DNAFuzz: Descriptor-Aware Fuzzing for USB Drivers

Zhengshu Wang*, Peng He *✉, Fuchen Ma†, Yuanliang Chen†, Shuoshuo Duan‡, Yiyuan Bai‡, and Yu Jiang†
*School of Cyber Science and Technology, Hubei University, Wuhan, China
†School of Software, Tsinghua University, KLISS, BNRist, Beijing, China
‡Shuimu Yulin Technology Co., Ltd

Abstract—**USB is a widely used interface standard in modern operating systems for connecting computers to various external devices. External devices can launch attacks by injecting random data into the host via USB, causing memory errors or even system-level crashes. Fuzzing has been proven to be an effective method to detect USB driver vulnerabilities. However, existing fuzzing methods generate testing inputs without considering the format and semantics of USB descriptors, which define device functionality. As a result, many test cases fail to pass the host's input validation mechanism, leading to ineffective testing.**

In this paper, we propose DNAFuzz, a USB driver fuzzer that generates descriptor-aware payloads. First, it utilizes USB specifications to parse the field definitions and item types of USB descriptors for modeling. Then, based on the field description list and semantic information, DNAFuzz designs mutation strategies to guide the generation of payloads. This approach improves the quality of test cases and the fuzzing effectiveness. Currently, we evaluated DNAFuzz on multiple versions of Linux kernel USB drivers and compared it with state-of-the-art fuzzers, including USBFuzz and Syzkaller. Results show that DNAFuzz significantly improves input quality, successfully increasing the proportion of tests with execution times exceeding 2 seconds by 358% and 65%. In addition, DNAFuzz detected 15 bugs, 11 of which have been fixed or confirmed by the corresponding maintainers.

I. Introduction

USB has become a widely adopted standard interface for connecting various external devices. As a bridge between the host and USB devices, USB drivers manage critical operations such as device identification, configuration, and data transfer. USB descriptors define device characteristics and functionalities, providing the basis for host identification and management. They support the USB devices and the hosts construct interactions in two phases: 1) The enumeration phase: the host identifies the device and initializes the driver by reading descriptors; 2) The data transfer phase: the host exchanges data packets with the device based on the parsed descriptor information.

However, the security issues of USB drivers are often underestimated. Malicious USB devices can exploit vulnerabilities at any stage to attack the host system, leading to data breaches, system crashes, or more severe security threats [1]. For instance, in January 2022, the financially motivated FIN7 [2], [3] group targeted transportation, insurance, and defense companies with BadUSB [4] attacks to deliver REvil and BlackMatter ransomware. Malicious USB drives emulated keyboards and injected keystroke sequences that opened PowerShell and executed commands to retrieve malware. The U.S. Department of Justice estimated the total losses caused by this group at $3 billion. By the first half of 2023, Mandiant [5] observed a threefold rise in infected USB drive attacks targeting sensitive data theft. The recently disclosed high-severity vulnerability CVE-2024-53104 [6], [7] further highlights the serious security risks in USB drivers. This vulnerability resides in the Linux kernel's USB Video Class (UVC) driver uvcvideo module, and stems from the function uvc_parse_format() failing to skip undefined video frames. As a result, uvc_parse_streaming() mishandles these frame types when calculating the frame buffer size, potentially causing out-of-bounds writes. Therefore, detecting potential vulnerabilities in USB drivers is imperative.

Fuzzing is a widely used and effective method for detecting vulnerabilities in host drivers, with mainstream tools including USBFuzz [8], Syzkaller [9], and Saturn [10]. USBFuzz emulates USB devices in software, injecting randomized inputs during driver I/O operations. Syzkaller generates large volumes of random system call sequences to uncover potential kernel vulnerabilities. Saturn employs host-device collaborative fuzzing to explore interaction logic and expose relevant bugs. These fuzzing tools have achieved notable success, uncovering numerous USB driver vulnerabilities. However, due to the complexity of the USB specifications and descriptor structures, existing methods often overlook descriptor format specification and semantic information when injecting mutated data in the enumeration phase. This results in many test cases failing to pass the host's strict input validation mechanisms, thereby hindering the effectiveness of USB driver fuzzing. In practice, ensuring mutated payloads conform to USB specifications poses two major challenges.

The first challenge is accurately modeling USB descriptors. The USB protocol is highly flexible and extensible, allowing devices to define their own characteristics and data structures depending on functionality, vendor requirements, and the specific USB class. As a result, the descriptor structures differ not only across device types but also among models within the same class. However, existing tools (such as Wireshark) [11] have limited capabilities in parsing complex USB descriptors, generally supporting only basic fields while

✉Peng He is the corresponding author.

lacking comprehensive semantic coverage. If the descriptor types and format specifications cannot be accurately parsed, it becomes difficult to map data to the correct fields. Furthermore, applying targeted mutations based on field types and lengths becomes challenging, which limits the effectiveness of mutation strategies and ultimately impacts the quality and efficiency of fuzz testing.

The second challenge is designing effective mutation strategies. Even though items in a USB descriptor have complex attributes, existing methods neither consider designing different mutation strategies based on descriptor structures nor pay attention to the semantic information of fields. Thus, the mutated descriptor data often fails to pass the host's input validation mechanisms. For example, the bDeviceClass field is an 8-bit unsigned integer that specifies the device class. If mutations are performed solely at the type and size level, arbitrary 1-byte values such as 0x15 or 0x72 could be generated. However, the bDeviceClass value carries specific semantics, and 0x15 does not correspond to any device class. Such mutated fields are rejected by host validation checks, hindering effective fuzzing of USB drivers.

To address these challenges, we designed a descriptor-aware payload generation-based USB driver fuzzing tool. First, to accurately model USB descriptors, we extracted and parsed various field definitions and item types [12], [13] from the descriptors based on the USB specification, and converted the descriptor data, represented by byte sequences, into a structured list of field descriptions. Then, to design efficient mutation strategies, we built upon the USB descriptors model and leveraged the converted field descriptions and semantic information to guide input generation and payload mutation, thereby improving test case quality and fuzzing effectiveness. We tested our tool on USB drivers across multiple Linux kernel versions. Compared to state-of-the-art fuzzers like USBFuzz and Syzkaller, our tool significantly improves input quality, successfully increasing the proportion of tests with execution times exceeding 2 seconds by 358% and 65%. Additionally, our tool detected 15 bugs, 11 of which have been fixed or confirmed by the respective kernel maintainers.

The main contributions of this paper are as follows:

- We designed a descriptor-aware fuzzing method to detect vulnerabilities in USB drivers.
- We implemented DNAFuzz, [1] a tool that models descriptors based on the USB specification, and leverages semantic information to guide payload generation and mutation. The tool can be tested on USB drivers across multiple Linux kernel versions.
- Compared to USBFuzz and Syzkaller, DNAFuzz increases the proportion of tests with execution times exceeding 2 seconds by 358% and 65%. In addition, 15 bugs were discovered during continuous fuzz testing.

[1]The prototype of DNAFuzz can be found at: https://anonymous.4open.science/r/DNAFuzz

II. Background

A. USB Protocol and its Communication

Currently, USB has become a widely adopted standard interface worldwide, supporting a vast number of device types. USB employs an asymmetric host-device architecture, with the host serving as the central control unit responsible for device identification and data transmission. All USB data interactions are initiated by the host, while devices simply respond to host requests. A key feature of the USB protocol is its descriptor mechanism [14], which defines device characteristics and functions. Each USB device communicates its information through a series of descriptors covering device type, configuration, interface, endpoint, and other aspects. Among these, the device descriptor provides basic device information. The configuration descriptor details the device's configuration scheme. The HID descriptor specifies functionalities related to human interface devices. Other descriptors further support device customization and functional extensions. The USB specification allows manufacturers to flexibly design device descriptors within the standard framework, which may lead to variations in implementation. Analyzing actual descriptor data therefore provides a more accurate reflection of device behavior. The diversity and hierarchical structure of the descriptor mechanism enable the USB protocol to flexibly accommodate a wide range of device requirements and offer efficient configuration options.

