Universidade de São Paulo
Instituto de Matemática, Estatística e Ciência da Computação
Bacharelado em Ciência da Computação

Detecção de Vulnerabilidades em Hosts USB utilizando Técnicas de *Fuzzing* Externo

Gabriel Geraldino de Souza

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Marcos Antonio Simplicio Junior

São Paulo
2025

Agradecimentos

Agradeço ao Professor Doutor Marcos Antonio Simplicio Junior e ao Professor Doutor Bruno de Carvalho Albertini, pela orientação, disponibilidade e pelas contribuições fundamentais ao desenvolvimento deste trabalho.

Agradeço, também, ao mestrando Gustavo Cerqueira Bastos pelo apoio, pelas sugestões técnicas e pela atenção durante as etapas de construção deste projeto.

Resumo

Gabriel Geraldino de Souza. **Detecção de Vulnerabilidades em Hosts USB utilizando Técnicas de *Fuzzing* Externo**. Monografia (Bacharelado). Instituto de Matemática, Estatística e Ciência da Computação, Universidade de São Paulo, São Paulo, 2025.

O protocolo *Universal Serial Bus* (USB) é uma das interfaces de comunicação mais onipresentes em sistemas computacionais, conectando desde periféricos simples a componentes de sistemas críticos, como caixas eletrônicos e dispositivos industriais. Essa ampla adoção, contudo, estabelece-o como uma significativa superfície de ataque. Este trabalho tem como objetivo principal avaliar a segurança de sistemas que utilizam o protocolo USB por meio de *fuzzing*, uma técnica de teste automatizada que consiste no envio de dados massivos, inválidos ou inesperados para a descoberta de vulnerabilidades. Este trabalho envolve uma revisão da literatura sobre o protocolo USB e falhas conhecidas, seguida pelo desenvolvimento de uma ferramenta de software capaz de interpretar, manipular e reproduzir capturas de tráfego USB (.pcap) em baixo nível, utilizando o módulo `raw-gadget` do *kernel* Linux. Esta ferramenta foi integrada a *fuzzers* como `syzkaller` e `radamsa`, para automatizar a geração de casos de teste e identificar falhas de segurança, como negação de serviço (DoS), corrupção de memória e outras anomalias no tratamento de pacotes. Os experimentos comparam a eficácia dos diferentes métodos de mutação em relação ao impacto prático das vulnerabilidades encontradas.

Palavras-chave: USB. *Fuzzing*. Análise de Vulnerabilidades. Teste de Software.

Abstract

Gabriel Geraldino de Souza. **Detection of Vulnerabilities in USB Hosts Using External *Fuzzing* Techniques**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

The Universal Serial Bus (USB) protocol is one of the most ubiquitous communication interfaces in computer systems, connecting everything from simple peripherals to critical system components, such as ATMs and industrial devices. This widespread adoption, however, makes it a significant attack surface. The main objective of this work is to evaluate the security of systems that utilize the USB protocol through *fuzzing*, an automated testing technique that consists of sending massive, invalid, or unexpected data to discover vulnerabilities. This work involves a review of the literature on the USB protocol and known flaws, followed by the development of a software tool capable of interpreting, manipulating, and reproducing low-level USB traffic captures (.pcap) using the `raw-gadget` kernel module. This tool was integrated with *fuzzers* such as `syzkaller` and `radamsa` to automate the generation of test cases and identify security flaws such as denial of service (DoS), memory corruption, and other anomalies in packet handling. The experiments compare the effectiveness of different mutation methods in relation to the practical impact of the vulnerabilities found.

Keywords: USB. *Fuzzing*. Vulnerability Analysis. Software Testing.

Lista de Abreviaturas

CVE	Vulnerabilidades e Exposições Comuns (*Common Vulnerabilities and Exposures*)
CWE	Enumeração de Fraquezas Comuns (*Common Weakness Enumeration*)
URL	Localizador Uniforme de Recursos (*Uniform Resource Locator*)
USB	Barramento Serial Universal (*Universal Serial Bus*)
URB	Bloco de Requisição USB (*USB Request Block*)
BCD	Decimal Codificado em Binário (*Binary-Coded Decimal*)
HNP	Protocolo de Negociação de Host (*Host Negotiation Protocol*)
SBC	Computador de Placa Única (*Single Board Computer*)
FPGA	Matriz de Portas Programáveis em Campo (*Field-Programmable Gate Array*)
TT	Tradutor de Transações (*Transaction Translator*)
LSB	Bit Menos Significativo (*Least Significant Bit*)
IME	Instituto de Matemática, Estatística e Ciência da Computação
USP	Universidade de São Paulo

Lista de Figuras

Lista de Tabelas

Lista de Programas

Sumário

Capítulo 1

Introdução

1.1 Contexto

Hoje, a sociedade moderna tem a tecnologia presente de forma universal em seu cotidiano. Essa presença é intrinsicamente sustentada por uma série de protocolos e interfaces de comunicação que, muitas vezes, operam de forma invisível ao usuário final. Dentre estas, poucas são tão essenciais quanto o *Universal Serial Bus* (USB). Desde sua concepção, o USB evoluiu de uma simples interface para conectar periféricos de baixa velocidade, como teclados e mouses, para um ecossistema complexo e de alta performance, capaz de transmitir dados a gigabits por segundo, fornecer energia para carregamento rápido de dispositivos e conectar sistemas das mais distintas naturezas.

Atualmente o protocolo USB está integrado a quase todos os dispositivos computacionais do dia-a-dia. Ele é o cerne da conexão de periféricos em computadores pessoais e notebooks, mas sua aplicação se estende muito além. É encontrado, também, em sistemas embarcados, infraestruturas críticas como caixas eletrônicos (ATMs), sistemas de controle industrial (ICS), equipamentos médicos, sistemas automotivos e dispositivos de Internet das Coisas (IoT). Essa disseminação massiva, embora tenha trazido conveniência e padronização sem precedentes, introduziu simultaneamente uma superfície de ataque vasta e muitas vezes subestimada.

A complexidade inerente ao protocolo USB, com suas diversas especificações, classes de dispositivos, múltiplos modos de operação e inúmeras implementações e interfaces cria um terreno fértil para a existência de vulnerabilidades. Uma falha na implementação de um drivers USB de um sistema operacional ou no *firmware* de um dispositivo periférico pode ter consequências severas, que vão desde a negação de serviço (DoS) até a execução de código espúrio e completa tomada de controle do sistema hospedeiro.

1.2 Problema e Motivação

A avaliação da segurança de implementações USB é, portanto, uma tarefa de crucial importância. Contudo, ela apresenta desafios significativos: muitas implementações de *drivers* e *firmwares*, por exemplo, são de código fechado (*closed-source*), o que impede a

análise estática de código e a revisão manual por parte de pesquisadores de segurança. Além disso, a interação com o hardware em baixo nível exige conhecimento especializado e ferramentas adequadas. Nesse cenário, abordagens de teste de segurança de caixa-preta (*black-box*), que não requerem acesso ao código-fonte do alvo, tornam-se difíceis, mas indispensáveis.

Uma das técnicas mais eficazes para a descoberta de vulnerabilidades em cenários de *black-box* é o *fuzzing*. O *fuzzing* é um processo de teste de software automatizado que envolve o fornecimento de dados massivos, inválidos, inesperados ou aleatórios como entrada para um programa. O objetivo é monitorar o comportamento do alvo em busca de anomalias, como travamentos, asserções de código ou vazamentos de memória, que possam indicar a presença de falhas de segurança exploráveis.

A motivação para este trabalho surge da necessidade de se ter uma ferramenta mais acessível e eficaz para realizar o *fuzzing* do protocolo USB em baixo nível. Embora existam soluções de hardware e software para tal, poucas se concentram na automação da geração de casos de teste mutacionais a partir de capturas de tráfego reais, de uma maneira que seja agnóstica ao dispositivo e ao sistema operacional. A capacidade de capturar uma comunicação USB legítima, aplicar mutações inteligentes e reproduzi-la contra um alvo permite simular uma gama quase infinita de interações anômalas, potencializando a descoberta de vulnerabilidades que seriam difíceis de encontrar por meio de análise manual ou testes tradicionais.

1.3 Objetivo

O objetivo principal deste trabalho é desenvolver e avaliar uma abordagem *black-box* para o *fuzzing* de sistemas que utilizam o protocolo USB, com foco na descoberta de vulnerabilidades não somente em drivers de sistemas operacionais e *firmwares* de dispositivos, mas também na camada de aplicação.

Para alcançar este objetivo principal, foram definidos os seguintes objetivos específicos:

- revisar a literatura existente sobre o protocolo USB, suas arquiteturas, especificações e as vulnerabilidades de segurança historicamente conhecidas;
- desenvolver uma ferramenta de software capaz de interpretar, manipular e reproduzir capturas de tráfego USB em baixo nível, de forma que seja possível emular um dispositivo USB e enviar pacotes arbitrários;
- adaptar essa ferramenta de forma que possa ser utilizada como um *fuzzer* de contexto geral, automatizando a geração de casos de teste por meio da mutação de pacotes USB e, possivelmente, integrando com *mutators* existentes, como o `radamsa`;
- realizar experimentos práticos contra alvos selecionados para identificar falhas de segurança, como negação de serviço (DoS), corrupção de memória e outras anomalias no tratamento de pacotes;
- analisar e comparar a eficácia dos diferentes métodos de mutação empregados, avaliando o impacto e a severidade das vulnerabilidades encontradas.

1.4 Estrutura do trabalho

O presente trabalho, além da introdução, é composto por cinco capítulos. O capítulo 2 apresenta uma visão geral do protocolo USB, incluindo sua arquitetura, funcionamento e principais características. O capítulo 3 discute a técnica de *fuzzing*, detalhando diferentes estratégias e aplicações. O capítulo 4 detalha o desenvolvimento da ferramenta de *fuzzing* USB, abordando desde a escolha do hardware até a implementação do software. O capítulo 5 descreve os experimentos realizados e os resultados obtidos, analisando a eficácia da ferramenta em diferentes cenários. Finalmente, o capítulo 6 finaliza o trabalho, discutindo suas contribuições e sugerindo direções para pesquisas futuras.

Capítulo 2

USB

O padrão *Universal Serial Bus* (USB) começou a ser desenvolvido no início da década de 1990 por um consórcio de empresas de tecnologia liderado por Intel, Compaq, Microsoft, IBM, DEC, NEC e Nortel. O objetivo era criar uma interface fundamentalmente nova para superar as limitações de usabilidade e a fragmentação de conexões legadas, como portas seriais (RS-232), paralelas e conectores PS/2 . Oficializado com a publicação da especificação USB 1.0 em 1996, o padrão não somente unificou a conexão de periféricos, mas introduziu o conceito de *Plug and Play*, simplificando drasticamente a configuração de hardware para o usuário final.

Embora a especificação original 1.0 tenha definido as bases, foi a revisão USB 1.1, lançada em 1998, que começou a popularizar o padrão, com as velocidades *low speed* (1.5 Mbits/s) e *full speed* (12 Mbits/s). Contudo, a onipresença do barramento consolidou-se com a chegada do USB 2.0 em abril de 2000. Esta versão introduziu o modo *high speed*, elevando a taxa de transferência para 480 Mbits/s – um aumento de 40 vezes sobre a versão anterior – permitindo que o USB substituísse interfaces de alta largura de banda e se tornasse o padrão dominante de mercado.

Versões subsequentes continuam a expandir a tecnologia, como o USB 3.0 (2008), que introduziu a *SuperSpeed* (5 Gbits/s). Além da performance de dados, houve uma evolução crítica na capacidade de fornecimento elétrico com o padrão *USB Power Delivery* (USB-PD). É importante notar que, para manter a promessa de universalidade, o USB preserva uma forte retrocompatibilidade. Essa decisão de design, embora benéfica para o usuário, resulta em uma pilha de protocolos complexa e extensa, que deve gerenciar múltiplos estados, velocidades e classes de dispositivos simultaneamente.

2.1 Arquitetura Física e Lógica

O USB opera sob uma arquitetura de barramento host-driven, com uma topologia de estrela em camadas. Um único dispositivo *host* (ou hospedeiro), como um computador, coordena toda a comunicação com múltiplos periféricos, também nomeados como *devices* ou *gadgets*.

O barramento USB suporta a conexão de até 127 dispositivos simultaneamente, incluindo o *host*, *hubs* e dispositivos finais; este é um limite estrutural da especificação USB, dado que o campo de endereço contém 7 bits e o endereço 0 é um endereço reservado para *broadcast*.

