Exploring Depths of WebAudio: Advancing Greybox Fuzzing for Vulnerability Detection in Safari

Jiashui Wang[†‡*], Jiahui Wang[†*], Jundong Xie[‡], Zhenyuan Li[†✉], Yan Chen[§], Peng Qian[†]
[†]*Zhejiang University*
[‡]*Ant Group*
[§]*Northwestern University*
[†]{12221251, wjh_13, lizhenyuan, pqian}@zju.edu.cn,
[‡]jundong.xjd@antfin.com,
[§]ychen@northwestern.edu

***Abstract*—WebAudio is a widely used audio processing API in popular browsers, which provides rich audio support for the exclusive browser Safari on macOS. Given its widespread use, it is critical to thoroughly test WebAudio to ensure its reliability. Traditional fuzzing techniques typically lack awareness of the input structure and fail to accommodate the unique characteristics of audio file formats, and cannot generate effective fuzzing input, thus falling short of effectively detecting vulnerabilities within WebAudio.**

In this work, we introduce PROTEUS, an advanced greybox fuzzer designed to achieve structure awareness through the use of input templates. Moreover, PROTEUS is equipped with high-level mutation operators, diverging from traditional bit-level manipulations, and incorporates a post-processing stage that repairs format constraints disrupted during mutation. These enhancements enable PROTEUS to explore new input domains effectively while maintaining file validity, significantly improving the depth and efficiency of the fuzzing process.

Our evaluation confirms the effectiveness of PROTEUS. In the experiment of fuzzing WebAudio using CAF files, our tool exposed significantly more vulnerabilities than the baseline Honggfuzz without compromising efficiency. Excitingly, we have identified a vulnerability that can be exploited to gain control of the browser. Generally, PROTEUS has discovered 36 zero-day vulnerabilities in WebAudio on macOS 10.15.3, with 11 of these assigned CVEs.

I. INTRODUCTION

WebAudio [1] is a web audio processing API that supports a wide array of file formats like CAF. It is extensively utilized in popular browsers such as Safari and Chrome. Existing web security research focuses on the DOM (Document Object Model) and JavaScript engines, with less attention given to modules like WebAudio. Recent browser updates [2] have incorporated built-in heap isolation mechanisms to mitigate out-of-bounds writing vulnerabilities, thereby complicating the execution of cyber attacks via individual modules. Despite these protections, we have discovered many vulnerabilities in Safari's audio decoding process. These vulnerabilities allow for arbitrary address writing that alters the length field of a JavaScript object, effectively circumventing heap isolation. Exploiting this vulnerability will provide attackers with potential access to Safari. Therefore, it is necessary to conduct a systematic security analysis of WebAudio.

Fuzzing has emerged as a leading technique for detecting vulnerabilities since 1901 [3]. Among various fuzzing methodologies, Coverage-Oriented Greybox Fuzzing (CGF) is particularly notable for its efficacy [4]. Unlike blackbox methods, which provide limited program insights, and whitebox methods, which are constrained by high program analysis and constraint resolution costs, CGF utilizes lightweight code instrumentation to navigate the program space through strategic mutations, effectively uncovering software vulnerabilities.

Applying CGF to WebAudio, however, presents significant challenges. First, CGF lacks structure awareness, primarily executing low-level mutation operators such as flipping, deleting, or adding random bits. These changes are insufficient to modify the structure significantly, which is crucial for thoroughly exploring valid program inputs. Second, the intricate nature of multimedia files, which include both a container format to encapsulate data and a codec format for compressing and decoding, complicates fuzzing efforts. Typically, CGF's elementary mutations lead to invalid formats that are dismissed by the program's parser before they can even reach the codec processing stage, thus missing potential vulnerability discovery areas. We call the goal of exposing more vulnerabilities in subsequent processing logic as **Fuzz Deeper**.

Two techniques are introduced into fuzzing to tackle these challenges, namely, dynamic taint analysis [5] and dictionary-based methods [6]. Despite these advancements, neither of them fully addresses the core issue: penetrating more deeply into the codec processing stage of the program. This area remains challenging for fuzzers, requiring more refined approaches to uncover vulnerabilities effectively. Meanwhile, blackbox fuzzers like PEACH [7] achieve structure awareness by employing an input model to parse seeds and manipulate

[*]Two authors contribute equally to this work.
[✉]Corresponding author: Zhenyuan Li (lizhenyuan@zju.edu.cn)

them into new files, which includes adding and removing blocks. Despite the structure awareness of blackbox fuzzers, the coverage feedback provided by greybox fuzzers significantly enhances the effectiveness of vulnerability discovery.

In this article, we proposed a new CGF called PROTEUS to address the aforementioned limitations by extending our well-designed structure awareness to CGF and applying high-level mutation operations. Using manually defined structure templates, our method converts seed corpus into intermediate representations and then performs a set of high-level mutation operators added by us on them. Then, by using the serialization method designed for special file formats, a large number of fuzzing inputs are generated by PROTEUS. Finally, a post-processing step is employed to preserve the container format's integrity of the generated inputs. Combined with the coverage-oriented feedback, PROTEUS facilitates a comprehensive exploration of the target program.

We implement PROTEUS based on the popular fuzzing tool - HONGGFUZZ [8] by integrating the structure awareness defined by an open-source serialization format - Protobuf [9] like PEACH-Pit. Hence, in our evaluation, we compare against both as baseline techniques. We crawled approximately 200,000 CAF files from public websites as the seed corpus for our experiments. Taking into account the differences in template granularity, we compared PROTEUS with HONGGFUZZ and Peach, respectively. Each group of experiments conducted 1,000,000 fuzz tests on WebAudio. The evaluation results indicate that, compared to HONGGFUZZ, PROTEUS can double the number of zero-day bugs found. PROTEUS discovered 36 zero-day bugs (with 11 CVEs assigned), while the baseline technique, HONGGFUZZ, only detected 19. Additionally, in comparison with Peach, PROTEUS found almost twice as many bugs. The time spent writing file templates is outweighed by the significant improvements in behavioral coverage and the number of bugs exposed in our experiments. It took us three working days to complete the definition of the CAF file templates. Once developed, file format specifications can be reused across different programs as well as for various versions of the same program.

