) and looked for known bug patterns.

We used VLC media player version 3.0.17 on Windows 10 (64-bit) [73], recording software bugs related to *libavcodec* in the same version tested by H26Forge. These inputs caused the VLC media player to crash on the NVIDIA machine setup (which we call *windows-nvidia* in Table II). In addition, we were able to find a new bug in version 3.0.21 of VLC media player (the latest release at the time of writing), where `RPC_X_NULL_REF_POINTER` errors are reported when VLC is interacting with *Microsoft Direct3D11*. We reported it to the developers but have not yet received a confirmation.

Next, we crafted a new way to store the inputs based on their entry signature. We inspected mismatching output frames to check for known hardware-related repetitive patterns, such as `x80` or `x00`, introduced in the hardware decoding process. This provides possible fingerprinting based on the specific input and the actual hardware decoder used, as proposed in the H26Forge paper. We identified cases in our *observable differences* cluster where the hardware decoder showed the mentioned patterns when the hardware could not successfully compute the decoding process. At the same time, the software correctly decoded the frame. Hence, we were able to reproduce the proposed hardware fingerprinting experiment successfully, and we consider this as a hardware layer fault in Table II.

Lastly, we converted our corpus entries into MP4 files with MP4Box and Minimp4. We play these in Firefox version 100 [49], as used in H26Forge, and the more recent release, 131.0.3, on a Windows 10 (64-bit) machine with an NVIDIA GPU (*windows-nvidia*). In this experiment, we found an information leak using our MP4-converted corpora and discovered a "reader" file that can read and leak part of the content from another MP4-converted corpus file played in a loop in another tab. We reported this information leak to Mozilla, which has confirmed the finding and is in contact with the developer and security team. This information leak specifically occurs on the *windows-nvidia* platform and is only present when hardware acceleration is enabled in the browser.

Our approach is orthogonal to the generative one used in H26Forge, but the findings here indicate that techniques are complementary. We neither aim to find all possible bugs nor the same bugs as H26Forge with this work; rather, to be able to find the same bug categories *but different bugs* as discovered in H26Forge. All H26Forge-confirmed bugs would have been flagged by a differential oracle by detecting changes in computed output, eliminating the need for the original technique's manual inspection. Our strategy sidesteps the need for complex generation by use of coverage by proxy.

We attempted to experiment on the same device category used in H26Forge to replicate the *Luma Chroma Thief* attack. However, our approach proved ineffective for this category of embedded devices, such as Dragonboard 410C [1], due to the limited processing power of the devices and the large number of inputs to test. Additionally, successfully crafting this experiment would require enforcing known patterns on our inputs and creating custom, grammar-based inputs, as was done in H26Forge. Though compatibility with low-performance devices is a non-goal for TwinFuzz, we recognize the need to test a greater number of inputs as a limitation of its applicability on these devices.

VI. Discussion

TwinFuzz, as described in Section IV and evaluated in Section V, serves as an initial prototype that exemplifies the methodology proposed in this paper. We now discuss potential improvements, alternative applications, and future research directions.

A. *Integration of* TwinFuzz *Harness with Other Fuzzers*

The harness described in Section IV-B uses the libFuzzer fuzzing engine [40], though it is certainly not restricted to libFuzzer. One could just as easily apply AFL++ [17], LibAFL [18], honggfuzz [21], and others. In particular, we note that fuzzers augmented with mutational grammar fuzzing capabilities, such as Nautilus [5], could be highly effective if a grammar is available for the targeted application. Mutational fuzzers that use non-domain-specific mutations are inherently limited to the inputs that may be generated by such mutations and may not effectively explore code paths with complex constraints, as seen in Section V-C. We chose not to evaluate other advanced fuzzers for our project because they will not significantly affect our design and approach.

B. *Identification of Observable Differences*

As described in Section V-D, identifying the root cause of observable differences is extremely difficult in practice and an open problem, specifically in the hardware domain. Existing solutions that cluster crashes based on the location of the crash [59] or on the coverage points incurred during execution [33] struggle to identify root causes of observable differences. While our strategy can identify when a memory safety-affecting bug occurred by means of an observable difference in output, we struggle to identify the root cause of these observable differences, hindering our ability to identify otherwise unobservable memory safety issues. Difficulties associated with fault localization in the presence of differential testing is a standing unsolved issue in differential testing [56], especially in the hardware domain. While we make a first step in this direction in Section V-D1 by clustering the differences by their characteristics, more work is needed to identify the root cause of these issues.