Hubs são dispositivos intermediários que expandem o número de portas disponíveis, permitem a conexão em cascata de múltiplos dispositivos, garantem flexibilidade à topologia, onde dispositivos podem ser conectados e desconectados dinamicamente, facilitam a gestão de energia e segmentação do tráfego.

É ilustrado na figura 2.1 uma topologia típica de barramento USB, com múltiplas camadas e especificações. Nesse exemplo, o *hub* raiz se comunica com o *hub* A em *high speed* (USB 2.0) e o mouse conectado ao *hub* A é isolado pelo *hub* utilizando um *Transaction Translator* (TT), permitindo que o mouse opere em *full speed* (USB 1.1) sem impactar a performance dos demais dispositivos. O funcionamento do TT é descrito pela primeira vez na seção 11.14 da especificação USB 2.0, conforme USB Implementers Forum, 2025b.

Do lado do *host*, a interação com o barramento é gerenciada por uma controladora de *hardware*, que também contém o *hub* raiz (*root hub*). Historicamente, existiram diferentes especificações, como a Universal *Host Controller Interface* (UHCI) e a *Open Host Controller Interface* (OHCI). A especificação USB 2.0 introduziu a *Enhanced Host Controller Interface* (EHCI), que se tornou o padrão para a operação em *high speed*, simplificando o ecossistema de drivers.

Fisicamente, as interface USB 1.1 e 2.0 consiste em um cabo de quatro vias: duas para alimentação, VCC e GND, e duas para a transmissão de dados, $D+$ e $D-$. O padrão 3.0 conta com um canal adicional, com linhas TX/RX adicionais para garantir maior velocidade. A comunicação é serial e os dados são transmitidos de forma diferencial sobre o par $D+/D-$, o que confere alta imunidade a ruído eletromagnético. Para garantir transições de sinal suficientes para a sincronização dos *clocks* entre o *host* e o *device*, o USB utiliza a codificação *Non-Return-to-Zero Inverted* (NRZI) com a técnica de *bit stuffing*, que insere um *bit* 0 após seis *bits* 1 consecutivos no fluxo de dados.

2.2 Estrutura de Comunicação

2.2.1 Pacotes

Os pacotes são as unidades atômicas de informação que trafegam no barramento. Os dados são transmitidos com o bit menos significativo (LSB) primeiro. Quase todos os pacotes USB são compostos por algum conjunto dos seguintes campos:

- **SYNC**: sequência de *bits* usada para sincronizar os *clocks* do transmissor e receptor. Presente em todos os pacotes;
- **PID (*Packet ID*)**: campo de 8 *bits* que identifica o tipo do pacote (token, dados, handshake, SOF). Cada `PID` é composto por 4 bits de identificação seguidos por seus 4 *bits* complementares, como uma simples forma de *checksum*, permitindo detecção de erros. Presente em todos os pacotes;

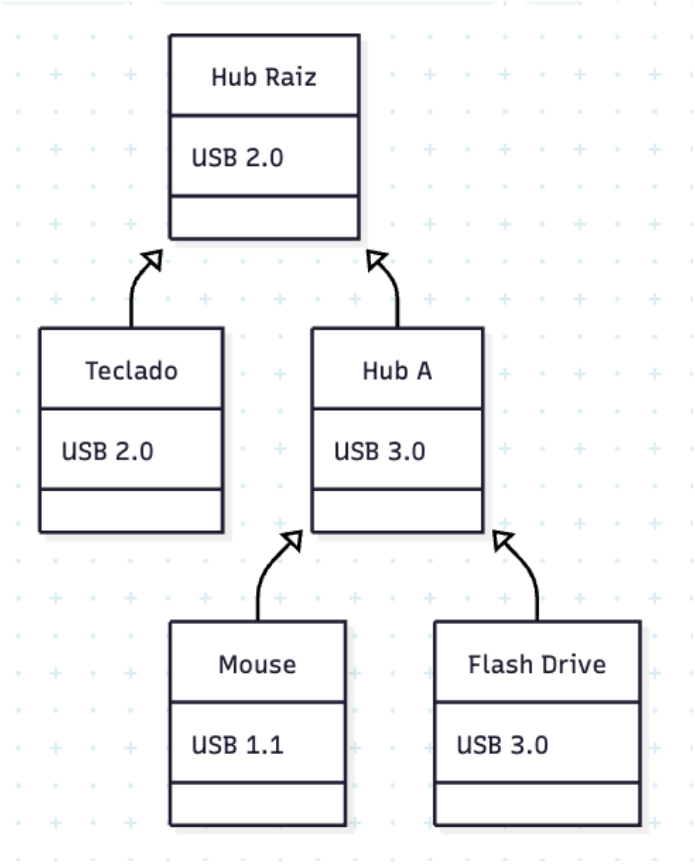

Figura 2.1: *Topologia USB, com múltiplas camadas e especificações*

- **endereço do dispositivo (ADDR)**: campo de 7 *bits* que especifica o endereço do dispositivo de destino da transação;
- **endpoint (ENDP)**: campo de 4 *bits* que identifica o endpoint específico dentro do dispositivo;
- **dados**: campo variável que contém a carga útil (*payload*), presente apenas em pacotes de dados;
- **CRC (*Cyclic Redundancy Check*)**: campo usado para detecção de erros, com 5 *bits* para pacotes de token e 16 *bits* para pacotes de dados;
- **EOP (*End of Packet*)**: sinaliza o fim do pacote. Presente em todos os pacotes.

O protocolo USB 2.0 define quatro categorias principais de pacotes, cada uma identificada por PIDs distintos:

- **pacotes de *token***: enviados para iniciar uma transação e podem ser do tipo `IN`, onde o host deseja ler informações, `OUT`, onde o host envia dados ao dispositivo e `SETUP`, para iniciar transações de controle;
- **pacotes de dados**: transportam a carga útil (*payload*). Existem diversos tipos de pacotes de dados (`DATA0`, `DATA1`, `DATA2`, `MDATA`), definidos pelo `PID`;
- **pacotes de *handshake***: indicam o status da transação. Podem ser `ACK` (confirmação),

`NAK` (indisponibilidade temporária), `STALL` (erro) e `NYET` (ainda sem resposta);

- **pacotes de *Start of Frame* (SOF)**: *broadcast* enviado pelo host em intervalos regulares para manter a sincronização do barramento. Essencial para agendar transferências periódicas (isócronas e de interrupção).

2.2.2 Transações

No nível lógico, a comunicação é organizada em transações. Como o barramento é centrado no *host*, é ele quem inicia as transações. Uma transação completa é uma sequência de pacotes que realiza uma transferência de dados útil e é tipicamente composta por três fases:

- **pacote de *token***: o cabeçalho da transação, enviado pelo *host*. Define a natureza da transação (`IN`, `OUT`, `SETUP`), o endereço do dispositivo e o endpoint de destino;
- **pacote de dados (opcional)**: contém a carga útil (*payload*). É enviado pelo *host* (em transações `OUT`) ou pelo *device* (em transações `IN`);
- **pacote de status (*handshake*)**: a fase final, onde o receptor acusa o recebimento (`ACK`), reporta indisponibilidade temporária (`NAK`) ou sinaliza um erro (`STALL`).

2.2.3 Transferências

Transferências são operações lógicas de alto nível que podem envolver múltiplas transações para completar o envio ou recebimento de dados. Existem quatro tipos principais de transferências USB, cada uma adequada a diferentes necessidades de comunicação:

- **transferências de controle**: são mandatórias e essenciais para a operação do barramento. Elas são usadas pelo *host* para operações de comando e status, mais notavelmente durante o processo de enumeração, para descobrir, configurar e endereçar um dispositivo recém-conectado;
- **transferências de interrupção**: ao contrário do que o nome indica, não são transferências exatamente baseadas em interrupções. São baseadas em uma sondagem periódica (*polling*) feito pelo *host* e são usadas para dados pequenos e não periódicos que exigem latência máxima, como cliques de um mouse;
- **transferências em lote (*bulk transfers*)**: destinadas a grandes volumes de dados sem requisitos de tempo real. Há garantia de entrega baseada no CRC e handshake ACK/NAK, com retransmissão em caso de erros. São usadas para dispositivos como impressoras e armazenamento em massa;
- **transferências isócronas (*isochronous transfers*)**: utilizadas para fluxos de dados que exigem entrega contínua e em tempo real, como áudio e vídeo. Pacotes corrompidos ou perdidos não são retransmitidos, priorizando a latência sobre a integridade dos dados.

As seções 5.6.4 e 5.7.4 da especificação USB 2.0 detalham um aspecto importante em relação às transferências periódicas (isócronas e de interrupção): em *full speed*, não mais

que 90% da banca pode ser alocada para transferências periódicas, o limiar é de de 80% para *high speed* USB Implementers Forum, 2025b.

2.3 Iniciação e Detecção de Velocidade

Apesar de toda a comunicação ser essencialmente controlada pelo *host*, a conexão é fisicamente iniciada pelo dispositivo. Quando um dispositivo é conectado à porta, ele deve anunciar sua presença ao *host* através de um resistor de *pull-up* de $1.5k\Omega$. A posição deste resistor determina a velocidade inicial do dispositivo:

- pull-up em $D+$: dispositivo de *full speed* ou *high speed*;
- pull-up em $D-$: dispositivo de *low speed*.

O *host* detecta a mudança de tensão causada por este resistor e inicia o processo de enumeração. Dispositivos com capacidade *high speed* devem, obrigatoriamente, se conectar inicialmente como *full speed*. Durante a enumeração, ocorre um específico onde o dispositivo demonstra sua capacidade de operar em 480 Mbits/s, e só então o host comuta o barramento para o modo de alta velocidade.

Uma extensão à especificação USB 2.0 é o *USB On-The-Go* (OTG), que permite que dispositivos atuem tanto como *host* quanto como periférico, dependendo do contexto da conexão, mas nunca simultaneamente. Nesse caso, o estado inicial de quem é *device* e quem é *host* é determinado pela orientação do cabo e a troca de papéis é negociada através do *Host Negotiation Protocol* (HNP), conforme descrito na seção 5.2 da extensão USB Implementers Forum, 2025a da especificação USB 2.0.

2.3.1 Enumeração e Descritores

Após a iniciação da conexão, o *host* inicia o processo de enumeração, que é a sequência de passos necessários para reconhecer, configurar e preparar o dispositivo para comunicação. Durante este processo, o *host* se comunica com o *endpoint* 0 do dispositivo (o *endpoint* de controle padrão) para ler uma hierarquia de estruturas de dados chamadas descritores (*descriptors*). Estes descritores informam ao *host* tudo o que ele precisa saber sobre o dispositivo.

A hierarquia de descritores é a seguinte, ilustrada na figura 2.2:

- *device descriptor*: existe apenas um por dispositivo. Contém informações globais, como a versão da especificação USB suportada, os identificadores de fornecedor e produto (`Vendor ID` e `Product ID`), e o número de configurações possíveis;
- *configuration descriptor*: um dispositivo pode ter múltiplas configurações, mas apenas uma pode estar ativa por vez. Este descritor especifica características de uma configuração, como o consumo de energia e o número de interfaces que ela agrupa. A seleção é feita pelo host com o comando `SetConfiguration`;
- *interface descriptor*: agrupa um conjunto de endpoints que implementam uma funcionalidade específica (e.g.: uma webcam pode ter uma interface para vídeo e outra para áudio). As interfaces permitem configurações alternativas (`bAlternateSetting`),

que podem ser trocadas em tempo de execução com o comando `SetInterface` para, por exemplo, alterar a largura de banda alocada;

- *endpoint descriptor*: descreve um único *endpoint*, que é um *buffer* de dados unidirecional (`IN` ou `OUT`). Define atributos cruciais como o tipo de transferência (*control*, *isochronous*, *bulk*, ou *interrupt*), o tamanho máximo do pacote de dados e, para *endpoints* isócronos e de interrupção, o intervalo de *polling*;

- *string descriptors*: opcionais, fornecem informações legíveis por humanos, como o nome do fabricante, o nome do produto e o número de série. Os outros descritores referenciam essas *strings* através de um índice.