In summary, this paper makes the following contributions:

- We designed a new CGF called PROTEUS that achieves structure awareness and generates inputs using high-level mutation operators. By leveraging the extended post-processing feature to fix the container format of inputs, PROTEUS can completely fuzz deeper into WebAudio.
- We evaluated audio files on WebAudio. The results show that our method found approximately twice the baseline zero-day bugs while maintaining coverage and efficiency similar to the baseline fuzzer-HONGGFUZZ. Overall, we discovered 36 zero-day bugs while 11 CVEs have been assigned.

II. BACKGROUND AND RELATED WORK

A. Services Framework of WebAudio

A typical process for playing a CAF (Core Audio Format) file by WebAudio is illustrated in Fig. 2. Specifically, steps 2, 3, and 5 involve analyzing segments of the memory buffer, making these stages particularly suitable for fuzzing.

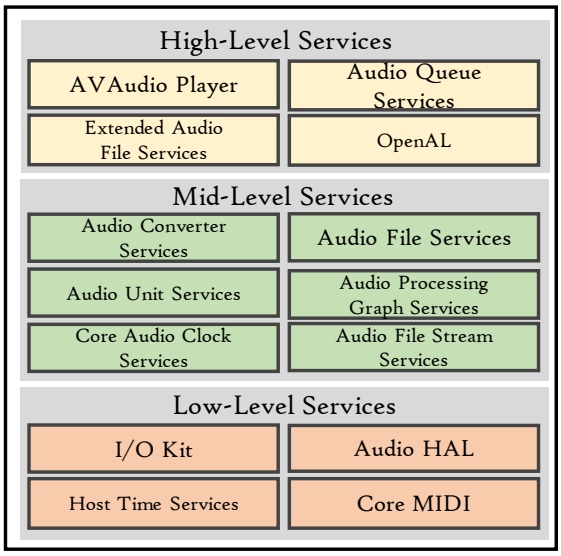

Fig. 1: Audio Services Frameworks.

On macOS, WebAudio is implemented through a range of service levels, each supported by Apple's extensive API infrastructure, as outlined in Apple's documentation [1]. The process of fuzzing WebAudio effectively examines these varied services. Choosing the right frameworks to target is critical for effective vulnerability testing, as each service level interacts differently with audio data. This differentiation is clearly depicted in Fig. 1 [10]. Our approach excludes low-level services linked predominantly to hardware interaction, focusing instead on mid to high-level services with greater potential for uncovering significant security flaws. Among the frameworks we focus on are Audio File Services, which deal with the extraction and handling of fundamental audio metrics such as sampling and bit rates, as well as the separation of audio frames; Audio File Stream Services, tailored towards the demands of streaming audio content; Audio Converter Services, which facilitate the transformation of audio files across different formats, thus testing the robustness of encoding and decoding processes; Extended Audio File Services that combine features of both audio file handling and conversion to provide a thorough testing scenario; and AVAudioPlayer, an advanced framework designed for complete audio playback management.

By targeting these specific aspects of WebAudio, we can more effectively construct a fuzzing harness that tests its resilience against potential security vulnerabilities. In our examination of WebAudio's services for fuzzing, we considered the complexity and interface requirements of each. The use of AVAudioPlayer, although advanced, necessitates a graphical interface which complicates the fuzzing process. Consequently, for greater efficiency in step 2 of our testing, we selected Audio File Stream Services. This choice was driven by their capability to directly read files from memory,

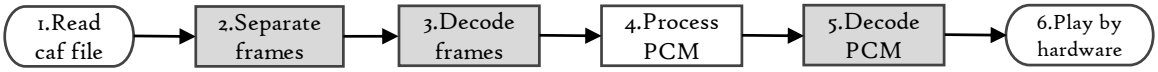

Fig. 2: Classic Playback Process for CAF Files.

streamlining the fuzzing operations significantly. For step 3, Extended Audio File Services were chosen due to their ease of use and the comprehensive functionalities in handling and converting audio files, further enhancing our fuzzing approach.

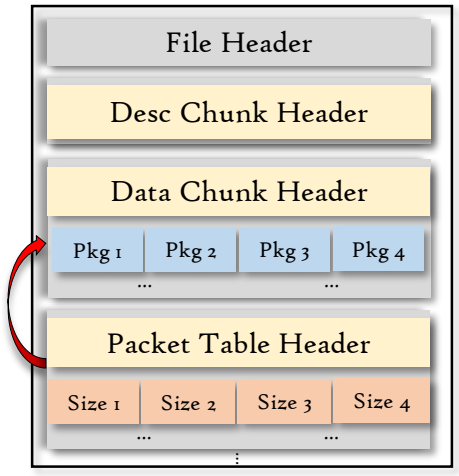

Fig. 3: Simplified CAF Format.

B. Encapsulation Format

Data in file systems is stored as binary sequences, which programs interpret by parsing the file to extract relevant information. Multimedia files have two main components: the Container Format and the Codec Format. The container format organizes the audio data, determines file extensions like MP3 and CAF, and structures the data into manageable units. It includes metadata and the actual audio content. The codec format handles the compression and encoding of raw data, affecting audio quality and file size by reducing data redundancy. Together, these formats enable efficient multimedia file management, storage, and playback.

