Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015

# Framework for Fuzzing USB stacks with Virtual Machines

Tobias Mueller[1]

**Abstract:** The Universal Serial Bus (USB) is a widely deployed technology that connects peripheral devices to computer systems. Despite its popularity and the vast number of existing USB enabled devices, assessing security properties of the USB key-components in an automated fashion has not yet been achieved by existing solutions. While malicious USB devices exist, they still require physical actions such as plugging the device in. At present, arbitrary USB behaviour cannot be implemented in software for easy consumption with virtual machines, thus making it hard to test USB stacks, USB drivers or applications using the functionality a driver exposes.

This paper presents a design to write software defined USB devices, using them to automatically fuzz-test key components of a USB enabled system, and to record abnormal behaviour of systems under test.

The design was prototypically implemented and successfully applied to a simple kernel driver. The results suggest that USB exposes a considerable attack surface and that real attacks are possible. They also demonstrate that the design developed is capable of uncovering flaws in kernel level drivers as well as user space applications. It is also capable of detecting USB stack fingerprints of various operating systems which can enable an attacker to build a physical USB device which, when plugged in to the victims machine, can be used to launch a targeted attack.

**Keywords:** security, usb, fuzzing

[1] Universität Hamburg

# 1 Introduction

The Universal Serial Bus (USB) [Co00] is a widely adopted technology which replaced serial and parallel IO ports in 1996. USB emerged because attaching peripheral devices to a PC was cumbersome and error-prone. This was primarily due to the traditional IO paradigm which mapped the devices into the CPU's IO address space and assigned an interrupt line (IRQ). The design of USB allows attaching, configuring and accessing peripheral devices with low cost and simplicity from the user's perspective. Other benefits include one interface for many devices, automatic configuration, hot pluggability or built-in power supply for the device [AI97]. A modern PC is equipped with USB ports to attach keyboards, mice, cameras, printers, scanners, hard-drives, mobile phones or other devices. Even embedded systems such as printers or mobile phones have the ability to attach devices via USB. Key features of USB include its versatility, its inexpensiveness and support by major operating systems.

When an operating system encounters a device attached via USB, it needs to load the appropriate driver to expose the functionality of the device to the user. Presently, although nowadays some drivers can be written in user-space[3], many drivers still reside in kernel space, not only for legacy reasons but also for performance or convenience purposes [CRK05].

Even though kernel developers are usually very experienced and kernel code is subject to repeated review, varying code quality for USB drivers can be expected for multiple reasons. Not only do many different devices exist which require numerous different drivers, but these devices will also have time-to-market constraints which do not allow for driver code security audits. Examining the history of the Linux kernel source code[4] supports that claim.

If those drivers in kernel space were vulnerable, an attacker could gain elevated privileges because it can be assumed that the kernel runs with the highest privileges on the system.

Several concrete attack scenarios exist. An attacker could manipulate elections if a USB-based voting device [ABS07] was used. Publicly available computers with USB interfaces could be attacked by simply plugging in a malicious device. Such a public computer could be a photo-terminal used to load photos to be printend from a pendrive or a digital camera, a PC in a library or an unsupervised machine in a shop. Even some aircraft have an in-flight entertainment system which allows USB devices to be plugged in to listen to songs on portable music players or view documents on a pendrive [Th06]. But the attacker does not necessarily need to physically attach a malicious device herself. Simply distributing a new and expensive looking device (i.e. a digital camera or music player) will most likely lure the victim to plug the device into her computer and thus allow the attacker gain control over it. It is also possible to run USB over IP networks (using USB/IP [Hi05]) and Wireless USB [Le07] uses radio technology as the transport layer for USB. Thus, physical access to the targeted machine is not necessarily needed.