1) Existing fault localization utilities: We attempted to use several existing strategies to identify the root cause of observable differences and encountered the following problems:

a) Dependence on crash location: We were able to utilize the `casr-libfuzzer` program offered in CASR [59] to cluster crashes discovered during fuzz campaigns, though it was only able to cluster based on the AddressSanitizer [62] output. Consequently, it could only cluster crashes caused by

observable memory safety violations but not those triggering observable differences. Moreover, CASR does not indicate root cause locations. For that reason, we were unable to use it to investigate observable differences further.

b) Assumptions regarding root cause: We tried to use IGOR [33] to perform root cause clustering on the observable differences but were unable to configure the tool to do so. We note, however, that it is unlikely that IGOR would successfully cluster observable differences due to a unique feature of observable differences: since a differential oracle is only queried *after* execution is complete, there may be multiple bugs triggered in a single testcase, or the observable execution trace might not include the relevant behavior (e.g., because bugs are triggered in code parts that cannot be instrumented). In this sense, root-cause clustering would need to be expressed as a hierarchy, whereas IGOR assumes that all testcases exhibiting the same bug necessarily reduce to the same minimal path. This has the implicit effect of removing paths that exercise one bug but retain the other and may prevent the discovery of some bugs.

2) Collaboration with developers: We reached out to the FFmpeg developers to determine the root causes of inconsistencies in frame data between various applications. Although the FFmpeg team has responded, they have not yet attempted to analyze or remediate the issues associated with the testcases submitted. In their initial response, they indicated that they believed that the vast majority of consistency issues would be induced by the components of the hardware acceleration stacks provided by third parties – consistent with our analysis in Section V-D – and that, in previous attempts to implement hardware compatibility, they have found many issues with both device firmware and the hardware itself. Moreover, they note that these components effectively undergo no automated testing in the context of FFmpeg and are *known to contain bugs* but with limited effort to resolve them, further motivating our work. However, since we are unable to automatically identify the components responsible for these issues, it is difficult for the FFmpeg developers to investigate further, as they are unable to inspect these third-party components. Future work on detecting root causes for non-implied oracles would simplify the remediation of such issues, make them more approachable for developers, and allow researchers to identify the responsible software component. We will continue to work with the FFmpeg team to resolve these remaining identified testcases.

C. Limitations of Coverage Evaluation

In Section V-C, we identified the code regions discoverable by both H26Forge and TWINFUZZ. We compared them to gain a better insight into their relative ability to explore the software-only implementation since collecting the hardware coverage is infeasible for a video decoder. The results are mixed; H26Forge does not cover a majority of the regions that TWINFUZZ does during a 24-hour period, as it is restricted to the generation of H.264 video files. Conversely, TWINFUZZ is unable to reach a non-negligible subset of the code regions discovered by H26Forge, as mutation alone is insufficient to generate some higher-complexity H.264 inputs. This phenomenon is well-known [8]: if the system under test implements a superset of the grammar used by an input generator, then the input generator will only cover code regions associated with a subset of the grammar accepted by the system under test. Likewise, the constraints imposed by grammars may prevent a fuzzer utilizing a simple mutator from producing an input that successfully satisfies the constraints of certain grammar features, preventing the corresponding code regions from being reachable [19].

Beyond these results, we must also emphasize that a region covered by a fuzzer *does not necessarily imply* that this code region is bug-free. While code coverage is a prerequisite for bug discovery, recent work suggests that different fuzzers have different but consistent relationships with the discovery of coverage and the discovery of bugs [12]. As a result, we must emphasize that the coverage experiments here merely provide insight into each fuzzer's ability to generate inputs that exercise code regions. These fuzzers are not necessarily capable of discovering all bugs in the associated code regions. Moreover, recent vulnerabilities in image codecs suggest that even well-fuzzed code regions may have latent vulnerabilities that cannot be found easily via fuzzing [26]. For this reason, discovering latent bugs in some video codecs may require even more specialized testing than mutational or grammar-based fuzzing alone.