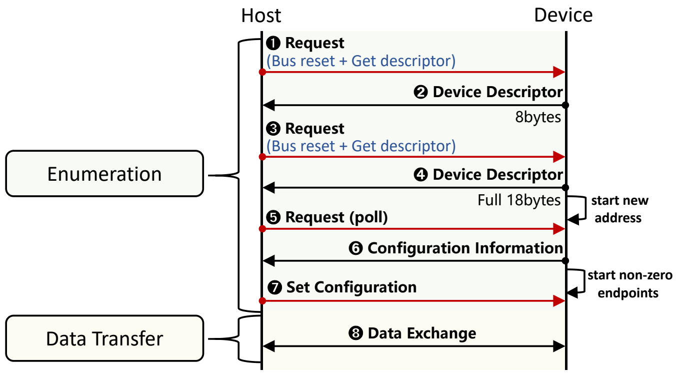


Fig. 1. USB communication process, consisting of two main phases: enumeration, where the host continuously polls the device for data requests, and formal data transfer.

As shown in Figure 1, the USB communication process is divided into the enumeration phase and the data transfer phase. During the enumeration phase, the host initializes the bus with a reset signal and sequentially issues two device descriptor requests: the first request obtains an initial 8-byte descriptor providing basic information, and the second request retrieves a complete 18-byte descriptor defining the device's functionality. Subsequently, the host polls for additional sub-descriptors, including port, endpoint, and HID descriptors, to fully capture the device configuration. Once enumeration is completed, the host and device enter the data transfer phase, exchanging data packets efficiently based on negotiated parameters.

B. USB Protocol Fuzzing

Fuzzing is an automated testing technique that effectively exposes software defects by injecting malformed or unexpected inputs into a program [15]. Modern fuzzing methods fall into two paradigms based on input generation:

- Mutation-based Fuzzing: This black-box technique generates test cases by randomly mutating valid inputs (e.g., protocol packets, file formats) through operations such as bit flips, truncations, or field modifications [16]. Tools like AFL [17] utilize genetic algorithms to iteratively mutate seed inputs while monitoring code coverage.
- Generation-based Fuzzing: Also known as model-driven fuzzing, this approach constructs inputs based on formal specifications (e.g., grammar rules, protocol definitions). Tools like Peach [18] generate test cases defining data models declaratively. This white-box method is more effective when targeting systems with strict input formats, but entails higher implementation complexity.

The USB protocol defines device functionality and communication rules through a standardized descriptor system, which includes a hierarchical descriptor structure and validation process. Device descriptors, configuration descriptors, interface descriptors, and endpoint descriptors form a four-level nested system, with syntax and semantic constraints between layers. During the device enumeration process, the host checks the descriptor structure, reference validity, and functional consistency through a validation mechanism [19]. However, this protocol architecture presents challenges for fuzz testing: custom descriptors increase rule complexity; the operating system's validation strategies create multiple layers of defense; and implicit dependencies between descriptor fields make it difficult for traditional mutation strategies to balance input validity and anomaly. Therefore, USB fuzzing requires a comprehensive understanding of the descriptor system, rather than relying solely on localized perturbations.

C. Threat Model

In this paper, we use the following threat model. We formally define the interaction model of the USB host software stack as $\varphi = \{U, K, B\}$. Specifically, U represents untrusted inputs from user space, which are generated by an attacker using a fuzzer to produce arbitrary or mutated data and injected into the kernel via system calls. K denotes the kernel-space components, including the USB Gadget Core, USB Core, various USB class drivers, and virtual host/device controller drivers. These components are assumed to be logically correct and therefore treated as trusted. B denotes the trust boundary, which is set at the Raw Gadget interface, separating untrusted inputs from the trusted kernel components.

In this model, the underlying hardware is emulated by dummy drivers and is therefore also considered trustworthy. Consequently, the attacker is able to interact with the kernel USB software stack from user space through the Raw Gadget interface, but the attack surface is strictly limited to the communication channel between user-space inputs and the kernel USB stack. Formally, we represent the potential attack interaction as $FuzzerInput \xrightarrow{USBStack} State_{error}$. If the inputs generated by the user-space fuzzer trigger abnormal execution paths or latent vulnerabilities in the USB software stack, the system may enter an erroneous state $State_{error}$. Otherwise, it should correctly process the inputs and continue to provide normal services.

III. Motivating Example

To better understand the challenges of injecting mutated payloads into the operating system kernel during the USB device enumeration, we analyze CVE-2024-56629 [20] as a representative case. This vulnerability persisted in the Linux kernel for eight years and remained undetected by other USB driver fuzzers until its recent discovery during real-world execution. Figure 2 shows the key steps that trigger this error.

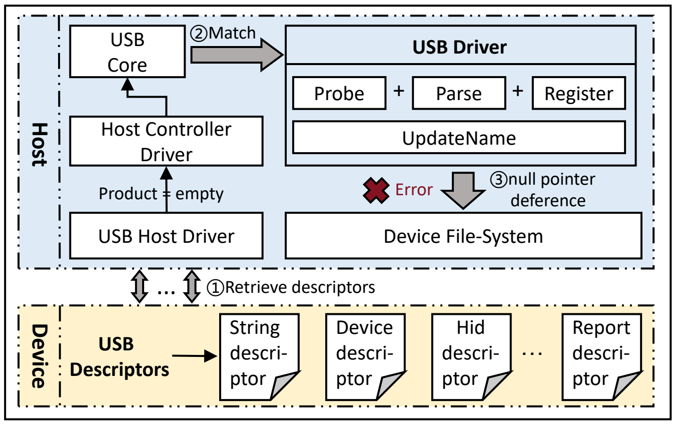


Fig. 2. Key steps to trigger CVE-2024-56629. After the USB device is connected to the host, the host continuously sends requests to retrieve device information (①). The USB core matches the corresponding USB driver based on the device information (②). The USB driver calls the UpdateName function to update the device name (③). During this process, a null pointer dereference error occurs due to the product field in the device information being empty.

A. Bug Triggering

During the device enumeration phase, if the device transmits incorrect or malicious data to the host, it may lead to a series of issues. A typical example is a kernel page fault, which occurs when the product field in the descriptor provided by the device is empty, leading to a null pointer dereference when the USB driver updates the device information. Figure 3 presents the core code snippet that triggers the error.

After the USB device is inserted into the host, the host's USB core establishes communication with the device through the host controller driver and initiates the enumeration process. During this process, the host continuously sends Get_Descriptor requests to retrieve various descriptor information from the device. Based on this information, the USB core matches and loads the appropriate driver for the device. Once the driver is matched, the USB driver calls the probe function to initialize and configure the device. The parse_and_register function parses device information and completes registration, while the update_name function

```
static void wacom_update_name(struct wacom *wacom
    , const char *suffix){
  struct wacom_wac *wacom_wac = &wacom->
      wacom_wac;
  struct wacom_features *features = &wacom_wac->
      features;
  ...
  if (hid_is_usb(wacom->hdev)) {
    struct usb_interface *intf =
        to_usb_interface(wacom->hdev->dev.
        parent);
    struct usb_device *dev =
        interface_to_usbdev(intf);
-   product_name = dev->product;
+   if (dev->product != NULL)
+     product_name = dev->product;
  }
}
```

Fig. 3. The core code snippet of CVE-2024-56629. Line - is the original code, and lines + are the fixed code.

is responsible for updating the device's product information. However, in this case, the product field provided by the device to the USB core is empty, leading to a null pointer dereference in the wacom_update_name function when processing the product string. This prevents registration of the device node in the file system and may cause a system crash. The fixed code (line+) adds a check during the USB driver initialization to ensure that the product field in the descriptor is non-empty. This improvement prevents the null pointer dereference error, effectively mitigating the risk of a system crash.

B. Limitation of Existing Methods

Next, we will demonstrate the limitations of existing state-of-the-art fuzzing tools, which prevent them from detecting this vulnerability. Additionally, we will discuss the efforts required to discover this vulnerability. Currently, USB driver fuzzers have shown promising results and can be categorized into two main types.