For instance, a simplified depiction of the CAF container (*.CAF) format is shown in Fig. 3. A CAF file is structured into chunks, beginning with a File-Header followed by various chunk types, each initiated by a Chunk-Header that specifies its size. The Desc-Chunk stores the file's metadata, the Data-Chunk contains all the audio packets, and the Packet-Table-Chunk logs each packet's size within the Data-Chunk. When parsing a CAF file, players first access the Packet-Table-Chunk to determine packet sizes before retrieving the corresponding packets from the Data-Chunk. While some chunks are optional, others are mandatory. Each chunk starts with a size field, indicating its overall size, and there are dependencies among chunks. For example, the size of each packet in the Data-Chunk is dictated by a specific field in the Packet-Table-Chunk.

C. Related Works

1) Blackbox fuzzing: Blackbox fuzzing utilizes file format specifications to generate program inputs, extensively using input grammars for test input generation as evidenced by tools like PEACH Fuzzer [7], Spike [11], Domato [12], and LangFuzzy [13]. Specifically, LangFuzzy is designed to detect bugs in JavaScript engines by mutating seed inputs and replacing code snippets based on analyzed sample inputs [14]. Our work on PROTEUS can be seen as integrating the structure awareness capability into coverage-based greybox fuzzing.

2) Whitebox fuzzing: Whitebox fuzzing leverages the internal structures of programs to optimize test exploration, employing symbolic execution engines such as KLEE [15] and S^2E [16]. It incorporates techniques like grammar-based obfuscation [17] to produce context-independent, grammatically valid files and model-based approaches [18] to impose semantic constraints beyond the limits of context-free syntax. However, whitebox fuzzing demands extensive symbolic execution and complex constraint-solving, distinguishing it from our methodology.

3) Coverage-based greybox fuzzing (CGF): CGF, exemplified by the AFL [19] fuzzer and its variants [20], [21], offers a balance between blackbox and whitebox techniques. CGF employs minimal instrumentation to direct fuzzing towards unexplored code areas without the significant overhead associated with whitebox methods [22]. AFLSmart [4], which integrates structure awareness using the PEACH Pit template [7] atop AFL, shares similarities with our approach but struggles with the encapsulation format of multimedia files and support for sourceless libraries, as detailed in §II-B.

4) Boosted greybox fuzzing: Boosted greybox fuzzing, such as AFLFAST [20], leverages Markov chain modeling to target regions that remain uncovered by standard AFL. This approach enables the faster discovery of known vulnerabilities and the identification of novel bugs. AFLGO [23] performs reachability analysis to prioritize seeds estimated to be closer to a given location or target, enhancing the likelihood of reaching previously inaccessible code paths. Angora [21], an extension of AFL, improves coverage by employing gradient descent-based search to solve path conditions without resorting to symbolic execution. SlowFuzz [24], on the other hand, prioritizes inputs with higher resource usage counts for mutation, aiming to uncover vulnerabilities susceptible to complexity attacks. These advancements enhance the effectiveness of greybox fuzzing along various dimensions, apart from input structure awareness, and are generally complementary to our approach.

5) Restricted mutations: Restricted mutations are also employed by specific optimization methods in other works. VUzzer [5] utilizes data flow and control flow analysis of the test subject to detect the locations and types of input data that

should be mutated or kept constant. Steelix [25] focuses on developing customized mutation operations for magic bytes, such as the special words RIFF, fmt, or data found in a WAVE file. SymFuzz [26] learns the dependency relationships among bits in the seed input through symbolic execution to calculate an optimal mutation ratio given the program under test and the seed input. The mutation ratio refers to the number of seed bits that are flipped during mutation-based fuzzing. These works encompass specific optimizations for restricting mutations. However, they do not incorporate awareness of input formats to generate valid inputs, which is achieved through our high-level mutation operators.

III. Approach

To tackle the limitations of traditional CGFs lacking structure awareness of input files and inability to fuzz deeper, we introduce PROTEUS, a new CGF distinguished by its sophisticated structure awareness, advanced mutation operations, and format fixing feature. We will first outline the overall design and workflow of PROTEUS, followed by a detailed explanation of its key components.

A. Overview

Fig. 4 provides an overview of PROTEUS. The workflow begins with manually defined structure templates, which use formal language to describe the target file's format. These templates allow the transformation of the crawled seed corpus into intermediate representations. Instead of employing traditional bit-level mutation operators, we apply high-level mutation operators, significantly increasing the diversity of the generated files.

Besides, before submission to PROTEUS, a post-processing step assesses and rectifies any constraints in the container format. Fuzzing feedback, captured through instrumentation, directs PROTEUS to focus on inputs that enhance coverage if a generated test case leads to a crash, PROTEUS records and reports the issue upon fuzzing completion. Moreover, we leverage a mutation tool known as libProtobuf-mutator [27], which utilizes Google's Protocol Buffers (Protobuf) [9] to define structure templates. Additionally, we enhance the capabilities of PROTEUS by integrating and extending modules from the existing greybox fuzzer, HONGGFUZZ [8], known for its robust support for open-source programs.

B. Defining Input Templates

Protobuf is a tool provided by Google that depicts various structured data. The file used to define the Protobuf format is called a proto file (*.proto). Proto files typically contain several `message` types, where each `message` is a data structure. In the `message` data structure, fields serve as the fundamental building blocks, with each field encompassing three key elements: the field type (specifying the data type), the field name, the field rules, and the field label (a distinctive, positive integer for identification purposes). Since our work pays more attention to CAF files to fuzz WebAudio, we manually defined the complete CAF proto file according to the specifications. Part of the proto file we defined is shown in Fig. 5. As mentioned before, summarily, a CAF file is composed of a file header and several chunks, hence we define the top-level `message` as `CAFFile`, which consists of `FileHeader` and `Chunks`. Protobuf uses a syntax similar to C arrays to define repeated chunks.