D. Discussion on False Positive Visual Differences

In the context of our work, we define observable differences as differences in the number of frames, a frame's dimensions, or its content. While the first two are certainly incorrect behavior, it could be argued that a very small difference in the frame's pixel content would not be visible to humans and could be the result of a conscious trade-off made by the hardware designers. Without deep knowledge of the respective designs, this is impossible to tell. However, the vast majority of testcases actually produce exactly matching frames, which indicates that the effects we see are more probably corner cases unnoticed by manufacturers' testing. Moreover, our *indirect proxy* fuzzing approach uncovers memory security vulnerabilities that could not be found with a "traditional" fuzzing setup targeting the hardware acceleration stack (cf. Section V-B). Consequently, the manufacturers have probably not tested their stacks as deeply as TWINFUZZ does. We interpret this as an additional hint that the bug reports we get from our oracle are indeed unexpected and not false positives.

E. Indirect Proxy between Software and Hardware Coverage

As mentioned in Section III, collecting hardware coverage, especially on proprietary hardware designs, is infeasible in practice. Our approach does not require it. Instead, we treat the system as a black box, making the presented technique more generalizable. Recall that the CFG edge coverage used in modern greybox fuzzers is already a heuristic used to approximate coverage of the problem's input space and, thus, a program's state space. Given a second program that fulfills the same task and takes the same input format, we can

reasonably expect that its state space is similarly structured to the first program's, even if there is no strict one-to-one mapping because of slight implementation differences. Using this knowledge, we can test the second program by *indirect proxy*: Inputs that generate meaningful coverage in the "white-box" program will most probably also generate coverage in the "black-box" program, exercising it without direct coverage. This approximation works well, as we show in Section V-B: a hardware-decoding harness with the same resources, but without indirect proxy guidance, finds *none* of the bugs that TwinFuzz found. As a result, we may infer that we do indeed discover greater coverage even in hardware, as bug discovery is a strong indicator of fuzzer coverage [12].

Bug localization in differential testing, i.e., telling where in the code the output difference originates, is an unsolved problem outside the scope of this work. Assuming such a localization oracle for the software side, the proxy coverage would likely provide meaningful insight into the relevant (black-box) acceleration procedure because of the similarly structured state space.

F. Impact of Differential Bugs in Real-World Scenarios

As introduced in Section V-F on fingerprinting, we extended the discussion to demonstrate the real-world impact of the differential bugs found with TwinFuzz by developing a web fingerprinting demo using open-source JavaScript libraries and browser APIs [48], [58]. This demo processes multiple H.26x files and decodes them using the browser-provided, hardware-accelerated HTML5 media APIs. We selected a subset of inputs that trigger observable differences (see Section V-D) discovered in the fuzzing campaign. By reproducing these in the web demo on our test machines, we could identify which clusters a machine belongs to, as identified in Figure 6. Using this building block, one could construct an automatic device fingerprinting system, and we are communicating with browser vendors on how to address this problem.

VII. Related Work

Closest to our work, Vasquez et al. presented H26Forge, a framework for analyzing, generating, and manipulating H.264 encoded videos, particularly those syntactically correct but semantically non-compliant with the specification [72]. To this end, the authors have manually implemented a large part of the H.264 video standard [32]. The tool successfully identified multiple memory corruption vulnerabilities in several video decoders. These included kernel-level vulnerabilities in iOS, memory corruption bugs in the Firefox and VLC media players on Windows, and kernel memory bugs affecting both the video accelerator and the application processor in a number of Android devices. Beyond H26Forge, discovering vulnerabilities in hardware acceleration stacks demands significant manual effort and a deep understanding of codec design and implementation. As it is the only existing academic work that is explicitly designed to test hardware-accelerated encoding and decoding that we know of, we use it as a comparative metric for evaluation in Section V-C.