[3] Using libusb, which provides a high level API to communicate with a USB device

[4] i.e. by executing `git log -grep='overflow'` in the USB driver directory

```
class USBInDeviceDescriptor(Packet):
    name = 'DeviceDescriptor'

    fields_desc = [
        LEShortField('bcdUSB', 0x0200),
        ByteEnumField('bDeviceClass', 0, CLASS_ENUMS),
        ByteEnumField('bDeviceSubClass', 0, SUBCLASS_ENUMS),
        ByteEnumField('bDeviceProtocol', 0, PROTOCOL_ENUMS),
        ByteField('bMaxPacketSize', 64),
        LEXShortEnumField('idVendor', 0xffff, VENDOR_ENUMS),
        LEXShortField('idProduct', 0x1337),
        LEShortField('bcdDevice', 0x2342),
        ByteField('iManufacturer', 0),
        ByteField('iProduct', 0),
        ByteField('iSerialNumber', 0),
        ByteField('bNumConfigurations', 0),
    ]
```

List. 1: Python code representing a device descriptor using Scapy

Given the popularity of USB and the expected privileges, exploiting USB is very attractive to an attacker. Consequently, being able to uncover flaws and to fix them is important. Finding bugs, however, often is a non-trivial task, especially in the case of USB, quite simply because providing arbitrary input for the USB driver is not easily possible today, without a USB development board which allows delivery of arbitrary data.

This work extends an existing approach which concentrated on mutation-based fuzzing of the communication between a physical device and a host [JJ10]. Contrasting that existing work, a method to attach USB devices to a virtual machine and to generate arbitrary packets in software is presented here. As the USB devices are written in software , tests can be run without needing to physically attach real USB hardware. Additionally, tests are conducted using packets with payload produced by “fuzzing”.

The contributions of this paper are:

- software-defined USB devices with fuzzing capabilities,
- a framework for executing automated fuzz tests, and
- reliable detection of operating systems through fingerprinting.

This paper is structured as follows. Section 2 describes fuzzing techniques and why fuzzing is a viable approach to identify vulnerabilities. Section 3 lists the components involved around a USB-enabled system. Section 4 discusses how the components can be tested using the described techniques. Section 5 details how the implementation was performed and results obtained. Section 6 concludes this paper and gives an overview of future work.

## 2 Fuzzing

This section describes the fuzzing technique that will be used to search for vulnerabilities in USB stacks, drivers and applications.

"Fuzzing" (or "fuzz testing") was coined in the late 80's while trying "*to evaluate the robustness of various UNIX utility programs, given an unpredictable input stream*" [TDM08]. With the development of better tools, such as "american fuzzy lop"[5] or "Address Sanitizer"[6], fuzzing gained popularity as a security method. Fuzzing results are likely to be exploitable, because the input, which is necessary to produce the erroneous state, is generated during the fuzzing process. Fuzzing is also well suited to expose bugs in kernel level drivers as shown by Keil and Kolbitsch in [KK07].

Tradionally, fuzzing is used to conduct blackbox and state-less tests with randomly generated bytes. Blackbox because the fuzzer does not know anything about the tested program. Since knowledge of internals of the program is not required, this general purpose approach is widely applicable. The drawback of this method, however, is its inability to test either specific input fields or complex protocols. State-less because the fuzzer generates bytes independently of each other regardless of the context or semantics of that byte. Thus, if the input is expected to contain a checksum over the randomly generated bytes and that checksum is not properly updated, the program aborts processing and the fuzzer tests only the checksum validation code. Those fuzzers are labelled "dumb".

"Smart" fuzzers take into account properties of the expected input and can thus force the tested program to execute more code paths. Also, specific fields of the expected input can be tested, i.e. the field indicating a length. This allows the generated payload to penetrate deeper into the tested program, thus testing more aspects of the program. This is also known as "schema-based fuzzing" because a known pattern is modified [NN08].

Because the USB protocol is stateful and most packets have a fixed structure, this paper follows such a schema-based approach using Scapy [Bi10a]. Scapy is a framework that is designed to interact with packets on ethernet networks. It is primarily used to craft and manipulate packets on various network layers resulting in a powerful networking framework to rapidly create tools for conducting various tests, "*but instead of dealing with a hundred line C program, you only write 2 lines of Scapy*" [Bi10b]. It also provides a `fuzz()` function to generate packets with random yet appropriate values for the packet's fields. This includes the width in bits of the generated value as well as adjusting modelled dependencies such as checksums or length indicating fields. Listing 1 shows how Scapy can be use for USB packets, instead of network packets.