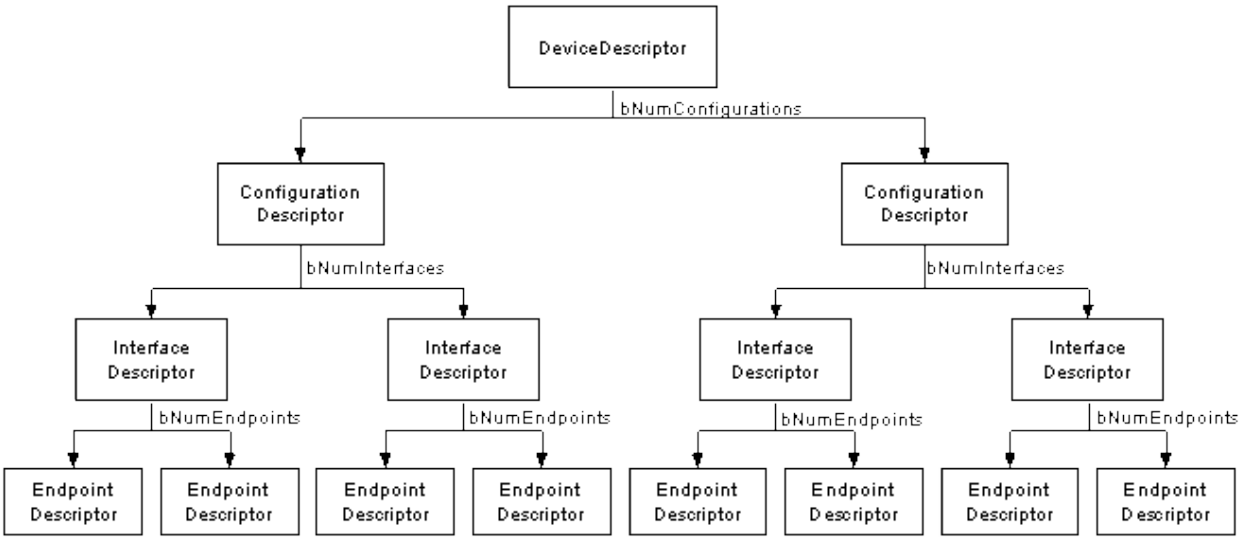

Figura 2.2: *Hierarquia de descritores USB. Retirada de* BeyondLogic, 2025

Todos tipos de descritores tem um formato binário rígido, com campos de tamanho fixo e posições predefinidas, conforme detalhado na seção 9.6 da especificação USB 2.0 USB Implementers Forum, 2025b. Um padrão entre todos descritores é o primeiro *byte*, que indica o tamanho total do descritor em *bytes*, e o segundo *byte*, que identifica o tipo do descritor (e.g.: $0x01$ para *device descriptor*, $0x02$ para *configuration descriptor*, etc.). Os demais campos variam conforme o tipo do descritor, mas seguem uma estrutura consistente que facilita a análise e interpretação pelo *host*. A tabela 2.1 exemplifica a estrutura do device descriptor.

Offset	Campo	Tamanho	Valor	Descrição
0	`bLength`	1	18	Tamanho do descritor em bytes
1	`bDescriptorType`	1	1	Tipo do descritor (device)
2	`bcdUSB`	2	BCD	Especificação com a qual o dispositivo é compatível, em *Binary-Coded Decimal* (BCD)
4	`bDeviceClass`	1	Class	Código de classe definido pela USB-IF. Exemplos de classe são *mass storage* ou áudio. Valores $0x00$ e $0xFF$ são casos especiais: definido pela interface e pelo fornecedor, respectivamente
5	`bDeviceSubClass`	1	SubClass	Código de subclasse definido pela USB-IF. Deve ser $0x00$ caso `bDeviceClass` também seja nulo, e $0xFF$ caso a subclasse seja indefinida
6	`bDeviceProtocol`	1	Protocol	Especifica o protocolo de comunicação usado pelo dispositivo. Os valores $0x00$ e $0xFF$ indicam que não há protocolo específico de classe, e protocolo específico do fornecedor, respectivamente
7	`bMaxPacketSize0`	1	Número	Tamanho máximo do pacote suportado pelo *endpoint* 0
8	`idVendor`	2	ID	`Vendor ID` atribuído pela USB-IF
10	`idProduct`	2	ID	`Product ID` atribuído pelo fornecedor
12	`bcdDevice`	2	BCD	Número de versão definido pelo desenvolvedor do dispositivo, em BCD
14	`iManufacturer`	1	Índice	Índice do *string descriptor* que descreve o fabricante
15	`iProduct`	1	Índice	Índice do *string descriptor* que descreve o produto
16	`iSerialNumber`	1	Índice	Índice do *string descriptor* que descreve o número serial
17	`bNumConfigurations`	1	Número	Número de possíveis configurações do dispositivo

Tabela 2.1: *Estrutura do Device Descriptor USB. Adaptada de USB Implementers Forum, 2025b*

Após a leitura dos descritores, o *host* seleciona uma configuração, carrega o *driver* apropriado com base nas informações de `Vendor/Product ID` (VID/PID) e classe do dispositivo, e o dispositivo está pronto para ser utilizado.

2.4 Captura de tráfego

A captura de tráfego USB em um sistema pode ser realizada em diferentes níveis – *hardware*, *kernel*, espaço de usuário — dependendo dos objetivos e do acesso disponível.

No nível de hardware existem existem analisadores de protocolo (*protocol analyzers*) e analisadores lógicos (*logic analyzers*) que interceptam e encaminham sinais elétricos, permitindo a reconstrução de tráfego com alta fidelidade temporal; é a única forma de observar falhas que decorrem de condições físicas.

Também é possível fazer essa captura no nível de *software*, em duas camadas principais: no espaço do kernel ou no espaço do usuário. A abordagem no espaço do usuário é muito limitada, dado que essa técnica geralmente se baseia em "interceptar"(*hooking*) chamadas de API de alto nível. Um exemplo no espaço do usuário é capturar chamadas ao método `libusb_bulk_transfer` da libusb, todavia isso é restrito a programas aos quais se tem controle.

A abordagem em nível de *kernel* é muito mais comum e robusta, tipicamente utilizando o módulo `usbmon`. Ele opera como um *sniffer* de protocolo, interceptando os *USB Request Blocks* (URBs) – estruturas de dados do *kernel* que encapsulam todas as informações necessárias das transações USB Corbet *et al.*, 2005. O `usbmon` se posiciona entre os *drivers* de dispositivo e os *drivers* da controladora *host*, fornecendo uma visão completa e de baixo nível de toda a comunicação USB do sistema. Ferramentas do espaço de usuário, como o `Wireshark` consomem os dados do `usbmon` para decodificar esse tráfego em um formato legível. Uma solução análoga em sistemas Windows é o driver de filtro chamado `USBPcap`, que cumpre a mesma função que o `usbmon`.

2.5 Abstrações de Software para Interação USB

Embora a comunicação em baixo nível seja gerenciada pelo *kernel* e seus *drivers* de controladora (*host controllers*), a interação de aplicações de espaço de usuário (*userspace*) com dispositivos USB é, na maioria dos casos, mediada por bibliotecas de abstração. Essas bibliotecas facilitam a criação de *drivers* customizados, ferramentas de análise e interação direta com dispositivos USB sem a necessidade de desenvolver código de baixo nível.

A biblioteca mais fundamental neste ecossistema é a `libusb` (libusb Developers, 2025). Trata-se de uma API em C, de código aberto e multiplataforma (Linux, macOS, Windows), que fornece acesso de baixo nível a dispositivos USB a partir do espaço de usuário. Sua principal função é permitir que uma aplicação assuma o controle direto de uma interface USB, sem passar pelos *drivers* de classe do *kernel* (como `usb-storage` ou `usbhid`). Quando o sistema operacional já possui um *driver* associado à interface, a aplicação pode solicitar que a `libusb` desvincule esse driver (`libusb_detach_kernel_driver`), e então reivindicar a interface (`libusb_claim_interface`). Uma vez que a interface está sob controle da aplicação, a `libusb` expõe funções para comunicação direta com seus *endpoints*, permitindo o envio e recebimento de transferências de controle, lote, interrupção e isócronas.

Sobre esta base, outras bibliotecas de nível superior foram criadas. O `PyUSB` e o `usb4java`, por exemplo, são *wrappers* em Python e Java, respectivamente, para a `libusb`, que oferece a mesma funcionalidade de baixo nível, mas com a conveniência e agilidade de prototipagem das linguagens de mais alto nível. Também em um nível de abstração mais alto, existe a `HIDAPI`. Esta biblioteca foca exclusivamente em dispositivos da classe *Human Interface Device* (HID) e abstrai completamente os detalhes do protocolo USB. A `HIDAPI`

permite que uma aplicação leia e escreva relatórios HID sem precisar lidar com endpoints, descritores ou tipos de transferência, fornecendo uma API mais simples e orientada ao uso típico de teclados, mouses, controles e sensores HID.

Capítulo 3

Fuzzing

fuzzing é uma técnica de teste automatizado em softwares por meio do fornecimento massivo de entradas inválidas, inesperadas ou aleatórias. A ideia central é avaliar como o programa se comporta diante dessas entradas, identificando anomalias que indiquem falhas de robustez ou de segurança. Tais anomalias podem se manifestar na forma de travamentos (*crashes*), execuções não terminantes (*loops* infinitos), uso excessivo de recursos computacionais, ou vulnerabilidades clássicas de memória, como *buffer overflows* e *use-after-free* (UAF). Embora seja uma ténica antiga, inicialmente descrita em Miller *et al.*, 1990, que cunhou o termo *fuzzing* após o programa chamado `fuzz`, a técnica evoluiu para métodos mais inteligentes e guiados por métricas internas do programa.

Os tipos de *fuzzing* podem ser classificados segundo o nível de conhecimento do alvo e o modo de geração de entradas. No que diz respeito à literatura, *fuzzing* apresenta uma taxonomia consolidada, na qual os principais trabalhos utilizam terminologias e categorias bastante uniformes para descrever estratégias, arquiteturas e técnicas adotadas pelos *fuzzers*. Um dos estudos mais abrangentes nesse sentido é Manès *et al.*, 2018, no qual a classificação dada a seguir segue em conformidade.

Em um *fuzzing black-box*, o sistema sob teste é tratado como uma caixa-preta: o *fuzzer* tem acesso somente a entradas e saídas, limitando-se a observar o que lhe é fornecido, por canais diretos ou laterais. Essa limitação muitas vezes resulta em uma exploração menos eficiente, já que o *fuzzer* não tem informações sobre a estrutura interna do programa, dificultando a correlação entre a entrada fornecida e o estado interno atingido pelo sistema. Uma estratégia de *fuzzing white-box* foi inicialmente apresentada pela Microsoft em Godefroid *et al.*, 2012, utilizando instrumentação, execução simbólica e informações internas do sistema, permitindo uma exploração mais direcionada, com melhor cobertura de código e melhor detecção de falhas.

Entre esses extremos, há ainda uma abordagem *grey-box*, que combina a eficiência dos simples testes aleatórios com a inteligência da instrumentação parcial do programa. Tanto as estratégias *black-box* quanto *grey-box* possibilitam um *fuzzing* guiado por cobertura de código (*coverage-guided*), onde a geração de entradas é orientada por métricas de cobertura: entradas que exercitam novos trechos de código recebem "energia"para serem mutadas e exploradas ainda mais. Esse conceito de energia é explorado em Böhme *et al.*, 2016,

que mostra como a exercitação de novos caminhos de código pode ser priorizada com base em um modelo cadeia de Markov, distinguindo trechos de código menos comumente alcançados.

Preliminarmente, outro componente essencial do processo de *fuzzing* é o *corpus*, o conjunto de entradas que serve de ponto de partida para as mutações. Um *corpus* bem projetado cobre a maior variedade possível de caminhos de execução, aumentando a eficiência do teste. O *fuzzer* gera novas entradas aplicando mutações sobre o *corpus*: inserção, remoção, substituição de bytes, recombinação de partes de diferentes amostras ou até transformações estruturais específicas do formato de dados em teste. Essas mutações buscam produzir pequenas variações que levem o programa a novos estados de execução.

Em termos de geração de dados, existem duas famílias principais e conceitualmente distintas: *mutational* e *generational*. *fuzzers* mutacionais partem de um *corpus* existente e aplicam mutações sintáticas ou semânticas para produzir variantes, enquanto *fuzzers* generacionais criam entradas do zero, baseando-se em modelos ou gramáticas que descrevem a estrutura esperada dos dados. Cada abordagem tem suas vantagens e desvantagens: *fuzzers* mutacionais são mais simples de implementar e podem ser eficazes quando o corpus inicial é representativo, mas enfrentam dificuldade em explorar profundamente formatos complexos. *fuzzers* generacionais, por outro lado, podem alcançar uma cobertura mais ampla de formatos específicos, mas exigem um conhecimento prévio detalhado sobre a estrutura dos dados e mutações estruturais podem ser mais complexas. Na prática, ambos métodos podem ser aplicados de forma complementar, inicialmente complementando o corpus com entradas geracionais e adições implementadas durante a execução por meio de mutações.