Another related work is SiliFuzz [38], where the authors use a similar coverage-by-proxy technique to compare the behavior of a CPU emulator with a post-silicon CPU to discover inconsistencies. SiliFuzz offers a perfect example of *direct* coverage-by-proxy. Instead of providing direct proxy and domain-specific inputs, we build an abstraction over the entire hardware acceleration stack, sidestepping the complexities of platform-specific differences. Following this direction, we do not use a direct proxy technique to test a specific platform like SiliFuzz. Instead, we use an *indirect* proxy technique to test that an arbitrary platform performs some higher-level task implemented by the local software-only implementation. This approach allows us to inspect and find bugs across many implementations of a common task, regardless of their introspectability. Like SiliFuzz, when applying our strategy, it tends to be difficult to identify an exact root cause of testcases that cause different outputs to be emitted by the system under test, as explained in Section V. This issue could potentially affect our strategy even more, as the input domain over which we are testing is even further abstracted from the actual components under test.

Distantly related is "Fuzzing Hardware like Software" by Trippel et al. [70], who propose fuzzing hardware models by converting them to software models and measuring the corresponding software metrics. We considered evaluating the correlation between the coverage of the software-only implementation and the coverage of the hardware used by the acceleration stack. We discussed this problem with the Intel Video and Graphics teams. The suggestion was likened to measuring the coverage of an interpreter to measure the coverage of a script; since the hardware merely executes firmware that implements the video decoding algorithm, the hardware's coverage would likely be very distantly related to the actual test. Moreover, they indicated that pre-silicon simulation of such complex models tends to take several days and would simply not scale for the purposes of performing an evaluation. Since firmware coverage cannot be captured and hardware coverage is infeasible and impractical to capture, our strategy is the only realistic way currently available to test these stacks in their entirety.

VIII. Conclusion and Future Work

In this paper, we presented a technique that enables fuzzers to exercise and discover bugs in uninstrumentable hardware acceleration stacks. We implemented a prototype of this strategy to test a wide variety of video hardware acceleration implementations. Using FFmpeg as a common interface, our implementation allows us to provide complex but valid inputs to a hardware acceleration stack. Moreover, we propose and implement an analysis technique that can more precisely identify the root cause of unexpected behavior discovered by differential oracles in the presence of many implementations. With these combined, we were able to determine 15 clusters of testcases that represent one or more root causes of the measured differences in output and offer vendors more clarity in their investigation of these issues. With the same experimental setup,

we identified five memory safety bugs and responsibly disclosed them to vendors. Moreover, we apply fingerprinting techniques to web browsers and applications, which can identify the device playing the inputs from our clustered data or cause the application to crash. We compared our approach with state-of-the-art tools, comparing and exploring the capacity of our tool to application findings. During our comparison with H26Forge, we successfully replicated the same bug classes identified in VLC, specifically those related to hardware acceleration components on Windows and the libavcodec library. Additionally, we discovered an information leak in Firefox and successfully reproduced hardware-decoder fingerprinting using our setup. Moreover, we performed experiments that indicate the relative effectiveness of simple mutational and grammar-based fuzzing, highlighting the need for a diverse testing strategy for highly complex video decoding implementations.

In future work, we would like to see this strategy extended to new domains where the observability of the system under test is low. This extends far beyond hardware acceleration stacks alone and may be applicable to testing closed-source, obscured, or unobservable systems. Our analysis technique for clustering testcases identified by a differential oracle will similarly have applications beyond this strategy and may prove helpful in other testing pipelines where a root cause is not easily determined, but many comparable systems are under test. For video hardware acceleration specifically, we highlighted the need for more than just simple mutational and grammar-based fuzzing. We clarified that both (or potentially something more powerful) would be needed to test these stacks in depth. We hope this result inspires further research into testing this highly complex and widely depended-upon domain.

Acknowledgements

This work was supported by the European Research Council (ERC) under the consolidator grant RS^3 (101045669). In addition, the original concept of TwinFuzz resulted from discussions with members of Intel's hardware and video security research teams, who we would like to thank for their insight into the current state of testing of video hardware acceleration and feedback regarding our findings. We would like to thank the developers of libFuzzer [40] for the development of their fuzzer runtime upon which our work is developed. Several devices tested during the process of our evaluation were provided on loan by colleagues, for which we would especially like to thank Michael Schwarz and members of his research group. We would like to thank each of the security teams that responded to and remediated the bugs that we reported, including, but not limited to, Google's Chromium security team, Mozilla's Firefox security team, Intel PSIRT, NVIDIA PSIRT, VideoLAN, and most of all the FFmpeg team. Finally, we would like to thank the Saarbrücken Graduate School of Computer Science for their support.