## 3 USB Components

This section describes the key components that an operating system needs in order to support USB. Some of these components will later be targets of attacks.

[5] `http://lcamtuf.coredump.cx/afl/`
[6] `https://code.google.com/p/address-sanitizer/`

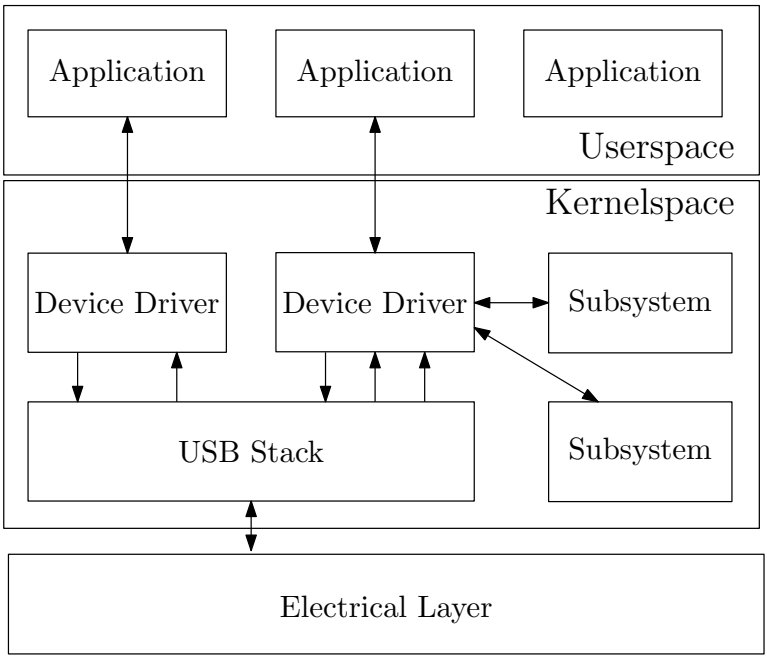


Fig. 1: Key components of a system with USB support

Figure 1 shows the fundamental parts of a USB system. At the bottom a USB device is connected with the USB controller on the host at the electrical layer. The USB controller is controlled by the USB stack which takes care of low level USB message passing and other USB protocol-related aspects such as error correction. The USB stack sends and receives messages from either sources or sinks on the device, called endpoints. Connections to endpoints build logical pipes between the host and the device which are exported to drivers. Upon the USB stack, various drivers are implemented which use the provided USB messaging capabilities to exchange messages either via control, isochronous, bulk or interrupt transfer packets. The drivers may reuse already existing functionality from one or more subsystems (i.e. SCSI for hard-drives or v4l for video cameras). Applications can then use the device through one of the various interfaces that a driver could expose (i.e block device or `mmap()`).

After a device has been plugged in, the operating system asks the device for its details in order to establish the pipes and to know which driver to load. This process is called "enumeration" and it involves so called descriptors to be sent from the device to the host. Every USB device is asked by the USB standard to answer a set of commands that might be issued during enumeration.

As shown in Figure 2, four descriptor types exist: device, interface, endpoint and string descriptor. The very first descriptor requested by the operating system is the device descriptor which describes basic aspects of the device in question such as a globally unique