The message `Chunk` is defined using the one-of syntax, enabling it to contain any field within it. We defined three types of `Chunk`, namely `AudioDataChunk`, `AudioDescriptionChunk`, and `PacketTableChunk`. As previously mentioned, `AudioDataChunk` primarily serves to house the metadata of the CAF file, whereas `AudioDescriptionChunk` is responsible for storing all data packets. Additionally, `PacketTableChunk` keeps track of the size of each data packet. Once the overarching structure has been firmly established, we proceed to meticulously define the components of the `FileHeader` and each `Chunk`. The `FileHeader` comprises three distinct fields, namely, `file_type`, `file_version`, and `file_flags`. Moving forward, the `AudioDescriptionChunk` is segmented into a header and a body. The header contains a four-byte chunk type for identification and an eight-byte chunk size to indicate its extent. The body, on the other hand, encompasses an eight-byte double-type sample rate, alongside various other fields, to capture the audio's characteristics. Lastly, `AudioDataChunk` and `PacketTableChunk` adhere to similar definitions, ensuring a consistent and cohesive data representation.

C. Applying Mutation

The primary objective of the current stage is to craft input cases specifically tailored for fuzzing. Given the uniqueness of the mutation process, including the need to avoid mutating certain fields or specify the length of string-type variables, the straightforward input templates devised in the previous stage III-B fall short of fulfilling these specialized requirements. To overcome this limitation, we leverage the extensible feature-custom options offered by Protobuf. After obtaining the enhanced input template, we implement a deserialization code that transforms the collected seed corpus into intermediate representations. These intermediate representations then undergo our well-designed high-level mutation operators, purpose-built for the mutation process. After successfully executing the aforementioned operations, the intermediate representations are indicated as being forwarded to the subsequent stages. The following discussion delves into the design principles underlying these modules.

1) Custom Options.: Protobuf offers a feature called custom options that enable extensive personalization and extensions, fulfilling our previously mentioned aspirations. Drawing inspiration from Google's Project Zero fuzzing project [28], we have crafted a suite of options that align with the definition specifications of Protobuf extensions. These options can be broadly categorized into two groups: those that impact the message level and those that influence the field level, based on their respective application scopes. Table I presents part of the

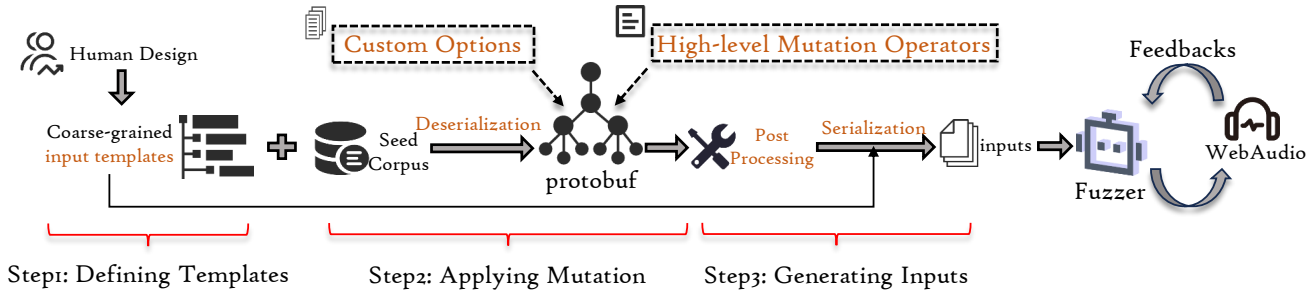

Fig. 4: Architecture of PROTEUS.

```
syntax = "proto2";
message CAFFile {
  required FileHeader file_header = 1;
  repeated Chunk chunks = 2;
}
message Chunk {
  oneof the_chunk {
    AudioDataChunk audio_data_chunk = 1;
    AudioDescriptionChunk audio_desc_chunk = 2;
    PacketTableChunk packet_table_chunk = 3;
  }
}
message FileHeader {
  required uint32 mFileType = 34;
  required uint32 mFileVersion = 35 ;
  required uint32 mFileFlags = 36;
}
message AudioDescriptionChunk {
  required uint32 mChunkType = 37;
  required int64 mChunkSize = 38;
  message DataSection {
    required double mSampleRate = 39;
    required uint32 mFormatFlags = 41;
    ...
  }
  required DataSection data_section = 46;
}
```

Fig. 5: Part of the Proto File that Defines CAF Files.

options we defined. Notably, there is a single option designated for the message, named `msg_is_const`. When specified for a message, this option ensures that the entire message is omitted during mutation processes. On the other hand, the options affecting fields can be further subdivided into those about the mutation stage and those related to the serialization and deserialization stages. At the mutation stage, three options stand out. The `custom_type` option complements Protobuf's types by incorporating missing fundamental types such as uint8, uint16, int8, and int16. The `fixed_length` option, which can only be applied to fields of type bytes or string, ensures that the data remains of a consistent length throughout the mutation process. Lastly, the constant option serves to determine whether a field should undergo mutation. During the serialization and deserialization stages, we also introduced three options. The `variable_big` and `variable_small` options represent variable-length integers and stipulate that the corresponding fields must adhere to specific encoding rules during these processes. The `null_terminated` option, exclusive to string-type fields, indicates whether an additional $'\backslash 0'$ character should be considered during serialization and deserialization.