References

[1] 96Boards, "DragonBoard 410c," https://www.96boards.org/product/dragonboard410c/, 2024, accessed: 2024-11-28.

[2] Alliance for Open Media, "AV1 Specification," https://aomedia.org/specifications/av1/, 2018, accessed: 2024-11-29.

[3] ALSA Project, "The advanced linux sound architecture (alsa) - library," https://github.com/alsa-project/alsa-lib, 2023, accessed on April 23, 2024.

[4] ARM Labs, "Armv8.5-a memory tagging extension," https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf, 2022.

[5] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi, and D. Teuchert, "Nautilus: Fishing for Deep Bugs with Grammars," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[6] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with Input-to-State Correspondence," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[7] B. Beizer, *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[8] B. Bendrissou, C. Cadar, and A. F. Donaldson, "Grammar mutation for testing input parsers," in *International Fuzzing Workshop (FUZZING)*, 2023.

[9] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "JIT-Picking: Differential Fuzzing of JavaScript Engines," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[10] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[11] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," *IEEE Transactions on Software Engineering*, 2017.

[12] M. Böhme, L. Szekeres, and J. Metzman, "On the Reliability of Coverage-Based Fuzzer Benchmarking," in *International Conference on Software Engineering (ICSE)*, 2022.

[13] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[14] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory *et al.*, "Sok: Enabling security analyses of embedded systems via rehosting," in *Proceedings of the 2021 ACM Asia conference on computer and communications security*, 2021.

[15] "FFmpeg — ffmpeg.org," https://ffmpeg.org/, FFmpeg Project, 2023.

[16] "libavcodec Documentation," https://www.ffmpeg.org/libavcodec.html, FFmpeg Project, 2024, accessed: 2024-11-28.

[17] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining Incremental Steps of Fuzzing Research," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[18] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[19] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[20] X. Gong and P. Pi, "Bypassing the maginot line: Remotely exploit the hardware decoder on smartphone," *Black Hat*, 2019.

[21] "Honggfuzz," Google, 2023, https://honggfuzz.dev/.

[22] "OSS-Fuzz - Continuous Fuzzing for Open Source Software," https://github.com/google/oss-fuzz, Google, 2024.

[23] M. A. Gulzar, Y. Zhu, and X. Han, "Perception and practices of differential testing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019.

[24] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[25] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel *et al.*, "Toward the Analysis of Embedded Firmware through Automated Re-hosting," in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2019.

[26] B. Hawkes, "The WebP 0day," Isosceles Blog, 2023, https://blog.isosceles.com/the-webp-0day/.

[27] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A Ground-Truth Fuzzing Benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, 2020.

[28] Y. V. Hoskote, D. Moundanos, and J. A. Abraham, "Automatic extraction of the control flow machine and application to evaluating coverage of