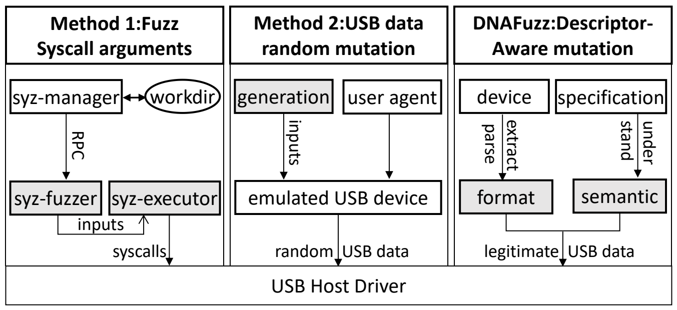


Fig. 4. Existing fuzzing methods to explore USB kernel drivers and our key insight.

Method 1: Fuzzing Syscall Arguments. The primary USB driver fuzzing approach is generating large volumes of random system call sequences, widely adopted by existing tools [9], [21]–[23]. Figure 4 illustrates this testing process. For example, Syzkaller, one of the most advanced fuzzing tools, employs built-in system call descriptions to generate valid or semi-valid system call sequences for testing. Saturn and Healer, on the other hand, guide USB input generation for USB driver fuzzing by learning host-device cooperation [10] and system call correlations. However, the core of this syscall fuzzing method lies in mutating the parameters and sequences of system calls, while lacking specific mutation algorithms for USB data. As a result, it is difficult to perform fine-grained mutations on key fields within the USB protocol. Due to this limitation, during extended real-world testing, the fuzzing tools never set the product field in the USB descriptor to empty, thereby failing to trigger this error.

Method 2: Random Generation of USB Data. This method simulates the device providing completely random data to the driver. USBFuzz is a typical implementation, as shown in Figure 4. USBFuzz extends the fuzzing engine AFL to generate inputs simulating device–host communication during driver I/O operations. While this method can achieve mutations of device data, the mutation process is highly random, ignoring USB protocol specifications and descriptor semantics. Specifically, regarding the aforementioned bug, when mutating the descriptor data, USBFuzz cannot distinguish which values belong to the product field and which belong to other fields. It also fails to semantically infer which fields can reasonably be set to empty. Therefore, due to its randomness, USBFuzz is unlikely to set the product field to empty, thus failing to trigger the bug. The lack of semantic constraints in mutation also results in many test cases failing the host's input validation mechanisms, reducing the fuzzing effectiveness.

The key insight of DNAFuzz: Unlike existing fuzzers, we guide the mutation of USB data based on USB specifications and descriptor semantics, thereby generating more protocol-compliant and targeted test cases. This approach not only increases the pass rate of test cases through the host's input validation mechanism but also significantly improves the fuzzing coverage for USB driver vulnerabilities. For example, for the bug mentioned above, we first parse the product field in the descriptor as a UTF-16 encoded string. Considering the semantic meaning of this field as the product name with few constraints, we keep other fields in the string descriptor unchanged and set the specific product field to an empty string, successfully triggering the bug.

IV. Design

To address the above issue, we designed a descriptor-aware payload generation tool, DNAFuzz, for fuzz testing USB drivers. Figure 5 illustrates the main components and overall workflow of DNAFuzz, divided into two stages: USB descriptor modeling IV-A and USB package generation IV-B. In the USB descriptor modeling stage, DNAFuzz extracts descriptor data from real USB devices. Then, based on the USB specification, it accurately parses descriptor structure and the item fields, converting raw byte sequences of descriptor data into a structured list of field descriptions, $field\_desc$. In the USB package generation stage, DNAFuzz uses the parsed field types, semantic information, contextual constraints from the USB specification, and cross-descriptor relationships to guide the design of the mutation strategy. This enables

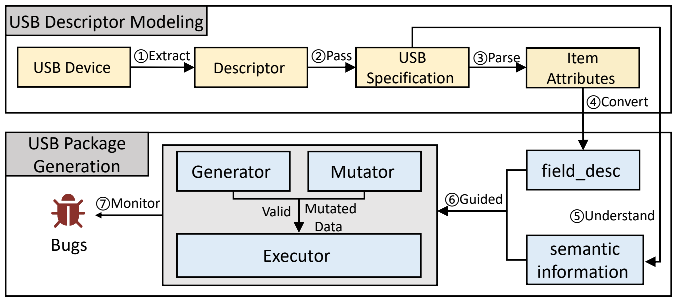


Fig. 5. DNAFuzz Overview: First, descriptor data is extracted from the real USB device (①). Then, the items within the descriptor are parsed based on the USB specification (②,③), and converted into field descriptions (④). Semantic information is understood from the USB specification (⑤), which, along with the field descriptions, guides the design of the mutation strategy (⑥). Finally, instrumentation is added to the host driver to monitor system errors (⑦).

the generation of protocol-compliant and targeted mutated packets. In addition, we add instrumentation to the host driver to monitor potential errors or crashes during execution.

A. USB Descriptor Modeling

To efficiently fuzz test host drivers, it is essential to design mutation strategies that adhere to protocol specifications and are targeted. However, due to the high flexibility and extensibility of the USB protocol, along with the complexity of descriptor types and structures, accurate parsing of descriptor data is critical. If the USB device descriptors cannot be accurately parsed, data fields cannot be identified or mutated based on their characteristics, causing mutation strategies to fail. Therefore, DNAFuzz parses descriptor data extracted from real USB devices based on the USB specification. [2] Specifically, device descriptors are stored as byte sequences, and DNAFuzz parses them to extract structures and fields, converting the data into a structured list of field descriptions. DNAFuzz achieves a high success rate in parsing USB descriptor data and performs effective validation of the device information, providing a reliable foundation for the subsequent mutation strategy design.

USB Device Data Extraction. To cover a broader range of host drivers, DNAFuzz extracts all standard and proprietary descriptors from real USB devices across different Usage IDs. Initially, DNAFuzz monitors the communication path between the host and USB device, intercepting all data packets transmitted over the USB bus, including control transfers during device initialization, descriptor retrieval, bulk transfers for large data volumes, and interrupt transfers for input devices such as keyboards and mice. After packet capture, DNAFuzz classifies them based on different transfer types and the direction of data flow. It then extracts key information such as packet header details, data payloads, and CRC checks, essential for understanding USB device behavior and protocol implementation. Given that the USB specification defines 24 major device classes, each containing multiple subclasses and protocols, actual device descriptors exhibit significant diversity and complexity. We extracted 6 Usage Pages and 23 Usage Names [12] from real USB devices, covering 23 types of USB drivers, including mice, keyboards, touchscreens, and others.

[2]The complete list of fields, their constraining mechanisms, and corresponding mutators is available at: https://anonymous.4open.science/r/DNAFuzz/USBSpecsManualStudy.md

Descriptor Data Parsing. USB descriptors can be categorized into standard descriptors and proprietary descriptors. Standard descriptors include five types (e.g., device and configuration descriptors), with field order and length explicitly defined by the USB specification. For example, DNAFuzz parses device descriptors strictly according to the protocol: starting with the length field in the first byte, followed by descriptor type (USB_SCAPY_TYPE_DEVICE = 0x01), USB version, class, subclass, and others. Since field misalignment can cause parsing errors, DNAFuzz introduces a boundary control mechanism to ensure that each field is extracted at the correct position. It is worth noting that some proprietary descriptors are defined in ways similar to standard descriptors.