TABLE I: Part of the Custom Options defined for Protobuf(we abbreviate serialization and deserialization stages as s/d)

Option	Description
MSG_IS_CONST	Whether the data structure will mutate
CUSTOM_TYPE	supplement missing basic data types
FIXED_LENGTH	Whether the string is fixed in length
CONSTANT	Whether the field will mutate
VARIABLE_BIG	Big-endian variable-length integer
VARIABLE_SMALL	Small-endian variable-length integer
NULL_TERMINATED	Whether there is a $'\backslash 0'$ at the end

2) Deserialization.: Based on the past fuzzing experience, we recognize that leveraging seed files is crucial for enhancing the effectiveness of fuzzing techniques. Consequently, we developed a code module specifically designed to deserialize binary files into Protobuf intermediate representations. This deserialization process transforms the collected seed corpus into memory-based data structures, ready for utilization in subsequent stages. This approach enables us to possess a comprehensive set of intermediate representations from the outset, upon which subsequent high-level mutation operations can be efficiently performed. We will omit the intricacies of the code implementation in this discussion. Algorithm1 describes the sketchy program structure. In essence, we have harnessed the reflection functionality offered by Protobuf. When dealing with messages of fixed length, we dynamically acquire the attributes of the fields within the message, compute its size, extract the corresponding data from the input binary file, and convert it into a message object. On the other hand, for messages of variable length, we tailor our approach by manually writing code to read and parse the fields based on the specific context.

3) High-level Mutation Operators.: We selected the libProtobuf-mutator [27] as our mutation library due to its

TABLE II: Part of High-level Mutation Operators

Mutation operator	Description
change magic value	Replace one with special value
add/sub value	Add or subtract a value
mangle_MemMove	Replace one substring with another
mangle_Random	Replace substrings with random values
mangle_CloneByte	Replace one byte with another
mangle_ASCIIval	Use ASCII characters for mutation

superior support for Protobuf. This library offers basic mutation operators tailored for Protobuf, encompassing operations like random bit flipping. Nevertheless, it's worth noting that basic mutation operators alone are inadequate for generating intricate audio files that are highly structured. To address this, we enhanced our arsenal with a set of advanced mutation operators, several of which are outlined in Table II. Among the new mutation operators, we incorporated more intricate mutation mechanisms for basic data types. Specifically, we introduced two mutation operators tailored for numerical types like INT32, including substitution with preset "magic" values and incrementing or decrementing by a small integer. For fields of STRING or BYTES type, we defined over ten mutation operators, ranging from string extension and reduction, among others.

Algorithm 1: Deserialization Algorithm

Data: Protobuf template T, Binary file F.
Result: A Protobuf object O.

```
INIT(O);
for m in T.messages do
    if ISFIXED(m) == True then
        l ← m.length();
        O.set(F, m, l);
    else
        // manually writing message-related
           function.
        O.set_custom(F, m, l);
    end
end
return O;
```

D. Generating Inputs

Given that the mutation operation is executed on the Protobuf intermediate representations, it becomes imperative to carry out further procedures that transform these in-memory data structures into binary files suitable for fuzzing. This process, which is the reciprocal of deserialization, involves serialization. Nevertheless, before serialization, we must take into account the fact that mutation inevitably disrupts certain container structure constraints inherent in the intermediate representations. To address this issue, we devised an additional post-processing step. The subsequent discussion elucidates the intricacies of this post-processing design and the serialization process.

1) Post-Processing.: As our objective is to fuzz deeper, we must ensure the container format remains accurate. This ensures a smooth pass through the container parsing stage, thereby enabling a more extensive exploration of subsequent codec stages. Using the post-processing feature involves two steps. Firstly, we register the `message` as the key in the Mutator class's mapping table, with the corresponding callback function as the value. Then, the post-processing process performs a post-order traversal of the Protobuf tree: it processes the child nodes of each message first, followed by the root node. During this traversal, it finds and calls the registered callback functions to repair the current message's constraints.

2) Serialization.: Since the above operations are executed on the Protobuf intermediate representations, we are expected to incorporate a serialization phase, ensuring we obtain a formal file suitable for fuzzing input. For audio files' Protobuf, we developed a serialization code tailored for PROTEUS. The serialization process is relatively straightforward, essentially involving a recursive traversal of the Protobuf's tree structure. During this traversal, each field is converted into a binary byte stream, thus generating the desired output. Algorithm 2 provides a concise overview of this serialization process. Upon initial invocation, the algorithm receives the `message` type parameter from the Protobuf's topmost layer and proceeds to generate a binary audio file. In essence, the algorithm systematically traverses each field within the `message`, outputting the corresponding bytes based on the field's type, and recursively applying this function for any nested `message` types encountered.

Algorithm 2: Serialization Program

Data: Protobuf object O.
Result: A binary file F.

```
INIT(F);
for f in O.fields do
    if ISMESSAGE(F) == False then
        F.WRITE(f, f.value);
    else
        // recursively call the
           serialization function.
        F.WRITE(f, SERIALIZATION(f));
    end
end
return F;
```

IV. EVALUATION

We build PROTEUS based on Google's Project Zero fuzzing project [28] and incorporated TrapFuzz [29] for instrumentation. We used the open-source community version of PEACH Fuzzer [7] for the experiments. All fuzzing activities were performed on macOS Catalina 10.15.3, using a computer equipped with a 2.19 GHz CPU, 32 cores, and 16GB of RAM.

A. Evaluation Methodology

To evaluate the effectiveness and efficiency of PROTEUS, we carried out extensive experiments. Initially, to investigate whether enhanced structure awareness and container format preservation boost vulnerability detection, we compared PROTEUS with the traditional greybox fuzzer, Honggfuzz (**RQ.1**)

[8]. Then, to assess whether coverage feedback leads to more thorough exploration of potential paths and vulnerability discovery, we set PROTEUS against the blackbox fuzzer, Peach (**RQ.2**) [7]. Given the challenges of direct coverage measurement in sourceless programs like WebAudio, we used the number of covered basic blocks as a proxy for assessing coverage effectiveness.