verification vectors," in *Proceedings of ICCD'95 International Conference on Computer Design. VLSI in Computers and Processors*. IEEE, 1995.
[29] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.
[30] "Build and debug open source media stack," Intel Corporation, 2018, https://www.intel.com/content/www/us/en/developer/articles/technical/build-and-debug-open-source-media-stack.html.
[31] "PreSiFuzz," Intel Corporation, 2023, https://github.com/IntelLabs/PreSiFuzz.
[32] ITU, "H.264: Advanced video coding for generic audiovisual services," 2021, https://www.itu.int/rec/T-REC-H.264-202108-I/en.
[33] Z. Jiang, X. Jiang, A. Hazimeh, C. Tang, C. Zhang, and M. Payer, "Igor: Crash Deduplication Through Root-Cause Clustering," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.
[34] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno, "Privacy oracle: a system for finding application leaks with black box differential testing," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.
[35] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities," in *USENIX Security Symposium*, 2022.
[36] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.
[37] K. Koscher, T. Kohno, and D. Molnar, "Surrogates: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.
[38] D. Kwan, K. Shtoyk, K. Serebryany, M. L. Lifantsev, and P. Hochschild, "SiliFuzz: Fuzzing CPUs by proxy," Google, Tech. Rep., 2021.
[39] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot, and H. Pfister, "Upset: visualization of intersecting sets," *IEEE transactions on visualization and computer graphics*, 2014.
[40] LLVM Project, "LibFuzzer - A Library for Coverage-guided Fuzz Testing," https://llvm.org/docs/LibFuzzer.html, 2024.
[41] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, 2021.
[42] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, 1998, http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf.
[43] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "FuzzBench: an open fuzzer benchmarking platform and service," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Aug. 2021.
[44] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM (CACM)*, 1990.
[45] MITRE Corporation, "Common Weakness Enumeration (CWE)," https://cwe.mitre.org, 2024, accessed: April 24, 2024.
[46] M. Y. Mo, "Gaining kernel code execution on an MTE-enabled Pixel 8," https://github.blog/2024-03-18-gaining-kernel-code-execution-on-an-mte-enabled-pixel-8/, 2024.
[47] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys (CSUR)*, 2018.
[48] "MDN Web Docs: Manipulating video using canvas," https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Manipulating_video_using_canvas, Mozilla Corporation, 2024.
[49] "The Firefox Project," Mozilla Corporation, 2024, https://www.mozilla.org/it/firefox/.
[50] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.
[51] National Institute of Standards and Technology, "CVE-2023-41061," 2023, https://nvd.nist.gov/vuln/detail/CVE-2023-41061.
[52] ——, "CVE-2023-41064," 2023, https://nvd.nist.gov/vuln/detail/CVE-2023-41064.
[53] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, 2012.
[54] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing Black-Box CPUs against Speculation Contracts," in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.
[55] L. E. Olson, S. Sethumadhavan, and M. D. Hill, "Security implications of third-party accelerators," *IEEE Computer Architecture Letters*, 2016.
[56] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 615–632.
[57] P. Ranganathan, D. Stodolsky, J. Calow, J. Dorfman, M. Guevara, C. W. Smullen IV, A. Kuusela, R. Balasubramanian, S. Bhatia, P. Chauhan, A. Cheung, I. S. Chong, N. Dasharathi, J. Feng, B. Fosco, S. Foss, B. Gelb, S. J. Gwin, Y. Hase, D.-k. He, C. R. Ho, R. W. Huffman Jr., E. Indupalli, I. Jayaram, P. Kongetira, C. M. Kyaw, A. Laursen, Y. Li, F. Lou, K. A. Lucke, J. Maaninen, R. Macias, M. Mahony, D. A. Munday, S. Muroor, N. Penukonda, E. Perkins-Argueta, D. Persaud, A. Ramirez, V.-M. Rautio, Y. Ripley, A. Salek, S. Sekar, S. N. Sokolov, R. Springer, D. Stark, M. Tan, M. S. Wachsler, A. C. Walton, D. A. Wickeraad, A. Wijaya, and H. K. Wu, "Warehouse-scale video acceleration: Co-design and deployment in the wild," in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.
[58] D. Samir, "jmuxer: A simple javascript mp4 muxer that works in both browser and node environment," https://github.com/samirkumardas/jmuxer, 2022.
[59] G. Savidov and A. Fedotov, "Casr-Cluster: Crash Clustering for Linux Applications," in *Ivannikov ISPRAS Open Conference (ISPRAS)*, 2021.
[60] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing," in *USENIX Security Symposium*, 2022.
[61] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "SoK: Prudent Evaluation Practices for Fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2024.
[62] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A Fast Address Sanity Checker," in *USENIX Annual Technical Conference (ATC)*, 2019.
[63] S. R. Shahamiri, W. M. N. W. Kadir, and S. Z. Mohd-Hashim, "A comparative study on automated software test oracle methods," in *2009 fourth international conference on software engineering advances*. IEEE, 2009.
[64] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for Security," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.
[65] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design & Test of Computers*, 2001.
[66] "Chromium," The Chromium Project, 2024, https://www.chromium.org/chromium-projects/.
[67] The Clang Team, "LLVM SanitizerCoverage," 2023, https://clang.llvm.org/docs/SanitizerCoverage.html.
[68] The Linux Kernel, "Kernel Sanitizer," 2017, https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html.
[69] The OpenSSL Project, "OpenSSL: The open source toolkit for SSL/TLS," April 2003, www.openssl.org.
[70] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing Hardware Like Software," in *USENIX Security Symposium*, 2022.
[71] A. Tversky, "Features of similarity," *Psychological Review*, vol. 84, no. 4, p. 327–352, 1977.
[72] W. R. Vasquez, S. Checkoway, and H. Shacham, "The Most Dangerous Codec in the World: Finding and Exploiting Vulnerabilities in H.264 Decoders," in *USENIX Security Symposium*, 2023.
[73] VideoLAN, "VLC Media Player," https://www.videolan.org/vlc/, 2024, accessed: 2024-11-28.
[74] G. Vranken, "Differential fuzzing of cryptographic libraries," May 2019, https://guidovranken.com/2019/05/14/differential-fuzzing-of-cryptographic-libraries/.
[75] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization." in *NDSS*, 2020.
[76] "VP9 Codec," https://www.webmproject.org/vp9/, The WebM Project, 2024, accessed: 2024-11-28.