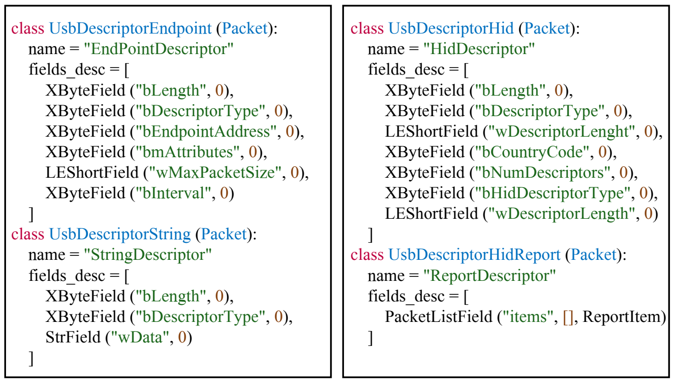
In contrast, parsing proprietary descriptors is more complex, as they are designed with greater flexibility to accommodate diverse device reporting needs. For example, the HID report descriptor consists of items that strictly define packet formats, functions, and constraints. It typically includes Main, Global, and Local items, representing semantic elements such as the Usage Page and Usage. A typical mouse descriptor has about 30 items, while more complex devices (e.g., digitizers) may include up to 200 items, covering nearly 400 Usages for inputs such as buttons, axes, and pressure. To improve storage and transmission efficiency, report data are stored in a compact format, but this also increases the difficulty of automated parsing. Existing tools (e.g., Wireshark) support only basic fields, such as report length, and lack key semantics like the Usage Page. To address this limitation, DNAFuzz builds on the external library $HidReportParse.so$ [24] and further implements a standardized parsing method for report descriptors. This method decomposes each item into three core fields: item tag, item type, and item size, which are used to identify its type and function. The specific recognition rules and standards are provided in Table I, which lists the item attributes corresponding to different prefixes. For instance, in

TABLE I
IMPORTANT ITEM PREFIXES INVOLVED IN THE REPORT DESCRIPTOR.

item type	item tag	Item Prefix, nn as Data Length
Main	Input	1000 00 nn
	Output	1001 00 nn
	Collection	1010 00 nn
	End Collection	1100 00 nn
Global	Usage Page	0000 01 nn
	Logical Minimum	0001 01 nn
	Logical Maximum	0010 01 nn
	Report Size	0111 01 nn
	Report ID	1000 01 nn
	Report Count	1001 01 nn
Local	Usage	0000 10 nn
	Usage Minimum	0001 10 nn
	Usage Maximum	0010 10 nn

the byte sequence 0x26, 0xff, 0x00, the prefix 0x26 indicates the item type Logical Maximum. The following two bytes, 0xff, 0x00, are interpreted in little-endian order as 0x00ff, which corresponds to a logical maximum value of 255.

Convert to Field. In defining the Field classes, DNAFuzz adheres to modular and structured design principles to ensure that the modeling results strictly conform to protocol specifications while supporting precise parsing and construction. By inheriting from base field classes (such as ByteField and ShortField), DNAFuzz customizes the logic for specific field types. Serialization and deserialization methods are overridden to enable accurate modeling of field values during transmission. For more complex structures, DNAFuzz introduces container types such as PacketListField and callback mechanisms to support the dynamic handling of nested fields, thereby enhancing the flexibility and extensibility of the modeling framework.



```
class UsbDescriptorEndpoint (Packet):
    name = "EndPointDescriptor"
    fields_desc = [
        XByteField ("bLength", 0),
        XByteField ("bDescriptorType", 0),
        XByteField ("bEndpointAddress", 0),
        XByteField ("bmAttributes", 0),
        LEShortField ("wMaxPacketSize", 0),
        XByteField ("bInterval", 0)
    ]
class UsbDescriptorString (Packet):
    name = "StringDescriptor"
    fields_desc = [
        XByteField ("bLength", 0),
        XByteField ("bDescriptorType", 0),
        StrField ("wData", 0)
    ]
```

```
class UsbDescriptorHid (Packet):
    name = "HidDescriptor"
    fields_desc = [
        XByteField ("bLength", 0),
        XByteField ("bDescriptorType", 0),
        LEShortField ("wDescriptorLenght", 0),
        XByteField ("bCountryCode", 0),
        XByteField ("bNumDescriptors", 0),
        XByteField ("bHidDescriptorType", 0),
        LEShortField ("wDescriptorLength", 0)
    ]
class UsbDescriptorHidReport (Packet):
    name = "ReportDescriptor"
    fields_desc = [
        PacketListField ("items", [], ReportItem)
    ]
```

Fig. 6. Example list of field descriptions, including two standard descriptors (Endpoint Descriptor and String Descriptor) and two proprietary descriptors (HID Descriptor and Report Descriptor).

Under this mechanism, DNAFuzz systematically enumerates each individual field in USB descriptors and specifies its size, order, type, and related attributes. Figure 6 presents examples of field descriptions, including two standard descriptors (Endpoint Descriptor and String Descriptor) and two proprietary descriptors (HID Descriptor and Report Descriptor). For standard USB descriptors and certain proprietary ones, the field type definitions remain consistent. For instance, in the Endpoint Descriptor, the bLength field is defined as an XByteField, which inherits from ByteField and represents a one-byte unsigned integer, while the wMaxPacketSize field is defined as an LEShortField, a two-byte unsigned integer encoded in little-endian format. In contrast, for more flexible proprietary descriptors such as the Report Descriptor, we designed a ReportItem class based on both the USB and HID specifications to parse and construct individual items.

Algorithm 1 illustrates the overall process of parsing and converting a report descriptor into field descriptions. First, the algorithm iterates through all the items in the *reportdesc* (Lines 1-2). When the item.type is main and the item is an input item (Lines 4-5), it determines whether padding [13] is required and the amount of padding based on the value and the product of report_count and report_size (Line 6). Next,

Algorithm 1: Descriptor parsing and field generating

```
   Input  : HidReport descriptor: $reportdesc$
   Output: Field descriptions: $reports$,
           List of report identifiers: $report\_id$
1  Function generateField($reportdesc$):
2      for $item \in reportdesc$ do
3          // Check if the type is main
4          if $isMain(item.type)$ then
5              if $isInput(item.tag)$ then
6                  field_desc.append=checkpad
                    ($report\_count$,$report\_size$) ;
7                  pagebuf=gen.replace($usage\_page$) ;
8                  while $index < report\_count$ do
9                      field_desc.append=convert
                        ($usage$) ;
10                     $index$ +=1 ;
11             else
12                 usage.clear() ;
13         if $isGlobal(item.type)$ then
14             intFromBytes ($item.value$) ;
15             if $isreportid(item.tag)$ then
16                 $reports.append(field\_desc)$ ;
17                 $report\_id.append(item.value)$ ;
18                 $field\_desc.append(report\_id)$ ;
19     if $reports.len == 0$ then
20         reports.append($field\_desc$) ;
21     return ($reports$,$report\_id$)
22 End Function
```

it creates a page buffer (Line 7) and parses the data items according to the report_count (Line 8). For each data item, the algorithm generates the corresponding field object using usage and report_size (Lines 9-12). If the item.type is global, the field size is extracted and converted to an integer (Lines 13-14). When encountering a report_id item, the algorithm adds the current field description and value size to the reports and report_id lists, thereby saving and processing the report ID (Lines 15-18), a key step of the algorithm. Finally, it ultimately returns all the field descriptions and report IDs (Lines 19-21).

B. USB Package Generation

In USB devices, descriptor data is interrelated, and any anomaly during the enumeration process can cause the driver to fail to load correctly. To achieve efficient payload generation from descriptors, we construct a mutation strategy framework centered on the USB descriptor model, and further incorporate an in-depth understanding of the semantic information of USB descriptors to optimize the mutation process and ensure the rationality of the generated mutated packets. In addition, we add instrumentation to the host driver to monitor potential errors or crashes during execution.

Mutation Strategy Framework. DNAFuzz implements a multi-level, multi-type mutation mechanism, generating test data through flexible mutation strategy design and implementation. The core of the mutation strategy consists of several key components. First, the MutationLevel enumeration defines three levels of mutation strength: mild, moderate, and deep. DNAFuzz generates mutated data accordingly. The Mutator class, as the core component of DNAFuzz, is responsible for generating and displaying mutated data. Specifically, the Mutator class provides three methods: gen() to generate mutated data, count() to return the number of possible mutations, and show() to output a string representation. Under the framework of the Mutator class, DNAFuzz implements specific mutators for various field types (e.g., IntMutator, StrMutator, ByteMutator), supporting data processing ranging from 1 to 128 bits. The SequenceStrategy class implements multiple data mutation strategies, including data packet reordering and data packet insertion. By manipulating the sequence of data packets or inserting additional binary packets, SequenceStrategy simulates various data flow patterns, further enriching the mutation scenarios. The MutationDedup class controls the deduplication scope and method through bitwise masking, supporting deduplication strategies at different granularities such as global, scene, or field levels, to avoid redundant mutations and enhance test data diversity.