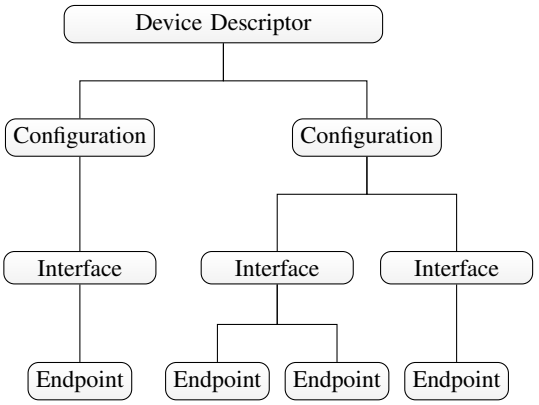

Fig. 2: USB Descriptor hierarchy

vendor ID, a product ID, the "class" of the device and the number of "configurations" which the operating system can select the device to run under. A configuration describes, among other things, whether the device is self-powered or how many "interfaces" the devices exposes. The class indicates that a device speaks a well defined protocol to expose its functionality. Defined classes include mass-storage, audio or video. A device can also announce to not belong to a prefined class or to implement several classes. In the latter case the class information is attached to the interface which is described by an interface descriptor. It also contains information about the number of available endpoints which in turn have their own descriptor defining the packet size and the interval for the host to poll for new messages [Co00].

Once the enumeration is complete, the kernel loads the appropriate driver. To find a suitable driver the kernel first[7] looks at the vendor ID and product ID. If it finds a driver that claims to be responsible for that device it will be loaded. If no driver is found a generic driver for the class of the device (or the interface) will be sought. The driver is then reponsible for exposing features of the device to the user [Ve08].

These components can be attacked using fuzz-testing. The electrical layer, however, is not of interest in this work because we focus on fully automated virtualised testing. Thus physical connections are out of scope. The USB stack can be attacked, i.e. by signalling many device attachments or by interfering with the enumeration process. Knowing the driver-loading mechanism and the structure of the descriptors allows us to trigger a certain driver to be loaded which in turn enables us to send fuzzed messages to the driver and thus assess its robustness. Furthermore, it is possible to send messages to applications if the higher level protocol is known. However, due to their limited applicability and impact, attacks on applications are not as compelling as their driver counterparts. Therefore, focus was given on rendering the latter possible.

[7] This is considering the Linux kernel, but other operating systems do it similarly

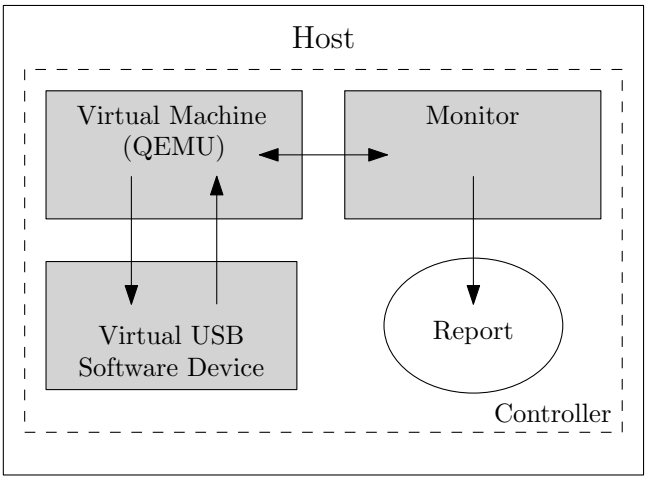


Fig. 3: Architecture of the automated fuzz-testing setup

## 4 Implementation

To find vulnerabilities in the components mentioned in Section 3 using the techniques described in Section 2, a prototypical architecture was built which allows automated testing. It extends an existing design [JJ10] such that no physical device needs to be connected to the host in order to find vulnerabilities.

Figure 3 depicts the built setup. To allow automated tests the host operating system runs a fully virtualised guest operating system. Note that it is possible to do the testing on the host itself. However, potential crashes of the kernel and lost log-files are inherent drawbacks of that approach. Instead of talking to physical devices, the virtual machine pipes USB communication in and out to a separate process on the host which behaves like a regular USB device. The virtual device then generates USB packets using fuzzing. A monitor watches the guest operating system and reports if unexpected behaviour, i.e. a crash, occurs. A controlling component is responsible for actually spawning the above mentioned components.