[77] "x265 Documentation," https://x265.readthedocs.io/en/master/, x265 Project, 2024, accessed: 2024-11-28.
[78] Y. Yang, T. Kim, and B.-G. Chun, "Finding consensus bugs in ethereum via multi-transaction differential fuzzing," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.
[79] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *Symposium on Network and Distributed System Security (NDSS)*, 2014.
[80] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/, 2024.

Appendix A
Extended security bug descriptions

A. *Bug 3: Global buffer overflow in* linux-intel

This bug was originally reported to FFmpeg. It is located in `avcodec/av1dec` and was caused by an integer underflow that led to an *out-of-bounds read*. As remediation, the developer simply adjusted the bounds check on the parameter that could lead to an integer underflow. Patched here: https://patchwork.ffmpeg.org/project/ffmpeg/patch/20231130122853.26758-1-michael@niedermayer.cc/

B. *Bug 4: Heap buffer overflow in* linux-intel

This bug was originally reported to Intel PSIRT via the Integriti Bug Bounty platform. The bug is a classical heap-overflow vulnerability in the Intel media-driver, which may lead to code execution. The bug was assigned to CVE-2024-23919. In the context of a browser, for example, this may lead to remote code execution via a video loaded on a webpage.

C. *Bug 5: Wild pointer dereference in* linux-intel

This bug was originally reported to Intel PSIRT via the Integriti Bug Bounty platform. We found this bug in our fuzzing environment, but Intel PSIRT indicated that they were unable to reproduce the behavior.

This bug is a wild pointer dereference, generally caused by an overflow inside one buffer into a region containing a buffer or by use-after-free, which leads to undefined behavior and data corruption. We have not investigated the impact further, but we suspect this may be used to perform arbitrary out-of-bounds reading and writing in certain scenarios. At this time, we are at least aware of an impact on availability (i.e., it causes a crash).

D. *Bug 6: Invalid pointer free in* linux-nvidia

We discovered bug 6 while fuzzing, but it was fixed by FFmpeg upstream before we were able to submit a report. This bug is located in `avcodec/nvdec`. In particular, it is an attempt to free NVDECContext→bitstream improperly.

As we did not publicly participate in the remediation of this bug, we provide the full analysis from the developers below:

> Using a field for both an ownership pointer and as a pure data pointer is bad and confusing. IMO there should be different pointers for this; or rather: bitstream is always set and used as data pointer and the another pointer for ownership which may alias bitstream. In fact, it looks like this is exactly what bitstream_internal is, it is just not used by H.264 and HEVC in this manner (they use bitstream for allocated stuff). With the approach outlined above, ff_nvdec_simple_end_frame() might become obsolete.

Originally patched here: https://patchwork.ffmpeg.org/project/ffmpeg/patch/20240206212640.9193-1-jamrial@gmail.com

Appendix B
Artifact Appendix

This appendix provides an overview of the artifact presented in our paper, along with instructions for running the artifact locally. Reproducing all of our results in full is a challenge, as access to all hardware platforms shown in Table I would be required.

The focus of the artifact is on the crash experiment described in Section V-A with a specific hardware/software configuration. If access to other platforms listed in Table I is available, the artifact can also be used on these platforms. To facilitate artifact evaluation, we have included a Dockerfile containing all necessary files to build our prototype, as described in Section III, for a *linux-intel* machine.