Additionally, DNAFuzz extends the Mutator class with specialized subclasses, such as MapMutator for mapping to other data types. Through these mechanisms, DNAFuzz flexibly generates mutated data tailored to different testing needs according to user-specified mutation levels and strategies.

Semantic Information Understanding. Based on the USB specification semantics, we design three key mechanisms: the self-limitation mechanism, the consistency constraint mechanism, and the structural rationality assurance mechanism, to guide the mutation strategy and generate high-quality test cases. Each mechanism fully considers the semantics of descriptors, ensuring the effectiveness of the mutation process and the rationality of the test cases. Specific examples are provided in Figure 7 to illustrate their application.

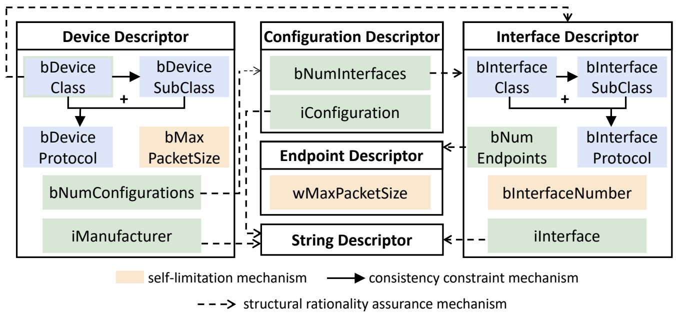


Fig. 7. Descriptor semantic relationships and three mechanisms: self-limitation mechanism, consistency constraint mechanism, and structural rationality assurance mechanism.

- **Self-Limitation Mechanism:** Ensures field values remain within valid ranges. For instance, the bMaxPacketSize field in the device descriptor, representing the maximum packet size for endpoint zero, is limited to 8, 16, 32, and 64. DNAFuzz ensures the packet size remains valid during mutation to prevent invalid cases.
- **Consistency Constraint Mechanism:** Ensures consistency between descriptor fields. For example, if bInterfaceClass is zero, bInterfaceSubclass must also be zero. Their combination also restricts valid bInterfaceProtocol values. DNAFuzz enforces these constraints to ensure rational mutations.
- **Structural Rationality Assurance Mechanism:** Ensures mutations reflect changes in device configuration. The bNumInterfaces value in the configuration descriptor affects the number and parsing method of subsequent interface descriptors. DNAFuzz uses it to guide host parsing and configuration to prevent descriptor mismatches or errors.

Moreover, through the collaboration of these mechanisms, DNAFuzz can adjust descriptor data, ensuring consistent relationships between fields. For example, in the consistency constraint mechanism, the bDeviceClass field limits bDeviceSubclass and bDeviceProtocol values, while in the structural rationality assurance mechanism, it determines the class definition of aggregated interfaces in the interface descriptor. This cross-mechanism collaboration improves mutation effectiveness and accuracy. [3]

V. Implementation

USB Descriptor Modeling. We implemented the extraction and parsing of USB descriptors based on the BusHound [25] tool and the external library HidReportParse.so. Before testing, we monitor changes on the bus to capture protocol packets and input/output operations from various USB devices. The collected descriptor data, in the form of byte sequences, is then passed to the fuzzing program, which uses external libraries and custom rules to parse the data attributes. This process allows us to extract and analyze USB descriptor information, providing reliable data support for subsequent fuzz testing.

USB Packet Generation. We implemented USB packet injection into the host using the Scapy [26] library and the extended Raw Gadget [27] component. First, we defined the registerFieldType function to register corresponding parsing methods for various field types in Scapy, enabling flexible protocol and packet processing and type mapping in the program. We then extended Raw Gadget, which simulates virtual USB devices sending data to the host under test. This solution requires no hardware support and relies on the virtual HCD/UDC module (Host Controller Driver/USB Device Controller). The virtual devices connect to the kernel running Raw Gadget, simulating various types of USB devices. As a pure software solution, it offers significant flexibility and scalability without requiring any modifications or upgrades to the hardware, allowing effective fuzzing of USB drivers across different kernel versions.

VI. Evaluation

To validate the effectiveness of DNAFuzz, we performed a series of experiments on the latest version of the Linux kernel.

[3]The complete list of fields, their constraining mechanisms, and corresponding mutators is available at: https://anonymous.4open.science/r/DNAFuzz/semantic-aware-mechanismList.md

Specifically, we used test cases generated with descriptor-aware generation as inputs for DNAFuzz, and compared the test-case quality and kernel code coverage between DNAFuzz and prior methods, demonstrating its ability to improve input quality and explore deeper execution paths. We further listed and verified discovered bugs to demonstrate DNAFuzz's effectiveness in vulnerability detection. We designed experiments to address the following questions:

- RQ1: How is the quality and effectiveness of the test cases generated by DNAFuzz?
- RQ2: How does DNAFuzz perform in vulnerability detection?
- RQ3: How does DNAFuzz perform in terms of improving code coverage?

Hardware and Software Environment. The experiments were conducted on a Linux server equipped with a 32-core Intel i9-14900 processor and 32 GiB memory, running 64-bit Ubuntu 22.04.4 LTS. The same configuration was applied to both virtual machine settings and related parameters. Specifically, all experiments ran concurrently with evenly distributed resources: each VM received 2 cores and 4 GiB memory. To minimize statistical errors, each experiment was repeated 10 times, and the average result was reported [28].

Guest OS Preparation. Our evaluation uses the latest Linux USB host driver and long-term supported kernel versions from v5.5 to v6.13. The related USB driver configuration was enabled and compiled into the kernel. We customized the target Linux host as follows: (i) enabled gcov (CONFIG_GCOV_KERNEL=y) [29] to collect code coverage; (ii) compiled and dynamically loaded common and critical USB drivers into the kernel; (iii) enabled multiple tools, including Kernel Address Sanitizer (KASAN) [30] and Undefined Behavior Sanitizer (UBSAN) [31] to detect vulnerabilities.

A. *Input Quality*

To evaluate the quality of inputs generated by DNAFuzz, we recorded the execution times of the tests and compared them with those of Syzkaller and USBFuzz. Experiments were conducted on four kernel versions (v5.5, v6.0, v6.12, and v6.13-rc2), representing the latest or long-term stable releases at submission. USBFuzz supports only Linux v5.5, because it relies on a software-simulated USB device that feeds random data during I/O operations and patches kernel assembly code, which changed significantly in later versions. To minimize measurement errors, we repeated each experiment 10 times and averaged the results.

Testing Quality Comparison. Manual analysis of test cases indicates that execution time is closely correlated with input quality. If a test fails early in the enumeration process, it terminates quickly; otherwise, the execution time increases as the enumeration progresses. When the simulated descriptor binds to a driver, the duration depends on the driver implementation. Longer execution times are more likely to trigger complex code paths in the driver, thereby increasing the likelihood of uncovering deeper vulnerabilities [8], [23]. We conducted statistical experiments on USB device enumeration times, and the results show significant variation across devices. For example, recognizing a mouse takes about 0.3 seconds, whereas recognizing a complex game controller may take more than 1.5 seconds. Prior studies have also shown that a test duration exceeding 2 seconds generally ensures that the test passes host validation and enters the data transfer phase. Therefore, we set 2 seconds as the threshold for evaluating the quality and effectiveness of test cases [8].

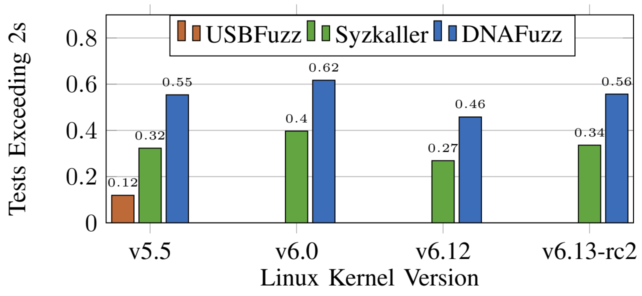


Fig. 8. Comparison of the proportion of tests lasting more than 2 seconds generated by DNAFuzz, USBFuzz, and Syzkaller. DNAFuzz achieves longer-lasting tests across all four versions.