The actual implementation uses QEMU [Be05] as virtualisation software to run the guest operating system. QEMU is a free virtualisation solution that fully virtualises different CPUs along with the necessary hardware such as hard-drive, network interface or USB controller. It allows complete and unmodified operating systems to be run in a virtual machine. It also allows the machine state to be saved and loaded at a later time. This can be used to prepare a virtual machine that is fully booted or has a special program started so that testing will take less time. QEMU supports physical USB devices to be passed from the host to the guest. It also supports simple virtual internal USB devices such as keyboard or mouse implemented inside QEMU to allow delivery of keystrokes or mouse movements from the host to the guest.

QEMU was enhanced in two steps: First with a USB packet filter functionality and then with support for external virtual USB devices. The latter is depicted in Figure 3. The USB packet filter intercepts communication between the guest operating system and a USB device attached to a Linux host[8]. The intercepted packets are piped out to a process on the host and read back in. The process on the host can thus either write the packets out to disk or modify them in place.

In order to carry out the schema-based fuzz-testing we need to obtain the schema first. Either valid communication patterns can be crafted by looking at the protocol specification or valid USB packets can be recorded for subsequent decoding. We chose to record valid communication by attaching a physical USB device to the host and thenn pass it through to QEMU. The guest running in QEMU started to communicate with the USB device and the above mentioned USB filter functionality was used to record the USB conversation.

Since the filter allows packets to be modified in place as they are in transit, this would already enable fuzz-testing. However, it would not be automatable since a physical device needed to be plugged in to the host. Also, only dumb fuzzing could be conducted as the obtained packets have not been decoded.

In order to build Scapy models for the packets, the USB specification [Co00] was used to dissect the packets and determine the types of the packet's fields, i.e. `Short` or `IntEnum`. This led to a proper description of the protocol and packets used for enumeration which in turn allowed easy packet generation and modification using Scapy's facilities.

### 4.1 USB Device Emulation

Although QEMU supports virtual internal USB devices as described above, they are not backed by an external program. To keep this framework modular, a second feature for QEMU was implemented: Support for external software-defined USB devices which are communicated with via a pipe so that USB packets are, as with the USB packet filter mentioned above, piped out to the host and read back in. Because the interface to the virtual machine consists of two named pipes only, the internals of the virtual machine are abstracted and implementing a software USB device is much simpler than writing a new virtual internal USB device for QEMU.

The actual fuzzing is done in the USB Device Emulation component which is responsible for generating USB packets and writing them to the virtual machine's USB in-pipe. Hence, the virtual USB device can do as simple things as replaying already obtained packets. Naturally, dumb-fuzzing can be applied on these packets.

Two generations of software USB devices were produced. The first and simple generation reads previously obtained packets and decodes them. A configuration then tells the software device which fields in which type of packet have to be fuzzed before sending them to the host. Obviously, this can resemble dumb-fuzzing by simply configuring all

[8] Porting this filter to other platforms such as BSD should be trivial

packet types and all fields to be fuzzed. The second generation is a stand-alone state-machine which autonomously answers packets from the host. To save time this automaton did not implement every command required by the USB specification but rather only the commands which were used in the initially obtained USB communication. This allowed for implementation of rudimentary USB devices such as web-cams or hard-drives. Again, packet types and fields to be fuzzed need to be configured.

### 4.2 Monitoring

In order to detect whether the tested operating system behaves unexpectedly (i.e. crashes), a monitoring component reports back every predefined number of seconds. It needs to implement a reasonable probe routine which is capable of detecting whether the guest operating system is still fully operational. The probe can be implemented as a simple check whether opening a TCP connection to a port on the guest is successful or as a complex routine that logs on to the guest, retrieves and analyses log-files.

In this framework, the monitoring component is realised as a Python module which is loaded in a separate thread by the controlling component. Thus communication between the controlling component and the monitor is possible using Python objects. The separate thread enables the monitor to accidentally block and not stall the execution of the controlling component. The controller regularly executes the monitor's `is_alive` function and assumes the guest operating system to be dysfunctional after a predefined number of probes have not been answered. The monitor is also reponsible for exposing information that is valuable for identifying potential malfunction, i.e. by writing out log files that were obtained from the guest operating system.

### 4.3 Controlling

A controlling component is responsible for starting the virtual machine, attaching virtual USB devices and alert when the virtual machine does not behave as expected.