Furthermore, to explore the impact of the extended modules on fuzzing performance, we conducted a series of ablation studies (**RQ.3**). We developed variants of PROTEUS, one without high-level mutation operators and another without the post-processing step, and compared their performance against the fully equipped PROTEUS under identical conditions.

We compiled an initial seed corpus of over 200,000 CAF files from the Internet for our experiments. In our comparative study with Honggfuzz, both PROTEUS and HONGGFUZZ were used to fuzz WebAudio 10,000,000 times. Similarly, for the comparison with PEACH, we crafted a Peach Pit with granularity comparable to that of PROTEUS's structure and subsequently subjected both PROTEUS and PEACH to 10,000,000 fuzzing iterations on WebAudio. For the ablation studies, each variant of PROTEUS, one lacking high-level mutation operators and the other without the post-processing step was applied to fuzz WebAudio 1,000,000 times. Further, we detail a real-world case study of exploiting vulnerabilities in Safari on macOS in §IV-E, to measure the practical effectiveness of PROTEUS in identifying significant security flaws (**RQ.4**).

B. RQ.1 Compared with Traditional Greybox Fuzzer

To ensure fairness in our experimental comparisons, we filtered inputs generated by HONGGFUZZ to focus exclusively on CAF file formats, given its lack of structure awareness, which can lead to generating various file formats.

Table III (column 3, 4) presents our comparison results with HONGGFUZZ. PROTEUS achieved coverage of 7.57% of the basic blocks in WebAudio, slightly less than HONGGFUZZ's 7.90%. This is expected since PROTEUS preserves the container format post-mutation, focusing less on exploring container format parsing blocks, which HONGGFUZZ might traverse more due to its lack of a similar preservation step. However, traditional fuzzers typically cover this aspect adequately. Hence, we focused more on probing deeper into the codec parsing modules.

In terms of crash discovery, PROTEUS identified 36 unique crashes compared to HONGGFUZZ's 19, validating the advantage of its approach in enabling more inputs to reach the codec stage, where more vulnerabilities are likely undisclosed. Fig. 6 and Fig. 7 further illustrate these findings: Fig. 6 shows that both fuzzers quickly reached their maximum block coverage, while Fig. 7 demonstrates that PROTEUS rapidly surpassed Honggfuzz in crash discoveries, indicating effective penetration into deeper codec modules. Additionally, file generation efficiency was comparable between the two fuzzers, with PROTEUS at 59.26 files/s and Honggfuzz at 62.3 files/s, suggesting that the extended components in PROTEUS will not add a significant computational burden.

Overall, despite similar file generation rates and slightly lower coverage metrics, PROTEUS has proven more effective at delving into internal codec modules and uncovering crashes, highlighting its enhanced capability in identifying deeper-seated vulnerabilities.

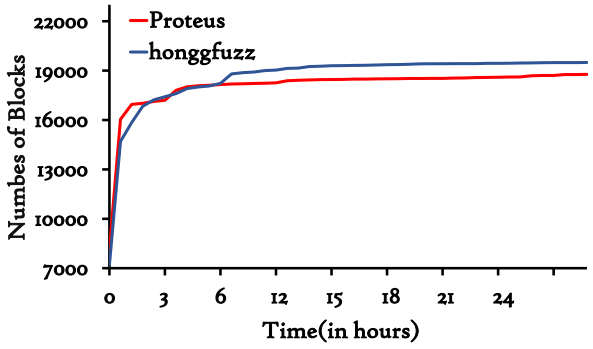

Fig. 6: Number of Blocks over Time.

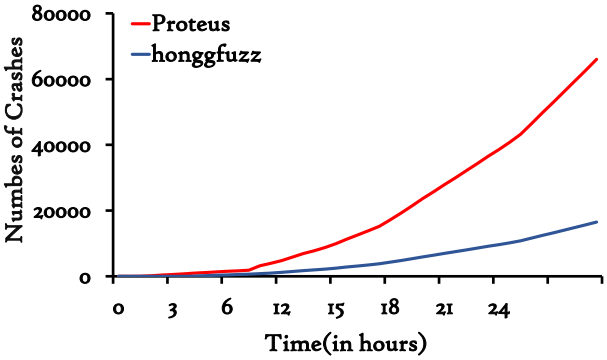

Fig. 7: Number of Crashes over Time.

C. RQ.2 Compared with Blackbox Fuzzer

The blackbox fuzzer, PEACH [7], utilizes PEACH Pit templates to define formats and achieve structure awareness. However, creating a complicated Peach Pit template requires specialized knowledge and significant time, prompting us to compare PROTEUS and PEACH using simpler, coarse-grained templates. Besides, PEACH does not provide real-time feedback on the exploration of basic blocks, so we did not compare the relationship between code coverage and bug detection over time for this experiment.

Table III (column 1, 2) presents our comparison results, where PROTEUS explored 15,227 basic blocks, surpassing PEACH which explored 14,135 blocks. This superior performance is attributed to PROTEUS's extension of Coverage-based Greybox Fuzzing (CGF) principles, utilizing lightweight instrumentation to direct the fuzzer toward unexplored code regions and retain inputs that explore new paths based on feedback from the instrumentation. In terms of crash detection, PROTEUS identified 9 unique crashes compared to PEACH's 5, demonstrating enhanced coverage and a greater ability to unearth vulnerabilities. Additionally, PROTEUS processed

TABLE III: Comparsion among PEACH, HONGGFUZZ, and PROTEUS

	PEACH	Proteus(simple)	HONGGFUZZ	Proteus
format model complexity	SIMPLE	SIMPLE	COMPLICATED	COMPLICATED
efficiency(files/sec)	72.2	85	62.3	59.26
coverage blocks	14135	15227	19611	18777
coverage rate	6.64%	7.16%	7.90%	7.57%
unique crashes	5	9	19	36

inputs at a rate of 85 files/s, faster than Peach's 72.2 files/s, likely due to PROTEUS's extension of HONGGFUZZ, which retains robust support for multithreading—a feature not as advanced in PEACH.