Finalmente, após encontrados resultados considerados interessantes com a execução do *fuzzer*, há o processo de minimização, também conhecido como delta debugging, para reduzir entradas interessantes até a menor forma que ainda reproduz o comportamento alvo – a entrada utilizada pelo *fuzzer* tende a não ser "minimal", dificultando uma possível posterior depuração. Com os resultados, e quando possível, inicia-se o fluxo de pós-processamento: agrupar e deduplicar bugs equivalentes (por *stack trace*, cobertura, instrumentação como `KASAN` ou `KMSAN`) para obter um diagnóstico e gerar um relatório "maximal"reproduzível; esses passos transformam ruído bruto de crashes em bugs acionáveis e úteis para correção.

3.1 Trabalhos Relacionados

Ao longo dos anos, diversos trabalhos apresentaram implementações e novas estratégias para ferramentas de *fuzzing*, cada uma com suas características, pontos fortes e limitações.

O *American Fuzzy Lop* (AFL), lançado em 2013 e continuado através do projeto AFL++ (Fioraldi *et al.*, 2020), é indiscutivelmente o *fuzzer* que popularizou o *fuzzing* guiado por cobertura (*coverage-guided*). Sua principal inovação foi o uso de uma instrumentação leve para monitorar quais caminhos de código uma entrada exercita. Com essa informação, o AFL emprega um algoritmo genético para priorizar e mutar as entradas do corpus que descobrem novos caminhos, permitindo-lhe explorar eficientemente a lógica interna de

programas complexos sem a necessidade de análise de código-fonte. Dada sua eficácia e simplicidade de uso, o AFL foi responsável pela descoberta de múltiplas vulnerabilidades em *softwares* importantes (Zalewski, 2025) e seu design se tornou a base para inúmeras ferramentas subsequentes e pesquisas acadêmicas na área (Manès *et al.*, 2018). As mutações empregadas pelo AFL incluem duas categorias principais: determinística e *havoc*. Dentre as mutações determinísticas, destacam-se *bit flips* e substituições por valores interessantes, como *edge cases* de inteiros ($0, 1, -1, 2^8 - 1, ...$). Mutações *havoc* são utilizadas após esgotar o modo determinístico, com mutações aleatórias empilhadas (*stacked mutations*).

Radamsa (Helin, 2025) é um projeto que se auto descreve como um gerador de casos de teste para testes de robustez. Ele é projetado para ser simples, rápido e eficaz na geração de entradas mutadas a partir de um *corpus* inicial, e operar completamente às cegas. Como suporta e aplica mutações a qualquer tipo de entrada, é frequentemente utilizado em conjunto com outras ferramentas de *fuzzing*, como o previamente mencionado AFL++, para aumentar a diversidade das entradas testadas e melhorar a cobertura do código. A ausência de qualquer tipo de integração com instrumentações inviabiliza uma análise mais profunda do desempenho do mesmo, como já explorado em Pramanik e Tayade, 2019.

Outro importante *fuzzer* com muitos *bugs* encontrados é o `syzkaller`, software de *fuzzing* desenvolvido pelo Google, focado em encontrar vulnerabilidades no *kernel* Linux. O syzkaller é um *fuzzer* guiado por cobertura sem supervisão, instrumentado principalmente pelo `kcov` para análise de cobertura de código e *sanitizers* do *kernel* como `KASAN` e `KMSAN` para detecção de erros. O suporte ao subsistema USB se dá através do módulo do *kernel* `raw-gadget`, apresentado em Konovalov, 2019. Um dos pontos interessantes do `syzkaller` é seu repositório público com milhares de *bugs* encontrados no kernel Linux, centralizados através do `syzbot` (Source, 2025). Zou *et al.*, 2022 mostra que muitos dos bugs de baixa severidade encontrados podem ser escalados para um impacto de risco maior.

O trabalho **ramsauer2019black-box** propõe uma abordagem de *fuzzing black-box* utilizada para avaliar a segurança do protocolo de rede MQTT. O processo de geração de entradas empregado é simples e utiliza apenas o `radamsa` para mutação de entradas. Uma abordagem interessante, por outro lado, é uma medida de contorno para o cenário `black-box`: na ausência de instrumentação direta do sistema alvo, o trabalho utiliza métricas externas para determinar se o sistema conectado está ativo, permitindo a identificação de entradas anômalas que causam falhas.

Dentre as muitas implementações de *fuzzers* existentes, Chen *et al.*, 2018 busca otimizar o processo de *fuzzing* ao introduzir uma abordagem de "*fuzzing* conjunto", onde múltiplos *fuzzers* são utilizados com sincronização do *corpus* entre eles. A ideia é aproveitar as forças individuais de cada *fuzzer* para explorar diferentes partes do espaço de entrada, aumentando a probabilidade de encontrar *bugs*. Essa abordagem é particularmente eficaz quando os *fuzzer*s envolvidos possuem estratégias de mutação e cobertura distintas, permitindo uma exploração mais abrangente do programa alvo. O problema, todavia, é quando a instrumentação no objeto de teste é limitada, dado que isso inviabiliza uma avaliação do desempenho de cada *fuzzer* para seleção e priorização.

Visando o escopo do presente trabalho, Schumilo *et al.*, 2014 apresenta, pela primeira

vez, uma abordagem de *fuzzing* de alta performance para drivers USB, chamada de `vUSBf`. O `vUSBf` define um conjunto de casos de teste e os pacotes são manipulados utilizando `scapy`. A execução dos testes ocorre com a virtualização paralela de múltiplos hosts utilizando `KVM` e `QEMU`, e os erros são detectados por meio de monitoramento de *logs* do kernel, que deve estar configurado para máxima verbosidade. Trabalhos antecessores, como Tonder e Engelbrecht, 2014 continham a necessidade de um dispositivo físico (usualmente `Facedancer`) para emular o dispositivo USB, ao mesmo tempo em que alguma instrumentação do sistema alvo era necessária.

3.2 Emulação de Dispositivos USB

Uma das tarefas que antecede a construção de um *fuzzer* é a escolha de como o dispositivo USB será emulado, de forma que pacotes possam ser enviados para dispositivos externos ao *fuzzer*. No caso de *fuzzers* usando sistemas embarcados baseados em FPGA, por exemplo, esse controle fica a cargo da implementação do hardware. Por conveniência e flexibilidade, a maior parte dos *fuzzers* USB utiliza o *kernel* Linux em dispositivos com suporte a USB OTG – que permite o uso do modo *gadget*.

Funções cruciais para o *fuzzer*, como um controle mais direto do hardware, são limitadas ao *kernelspace*. Embora seja tecnicamente possível codificar o *fuzzer* inteiro no espaço do *kernel*, isso certamente não é uma boa ideia devido ao risco de *bugs* que podem causar *kernel panic*, por exemplo. Então surge a necessidade de utilizar um módulo do *kernel*, que serve como uma ponte entre o *userspace* e *kernelspace* para utilizar funções restritas ao *kernel*. Há diversas opções de módulos do *kernel* Linux que podem ser utilizados para assumir o modo *gadget* e enviar pacotes manipulados. Dentre as principais opções existentes, destacam-se `FunctionFS`, `GadgetFS` e `raw-gadget`.

Cada uma dessas opções apresenta características distintas que influenciam diretamente na implementação e eficácia do *fuzzer*. A escolha entre essas opções deve considerar fatores como facilidade de uso, flexibilidade e desempenho.

O módulo escolhido para o desenvolvimento do *fuzzer* é o `raw-gadget`, que funciona com *bind* direto a controladores UDC e fornece uma interface de nível muito baixo que encaminha as requisições USB para o *userspace* com verificações mínimas, o que permite um controle mais fino dos dados. É um módulo que foi desenvolvido visando *fuzzing*, mas mesmo assim apresenta algumas sanitizações que limitam a reprodução de alguns *bugs*.

Uma das poucas limitações do `raw-gadget`, por exemplo, é a checagem de comprimento máximo de transferências. *Bugs* que levam a vazamento de memória, por exemplo, podem causar uma transferência de dados com tamanho inesperado. Limitar o tamanho máximo de transferências pode impedir a reprodução de tais *bugs*. Esse cenário já apareceu em uma lista de discussão do *kernel* e um *patch* foi enviado pelo mantenedor, após a vulnerabilidade `CVE-2025-38494` ser reportada e não ser efetivamente reproduzível com o `raw-gadget`. Konovalov, 2025b

Em contraste, `FunctionFS` foi pensado para montar funções padrões e válidas dentro de um gadget. Justamente por isso, oferece mais segurança, coerência e previsibilidade

para ambientes de produção. O `GadgetFS`, por sua vez, é um módulo legado no qual o `FunctionFS` foi baseado; também com limitações similares ao `FunctionFS` em termos de flexibilidade e controle, o que é um obstáculo para o desenvolvimento de um *fuzzer* eficaz.

Programa 3.1 linux/v6.17.1/source/drivers/usb/gadget/function/f_fs.c: validação de descritores.

```
1   switch (_ds->bDescriptorType) {
2   [...]
3   case USB_DT_INTERFACE: {
4      struct usb_interface_descriptor *ds = (void *)_ds;
5      pr_vdebug("interface descriptor\n");
6      if (length != sizeof *ds)
7         goto inv_length;
8
9      __entity(INTERFACE, ds->bInterfaceNumber);
10     if (ds->iInterface)
11        __entity(STRING, ds->iInterface);
12     *current_class = ds->bInterfaceClass;
13     *current_subclass = ds->bInterfaceSubClass;
14  }
15     break;
16  [...]
17  default:
18     /* We should never be here */
19     pr_vdebug("unknown descriptor: %d\n", _ds->bDescriptorType);
20     return -EINVAL;
```

O trecho de código 3.1 mostra um impeditivo do `FunctionFS`: um tratamento e validação de descritores. Por outro lado, nesse mesmo cenário, `raw-gadget` não faz nenhuma validação ou interpretação dessa estrutura. Ele a trata como um bloco de dados opaco e a envia diretamente para o aplicativo de espaço do usuário, que então tem a responsabilidade total de interpretá-la e responder adequadamente.

3.3 Análise de Vulnerabilidades

Vulnerabilidades relacionadas a USB podem surgir tanto por bugs de implementações de *drivers* no *host* quanto por falhas na camada de aplicação. No nível do *kernel*, vulnerabilidades geralmente se manifestam como corrupção de memória, leitura no *kernelspace* fora dos limites, negação de serviço, ou execução espúria de código. Na camada de aplicação, falhas geralmente exigem maior conhecimento do contexto da aplicação e podem se apresentar de diversas formas, como mau tratamento de *format strings* em descritores de strings ou condições de corrida (TOCTOU).

Sergey Bratus, 2012 apresenta um bug encontrado, em 2012, no Skype: ao definir alguns descritores de string como `"%n%s%n%s%n%s"`, a aplicação fechava inesperadamente. O problema provavelmente residia em um uso de funções como `printf` sem a devida sanitização, levando a uma falha potencialmente explorável.

Davis, 2013 define negação de serviço (DoS), através de *use-after-free* (UAF) ou desreferência de ponteiros nulos, como uma das mais comuns falhas encontradas, e essas são falhas facilmente encontradas através de *fuzzing*. Konovalov, 2017 lista inúmeros exemplos de *bugs* do tipo encontrados pelo syzkaller, como `CVE-2017-16525`.

3.4 Métricas Externas

Um desafio inerente ao *fuzzing black-box* é a observabilidade do sistema, que dificulta a identificação de falhas. Faz-se, então, necessário elaborar oráculos de detecção de *bugs* que possam ser empregadas para detectar diferenciais entre casos de teste, utilizando métricas externas para guiar o processo de *fuzzing*. Operando às cegas, faz sentido combinar diversas métricas para inferir o estado do sistema alvo e utilizar uma avaliação de curto-circuito, dado que não há como determinar precisamente o comportamento do sistema e qualquer anomalia pode ser considerada um bug potencial.

Uma das técnicas para a detecção de diferencial no sistema alvo é a disponibilidade do processo de enumeração USB. Dentre os códigos do projeto, há renumerator.c, baseado no módulo raw gadget que, simplificadamente:

1. **em *loop*:**
2. inicia USB no modo *device*, com `USB_RAW_IOCTL_INIT` e `USB_RAW_IOCTL_RUN`;
3. processa pacotes do tipo `USB_RAW_EVENT_CONNECT`;
4. responde com descritores e configurações de `vendor ID` $0x05e3$ e `product ID` $0xfe00$, simulando ser um mouse da marca Razer;
5. desconecta o *device*.