As shown in Figure 8, we present the proportion of tests generated by DNAFuzz with execution times exceeding 2 seconds, compared to state-of-the-art tools. On average, the proportion of tests generated by DNAFuzz lasting longer than 2 seconds is 358% higher than USBFuzz and 65% higher than Syzkaller. Specifically, for kernel v5.5, although DNAFuzz outperforms existing fuzzers, approximately 45% of the tests still execute in under 2 seconds. There are two points where the test execution remains short. One occurs on devices with minimal descriptor structures and simple functions, where the entire testing process, including enumeration and transfer, finishes within 2 seconds. The other scenario can be considered a fuzzing failure. For example, when the bInterfaceClass field in the interface descriptor identifies the device as a mass storage or wireless device, such devices typically require multiple protocols (such as UASP and BOT protocols) and involve encryption processes. However, DNAFuzz currently lacks support for these protocol-specific semantics, leading to premature termination of the generated tests. These factors together result in shorter execution times for some tests. Furthermore, we conducted comparative experiments using Wireshark as the parser in the DNAFuzz modeling stage. The results indicate that in this scenario, 39% of the test execution times exceed 2 seconds, which shows a significant disparity compared to DNAFuzz's performance on this metric.

The improvement in high-quality test generation indicates that DNAFuzz effectively explores more host-driven code paths by leveraging a descriptor-aware mutation strategy. This strategy, guided by USB specifications and descriptor semantics, enables DNAFuzz to generate protocol-compliant test cases, increasing the success rate of mutation data passing enumeration validation and triggering deeper USB driver logic.

TABLE II
DNAFuzz identified 15 vulnerabilities, and with the assistance of maintainers, 11 of them have been recognized and confirmed.

#	Kernel Operation	Bug Type	Bug Symptom	Version Appeared	Status
1	hid_set_field	Undefined Behavior Bug	shift-out-of-bound	6.0 - 6.12.0	Fixed
2	check_uevent_buffer	Memory Bug	logic bug	6.7.0	Fixed
3	handle_invalid_bp_value	Unexpected state reached	logic bug	6.7 - 6.13 - rc1	Confirmed
4	vmw_send_msg	Unexpected state reached	logic bug	6.7 - 6.13 - rc1	Confirmed
5	rcpu_preempt_kthread	Unexpected state reached	soft lockup	6.12 - 6.13 - rc1	Confirmed
6	_do_syscall_ioctl	Memory Bug	vmalloc-out-of-bound	6.7.0 - 6.12.0	Confirmed
7	hid_generic_probe	Undefined Behavior Bug	shift-out-of-bound	6.13.0 - 6.13 - rc2	Confirmed
8	hid_report_raw_event	Undefined Behavior Bug	shift-out-of-bound	6.1.0 - rc4	Fixed
9	wacom_update_name	Memory Bug	NULL pointer dereference	6.6.0	Fixed
10	raw_release	Memory Bug	slab-use-after-free	6.12 - 6.13 - rc2	Fixed
11	usb_ep_free_request	Undefined Behavior Bug	logic bug	6.12.0	Confirmed
12	hid_hw_start	Undefined Behavior Bug	shift-out-of-bound	6.0 - 6.12.0	Reported
13	arch_stack_walk	Unexpected state reached	soft lockup	6.12.0	Reported
14	stack_trace_save	Unexpected state reached	soft lockup	6.9.0	Reported
15	sysvec_apic_timer_interrupt	Undefined Behavior Bug	logic bug	6.12.0	Reported

B. Bug Finding

To address RQ2, we conducted a two-week testing campaign on multiple recent and long-term stable versions of Linux, during which we discovered 15 unique bugs in the host drivers and USB core. Of these, 9 were detected by KASAN and UBSAN, involving memory and undefined behavior bugs, including use-after-free (1), NULL pointer dereference (1), logic bugs (3), and out-of-bounds memory access (4). The remaining 6 bugs were caused by certain faults or unexpected states in the host. Although developers may be aware of these unexpected states, appropriate handling measures have not been taken yet. We reported these bugs to the developers, assisting them in reproducing and evaluating their potential impact. Currently, 11 have already been fixed or confirmed.

As shown in Table II, we provide detailed information on all discovered vulnerabilities and their affected kernel versions, identified via binary search [32]. Memory bugs and undefined behaviors in USB drivers pose serious risks to kernel stability and security. For example, the kernel operation hid_set_field (bug#1) triggers an out-of-bounds shift, causing undefined behavior that crashes the system or results in denial of service. Similarly, wacom_update_name (bug#9), when the name is null, dereferences the pointer directly, triggering a Kernel Panic and degrading system availability. Most of these vulnerabilities are critical, including many that have remained undetected and unreported in the USB driver codebase for decades. Despite the extensive computational resources used by Syzkaller and USBFuzz in continuously testing Linux host drivers, they discovered only a subset of the vulnerabilities listed in the table, with Syzkaller covering 4 bugs and USBFuzz covering just 1. In terms of detection efficiency, our approach also outperforms the others: for bug#1, #2, #8, and #10, Syzkaller's average detection time was 2.1 hours compared to our 1.7 hours; for bug#12, USBFuzz required 3.4 hours, whereas our approach took 2.6 hours. These results demonstrate that DNAFuzz's descriptor-aware payload generation significantly improves fuzz testing effectiveness for USB drivers.

C. Coverage Improvement

To address RQ3, we monitored the fuzz testing process on multiple Linux host versions, comparing code coverage with state-of-the-art fuzzers. Additionally, to further validate the coverage capability of the descriptor-aware fuzz testing method on USB device drivers, we selected 9 common USB device drivers and compared the coverage achieved by Syzkaller, USBFuzz, and DNAFuzz for these device drivers.

Branch Coverage Comparison. We instrumented the USB host-related code (USB core framework, host controller drivers, and USB device drivers) to compare the code coverage of Syzkaller, USBFuzz, and DNAFuzz. During the 24-hour testing period, we extracted coverage data every ten minutes and computed the statistical average for each fuzzing tool over ten executions.

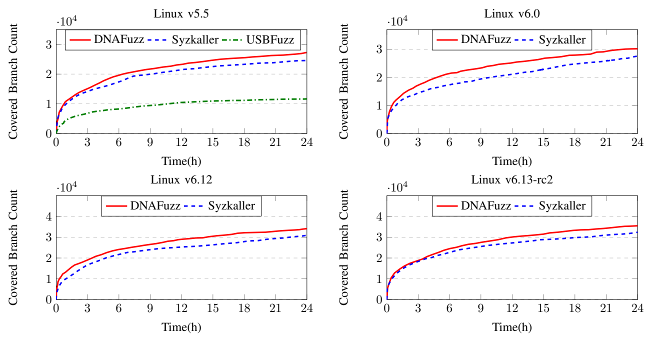


Fig. 9. Comparison of branch coverage between DNAFuzz, Syzkaller, and USBFuzz over 24 hours. In all four versions, DNAFuzz achieves superior coverage statistics within the same amount of time.

Figure 9 compares the branch coverage across the three fuzzers. The results indicate that DNAFuzz achieved higher branch coverage within the same testing time. All tools show significant coverage improvement during the first 8 hours, but the growth rate slows as time progresses. Specifically, Syzkaller and DNAFuzz exhibit similar coverage growth during the initial 3 hours, but the growth rate of Syzkaller started to slow down, while DNAFuzz's slowdown was relatively more gradual. In contrast, USBFuzz maintained lower cover-

TABLE III
DNAFUZZ, SYZKALLER, AND USBFUZZ BRANCH COVERAGE STATISTICS ON USB HOST-RELATED CODE.

Version	DNAFuzz	Syzkaller	USBFuzz	Improvement
5.5	27287	24583	11692	+10.9% / +133%
6.0	30164	27674	-	+8.9%
6.12	34316	31329	-	+9.5%
6.13 - rc2	35498	32451	-	+9.3%
Average	31816	29009	11692	+9.4% / +133%

age throughout the 24-hour test. Table III provides the coverage improvement statistics of DNAFuzz relative to Syzkaller and USBFuzz. Compared to USBFuzz, DNAFuzz increased branch coverage by 133% on Linux v5.5; across four kernel versions, DNAFuzz's coverage improved by an average of 9.3% over Syzkaller.