To run many fuzz-tests, even on multiple machines, the design presented in this paper does not require elevated privileges and is able to be run in many instances in parallel. Also, the components can easily be exchanged, as long as the interfaces are implemented. The interfaces are simple by design: Two named pipes for the emulated USB device, a Python module for the monitoring instance and a process to be called for the virtual machine. This implies that any type of virtual USB device, not only a fuzzer, can be used in this framework. Given the ability to prototype a USB device in software using a rather high level description for Scapy, this framework might be helpful for driver programmers to test whether their implementation fulfils functional requirements.

# 5 Evaluation

This section summarises the findings the implementation mentioned in Section 4 yielded.

While testing the second generation emulated USB device with different guest operating systems different behaviour in the enumeration process was observed. Variations are caused by different implementations of the USB stack. Table 1 shows different behaviour

| **Operating System** | **Packet Sequence** | **Retries** | **Remarks** |
|---|---|---|---|
| Windows XP SP 2&3, 7 | SETUP, IN, OUT | 3 | IN length: 64 |
| Linux 2.6.33 | SETUP (9 times), RESET | 4+2 | 4 get descr. then 2 set addr. |
| OpenBSD 4.7 | SETUP, IN, OUT | 7 | IN length: 8 |
| FreeBSD 8.0 | SETUP, IN, OUT | 6 | sets address right away |

Tab. 1: USB Stack Fingerprints of various operating systems

of various USB stacks. These samples were obtained by attaching a software USB device that answers with zero bytes only. It it thus possible for a USB device to tell operating systems apart. This enables an attacker to launch platform specific attacks.

It was also observed, that QEMU's virtual USB controller sent uninitialized memory to the device with "IN" transactions, which is similar to Etherleaking which exposes memory via padding for Ethernet packets [Bi10b]. Even if this "USB Leaking" does not happen with real hardware USB controllers it can be used by a USB device to be able to detect whether it is attached to a virtual machine.

The USB stack of the guest operating systems were tested rather by accident than on purpose. While testing whether the implemented USB software devices work, they often crashed and disrupted the communication and produced errors in the guest's USB stack. All errors were handled gracefully by generating a proper error message and thus no flaws were exposed. Trying to rapidly attach many USB devices only uncovered bugs in QEMU and not in the USB stack of the guest operating system. In fact, QEMU monopolizes the CPU after attaching 40 devices to the guest.

To test whether USB drivers could be exploited, a Linux driver with a buffer overflow vulnerability was dedicatedly written. The framework was configured to make the guest load the driver. Upon receiving a specially formatted message, the driver crashed the kernel and left the system in an unoperational state. The monitor realised the machine malfunctioning and reported it to be dead. The controller raised an alarm and made necessary informaton available that is needed to reproduce the crash. Existing drivers, however, were not tested due to time constraints.

The previous paragraphs show that this prototypical implemenation is capable of finding flaws in the identified key components of a USB enabled system. Future work should concentrate on making the presented framework test existing drivers of different operating systems.

# 6 Conclusion

This section will conclude the achievements, discuss limitations and show how future work can further improve the presented work.

Firsly, the need for evaluating the security of USB key components was identified as being necessary when assessing the security of a USB enabled system in Section 1. The reason being the USB drivers in kernel space, which are a very compelling target for an attacker not only because of the elevated privileges but also because of the wide deployment of USB. Secondly, fuzzing techniques were introduced, classified and discussed in Section 2. Furthermore, it was shown that fuzzing is well suited to expose bugs in kernel level drivers. Thirdly, an overview of USB and its key components was given in Section 3. Three layers were identified as being attackable with the presented framework: USB stack, drivers on top and applications using the exposed functionality. Fourthly, a design to automatically fuzz-test the identified components was presented and work that was necessary to build the architecture was outlined in Section 4. In order to build the framework various free software products had to be patched, including Scapy and QEMU. The former was improved by adding new fields to describe packets used in USB communication. The latter was enhanced with a USB packet filter as well as support for external software USB devices backed by named pipes.