In summary, with similar template granularity, PROTEUS outperforms PEACH across multiple metrics: code coverage, file generation efficiency, and crash discovery, highlighting the advantages of integrating CGF features into blackbox fuzzing contexts.

D. RQ.3 Ablation Study

To rigorously evaluate the impact of each component of PROTEUS on its overall performance, we conducted a series of ablation experiments. These experiments were designed to isolate and assess the individual contributions of various components, allowing us to understand their specific effects on the efficiency and effectiveness of the fuzzing process.

1) Without Post-Processing.: Firstly, we focused on evaluating the role of post-processing, which is primarily aimed at repairing the container format after mutation operations. To assess the impact of this component, we developed a script that checks the validity of the generated files. As detailed in Table IV, our results show a significant improvement in the validity of CAF files when post-processing is applied: validity rates increased to 20%, up from 15% without this component. This 5% increase underscores the essential role of post-processing in enhancing file validity, which in turn allows the fuzzer to engage more effectively with the WebAudio ecosystem. Furthermore, the performance of PROTEUS in detecting crashes notably declined in the absence of post-processing, emphasizing the importance of this component in uncovering vulnerabilities.

2) Without Mutation Operations.: Secondly, we conducted an experiment to assess the significance of mutation operations. The high-level mutation operators play a pivotal role in determining the quality of the generated files and steer the direction of the fuzzer's exploration. Table IV presents the ablation results, revealing that PROTEUS's crash discovery capabilities were significantly diminished without high-level mutation. This outcome is unsurprising, as even if post-processing repairs bit-level mutations that damage the format (as reflected in the similar file validity), this level of mutation alone is insufficient to comprehensively explore the entire program space. Overall, both post-processing and high-level mutation operators contribute positively to enhancing vulnerability detection. The ablation experiments provide compelling evidence of their importance, highlighting the necessity of each component for optimal fuzzing performance.

E. RQ.4 Real-World Impact (CVE-2021-1747, etc.)

We categorized the 11 CVEs detected by PROTEUS into three distinct groups labeled Out-of-bounds Writing, Memory Leak, and Buffer Overflow. Table V presents an illustrative example from each of these CVE categories. One of the most significant CVEs exploited by PROTEUS is `CVE-2021-1747`, which has a 7.8 CVSS Score according to CVE Details [30].

Specifically, This vulnerability exists in the `ACOpusDecoder::AppendInputData` function of the WebAudio module, which will cause out-of-bounds write when parsing CAF files. Fig. 9 shows the specific code. Annotation (1) is a code similar to the bound-checking, but it does not take effect, and annotation (2)'s `memcpy` function is called, causing out-of-bounds write.

Moreover, we further turned this out-of-bounds write into an arbitrary address write. We first conducted a reverse analysis of the vulnerability in the `ACOpusDecoder` structure, where the buf field (1500 bytes) is written out of bounds. The fields `buf_size`, `controlled_field`, and `log_obj` are controllable. In the `ACOpusDecoder::ProduceOutputBufferList` function, by controlling the values of several variables to ensure the return value is greater than or equal to 0, arbitrary address writing is triggered. However, this causes the program to crash. Further analysis revealed that the packet parsing function does not check packet length, leading to out-of-bounds parsing. By constructing two overlapping packets, it is possible to achieve arbitrary address writing while avoiding a crash.

Even with the capability to write to arbitrary addresses, if the program's ASLR protection is robust, an information leak vulnerability is required to exploit it effectively. However, issues within Safari's heap implementation allow us to spray-controlled values at fixed addresses through heap spraying. Given the ability to write to arbitrary addresses, the initial approach is to overwrite the length field in a JSArray or the length field in an ArrayBuffer. Due to Safari's Gigacage mechanism, overwriting the length field of an ArrayBuffer does not permit meaningful content to be read or written. Consequently, we ultimately chose to target the JSArray.

Fig 8 illustrates the entire exploitation process. Initially, a single JS thread performs heap spraying and constructs an audio file, followed by invoking the `decodeAudioData` function to decode the audio, which initiates the audio-A

TABLE IV: Ablation Study

	coverage blocks	coverage rate	case rate	unique crashes
w/o mutation operators	15680	7.37%	20.54%	4
w/o post processing	14794	6.95%	15.42%	10
w/all component	15815	7.73%	20.46%	23

TABLE V: Three representative vulnerabilities exploited by PROTEUS

CVEs	Type	Description	Security
CVE-2021-1747	Out-of-bounds Writing	Audio library out-of-bounds writing	HIGH
CVE-2021-1808	Memory Leak	Reading may cause memory leakage	HIGH
CVE-2021-30960	Buffer Overflow	Audio library buffer overflow	HIGH