These experimental results not only highlight the differences in the detection strategies of the various fuzzing tools but also reveal performance disparities between them. USBFuzz, by simulating devices, demonstrates good coverage of USB device drivers but performs poorly when targeting the USB core framework, resulting in a significant gap between USBFuzz and other fuzzers. In contrast, DNAFuzz outperforms Syzkaller in terms of coverage, primarily due to its ability to precisely model USB descriptors and incorporate semantic understanding, thereby effectively guiding the generation of mutated data. The similarity in coverage growth between the two tools can be attributed to the relative ease of covering many common kernel modules (such as usbcore and hub). In addition, while Syzkaller relies heavily on manual analysis and custom protocol message generation, DNAFuzz significantly reduces engineering costs through automation.

Common USB Driver Coverage Comparison. To further validate DNAFuzz's performance in terms of coverage, we utilized kcov and gcov to extract and compare the branch coverage of Syzkaller, USBFuzz, and DNAFuzz for 9 common USB device drivers. As shown in Table IV, the descriptor-aware generation strategy improved coverage by 25% and 36% over Syzkaller and USBFuzz, respectively, further demonstrating DNAFuzz's advantage in exploring the core logic of USB drivers. Meanwhile, DNAFuzz performs state resets by automatically disconnecting and re-establishing the connection with the USB driver after each test case, thereby avoiding full system reboots. This mechanism not only improves testing efficiency but also enables stateful fuzzing, allowing the exploration of code paths that depend on state continuity.

TABLE IV
BRANCH COVERAGE OF SYZKALLER, USBFUZZ, AND DNAFUZZ ACROSS NINE COMMON USB DEVICE DRIVERS.

Driver Name	Total Blocks	DNA-Fuzz	Syzk-aller	USB-Fuzz	Impr vs syzkaller	Impr vs USBFuzz
ftdi_sio	465	113	102	98	+11%	+15%
option	46	25	14	16	+79%	+56%
analog	206	68	57	49	+19%	+39%
cypress_m8	284	105	76	82	+38%	+28%
usbhid	451	146	137	106	+7%	+38%
sierra	189	51	44	38	+16%	+34%
synaptics	455	144	101	92	+58%	+57%
trackpoint	82	56	31	27	+43%	+107%
cp210x	373	119	97	101	+23%	+18%
Total	2551	827	659	609	+25%	+36%

Note: Drivers 1, 2, 4, and 6 are USB serial devices. Drivers 3 and 9 are joysticks. Drivers 5 and 8 are mice. Driver 7 is a touchpad.

In addition, DNAFuzz exhibits consistent effectiveness in both multi-driver integration and single-driver testing. For example, in a 24-hour dedicated test on the trackpoint (mouse driver), its branch coverage reached approximately 68%, consistent with the results of multi-driver integration and fully demonstrates DNAFuzz's stability across different scenarios.

VII. DISCUSSION

A. Data Transfer Phase

The primary goal of DNAFuzz is to enhance input quality, enabling more test cases to successfully pass USB enumeration and reach the data transfer phase. To evaluate the role of the data transfer phase in USB driver fuzzing, we modified DNAFuzz by removing all logic related to this phase, so that each test case terminates immediately after enumeration and proceeds to the next round. This reduced version is referred to as DNAFuzz-. We then ran DNAFuzz- on the USB host-related code for 24 hours and compared the final branch coverage with the full DNAFuzz results (Table III). As shown in Table V, DNAFuzz achieves 48.2% higher coverage on average.

TABLE V
DNAFUZZ AND DNAFUZZ- BRANCH COVERAGE STATISTICS ON USB HOST-RELATED CODE.

Version	DNAFuzz	DNAFuzz-	Improvement
5.5	27287	18217	+49.8%
6.0	30164	20481	+47.3%
6.12	34316	23165	+48.1%
6.13 - rc2	35498	23995	+47.9%
Average	31816	21465	+48.2%

In addition, we conducted a root cause analysis of the 15 bugs discovered by DNAFuzz, among which 4 were triggered during the data transfer phase. These results demonstrate that covering the data transfer phase is critical for improving the effectiveness of fuzzing. In DNAFuzz, this strategy not only significantly increases coverage but also exposes additional potential vulnerabilities.

B. Bug Reproducibility

DNAFuzz discovered 15 USB driver vulnerabilities, though some cannot be reliably reproduced. The reproducibility of bugs remains an open problem, with two primary causes. First, the complex concurrency of the Linux kernel is a key contributor. The concurrent execution of user-mode and kernel-mode threads may cause nondeterministic states, making bug reproduction difficult. Second, memory resources constrain the number of test rounds the fuzzer can record, which may restrict access to sufficient historical data during bug reproduction.

Currently, DNAFuzz employs a sliding-window strategy to record USB data packets in each test round, dynamically adjusting the window range over time. When an error occurs, the system begins with a small subset of packets from the sliding window, gradually expanding to the full window and replaying specific packets sequentially to maximize error reproduction and validation. In the future, we plan to use event-based or semantic compression and storage strategies to reduce redundancy, enabling the recording of more error execution data. This will help the fuzzer more accurately analyze kernel logs during reproduction and more effectively validate the triggered vulnerabilities.

C. Adapt to a New Kernel Version

To adapt DNAFuzz to a new Linux kernel version, we focused on adapting the Raw Gadget module for efficient kernel communication. By default, Raw Gadget processes control and data endpoint requests in a blocking manner, preventing non-blocking polling during fuzz testing. We modified control endpoint requests to support non-blocking mode, thereby improving response efficiency. Additionally, we adjusted the timeout to the maximum response time specified by the USB standard, optimizing data endpoint handling.

Different kernel versions may have varying function names, parameters, or interfaces, which can result in incompatibilities with Raw Gadget. This necessitates manual code inspection and modification, which demand familiarity with kernel variations and strong debugging skills. However, the core engine of DNAFuzz, including descriptor parsing and mutation packet generation, remains modular and generalizable. These core functions are less dependent on specific kernel versions or Raw Gadget implementations, allowing them to be migrated and reused across kernel versions. This allows adaptation efforts to concentrate on kernel interface adjustments without extensive modifications to DNAFuzz's core logic, thereby enhancing efficiency and flexibility.

VIII. Related Work

USB Testing. The USB driver, as a key component connecting the operating system with external devices, is crucial for system security and stability. Recently, various testing techniques have been proposed for USB drivers, including static analysis [33], [34], dynamic analysis [35]–[37], symbolic execution [38]–[40]. In addition, fuzzing is also a useful technique for USB driver testing. Researchers use simulation-based methods to emulate USB devices and reduce hardware dependency. QEMU [41], [42], as a widely used device emulator, supports a variety of peripherals, and tools like USBFuzz [8] have used it to simulate devices for fuzz testing. With the increasing number of devices and growing driver code complexity, researchers have proposed alternative simulation methods based on QEMU to enhance its ability to emulate a large number of devices. For example, printfuzz [43] automates device simulation, supporting device detection, hardware interrupt simulation, and device I/O interception, successfully simulating hundreds of devices and advancing fuzz testing. In addition, DR.FUZZ [44] designs a semantic-aware mechanism that understands the relevant data structures in drivers, enabling fuzzing through host validation.

Kernel Fuzzing. Many kernel fuzzing tools improve system call sequences to test the kernel and trigger crashes. These fuzz tests typically use the kernel's built-in gadget module as a peripheral to respond to various requests from the host-side driver. For example, Syzkaller [9] generates system call sequences for testing through built-in system call descriptions, focusing on basic system call sequence generation. To explore the kernel's complex logic, Healer [23] proposes a method that utilizes learned system call relationships to guide input generation and mutation, thereby enhancing the effectiveness of fuzz testing. Subsequently, KSG [21] uses the domain-specific language Syzlang to automatically generate system call specifications, enabling large-scale generation of system call sequences and significantly increasing test coverage. FUZZUSB [45] integrates static analysis with conformance-based hybrid fuzzing, while Saturn [10] introduces a host-cooperative fuzz testing approach. These two approaches effectively expand the breadth of testing.