A. *Description & Requirements*

This section provides all the details required to set up the experimental environment needed to run our artifact. To run the Dockerfile locally, your machine must support video hardware acceleration, have Intel VAAPI drivers, and an Intel x86-64 CPU.

1) *Access Instructions:* The artifact is publicly available via a Zenodo link[4]. The repository includes a Dockerfile for an *linux-intel* setup.
2) *Hardware Requirements:* To reproduce our experiments, your machine must have video hardware acceleration, Intel VAAPI drivers, and an Intel x86-64 CPU.
3) *Software Requirements:* The software required is the same as that needed to build and run FFmpeg with hardware acceleration enabled.
4) *Benchmarks:* No specific benchmarks are needed.

B. *Major Claim*

- (C1): We propose a technique for *indirectly* guiding an unmodified fuzzer to abstract over a hardware acceleration stack that is otherwise difficult to introspect.
- (C2): We propose a technique for *indirectly* guiding an unmodified fuzzer that abstracts over any acceleration stack that is otherwise difficult to introspect.
- (C3): We implement a prototype of our approach in a tool called TwinFuzz capable of fuzzing a specific hardware acceleration stack, providing an observable difference and potentially triggering memory safety bugs using memory sanitizers.

[4] https://doi.org/10.5281/zenodo.14261195

C. Artifact Local Installation

This subsection refers to the folder `local_setup` in the repository. To build and install our prototype locally using the Dockerfile, we have created a script to simplify the process. Run the following command:

```
./run_docker.sh
```

This script builds and runs the Docker image as follows.

To build the docker container, use the following command:

```
docker build -t twinframe-linux-intel .
```

To run the Docker container, use the following command:

```
docker run --rm --device=/dev/dri -e
    LIBVA_DRIVER_NAME="iHD" -it twinframe-
    linux-intel /bin/bash
```

D. Experiment Workflow

We provide a script to simplify the evaluation and to demonstrate the *functionality* of our prototype. This script performs differential fuzzing guided by the software implementation mentioned in C1, generating both software and hardware frames. It compares their outputs to detect observable differences as mentioned in C2 (e.g., they differ in content or size, or a crash occurs). These two contributions led to this prototype, which is our C3 contribution.

To run the evaluation with the corpus used in our experimental prototype (comprising 10 trials of 24 hours each), navigate to the `/out` directory within the container and run:

```
root:/out# ./evaluation_script.sh
```

You can customize the script by modifying parameters directly in the bash script, such as the number of trials and the duration of a single fuzzing campaign.

E. Results Reproduction

After running the default evaluation, you will find ten folders (each corresponding to a trial) named `ffmpeg_AV_CODEC_H264_fuzzer_trial_X` (Where `X` represents an integer from 1 to 10). Each folder contains a corpus folder and a potential list of discovered crashes. Additionally, you may find `hw_frame.pgm` and `sw_frame.pgm` files if TwinFuzz found observable differences.

To reproduce a crash, run the `ffmpeg_AV_CODEC_H264_fuzzer` binary from the `/out` directory and provide the input file corresponding to the crash:

```
root:/out# ./ffmpeg_AV_CODEC_H264_fuzzer
    trial_2/crash-artifact-2-crash-
    c21a135d3b3e925269f52437ff4c38503017a40e
```

After executing the crash input, the frame decoding process will continue until a frame mismatch is detected, after which AddressSanitizer will raise an abort signal. You should see an output similar to the following:

```
root:/out/
    ffmpeg_AV_CODEC_ID_H264_fuzzer_trial_2#
     ../ffmpeg_AV_CODEC_ID_H264_fuzzer
    crash-artifact-2-crash-
    c21a135d3b3e925269f52437ff4c38503017a40e

../ffmpeg_AV_CODEC_ID_H264_fuzzer: Running
     1 inputs 1 time(s) each.
Running: crash-artifact-2-crash-
    c21a135d3b3e925269f52437ff4c38503017a40e

--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
--> Decoding completed correctly
differed on data line 0 (640, 640)
HW/SW buffer contents differ; see {hw,sw}
    _frame.pgm
==2776== ERROR: libFuzzer: deadly signal
SUMMARY: libFuzzer: deadly signal
```

These steps offer a straightforward way to evaluate the availability and functionality of the code base from our project described in the paper.