O objetivo é simples: forçar um processo infinito de enumeração por parte do *host*, verificando a hipótese do *host*, por medidas de segurança, não tratar mais USB a partir de certo ponto, e servindo de base para o proceso de *fuzzing*.

Um outro exemplo de métrica externa que pode indicar uma negação de serviço é a falta de polling por lado do *host*. Se o device conectado tem um endpoint que requer *polling* e o *host* deixa de enviar pacotes de *polling*, isso pode indicar que o *device* enviou um pacote inválido e o *host* passou a ignorá-lo, ou que o *host* travou; em ambos casos, é um resultado anômalo interessante.

Capítulo 4

Desenvolvimento da ferramenta

Com base no capítulo 3, o presente trabalho propõe a construção de uma ferramenta de *fuzzing* USB para ser usado em testes do tipo *black-box*. Visando uma maior acurácia nos testes, a ferramenta também integra e combina resultados advindos de *fuzzing* em outros sistemas, permitindo uma rápida identificação de falhas presentes no *kernel* de sistemas operacionais comuns, por exemplo.

A ferramenta é composta por dois componentes principais: um dispositivo USB programável, responsável por emular dispositivos USB e enviar pacotes malformados; e um *software* executor, responsável por controlar o dispositivo USB, estruturar os pacotes a serem enviados e monitorar o sistema alvo em busca de falhas.

Embora a observabilidade dos resultados com os testes sendo feitos de maneira completamente fechada seja bastante comprometida, os testes de *fuzzing* necessitam de um oráculo de *bugs*, similar a um sistema de *feedback*, por dois simples motivos: é necessário identificar se a máquina chegou a algum estado, de alguma forma, considerado inesperado; é necessário saber qual conjunto de pacotes levou a máquina a tal estado.

Previamente, na seção 3.4, foi apresentado um script chamado `renumerator.c`, responsável por verificar se o processo de enumeração USB está ocorrendo normalmente. Uma versão modificada desse *script* será utilizada adiante como forma de oráculo. O processo é simples: após cada caso de teste, e um pequeno tempo de espera, o oráculo é acionado. O sucesso da enumeração tende a significar falha do teste, e esse processo aparenta cobrir a maior parte das possíveis falhas, como *crashes*. Outro método utilizado para detecção de leituras indevidas de memória no espaço do *kernel* será demonstrado, a partir da análise da saída de resposta a pacotes USB específicos. Alguns comportamentos inesperados, como um erro que leve a um menu desconhecido ou uma mensagem erro, também são uma forma de *feedback* valiosa, embora mais difícil de ser automatizada.

4.1 Escolha e configuração do *hardware*

A escolha do *hardware* é um ponto crucial para o desenvolvimento da ferramenta. É necessário que o dispositivo USB programável possua suporte a modo *gadget*, permitindo

que ele atue como um dispositivo USB completo, capaz de se comunicar com o *host* e enviar pacotes malformados conforme necessário. O suporte a Linux é outro ponto importante, visto que a implementação do *software* executor é muito mais simples nesse ambiente devido a existência de módulos do *kernel* que permitem manipulação de pacotes USB de forma direta. O custo e a disponibilidade do *hardware* também forem fatores a serem considerados, visando a viabilidade prática do projeto. A tabela 4.1 apresenta uma comparação entre as principais opções de *Single Board Computers* (SBCs) e microcontroladores, destacando as características relevantes para o projeto.

Nome	Linux	USB (modo host)	USB (modo gadget)
`ESP32-S3`	Muito limitado	0	1x USB 2.0
`Raspberry Pi Zero 2W`	Suportado	1x USB 2.0	1x USB 2.0
`Raspberry Pi 5`	Suportado	2x USB 2.0, 2x USB 3.0	1x USB 2.0
`BeagleBone Black`	Suportado	1x USB 2.0	1x USB 2.0

Tabela 4.1: *Comparativo entre SBCs e microcontroladores populares em relação ao suporte a Linux e USB*

Field Programmable Gate Arrays (FPGAs) foram uma opção considerada, mas o custo mais elevado e a complexidade de desenvolvimento tornaram essa alternativa inviável para o escopo do projeto. Microcontroladores como o `ESP32-S3` possuem suporte limitado a Linux, o que dificulta a implementação do *software* executor. `BleagleBone Black` e `Raspberry Pi 5` são opções viáveis, mas o `Raspberry Pi Zero 2W` tem um menor custo, é amplamente disponível e não apresenta limitações significativas em relação aos demais. Portanto, o `Raspberry Pi Zero 2W` foi escolhido como a plataforma ideal para o desenvolvimento da ferramenta.

Nota-se que, infelizmente, nenhuma opção considerada viável possui suporte nativo a USB 3.0 no modo *gadget*. Embora o USB 2.0 seja suficiente para a maioria dos testes de *fuzzing*, essa limitação pode impactar a capacidade de explorar vulnerabilidades específicas relacionadas ao USB 3.0.

O *setup* do `Raspberry Pi Zero 2W` tem alguns detalhes importantes, que impactam diretamente a configuração das portas e comunicação USB. Um dos problemas encontrados durante a configuração é que o módulo do *kernel* utilizado pelo *fuzzer*, `raw-gadget` (Konovalov, 2025a) não funciona com a versão 6.12 do *kernel* Linux, utilizada, por padrão, pelo `Raspberry Pi Imager`. A solução de contorno encontrada é fazer o uso de uma versão do `Raspberry Pi OS Bullseye`, com o *kernel* Linux na versão 5.20 fornecida em R. P. Foundation, 2023.

Após a instalação do sistema operacional, a configuração inicial pode ser feito com o *script* a seguir:

```
1    apt-get update -y
2    apt-get dist-upgrade -y
3    apt-get install -y raspberrypi-kernel-headers
4    printf "\ndtoverlay=dwc2" | sudo tee -a /boot/config.txt
5    printf "\ndtoverlay=disable-bt" | sudo tee -a /boot/config.txt
6    printf "\ndwc2" | sudo tee -a /etc/modules
```

```
7    git clone https://github.com/xairy/raw-gadget
8    reboot
```

setup.sh

Por padrão, o Raspberry tentará se conectar a rede Wi-Fi, se definida pelo imager. Isso traz alguns problemas, como o acesso ao Raspberry pela rede. Pode-se desabilitar as configurações wireless alterando a configuração de boot: `printf "ndtoverlay=disable-bt sudo tee -a /boot/config.txt`. Sem a configuração de rede, uma alternativa é fazer a conexão ao Raspberry via UART e obter um terminal serial. Com o terminal serial, também é possível ativar PPP sobre UART para criar uma interface de rede virtual, que será útil a seguir.

A instalação do `raw-gadget` deve ser feita após reiniciar o sistema com o *script* acima, para garantir que o SBC esteja no modo *gadget*. O processo de instalação do `raw-gadget` pode ser simplesmente feito com os comandos a seguir:

```
1    cd raw-gadget/raw_gadget
2    sudo make
3    sudo ./insmod.sh
```

raw-gadget_install.sh

Nesse ponto, o módulo `raw-gadget` já está ativo, com a placa operando em modo *gadget*. O nome do *driver* UDC estará listado em `/sys/class/udc/`. Resta configurar o `syzkaller`, que será utilizado pela integração com o Source, 2025. A primeira etapa é clonar o repositório, adaptar para o Raspbery Pi e executar o processo de *build*. Como o processo é lento, deve ser feito fora do Raspberry Pi, com compilação cruzada, e os arquivos copiados para a placa posteriormente. O script a seguir faz a compilação do `syzkaller` para a arquitetura ARM:

```
1    git clone https://github.com/google/syzkaller
2    alias syz-env="${PWD}/syzkaller/tools/syz-env"
3    cd syzkaller
4    DRIVER_NAME=# Resultado de $(ls /sys/class/udc/) dentro da placa
5    perl -0777 -i -pe 's/char device\[32\];\s*sprintf\(&device\[0\], "dummy_udc
         \.%llu", procid\);\s*int rv = usb_raw_init\(fd, speed, "dummy_udc", &
         device\[0\]\);/int rv = usb_raw_init(fd, speed, "${DRIVER_NAME}", "${
         DRIVER_NAME}");/g' executor/common_usb_linux.h
6    syz-env make generate
7    syz-env GOARM=5 make TARGETARCH=arm execprog
8    syz-env make TARGETARCH=arm executor
```

syzkaller_setup.sh

Com os arquivos copiados para a placa, o `syz-executor` já pode ser usado. Um único problema impede o uso do `syz-executor` e a integração com o `syzbot`: não há seeds carregadas.

4.2 Crawler do syz-bot

A fim de contornar o problema de ausência de *seeds* após a configuração inicial da placa, uma opção é simplesmente baixar todas *seeds* disponibilizadas pelo Source, 2025. O

script abaixo faz o *scraping* de todos *bugs* encontrados, com *logs* para a reprodução via `syzkaller`. Isso é facilitado pela interface *web* pública, com todas informações necessárias e sem restrições de acesso para *bots*.

```
import requests
from bs4 import BeautifulSoup
import hashlib
import time
import os

BASE_URL = "https://syzkaller.appspot.com"
SYZ_REPRO_LOG_PATH = "syz-repro-seeds"

bug_links = []

if not os.path.exists(SYZ_REPRO_LOG_PATH):
   os.makedirs(SYZ_REPRO_LOG_PATH)

def get_bug_urls(url):
   response = requests.get(url)
   if response.status_code != 200:
      print(f"{url}: {response}")
      return []

   soup = BeautifulSoup(response.text, "html.parser")
   rows = soup.find_all("tr")

   for row in rows:
      stat_cells = row.find_all("td", class_="stat")
      if not stat_cells:
         continue
      if any(cell.text.strip() in ("C", "syz") for cell in stat_cells):
         title_cell = row.find("td", class_="title")
         if title_cell and title_cell.a:
            bug_url = BASE_URL + title_cell.a['href']
            bug_links.append(bug_url)

def find_first_syz_href(bug_url):
   time.sleep(1) # syzbot is limited to 1 request per second
   response = requests.get(bug_url)
   if response.status_code != 200:
      print(f"{bug_url}: {response}")
      return None

   soup = BeautifulSoup(response.text, "html.parser")
   rows = soup.find_all("tr")
   for row in rows:
      repro_cells = row.find_all("td", class_="repro")
      for cell in repro_cells:
         a_tags = cell.find_all("a")
         for a in a_tags:
            if a.text.strip() == "syz":
               return BASE_URL + a['href']
   return None

def download_syz_repro(syz_url):
```

```
53        time.sleep(1) # syzbot is limited to 1 request per second
54        response = requests.get(syz_url)
55        if response.status_code != 200:
56           print(f"{syz_url}: {response}")
57           return None
58        return response.text
59
60     for bug_list_url in ["/upstream?manager=ci2-upstream-usb", "/upstream/fixed?
           manager=ci2-upstream-usb"]:
61        get_bug_urls(BASE_URL + bug_list_url)
62     print(len(bug_links))
63
64     for link in bug_links:
65        print(link)
66        syz_repro_url = find_first_syz_href(link)
67        if syz_repro_url:
68           print(f"Found syz repro URL: {syz_repro_url}")
69           syz_repro = download_syz_repro(syz_repro_url)
70           filename = hashlib.sha1(syz_repro_url.encode()).hexdigest()[:12]
71           with open(f"{SYZ_REPRO_LOG_PATH}/{filename}.log", "w") as f:
72              f.write(syz_repro)
```

syzbot_scraper.py

Note que nem todas seeds produzem resultados relevantes, e o *parsing* de *seeds* funcionais não é trivial: há *bugs* não reproduzíveis, ou com *syscalls* que não tem relação com USB (e, portanto, não reproduzíveis no *host*), por exemplo; essas *syscalls* podem ou não ter algum efeito no resultado. Para contornar essa limitação, um pequeno filtro foi aplicado a todos arquivos de log, selecionando apenas aqueles que contém chamadas de sistema relativas a USB, e sem chamadas que não são reproduzíveis externamente, como `name_to_handle_at`.

A reprodução de um arquivo de *log*, com pacotes enviados ao *host*, pode ser feita com o comando `sudo syz-execprog -executor ./syz-executor -slowdown=1 -threaded=1 -collide=1 -procs=1 -enable=” -debug <seed.log>`, onde `<seed.log>` é o arquivo de *log* obtido pelo *crawler*.