The presented architecture uses a software-based USB device to generate fuzzed packets. As of now, the software device supports enumeration and parts of the mass-storage protocol only. More protocols are needed to penetrate deeper into the drivers.

The architecture is universal in the sense that knowledge of the guest operating system is not required. If Linux is the guest operating system, we do, however, know about the inner workings. Unfortunately, this information is not taken account when building the fuzzed packets.

Finally, obtained results were presented in Section 5 and they identify the design as being capable of finding flaws in USB kernel drivers. Furthermore, USB stacks of different operating systems have been fingerprinted. This knowledge enables attackers to detect which host operating system they are attached to and to subsequently launch targeted attacks.

A limitation of the presented work is the lack of support for the already specified USB-3. While the general framework should also work with USB-3 enabled operating systems, additional features need to be implemented in the virtual USB device in order to test new features of USB-3 such as device initiated communication.

The fully virtualised approach may also not uncover time critical bugs. USB operates in frames of one millisecond and as the fuzzing framework is running in userspace it might not be able to guarantee processing in time. However, the way for easily carrying fuzz-testing with the ability to smartly and precisely fuzz fields of USB packets has been paved and the modular design of the framework makes it easy to be adapted.

## References

[ABS07] Arzt-Mergemeier, Joerg; Beiss, Willi; Steffens, Thomas: The digital voting pen at the Hamburg elections 2008: electronic voting closest to conventional voting. In: Proceedings of the 1st international conference on E-voting and identity. Springer-Verlag, Bochum, Germany, pp. 88–98, 2007.

[AI97] Anderson, Don; Inc., MindShare,: USB system architecture. Addison-Wesley Developers Press, Reading Mass., 1997.

[Be05] Bellard, Fabrice: QEMU, a fast and portable dynamic translator. In: Proceedings of the annual conference on USENIX Annual Technical Conference. USENIX Association, Anaheim, CA, pp. 41–41, 2005.

[Bi10a] Biondi, Philippe: , Scapy - a powerful interactive packet manipulation program. https://www.secdev.org/projects/scapy/, August 2010.

[Bi10b] Biondi, Philippe; Raynal, Fred; Martini, Sebastien; Kacherginsky, Peter: Scapy v2.1.1-dev documentation. April 2010.

[Co00] Compaq; Hewlett-Packard; Intel; Lucent; Microsoft; NEC; Philips: , Universal Serial Bus Specification, April 2000.

[CRK05] Corbet, Jonathan; Rubini, Alessandro; Kroah-Hartman, Greg: Linux device drivers. O'Reilly Media, Inc., 2005.

[Hi05] Hirofuchi, Takahiro; Kawai, Eiji; Fujikawa, Kazutoshi; Sunahara, Hideki: USB/IP: a peripheral bus extension for device sharing over IP network. In: Proceedings of the annual conference on USENIX Annual Technical Conference. USENIX Association, Anaheim, CA, pp. 47–60, 2005.

[JJ10] Jodeit, Moritz; Johns, Martin: USB Device Drivers: A Stepping Stone into your Kernel. In: 2010 European Conference on Computer Network Defense. IEEE Computer Society, Berlin, Germany, October 2010.

[KK07] Keil, S.; Kolbitsch, C.: Stateful Fuzzing of Wireless Device Drivers in an Emulated Environment. Black Hat Japan, 2007.

[Le07] Leavitt, Neal: For Wireless USB, the Future Starts Now. Computer, 40(7):14–16, 2007.

[NN08] Neystadt, John; Natanov, Nissim: , Application - Testing Software Applications with Schema-based Fuzzing, April 2008.

[TDM08] Takanen, Ari; DeMott, Jared; Miller, Charles: Fuzzing for Software Security Testing and Quality Assurance. Artech House, 1 edition, June 2008.

[Th06] Thales: , Thales Inflight Entertainment Systems, November 2006.

[Ve08] Venkateswaran, Sreekrishnan: Essential Linux device drivers. Prentice Hall, Upper Saddle River NJ, April 2008.