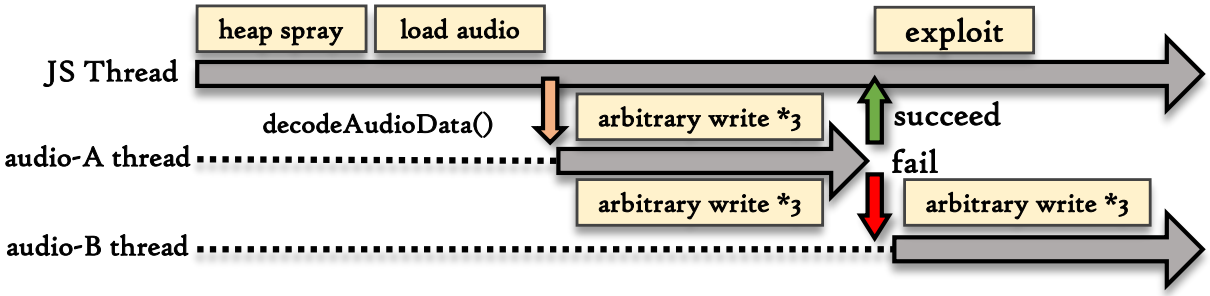

Fig. 8: The Entire Exploitation Process by Using CVE-2021-1747.

```
__int64 __fastcall
ACOpusDecoder::AppendInputData(
ACOpusDecoder*this, const void *a2,
unsigned int *a3, unsigned int *a4,
const AudioStreamPacketDescription *a5){
...
if ( a5 ){
v8 = a5->mDataByteSize;
if ( !a5->mDataByteSize || !*a4 ||
(v9 = a5->mStartOffset, (a5->mStartOffset + v8) > *a3)
|| this->buf_size )
 // (1). bound checking does not take effect here.
{...}
if ( v9 >= 0 ){
memcpy(this->buf, a2 + v9, v8);
 // (2). where out-of-bounds write
...
}...
```

Fig. 9: The Specific Code causing Out-of-Bounds Write.

thread. Assuming the memory layout corresponds to case 1, the audio-A thread triggers three arbitrary address writes. The JS thread checks for changes in the JSArray length after two seconds. If altered, it proceeds with subsequent exploit code. If unchanged, it assumes case 2, calls `decodeAudioData` again, and starts the audio-B thread to decode audio, triggering arbitrary address writes and repeats the JSArray length check.

V. DISCUSSION & FUTURE WORK

1) Generality: To investigate the generality of PROTEUS, we also evaluated it by fuzzing different programs and structured files. Specifically, we fuzzed WebAudio on MP4 files and discovered 5 new bugs. Additionally, we also fuzzed the font parsing library libFontParser.dylib which uses TTF files on the macOS system and discovered 8 new bugs. These bugs have been fixed in the new version released by macOS. Therefore, we consider that PROTEUS not only facilitates the fuzzing of different programs but also provides favorable support for various structured files. Quantitatively speaking, one of us spent one day defining the proto file for MP4 and a half day defining the proto file for TTF. Compared to defining a complete Peach-pit that requires one person to spend 5 days [4], PROTEUS greatly reduces the manual cost of security personnel while maintaining performance in exploiting vulnerabilities.

2) Future Work: The recent work by Godfrey et al. [31] highlights the potential for automatic learning of input formats. Although their study primarily focuses on the PDF format, we believe this research direction remains highly valuable. Looking ahead, we aim to further minimize the one-time manual effort required to specify input formats. Additionally, 010Editor [32], a widely-used format model that has been integrated into numerous fuzzers, will also be incorporated into PROTEUS. This integration, along with other format models, is intended to enhance usability.

VI. Conclusion

WebAudio, a widely utilized audio parsing library, has recently been exposed to harbor critical vulnerabilities. Hence, there is an urgent need for rigorous fuzzing to examine the library. However, traditional greybox fuzzers face challenges due to their limited understanding of input structure and the intricacies of audio file formats. To overcome these limitations, we introduce a novel CGF named PROTEUS. PROTEUS achieves a more thorough fuzzing by acquiring structure awareness, applying high-level mutation operators, and incorporating format repair capabilities. Our experimental evaluation of fuzzing WebAudio demonstrates that PROTEUS outperforms baseline fuzzing techniques, detecting a higher number of crashes without compromising efficiency. Specifically, PROTEUS discovered 36 zero-day bugs (with 11 CVEs assigned).

VII. Acknowledgements

This work was supported in part by the National Key Research & Development Project of China (2023YFB3106800), by the "Pioneer" and "Leading Goose" R&D Program of Zhejiang (2024C03288) and by the CCF-NSFOCUS "Kunpeng" Research Fund.

References

[1] "WebAudio API," https://developer.apple.com/library/archive/navigation/.
[2] C. Reis and S. D. Gribble, "Isolating web programs in modern browser architectures," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 219–232.
[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[4] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2019.
[5] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.
[6] "Afl-fuzz-making-up-grammar-with," https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html.
[7] "Peach Fuzzer," https://www.peach.tech/.
[8] "honggfuzz Fuzzer," https://github.com/google/honggfuzz.
[9] "protobuf," https://github.com/protocolbuffers/protobuf.
[10] "Audio Services," https://developer.apple.com/documentation/.
[11] "SPIKE," http://www.immunitysec.com/downloads/SPIKE2.9.tgz.
[12] "DOMATO," https://github.com/googleprojectzero/domato.
[13] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 445–458.
[14] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, vol. 12, pp. 366–375, 1972.
[15] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.
[16] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.
[17] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, 2008, pp. 206–215.
[18] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 543–553.
[19] "AFL," https://lcamtuf.coredump.cx/afl/.
[20] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032–1043.
[21] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.
[22] M. Böhme and S. Paul, "A probabilistic analysis of the efficiency of automated software testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 345–360, 2015.
[23] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2329–2344.
[24] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2155–2168.
[25] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 627–637.
[26] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 725–741.
[27] "libprotobuf-mutator," https://github.com/google/libprotobuf-mutator.
[28] "Project Zero," https://github.com/googleprojectzero.
[29] "TrapFuzzer," https://github.com/hac425xxx/trapfuzzer.
[30] "CVE-Details," https://nvd.nist.gov/vuln/detail/CVE-2021-1747.
[31] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 50–59.
[32] "010 Editor," https://www.sweetscape.com/010editor/.