4.3 *Fuzzing* com syzkaller

Uma das formas de excluir a dependência de *logs* de erros divulgados pelo projeto `syzbot` é fazer o uso do próprio `syzkaller` para realizar *fuzzing* genérico na pilha USB do kernel Linux. Essa abordagem pode revelar vulnerabilidades não previamente conhecidas e aumentar a abrangência dos testes. Para esse propósito, faz-se necessário o uso do `syz-manager`, componente do `syzkaller` responsável por orquestrar o processo de *fuzzing*, com uma configuração específica para esse propósito. Embora isso não aparente ser possível com a configuração padrão, o código que especifica o arquivo de configuração mostra o contrário.

Uma rápida análise de alguns *logs* de erros gerados pelo `syzkaller` com relação ao subsistema USB aponta o uso de pseudochamadas de sistema específicas, usualmente seguindo o formato `syz_usb_*`, como `syz_usb_ep_read`. As chamadas de sistema `open`,

Programa 4.1 syzkaller/pkg/mgrconfig/configgo: trecho do código-fonte do syzkaller que mostra a possibilidade de configuração de syscalls específicas

```
1    // List of syscalls to test (optional). For example:
2    // "enable_syscalls": [ "mmap", "openat$ashmem", "ioctl$ASHMEM*" ]
3    EnabledSyscalls []string `json:"enable_syscalls,omitempty"`
4    // List of system calls that should be treated as disabled (optional).
5    DisabledSyscalls []string `json:"disable_syscalls,omitempty"`
6    // List of syscalls that should not be mutated by the \textit{fuzzer} (
         optional).
```

`close`, `read`, `write`, `ioctl` também são frequentemente utilizadas. Demais configurações, como o *kernel* a ser utilizado pelo *fuzzer* ou aspectos relativos a performance ficam a critério do usuário.

4.4 Estratégias da ferramenta

A ferramenta, sucintamente, não é um único *software* monolítico destinado a realizar *fuzzing* USB. Ela é composta por alguns componentes que a assemelham a um *framework*, permitindo a implementação manual de estratégias de *fuzzing* para testar hipóteses mais direcionadas. Paralelamente, há um componente de *fuzzing* mais genérico, que visa explorar a superfície de ataque de forma mais ampla, a partir de uma captura de tráfego. Em resumo, a ferramenta desenvolvida possui três estratégias principais de *fuzzing*:

- *fuzzing* direcionado a hipóteses específicas, implementadas manualmente;
- *fuzzing* genérico, a partir de mutações aplicadas a um corpus de PCAPs;
- *fuzzing* genérico, guiado por vulnerabilidades conhecidas extraídas de um *fuzzing* instrumentado;
- simples reprodução de bugs conhecidos.

O terceiro caso é baseado em vulnerabilidades encontradas por Konovalov, 2017, com, como previamente citado, *bugs* reportados em Source, 2025. O `syzbot` é uma das escolha bastante promissora, dado seu baixo custo e velocidade de reprodução. Essa estratégia é reduzida, após um *parsing* e reprodução de logs, ao caso do *fuzzing* genérico a partir de mutações aplicadas a um *corpus* de PCAPs, conforme ilustrado em 4.1.

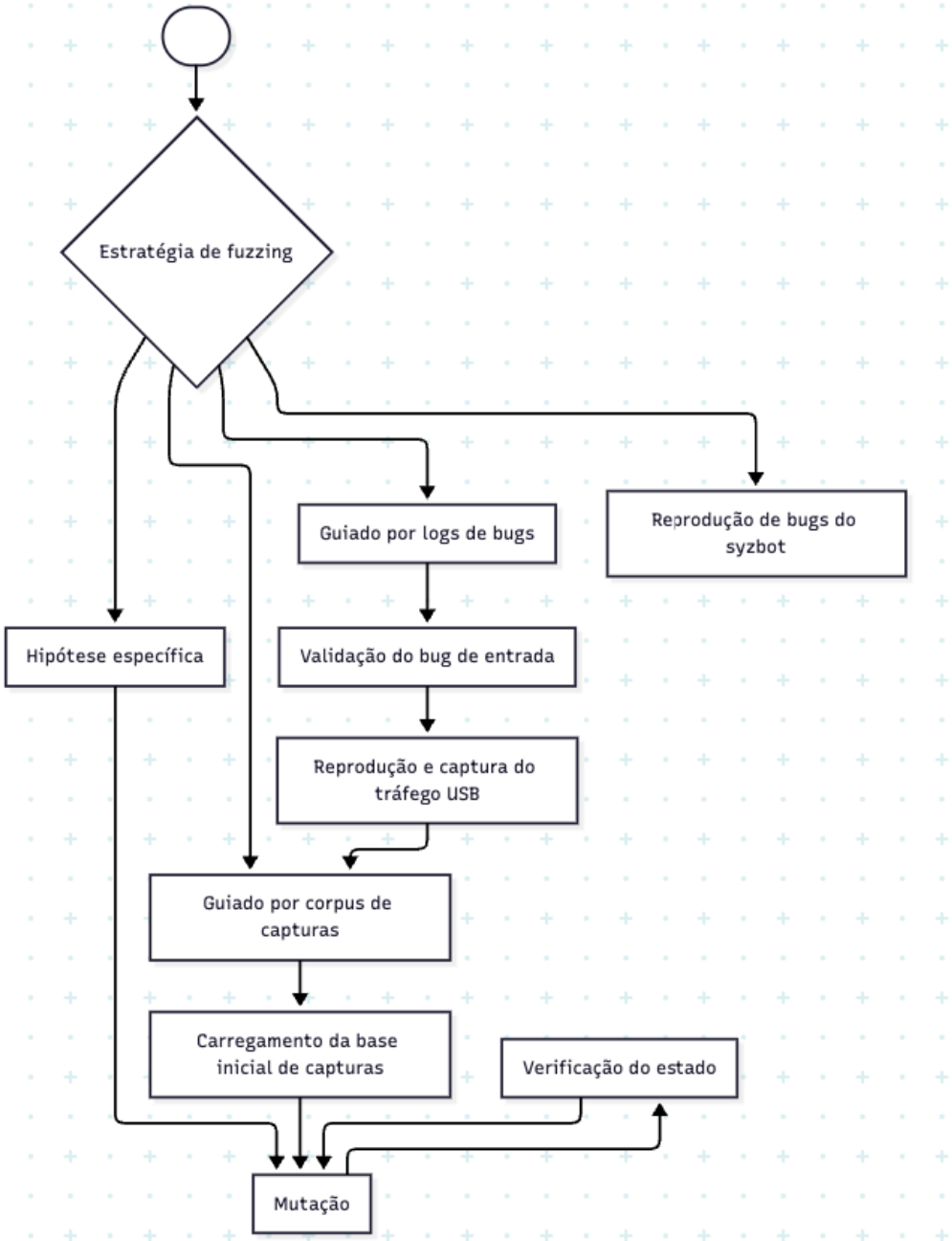

Figura 4.1: *Estratégias de fuzzing empregadas pela ferramenta*

4.5 Base inicial de capturas

O conjunto de capturas utilizado para os testes de *fuzzing* foi construído através de duas fontes principais:

- *logs* do `syzkaller`, obtidos via *crawler* (seção 4.2) do `syzbot` e instância local (4.3);
- tráfego USB capturado em PCAPs, obtidos via coleta local, através de diversos dispositivos como mouses e outros periféricos, e amostras do `Wireshark` (W. Foundation, 2025).

4.6 Validação e reprodução de *bugs* do `syzbot`

Embora os *logs* do `syzkaller` sejam reproduzíveis com ferramentas como o `syz_execprog/syz_executor`, esses *logs* não representam diretamente o tráfego do dispositivo USB em um formato padronizado, como PCAP. O conteúdo do arquivo de *logs* é composto por uma série de pseudochamadas de sistema que são interpretadas pelo executor do `syzkaller`. Embora seja tecnicamente viável fazer o *parsing* dessas chamadas e convertê-las para PCAPs, padronizando as entradas, esse processo é complexo e propenso a erros, devido à natureza abstrata das chamadas. Outro fator que precariza essa ideia é a restrição ao formato de uma implementação específica, utilizada pelo executor, que pode vir a mudar com o tempo.

Programa 4.2 Log gerado pelo syzkaller para o bug “memory leak in __hci_cmd_sync_sk“

```
1   # https://syzkaller.appspot.com/bug?id=4
        a86db64b1c9b392d18c1314c37e9de9facb2e40
2   # See https://goo.gl/kgGztJ for information about syzkaller reproducers.
3   #{"repeat":true,"procs":1,"slowdown":1,"sandbox":"none","sandbox_arg":0,"leak
        ":true,"close_fds":true,"vhci":true,"callcomments":true}
4   syz_usb_connect(0x5, 0x36, 0x0, 0x0)
5   r0 = userfaultfd(0x1)
6   ioctl$UFFDIO_API(r0, 0xc018aa3f, &(0x7f0000000000)={0xaa, 0x4d0})
7   bpf$MAP_CREATE(0x0, &(0x7f0000000000)=ANY=[@ANYBLOB="1700000007"], 0x50)
8   sendto$packet(0xffffffffffffffff, &(0x7f0000000000)='[', 0x1, 0x0, 0x0, 0x0)
9   r1 = syz_init_net_socket$bt_hci(0x1f, 0x3, 0x1)
10  bind$bt_hci(r1, &(0x7f0000000100)={0x1f, 0xffff, 0x3}, 0x6)
11  write$binfmt_misc(r1, &(0x7f0000000000), 0xd)
```

Como essa é uma tarefa que não é executada frequentemente, uma alternativa encontrada foi reproduzir o *log* e capturar o tráfego gerado em PCAPs. Essa abordagem, embora menos direta, garante que o tráfego capturado seja fiel ao comportamento do `syzkaller`, sem a necessidade de interpretar e converter as chamadas manualmente. Outro ponto positivo dessa ideia é a mitigação do problema de *logs* não minimizados: ao converter a captura para PCAP e filtrar por USB, há ao menos a garantia de que todo o *log* é, de fato, referente a USB.

O processo descrito é trivial: basta executar o *log* com o `syz-execprog`, como descrito anteriormente, enquanto uma ferramenta de captura de pacotes USB está ativa, como o `Wireshark`. Uma limitação dessa abordagem é que, ao capturar apenas o tráfego USB, perde-se o contexto das chamadas de sistema que prepararam o ambiente; *bugs* que dependem de interações complexas entre chamadas de sistema, por exemplo, podem não ser reproduzíveis.

4.7 Parsing do tráfego

Os arquivos de captura de tráfego USB, no formato PCAP, precisam ser interpretados e convertidos para estruturas de dados que possam ser manipuladas pela ferramenta de *fuzzing*. Para esse propósito, foi utilizado o framework `Pyshark` (KimiNewt, 2025), que é

um *wrapper* do `tshark`, permitindo a leitura e análise de arquivos PCAP de forma eficiente, com a divisão de campos dos pacotes USB oferecida pelo `Wireshark`.

O *parsing* dos pacotes consiste na extração dos campos relevantes de cada pacote USB, conforme definido da especificação. Para extrair o campo `idVendor` ou `idProduct`, por exemplo, faz-se necessário definir toda a estrutura de um `device descriptor`, iterar por todos pacotes USB da captura e, assim que um pacote desse tipo for encontrado, mapeá-lo para a estrutura definida.

Programa 4.3 Parsing de pacotes USB, utilizando pyshark: extração de device descriptors

```
1   import pyshark
2   import struct
3
4   def parse_device(data):
5      # bLength para device descriptor ésempre 18 bytes
6      if len(data) != 18: return False
7      # campos do device descriptor, conforme spec USB 2.0
8      fields = struct.unpack('<BBHBBBBHHHBBBB', data[:18])
9      # bDescriptorType para device descriptor é1
10     if fields[1] != 1: return False
11     # bMaxPacketSize só pode ser 8, 16, 32 ou 64 para USB 2.0,
12     if fields[6] not in [8, 16, 32, 64]: return False
13     print(f"Device Descriptor:")
14     print(f" bLength: {fields[0]}")
15     print(f" idVendor: 0x{fields[7]:04x} | idProduct: 0x{fields[8]:04x}")
16     return True
17
18  def process_payload(raw_bytes):
19     bLength = raw_bytes[0]
20     bDescriptorType = raw_bytes[1]
21     descriptor_data = raw_bytes[0 : bLength]
22     # faz o parsing do device descriptor, se a estrutura for compatível
23     if bDescriptorType == 1 and bLength == 18:
24        parse_device(descriptor_data)
25
26  def main():
27     capture = pyshark.FileCapture('usb_capture.pcap')
28     for pkt in capture:
29        # filtra apenas pacotes USB, com dados
30        if hasattr(pkt, 'usbll') and hasattr(pkt.usbll, 'data'):
31           hex_str = pkt.usbll.data
32           clean_hex = hex_str.replace(':', '')
33           # filtra pacotes sem dados relevantes
34           if(clean_hex == "Data <none>" or clean_hex == "00"): continue
35           raw_bytes = bytes.fromhex(clean_hex)
36           process_payload(raw_bytes)
37
38  if __name__ == "__main__":
39     main()
```

Dessa forma, todos campos relevantes dos pacotes USB podem ser extraídos e armazenados, para posterior manipulação e mutação pelo *fuzzer*.