Main Difference. Different from the above work, DNAFuzz is a USB driver fuzzer that generates descriptor-aware payloads. Most USB driver fuzzers are inefficient due to two reasons: 1) they do not parse model USB descriptors, making it difficult to design specific mutation algorithms, and 2) they ignore the semantic information in descriptor fields, limiting the mutation process's effectiveness and the rationality of test cases. In contrast, DNAFuzz focuses on parsing descriptor formats and understanding field semantics, enabling high-precision mutation strategies based on different fields. This enhances the quality of fuzzing inputs and improves USB driver fuzzing efficiency. Furthermore, DNAFuzz can be extended to different host versions with simple custom kernel configurations.

IX. Conclusion

This paper presents DNAFuzz, a descriptor-aware payload generation method for USB driver fuzzing. First, DNAFuzz precisely parses descriptor data based on the USB specification to achieve modeling of USB descriptors. Next, DNAFuzz combines the semantics of USB descriptors with the modeling results to design mutation strategies, effectively improving the quality of input data and enabling efficient fuzz testing of host drivers. DNAFuzz detects 15 vulnerabilities in USB drivers, 11 of which have been confirmed or fixed by developers. Compared with state-of-the-art tools, DNAFuzz improves input quality by up to 358% and 65% respectively. Our future work will focus on supporting a wider range of device types and operating systems, and exploring further possibilities in firmware programmability.

X. Acknowledgements

This work is supported by the Major Program (JD) of Hubei Province (2023BAA018), the Key R&D Program of Wuhan (2023010402010783) and the Wuhan Industrial Innovation Joint Laboratory (2024010902040444).

REFERENCES

[1] N. Nissim, R. Yahalom, and Y. Elovici, “USB-based attacks,” *Computers & Security*, vol. 70, pp. 675–688, 2017.

[2] K. Sigler, “Fin7 sends BadUSB devices to U.S. businesses as part of targeted ransomware campaign,” [hyphens]https://www.trustwave.com/en-us/resources/blogs/trustwave-blog/fin7-sends-badusb-devices-to-us-businesses-as-part-of-targeted-ransomware-campaign/, 2022.

[3] C. Cimpanu, “FBI: Fin7 hackers target us companies with BadUSB devices to install ransomware,” [hyphens]https://therecord.media/fbi-fin7-hackers-target-us-companies-with-badusb-devices-to-install-ransomware, 2022.

[4] N. Karsten, K. Sascha, and L. Jakob, “Badusb — on accessories that turn evil,” https://cdn.prod.website-files.com/6098eeb4f4b0288367fbb639/62bc77c194c4e0fe8fc5e4b5_SRLabs-BadUSB-BlackHat-v1.pdf, 2014.

[5] R. Joven and N. C. Kiat, “The spies who loved you: Infected USB drives to steal secrets,” https://cloud.google.com/blog/topics/threat-intelligence/infected-usb-steal-secrets/, 2023.

[6] U. S. government, “Cve-2024-53104 detail,” https://nvd.nist.gov/vuln/detail/CVE-2024-53104, 2025.

[7] RedHat, “Cve-2024-53104,” https://access.redhat.com/security/cve/cve-2024-53104, 2024.

[8] H. Peng and M. Payer, “USBFuzz: A framework for fuzzing USB drivers by device emulation,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2559–2575.

[9] Google, “Syzkaller - kernel fuzzer,” https://github.com/google/syzkaller, 2018.

[10] Y. Xu, H. Sun, J. Liu, Y. Shen, and Y. Jiang, “SATURN: Host-gadget synergistic USB driver fuzzing,” in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4646–4660.

[11] W. D. Community, “Wireshark: Network protocol analyzer,” https://www.wireshark.org/?hl=zh-cN, 2023.

[12] U. I. Forum, “HID Usage Tables 1.6,” https://www.usb.org/document-library/hid-usage-tables-16, 2025.

[13] ——, “Device class definition for human interface devices (HID),” https://www.usb.org/sites/default/files/hid1_12.pdf, 2025.

[14] armKEIL, “USB Component: USB Descriptors,” https://www.keil.com/pack/doc/mw/USB/html/_u_s_b__descriptors.html, 2024.

[15] 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.

[16] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[17] lcamtuf, “american fuzzy lop,” https://github.com/google/AFL, 2013.

[18] Google, “Peach fuzzer,” https://github.com/MozillaSecurity/peach, 2007.

[19] L. kernel’s development community, “The linux kernel documentation,” https://www.kernel.org/doc/html/latest/, 2025.

[20] CVE-2024-56629, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-56629, 2024.

[21] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang, “KSG: Augmenting kernel fuzzing with system call specification generation,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 351–366.

[22] S. Pailoor, A. Aday, and S. Jana, “MoonShine: Optimizing {OS} fuzzer seed selection with trace distillation,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 729–743.

[23] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, “Healer: Relation learning guided kernel fuzzing,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 344–358.

[24] pasztorpisti, “hid-report-parser,” https://github.com/pasztorpisti/hid-report-parser, 2024.

[25] Perisoft, “Bus hound version 7.05,” https://bushound.com/bushound/index.htm, 1998.

[26] gpotter2, Pierre, V. Guillaume, and W. Nils, “Scapy,” https://github.com/secdev/scapy, 2007.

[27] K. Andrey, C. Aristo, S. Nicolas, and G. Radoslav, “Raw gadget,” https://github.com/xairy/raw-gadget, 2020.

[28] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén *et al.*, *Experimentation in software engineering*. Springer, 2024, vol. 236.

[29] T. kernel development community, “Using gcov with the linux kernel,” https://docs.kernel.org/dev-tools/gcov.html.

[30] ——, “Kernel address sanitizer,” https://docs.kernel.org/dev-tools/kasan.html, 2023.

[31] ——, “Undefined behavior sanitizer,” https://docs.kernel.org/dev-tools/ubsan.html, 2023.

[32] ——, “Reporting issues,” https://www.kernel.org/doc/html/latest/admin-guide/reporting-issues.html, 2023.

[33] J.-J. Bai, T. Li, K. Lu, and S.-M. Hu, “Static detection of unsafe DMA accesses in device drivers,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1629–1645.

[34] X. Tan, Y. Zhang, X. Yang, K. Lu, and M. Yang, “Detecting kernel refcount bugs with {Two-Dimensional} consistency checking,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2471–2488.

[35] K. Denney, E. Erdin, L. Babun, M. Vai, and S. Uluagac, “Usb-watch: a dynamic hardware-assisted usb threat detection framework,” in *Security and Privacy in Communication Networks: 15th EAI International Conference, SecureComm 2019, Orlando, FL, USA, October 23-25, 2019, Proceedings, Part I 15*. Springer, 2019, pp. 126–146.

[36] A. Negi, S. S. Rathore, and D. Sadhya, “Usb keypress injection attack detection via free-text keystroke dynamics,” in *2021 8th International Conference on Signal Processing and Integrated Networks (SPIN)*. IEEE, 2021, pp. 681–685.

[37] R. Dumitru, D. Genkin, A. Wabnitz, and Y. Yarom, “The impostor among {US (B)}:{Off-Path} injection attacks on {USB} communications,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5863–5880.

[38] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. Butler, “Firmusb: Vetting usb device firmware using domain informed symbolic execution,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2245–2262.

[39] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, “Selective symbolic execution,” in *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[40] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, “{FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution,” in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 463–478.

[41] F. Bellard, “Qemu, a fast and portable dynamic translator.” in *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46. California, USA, 2005, pp. 10–5555.

[42] Q. Team, “Qemu:a generic and open source machine emulator and virtualizer,” https://www.qemu.org/, 2018.

[43] Z. Ma, B. Zhao, L. Ren, Z. Li, S. Ma, X. Luo, and C. Zhang, “Printfuzz: Fuzzing linux drivers via automated virtual device simulation,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 404–416.

[44] W. Zhao, K. Lu, Q. Wu, and Y. Qi, “Semantic-informed driver fuzzing without both the hardware devices and the emulators,” in *Network and Distributed Systems Security (NDSS) Symposium 2022*, 2022.

[45] K. Kim, T. Kim, E. Warraich, B. Lee, K. R. Butler, A. Bianchi, and D. J. Tian, “Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2212–2229.