Um ponto importante a ser destacado é que o *parsing* demonstrado acima pode abranger alguns erros do tipo falso-positivo, onde um pacote é identificado com um determinado descritor, e na verdade não o é. Isso ocorre devido à natureza dos dados capturados, que podem conter ruídos ou pacotes malformados que seguem a estrutura de determinado tipo. Durante os experimentos realizados, notou-se que esse tipo de erro não impacta significativamente o processo de *fuzzing*. As restrições impostas pelo *parser*, como ao restringir os valores do campo `bMaxPacketSize`, no exemplo acima, ajudam a mitigar esse problema.

4.8 Mutação de pacotes

Seguindo a lógica do *fuzzer*, descrita na seção 4.4, a mutação dos pacotes USB é um passo que sucede o carregamento e o parsing das entradas. Não é inteligente testar todos os possíveis valores para cada campo do pacote, dado que essa tarefa exaustiva é inviável na prática. O espaço de entradas é muito grande – um `configuration descriptor` pode assumir, sem ferir à especificação, qualquer valor em seus 18 *bytes*, um número na ordem de 10^{13} – e o processo de *fuzzing* não é facilmente escalável.

O processo de mutação de pacotes implementado na ferramenta itera sobre cada campo do pacote USB a ser mutado e aplica as seguintes mutações, descritas em Manès *et al.*, 2018 e Pramanik e Tayade, 2019, utilizadas por *fuzzers* como Fioraldi *et al.*, 2020, honggfuzz e **ramsauer2019blackbox**:

- mutação aritmética: incrementa ou decrementa o valor do campo em r, onde r é um inteiro aleatório tal que $1 \leq r \leq 35$;
- baseado em dicionário: valores como *format strings*, úteis para *bugs* como o apontado na seção 3.3, e valores próximos a limites comuns, usados em estratégias de análise de valor limite, como 0, -1 ou 255;
- integração com Helin, 2025;
- sobreescrita com valores de outro campo do mesmo pacote.

Nenhuma técnica de mutação utilizada visa quebrar fortemente a estrutura de um pacote USB. Como o protocolo é fortemente estruturado, mutações que alterem significativamente a estrutura do pacote, como adicionar ou remover muitos *bytes*, dificilmente levarão a resultados úteis, visto que o pacote provavelmente será descartado pelo *host* durante o processamento.

E, similarmente ao que Fioraldi *et al.*, 2020 faz com o seu denominado modo *havoc*, após esgotarem as mutações implementadas acima, o *fuzzer* aplica, exaustivamente, *bit flips*. É difícil de determinar quantos *bits* devem ser alterados, embora Cha *et al.*, 2015 mostre uma estratégia para chegar a um valor ideal, a aplicação é impraticável no contexto do presente *fuzzer*, que simplesmente aplica a mutação de $k \in [1, \lfloor n/2 \rfloor]$ *bits*, onde n é o tamanho da entrada mutável.

4.9 Reprodução dos pacotes

Após a mutação dos pacotes USB, o próximo passo é a reprodução desses pacotes no sistema alvo. A ferramenta desenvolvida utiliza o módulo `raw-gadget` (Konovalov, 2025a), que permite o envio de dados arbitrários diretamente ao *host*. Como USB é fortemente centrado no host, o *fuzzer* atua como um dispositivo USB completo, completando o processo de enumeração como um dispositivo comum, com as devidas mutações aplicadas.

4.10 Interface

Foi implementada uma simples interface web que possibilita a execução das estratégias de *fuzzing* desenvolvidas e o acompanhamento de seus resultados em tempo real. Essa interface foi desenvolvida utilizando o *framework* `Flask`, responsável pela camada de aplicação HTTP, em conjunto com `Socket.IO`, tecnologia que permite comunicação bidirecional em tempo real entre cliente e servidor por meio de `WebSockets`.

Atualmente, o servidor é acessível na porta 5000/TCP, sem mecanismos de autenticação. A interface tem como objetivo fornecer uma visualização mais clara e estruturada dos resultados do processo de *fuzzing*, reduzindo a necessidade de conexões SSH para monitoramento.

Figura 4.2: *Interface web da ferramenta*

Capítulo 5

Experimentos e resultados

Como uma forma de validar a ferramenta desenvolvida, foram realizados experimentos com o objetivo de avaliar sua eficácia na detecção de vulnerabilidades, de forma comparativa. A análise proposta nesta seção parte de bugs já conhecidos, encontrados na pilha USB do kernel Linux por um *fuzzing* guiado por cobertura, ou de outros erros descritos na literatura.

Foram selecionados alguns casos de teste que condizem com a proposta deste trabalho, formuladas hipóteses e geradas as devidas entradas relevantes para o *fuzzer*, a fim de apresentar os resultados e limitações da ferramenta proposta. Os detalhes dos experimentos conduzidos são apresentados nas seções seguintes.

5.1 Negação de serviço

A classe de vulnerabilidades mais fácil de ser testada pela ferramenta proposta são erros de negação de serviço. Um oráculo é facilmente construído para detectar tais falhas, uma vez que o sistema hospedeiro pode ser monitorado para observar se ele permanece responsivo durante o processo de enumeração do dispositivo USB.

Além das diversas falhas desse tipo destacadas em Konovalov, 2017, outros trabalhos como Euntae Jang, 2025 mostram que usualmente, travamentos do *kernel* levam a uma reinicialização do sistema e travamento temporário da enumeração por parte do sistema hospedeiro. Isso valida exatamente o que a implementação do oráculo proposta monitora.

5.2 Vazamento de memória do espaço do kernel

Outra classe de bugs comum é a leitura indevida de memória no espaço do kernel. A seção 3.2 destaca, brevemente, a vulnerabilidade `CVE-2025-38494`. Essa vulnerabilidade é causada por um *bug* de *underflow* aritmético em `drivers/hid/hid-core.c`, componente do *kernel* Linux que fornece funcionalidades básicas para *Human Interface Device* (HID).

Essa vulnerabilidade é explorável através de um dispositivo USB com um descritor de relatório HID malformado. Após a enumeração do dispositivo, o sistema hospedeiro faz uma requisição HID (transferência de controle) do tipo `HID_REQ_SET_REPORT` ao dispositivo USB com um relatório contendo memória do espaço do *kernel*.

A primeira observação a ser feita é que a vulnerabilidade é explorável através do *fuzzer* proposto. A seção 4.4 descreve a capacidade da ferramente de reproduzir bugs encontrados em sistemas com instrumentação disponível, e essa é uma falha publicada por Source, 2025.

Todavia, a simples capacidade de reproduzir o *bug* não é suficiente para validar a eficácia da ferramenta. A reprodução do *bug* simplesmente gera como retorno alguns *kilobytes* de memória; sem distinção da fonte desses dados, não é possível inferir se eles são provenientes de uma vulnerabilidade. Faz-se, então, necessário que a ferramenta seja capaz de reconhecer a vulnerabilidade de forma autônoma.

Um oráculo simples para essa vulnerabilidade, que traz consigo uma maior presença de falsos positivos e falsos negativos, é a observação da saída e uma importante característica do endereçamento canônico (*canonical addressing*) em arquiteturas `x86-64`. Nessa arquitetura, os ponteiros do *kernel* possuem os bits mais significativos iguais ao *bit* 47 (sinal); em binário, 1111 equivale a $0xF$. Uma heurística simples, portanto, é verificar se os dados retornados contêm grupos consecutivos de 8 bytes (tamanho de um ponteiro em `x86-64`) com os bits mais significativos iguais a $0xFFFF$.

Durante os testes da ferramenta, e reprodução de diversos *bugs* de acesso de memória no espaço do *kernel*, outro padrão observado foi a presença do texto `localhost`. A causa raiz dessa ocorrência não foi investigada a fundo. De qualquer forma, esse padrão foi incorporado como uma heurística adicional para detecção de vazamentos de memória.

Essa estratégia de oráculo foi implementada na ferramenta desenvolvida, mas não é utilizada por padrão devido a necessidade de análise e confirmação do resultado, visto que a precisão sobre do oráculo pode ser muito prejudicada.

5.3 *Format strings*

A seção 3.3 apresenta um *bug* encontrado no aplicativo Skype, em Sergey Bratus, 2012. O *bug* em questão é uma vulnerabilidade de *format string*, onde um dispositivo USB malicioso, durante o processo de enumeração, envia uma *string* de formato especialmente formatada para o sistema alvo em seu descritor de *string* apontado pelo campo `iManufacturer`.

Este *bug* é interessante pois, diferentemente dos outros exemplos apresentados neste capítulo, se trata de uma falha na camada de aplicação, em um software de terceiros, de código fechado. A ferramente desenvolvida tem uma limitação notória nesse cenário: além de não ter acesso ao estado do sistema hospedeiro, a informação sobre o estado de um software dentro do sistema alvo também é dificilmente inferida através de um oráculo ou canal lateral, utilizado em outros testes.

Um oráculo depende do nível de observabilidade que a ferramenta tem sobre o sistema

alvo. Embora o presente trabalho vise a criação de uma ferramenta de *fuzzing* USB capaz de operar "às cegas", qualquer capacidade de instrumentação ou monitoramento mais avançado do sistema alvo pode ser aproveitada para melhorar a eficácia do *fuzzer*. Na ausência de tal capacidade, a ferramenta irá operar continuamente e um oráculo disponível será, por exemplo, uma observabilidade visual, indicando se o aplicativo alvo (neste caso, o Skype) está funcionando.

Embora o bug em questão não tenha sido profundamente documentado, é possível inferir que o *fuzzer* proposto é capaz de atingir o mesmo resultado. A seção 4.8 descreve a capacidade da ferramenta de mutar pacotes USB; dentre as mutações aplicadas, estão dicionários e *bit flips*.

O dicionário utilizado pela ferramenta contém, de forma enviesada, uma *string* de formato típica que inclui o erro aqui apresentado, embora o estado de erro também seja alcançável através de mutações de *bit flip*. Portanto, é razoável concluir que a ferramenta desenvolvida é capaz de encontrar o mesmo *bug* no Skype.

A velocidade de tal descoberta depende de diversos fatores, incluindo a performance do sistema hospedeiro, randomicidade dos *bit flips* e *corpus* inicial. Com o *corpus* composto pelas capturas apresentadas nesse trabalho, inferiu-se que o *bug* poderia ser encontrado antes da ferramente alcançar o modo exaustivo descrito na seção 4.8, pois, dentre a base construída, com aproximadamente 750 entradas, dispositivos com descritores semelhantes estão presentes. Isso corresponde ao, máximo, um número da ordem de 10^7 mutações.

5.4 Performance

A melhor maneira encontrada para medir a performance da ferramenta foi através do monitoramento do número de enumerações de um dispositivo USB por segundo. A coleta ocorreu de forma similar ao que foi descrito na seção 3.4: o dispositivo USB foi conectado ao sistema alvo, e a ferramenta começou o processo de enumeração contínua, simulando ser um mouse. Foram realizadas 5 medições, cada uma com duração de 60 segundos, e a média do número de enumerações por segundo foi calculada. A tabela 5.1 apresenta os resultados obtidos.

Sistema Hospedeiro	**Enumerações por segundo**
Macbook Air M1	5.10
Smart TV LG UN7300	2.28
PlayStation 4	1.86
Raspberry Pi Zero 2W	1.40
Média	**2.66**

Tabela 5.1: *Performance da ferramenta desenvolvida medida em número de enumerações por segundo.*

Essa métrica é útil avaliar a eficiência da ferramenta, pois reflete diretamente a capacidade do *fuzzer* de explorar o espaço de entrada do sistema alvo em um determinado período de tempo.

Observa-se que a performance varia significativamente entre diferentes sistemas hospedeiros, o que pode ser atribuído às diferenças de capacidade de processamento e demonstra onde está a maior limitação da ferramenta desenvolvida. O valor médio de 2.66 enumerações por segundo não é problemático para a reprodução de *bugs* conhecidos, mas pode ser insuficiente para um grande número mutações em um intervalo tempo pequeno.

Capítulo 6

Conclusão

O presente trabalho abordou a segurança do protocolo USB, uma interface onipresente e crítica em sistemas computacionais modernos, mas que apresenta uma vasta superfície de ataque frequentemente subestimada. Diante da dificuldade de realizar auditorias em *drivers* de código fechado, o objetivo principal foi o desenvolvimento e avaliação de uma ferramenta de *hardware* e *software* capaz de realizar *fuzzing* em uma abordagem *black-box*, sem necessidade de acesso ao código-fonte ou instrumentação interna do sistema alvo.

Para viabilizar essa análise, foi desenvolvida uma ferramenta baseada no `Raspberry Pi Zero 2W`, escolhido por seu custo acessível e suporte a *USB On-The-Go*. A solução de *software* se fundamentou no uso do módulo de *kernel* `raw-gadget`, que permitiu a manipulação de pacotes em baixo nível e a emulação de dispositivos USB completos, superando as limitações de validação impostas por *drivers* padrão como o `FunctionFS`. A ferramenta operou como um *framework* integrador, capaz de aplicar mutações genéricas em capturas de tráfego (PCAP) e, também, reproduzir vulnerabilidades conhecidas a partir de *logs* do `syzkaller` e do `syzbot`.

Os experimentos demonstraram que, apesar dos desafios inerentes à observabilidade em testes *black-box*, é possível inferir o estado de segurança do *host* através de métricas externas. A utilização de uma técnica de *feedback* baseada no processo de enumeração de dispositivos pode ser eficaz para detectar falhas de negação de serviço (DoS) e travamentos no sistema hospedeiro. Dessa forma, o projeto cumpre seu objetivo de entregar uma solução com alta flexibilidade, validando a eficácia do *fuzzing* externo na identificação de vulnerabilidades críticas em implementações da pilha USB.

6.1 Trabalho futuro

O desenvolvimento desta ferramenta abriu diversas frentes de pesquisa que, devido a limitações de *hardware* e escopo temporal, não puderam ser abordadas neste trabalho, mas que representam passos naturais para a evolução do *fuzzing* USB.

No âmbito do suporte a protocolos, a expansão para a especificação 3.0 do USB é uma prioridade. A escolha do `Raspberry Pi Zero 2W` limitou os experimentos às velocidades

do USB 2.0. A migração para plataformas de *hardware* que suportem USB 3.0 em modo *gadget* é trivial em nível de *software* e permitiria explorar uma superfície de ataque mais ampla e complexa. Adicionalmente, a investigação do protocolo *USB Power Delivery (USB-PD)* pode ser um campo promissor, conforme recentemente demonstrado em Kim *et al.*, 2023. Ataques focados na negociação de tensão e corrente podem explorar vulnerabilidades físicas e lógicas nos controladores de energia dos hosts, uma vertente ainda pouco explorada.

Em relação às capacidades do *software*, uma melhoria significativa seria a implementação de suporte a transferências isócronas, atualmente não suportadas pelo módulo `raw-gadget`. Isso permitiria o *fuzzing* de *drivers* de dispositivos de *streaming* em tempo real, por exemplo, que dependem desse tipo de transferência para garantir latência.

Por fim, para aprimorar a metodologia de ataque, um modo de operação como *proxy* USB, permitindo que a ferramenta atue simultaneamente como *host* (para o dispositivo real) e *device* (para o alvo), interceptando e mutando o tráfego em tempo real, de forma *Man-in-the-Middle* (MITM) é interessante. Essa arquitetura também facilitaria a exploração de condições de corrida, como em vulnerabilidades do tipo *Time-of-Check to Time-of-Use* (TOCTOU), onde o atacante altera os dados entre o momento da validação e o seu uso efetivo — uma técnica já explorada em projetos como Secure, 2024, mas que carece de implementações acessíveis em ferramentas mais generalistas.

Referências

[BeyondLogic 2025] BeyondLogic. *USB Descriptors*. URL: https://www.beyondlogic.org/usbnutshell/usb5.shtml (acesso em 11/09/2025) (citado na pg. 10).

[Böhme *et al.* 2016] Marcel Böhme, Van-Thuan Pham e Abhik Roychoudhury. "Coverage-based greybox fuzzing as markov chain". Em: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pgs. 1032–1043. ISBN: 9781450341394. DOI: 10.1145/2976749.2978428. URL: https://doi.org/10.1145/2976749.2978428 (citado na pg. 15).

[Cha *et al.* 2015] Sang Kil Cha, Maverick Woo e David Brumley. "Program-adaptive mutational fuzzing". Em: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. SP '15. USA: IEEE Computer Society, 2015, pgs. 725–741. ISBN: 9781467369497. DOI: 10.1109/SP.2015.50. URL: https://doi.org/10.1109/SP.2015.50 (citado na pg. 30).

[Chen *et al.* 2018] Yuanliang Chen, Yu Jiang, Jie Liang, Mingzhe Wang e Xun Jiao. "Enfuzz: from ensemble learning to ensemble fuzzing". Em: *CoRR* abs/1807.00182 (2018). arXiv: 1807.00182. URL: http://arxiv.org/abs/1807.00182 (citado na pg. 17).

[Corbet *et al.* 2005] Jonathan Corbet, Alessandro Rubini e Greg Kroah-Hartman. "Interrupt handling". Em: *Linux Device Drivers*. 3ª ed. Sebastopol, CA: O'Reilly Media, 2005. Cap. 13 (citado na pg. 12).

[Davis 2013] Andy Davis. *Lessons learned from 50 bugs: Common USB driver vulnerabilities*. 2013. URL: https://www.nccgroup.com/media/kkpb02u0/_usb_driver_vulnerabilities_whitepaper_v2.pdf (acesso em 26/10/2025) (citado na pg. 20).

[Euntae Jang 2025] Jonghyuk Song Euntae Jang Donghyon Jeong. *DEF CON 31 Car Hacking Village - Automotive USB Fuzzing*. URL: https://www.youtube.com/watch?v=W_vQ5s1bB30 (acesso em 09/09/2025) (citado na pg. 33).

[Fioraldi *et al.* 2020] Andrea Fioraldi, Dominik Maier, Heiko Eissfeldt e Marc Heuse. "AFL++: combining incremental steps of fuzzing research". Em: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, ago. de 2020 (citado nas pgs. 16, 30).

[R. P. Foundation 2023] Raspberry Pi Foundation. *Raspberry Pi OS (armhf) 2023-05-03 (kernel 5.20)*. 2023. URL: https://downloads.raspberrypi.com/raspios_armhf/images/raspios_armhf-2023-05-03/ (acesso em 23/10/2025) (citado na pg. 22).

[W. Foundation 2025] Wireshark Foundation. *SampleCaptures*. 2025. URL: https://wiki.wireshark.org/samplecaptures (acesso em 26/10/2025) (citado na pg. 27).

[Godefroid *et al.* 2012] Patrice Godefroid, Michael Y. Levin e David Molnar. "Sage: whitebox fuzzing for security testing". Em: *Commun. ACM* 55.3 (mar. de 2012), pgs. 40–44. ISSN: 0001-0782. DOI: 10.1145/2093548.2093564. URL: https://doi.org/10.1145/2093548.2093564 (citado na pg. 15).

[Helin 2025] Aki Helin. *Radamsa - A general-purpose fuzzer*. URL: https://gitlab.com/akihe/radamsa (acesso em 11/09/2025) (citado nas pgs. 17, 30).

[Kim *et al.* 2023] Kyungtae Kim *et al.* "Fuzz the power: dual-role state guided black-box fuzzing for USB power delivery". Em: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, ago. de 2023, pgs. 5845–5861. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/kim-kyungtae (citado na pg. 38).

[KimiNewt 2025] KimiNewt. *Pyshark: Python wrapper for tshark, allowing python packet parsing using wireshark dissectors*. 2025. URL: https://github.com/KimiNewt/pyshark (citado na pg. 28).

[Konovalov 2017] Andrey Konovalov. *Linux kernel: multiple vulnerabilities in the USB subsystem*. 2017. URL: https://www.openwall.com/lists/oss-security/2017/11/06/8 (acesso em 26/10/2025) (citado nas pgs. 20, 26, 33).

[Konovalov 2019] Andrey Konovalov. *Coverage-guided USB fuzzing with Syzkaller*. 2019. URL: https://www.offensivecon.org/speakers/2019/andrey-konovalov.html (acesso em 23/10/2025) (citado na pg. 17).

[Konovalov 2025a] Andrey Konovalov. *USB Raw Gadget — a low-level interface for the Linux USB Gadget subsystem*. 2025. URL: https://github.com/xairy/raw-gadget (acesso em 23/10/2025) (citado nas pgs. 22, 31).

[Konovalov 2025b] Andrey Konovalov. *Patch na lista de discussão do kernel Linux sobre a limitação no tamanho de transferências no Raw Gadget*. URL: https://lore.kernel.org/linux-usb/a6024e8eab679043e9b8a5defdb41c4bda62f02b.1757016152.git.andreyknvl@gmail.com/ (acesso em 09/09/2025) (citado na pg. 18).

[libusb Developers 2025] libusb Developers. *libusb: C library for USB device access*. 2025. URL: https://libusb.info/ (acesso em 11/09/2025) (citado na pg. 12).

[Manès *et al.* 2018] Valentin J. M. Manès *et al.* "Fuzzing: art, science, and engineering". Em: *CoRR* abs/1812.00140 (2018). arXiv: 1812.00140. URL: http://arxiv.org/abs/1812.00140 (citado nas pgs. 15, 17, 30).

[Miller *et al.* 1990] Barton P. Miller, Lars Fredriksen e Bryan So. "An empirical study of the reliability of unix utilities". Em: *Commun. ACM* 33.12 (dez. de 1990), pgs. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279 (citado na pg. 15).

[Pramanik e Tayade 2019] Arijit Pramanik e Ashwin Tayade. "Study and comparison of general purpose fuzzers". Em: University of Wisconsin-Madison, 2019. URL: https://wcventure.github.io/FuzzingPaper/Paper/J19_Study.pdf (citado nas pgs. 17, 30).

[Schumilo *et al.* 2014] Sergej Schumilo, Ralf Spenneberg e Hendrik Schwartke. *Don't Trust Your USB! How to Find Bugs in USB Device Drivers*. 2014. URL: https://blackhat.com/docs/eu-14/materials/eu-14-Schumilo-Dont-Trust-Your-USB-How-To-Find-Bugs-In-USB-Device-Drivers-wp.pdf (citado na pg. 17).

[Secure 2024] Anvil Secure. *USB-Racer*. 2024. URL: https://github.com/anvilsecure/usb-racer (acesso em 26/10/2025) (citado na pg. 38).

[Sergey Bratus 2012] Travis Goodspeed Sergey Bratus. *Facedancer USB: Exploiting the Magic School Bus*. 2012. URL: https://recon.cx/2012/schedule/events/237.en.html (acesso em 26/10/2025) (citado nas pgs. 19, 34).

[Source 2025] Google Open Source. *syzbot*. URL: https://syzkaller.appspot.com/ (acesso em 16/09/2025) (citado nas pgs. 17, 23, 26, 34).

[Tonder e Engelbrecht 2014] Rijnard van Tonder e Herman Engelbrecht. "Lowering the USB fuzzing barrier by transparent Two-Way emulation". Em: *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, ago. de 2014. URL: https://www.usenix.org/conference/woot14/workshop-program/presentation/van-tonder (citado na pg. 18).

[USB Implementers Forum 2025a] USB Implementers Forum. *On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification*. URL: https://www.usb.org/document-library/usb-20-specification (acesso em 11/09/2025) (citado na pg. 9).

[USB Implementers Forum 2025b] USB Implementers Forum. *Universal Serial Bus 2.0 Specification*. URL: https://www.usb.org/document-library/usb-20-specification (acesso em 11/09/2025) (citado nas pgs. 6, 9–11).

[Zalewski 2025] Michał Zalewski. *AFL Bug-o-rama Trophy Case*. URL: http://lcamtuf.coredump.cx/afl/#bugs (acesso em 11/09/2025) (citado na pg. 17).

[Zou *et al.* 2022] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang e Zhiyun Qian. "SyzScope: revealing High-Risk security impacts of Fuzzer-Exposed bugs in linux kernel". Em: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, ago. de 2022, pgs. 3201–3217. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/zou (citado na pg. 17).